Generative Programming

Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models

by

Dipl.-Inf. Krzysztof Czarnecki

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doktor-Ingenieur

Department of Computer Science and Automation

Technical University of Ilmenau

October 1998

Oral defense held on August 9, 1999

Abstract

Current object-oriented (OO) and component technologies suffer from several problems such as the lack of analysis and design methods for the development for reuse, lack of effective techniques for dealing with many variants of components, loss of design knowledge due to the semantic gap between domain abstractions and programming language features, and runtime performance penalties for clean and flexible design.

This thesis proposes Generative Programming (GP) as a comprehensive software development paradigm to achieving high intentionality, reusability, and adaptability without the need to compromise the runtime performance and computing resources of the produced software.

In the area of analysis and design for GP, we investigate Domain Engineering (DE) methods and their integration with OO analysis and design (OOA/D) methods. The main difference between DE methods and OOA/D methods is that the first are geared towards developing whole families of systems while the latter focus on developing single systems.

We identify feature modeling as the main contribution of DE to OOA/D. Feature models represent the configurability aspect of reusable software at an abstract level, i.e. without committing to any particular implementation technique such as inheritance, aggregation, or parameterized classes. We give a precise and extended formulation of the feature diagram notation and investigate the relationship between feature modeling, OO modeling and Aspect-Oriented Programming.

In the area of implementation for GP, we study various metaprogramming technologies. We identify modularly extensible programming environments as the ideal programming platform for GP, which allows implementing domain-specific optimizations, domain-specific displaying and editing, domain-specific debugging and code analysis, new composition mechanisms, etc., in a scalable way. We also propose new implementation techniques such as configuration generators based on mixin models with automatic configuration and configuration repositories and make several contribution to template metaprogramming.

Based on the analysis of the areas mentioned above, we propose a new Domain Engineering method for the development of algorithmic reusable libraries (DEMRAL), which integrates various DE, OO, and AOP concepts. We validate the method by applying it to the domain of matrix computations, which results in the development of the Generative Matrix Computation Library (GMCL). We provide two implementation GMCL, one using generative programming techniques in C++ and another one in Intentional Programming (an modularly extendible programming environment).

In addition to validating the usefulness of DEMRAL, the GMCL case study provides a concrete comparison of two generative implementation technologies. The C++ implementation of the matrix component (which is a part of C++ GMCL) comprises only 7500 lines of C++ code, but it is capable of generating more than 1840 different kinds of matrices. Despite the large number of provided matrix variants, the performance of the generated code is comparable with the performance of manually coded variants. The application of template metaprogramming allowed a highly intentional library API and a highly efficient library implementation at the same time. The implementation of GMCL within the Intentional Programming system (IP) demonstrates the advantages of IP, particularly in the area of debugging and displaying.

Acknowledgements

First I would like to thank my wife Katja for her love, encouragement, and patient support. I am also grateful to my parents for giving me best of their life and the necessary education.

I would like to thank Dietrich Reschke, the chairman of supervisory committee for his invaluable advice, excitement, and generous support for this work. I am very grateful to Ulrich Eisenecker, one of the supervisors of this thesis, for encouraging me to work in this area, for giving me his professional guidance and advice throughout all phases of this work, for sharing many inspiring thoughts during our close research cooperation on Generative Programming, and for carefully reading and commenting hundreds of pages of all my professional writing. I also would like to express special thanks to Mehmet Aksit for his valuable suggestions and encouragement and for kindly accepting to be one of the supervisors of this thesis. Many thanks also go to Reinhold Schönefeld, who supervised the initial phase of my doctorial research.

During my research on this thesis I had the pleasure to supervise two brilliant students Tobias Neubert and Johannes Knaupp. Tobias implemented the full version of the generative C++ matrix component and is also responsible for many improvements of the configuration generator concept. Johannes contributed several improvements of the template metaprogramming techniques, extended the C++ matrix component, and developed a configuration generator for the family of LU matrix factorization algorithms (work directly building on this thesis).

This thesis would not have been finished without the generous support and encouragement of my friends and colleagues at Daimler-Benz AG Research and Technology in Ulm, especially my managers Wolfgang Hanika and Roland Trauter and my colleagues from the OSVA project team Wolfgang Köpf and Michael Jungmann.

Special thanks go to Charles Simonyi, Greg Shaw, and the entire Intentional Programming (IP) team at Microsoft Research for patiently teaching me in the ways of IP and helping me with the IP implementation of the matrix case study during my research visit in January 1998.

I would also like to thank Gregor Kiczales, Crista Lopes, John Lamping, and other members of the Aspect-Oriented Programming (AOP) team at Xerox PARC for explaining to me the AOP ideas, providing me with an early version of AspectJ, and sharing many stimulating thoughts that lead to the development of Dynamic Cool during my research visit in October/November 1997.

I am grateful to Mark Simos who shared with me both his invaluable insights into the area of Domain Engineering and his excitement for this area. I also would like to express my thanks to Don Batory for the advice and positive feedback he gave me. Attending his tutorial on generators at ICSR’96 provided me with important motivation for doing this research. Many thanks also go to Karl Lieberherr for his positive feedback and the many stimulating discussions we had.

During the Dagstuhl seminar on “Generic Programming” in March 1998, Ulrich Eisenecker, Robert Glück, Daveed Vandevoorde, and Todd Veldhuizen and I started an informal working group on Active Libraries. Our numerous discussions at Dagstuhl and later per e-mail greatly influenced the last steps of this work. I would like to thank all of the group members.

During several conferences, I had the pleasure to meet many interesting people from the reuse community and the object-oriented community who gave me useful feedback and shared with me their ideas. My thanks go to Ted Biggerstaff, Ulrich Breymann, Mike VanHilst, Sholom Cohen and the participants of the WISR’97 working group on Domain Engineering and Object Technology, Andrew Lumsdaine, Satoshi Matsuoka, David Quinlan, Jeremy Siek, Yannis Smaragdakis, Douglas Smith, Partick Steyart, Bruce Weide, and many others. I also would like to thank many other researchers, including Ira Baxter, John Favaro, Martin Griss, and Jim Neighbors, who played important roles in helping me form my ideas.

In addition to the supervisors, several other researchers, including William Harrison, Gregor Kiczales, Karl Lieberherr, Harold Ossher, Greg Shaw, Mark Simos, and Peri Tarr gave me invaluable comments on various chapters of earlier drafts of this thesis.

I also would like to thank Bogdan Franczyk and Witold Wendrowski for their professional support and friendship.

Funding for this work has been provided by Daimler-Benz AG and German Federal Ministry of Education, Science, Research and Technology (BMBF) through the OSVA project.

Knowing that this list is certainly not exhaustive, I hope that those who I have forgotten to mention will forgive me.

Overview

Detailed Table of Contents

List of Figures

List of Tables

Part I

SETTING THE STAGE

Chapter 1 What Is this Thesis About?

1.1 In a Nutshell

This thesis proposes Generative Programming (GP) as a comprehensive software development paradigm to achieving high intentionality, reusability, and adaptability without the need to compromise the runtime performance and computing resources of the produced software. In the area of analysis and design for GP, Domain Engineering methods and their integration with OO analysis and design methods are investigated. Furthermore, implementation technologies for GP (especially metaprogramming technologies) are studied. Based on both analyses, a Domain Engineering method for the development of algorithmic reusable libraries (DEMRAL) is proposed. The method is then used to develop a generative matrix computation library (GMCL). Finally, GMCL is implemented using generative programming techniques in C++ and in Intentional Programming (an extendible programming environment). This case study validates the usefulness of DEMRAL and provides a concrete comparison of two generative implementation technologies.

1.2 Who Should Read It and How to Read It

This work covers a broad spectrum of topics related to Generative Programming including theoretical, methodological, and technical issues. Thus, it is relevant to researchers, methodologists, and practitioners.

If you are a researcher, this work offers you a comprehensive reference in areas of Domain Engineering (Chapter 3), integration of Domain Engineering and OOA/D (Chapter 4), Generators (Chapter 6), and Aspect-Oriented Programming (Chapter 7). The new Domain Engineering method described in Chapter 9 will give you a valuable example of a specialized Domain Engineering method incorporating OO and aspect-oriented concepts. Chapter 10 documents the (as of writing) most comprehensive and publicly available case study in Domain Engineering. Several contribution of this work to the state-of-the-art research are listed in Section 1.9.

If you are a methodologist, you will learn why current OOA/D methods do not support development for reuse (e.g. development of frameworks and component libraries) and how to change that. The most interesting chapters for a methodologist are Chapters 3, 4, 5, 9, and 10. Specifically, you will

- learn about the role of Domain Engineering and how it addresses software reuse (Chapter 3);
- learn why it is important to integrate Domain Engineering with OOA/D and how to do it (Chapters 4 and 5);

- get a concrete method illustrating such an integration (Chapter 9) and a well documented case study of applying this method (Chapter 10);
- learn about directions that will inevitably influence methods in future (e.g. Metaprogramming in Chapter 6 and Aspect-Oriented Programming in Chapter 7).

If you are a practitioner, you will learn how to apply generative programming using today's technologies. You will also learn about new directions that will revolutionize software development within the next decade. The most interesting chapters for a practitioner are Chapters 6 (especially Section 6.4.2), 7, 8, and 10. Specifically, you will

- get a "canned" solution for writing reusable and highly-efficient C++ code, i.e. achieving reusability and adaptability without incurring unnecessary runtime costs (Chapters 6, 8, 9, and 10);
- learn about new directions in software development, which are already considered as strategic in the industry, e.g. Microsoft's Intentional Programming (in Section 6.4.3);
- learn some principles of good design (e.g. Aspect-Oriented Programming in Chapter 7) which, to a limited extent, can already be applied using current tools and languages.

1.3 Motivation

1.3.1 Advantages of the Object-Oriented Paradigm

It was also because of the advantages of the object-oriented (OO) paradigm that the 90's have seen such a wide use of OO technologies. We can summarize these advantages using the object model:

- *Classes and objects*: Classes correspond to the concepts of a problem domain and objects represent the concrete exemplars of these concepts. Classes provide a natural way to break down complex problems into smaller problems and to effectively describe sets of similar objects.
- *Encapsulation and information hiding*: Thanks to encapsulation, we can base our designs on interfaces and exchange objects with compatible interfaces. Information hiding means that the internals of an object remain hidden which promotes modularity.
- *Inheritance*: Inheritance allows us to compose and modify the characteristics of objects. Inheritance represents one of the key mechanisms for organizing hierarchical designs and achieving code reuse.
- *Dynamic polymorphism*[1] *and dynamic binding*: Using polymorphism, we can write code that works with different types. Thanks to dynamic binding, we can vary these types at runtime. Both mechanisms play the pivotal role in writing compact and flexible code and in achieving better reusability.
- *Object identity*: Object identity gives us a natural way to reference objects.

[1] Dynamic polymorphism in OO refers to the ability to respond to the same message by executing different method implementations depending on the type of the receiver, i.e. the implementation is selected at runtime. Dynamic binding is an implementation mechanism for achieving dynamic polymorphism. With static polymorphism, on the other hand, the implementation is selected at compile time (e.g. as in static method calls on instances of C++ template parameters).

There are also other advantages, e.g. the use of the same modeling concepts in OOA/D[2] and in the implementation, the significant level of reuse offered by well-designed frameworks and class libraries, the suitability of OO for distributed computing because of their underlying metaphor of sending messages back and forth, etc.

1.3.2 Problems That Still Remain

Despite the advantages listed in the previous section, industrial experience shows that OO technologies fail short to provide all the expected benefits (see e.g. [Web95, pp. 21-22]), particularly, in the areas of

- reuse,
- adaptability,
- management of complexity, and
- performance.

Frameworks and class libraries were long touted as the OO technologies to achieve reuse. While there are examples of successful, well-designed frameworks and class libraries (e.g. Apple's MacApp [Ros95] or Texas Instruments' ControlWORKS [CW]), the number of failures is probably much higher (e.g. Taligent's CommonPoint; most failures remain unpublished, though). The main problems of frameworks and class libraries include the following:

Lack of analysis and design methods for reusable frameworks and class libraries Most OOA/D methods focus on the development of single systems rather than reusable models for classes of systems. Frameworks and class libraries are currently developed ad-hoc rather than systematically. They are often created as a byproduct of application development through an iterative and opportunistic process of abstraction and generalization.

Generalization is usually achieved by introducing *variation points* (also referred to as "hot spots" [Pre95]). Variation points allow us to provide alternative implementations of functional or non-functional features, e.g. different formulas for calculating taxes, different persistency mechanisms, etc. Adding more variation points and alternative features increases the *horizontal scope* of a framework meaning that more specific applications can take advantage of it (see Figure 1 for the explanation of the concepts of horizontal and vertical scope).

There are serious problems with an ad-hoc generalization process. On the one hand, there is a good chance that some of the relevant variation points will not be identified. On the other hand, many developers posses a natural tendency to over-generalize which may result in the introduction of unnecessary variation points and thus superfluous complexity.

Problems with horizontal scaling The reusability of a framework or a class library can be increased by adding new features and variation points. Unfortunately, as the number of variation points increases, the complexity of the framework rapidly grows while its performance usually deteriorates.[3] Variation points are typically implemented using design patterns described in [GHJV95], such as the strategy or the state pattern. With each additional variation point, several new classes have to be introduced and some old classes need to be refactored. This results in so-called "fragmentation of the design" and thus increased complexity. While enlarging the horizontal scope of a framework inevitably increases its complexity (since new cases have to be covered), some of this complexity results from the fact that OO modeling and implementation languages do not provide language constructs to express design patterns more declaratively (or intentionally) and to replace the corresponding, more complex, implementation

[2] Object-Oriented Analysis and Design Methods

[3] The relationship between horizontal scaling and performance is discussed in [Big94] and [Big97] as the *library scaling problem*.

idioms.[4] Furthermore, when using framework technology, adding certain kinds of features (such as synchronization) might add more complexity than when adding other kinds of features. We discuss this problem next. Finally, design patterns are usually implemented using dynamic binding, even if some variation points remain frozen during runtime. Thus, each design pattern typically introduces additional levels of indirection which remain at runtime and contribute to performance degradation. In summary, enlarging the horizontal scope of a framework or a class library causes both complexity explosion and rapid performance degradation.

Problems with separation of concerns, code tangling, and lack of linear modifiability Adding certain kinds of features such as synchronization or some kinds of optimizations to a simple and clean functional OO model usually leads to a great deal of extra complexity. This is so since such features are usually expressed by small code fragments scattered throughout several functional components. Kiczales et al. refer to such features as *aspects* [KLM+97] and to the intertwined mixture of aspects and functional components as *tangled code*. Tangled code is difficult to read, maintain, adapt, extend, and reuse. It is worth noting that the code tangling problem tends to occur in later phases of the conventional development process. We usually start with a clean, hierarchical functional design, then manually add various aspects (e.g. code optimizations, distribution, synchronization), and the code becomes tangled. In addition to inserting aspect code at different locations, adding an aspect to an existing design often requires refactoring. For example, when we want to synchronize only a portion of a method (e.g. for performance reasons), we usually factor out this portion into a separate method. This is aggravated by the fact that different aspects usually require refactoring along different structures. Finally, the separation of aspects is insufficient not only in the implementation code, but also in the design models (e.g. class diagrams). A related concept to aspects and code tangling is *linear modifiability* [Sim98]. We say that a system is linearly modifiable with respect to some set of properties if modifying any of these properties requires only a well-localized change. Of course, we want any important system property to be linearly modifiable. As described above, in conventional OO software, aspects represent properties which are not linearly modifiable.

“Object collisions”[5] Even if the functionality of two or more class libraries or frameworks seems to be complementary, their simultaneous reuse in one application could be impossible because they use different error handling designs, memory management schemes, persistency implementations, synchronization schemes, etc. There might be different causes for this situation to occur. First, the lack of appropriate analysis and design method could have prevented the identification of the relevant variation points in the frameworks in the first place. Second, the horizontal scaling or the code tangling problem can make it hard to implement all relevant variation points.

---

[4] An *idiom* is usually understood as an implementation of a higher-level modeling concept in a specific programming language which otherwise does not provide an explicit construct for this concept.

[5] This phrase has been coined by Berlin [Ber90].

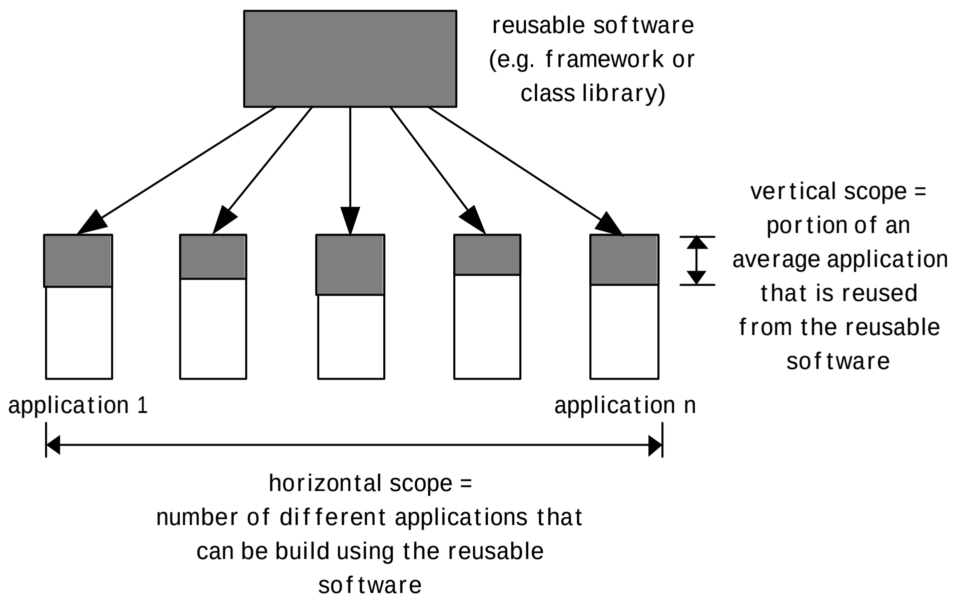

reusable software (e.g. framework or class library)
vertical scope = portion of an average application that is reused from the reusable software
application 1
application n
horizontal scope = number of different applications that can be build using the reusable software

Figure 1 *Vertical and horizontal scope of reusable software*

Loss of design information Conventional programming involves the manual translation of domain concepts living in the programmer's head into a source program written in a concrete, general-purpose programming language such as C++ or Java. This translation inevitably causes important design information to be lost due to the semantic gap between the domain concepts and the language mechanisms available in the programming language. Some of the deign knowledge could be recorded in the form of comments, but the computer cannot take advantage of such information. This loss of design information not only makes programs harder to understand, but also creates the two hard problems: the *legacy problem* and the *evolution problem*. In a time, where a new programming language enters the mainstream every few years (e.g. C++, Smalltalk, and then Java), companies have a truly hard time protecting their investment in Information Technology (IT). Today, most designs are expressed in a concrete programming language meaning that the larger share of the design information is lost. Thus, it is not possible to retarget existing application on to new platforms without significant costs. The evolution problem causes the IT costs even more to climb. Software systems, instead of being constantly improved, deteriorate over time and have to be eventually phased out.

1.4 Generative Programming

The previous section listed some problems of current OO technologies. It turns out that problems such as

- lack of A/D methods for developing reusable software in the industrial practice,
- loss of design knowledge due to the semantic gap between domain abstractions and programming language features, and
- runtime performance penalties for clean and flexible design

are not limited to OO technologies, but equally apply to other current technologies, e.g. component technologies such as COM or CORBA.

Generative Programming (*GP*) is about designing and implementing software modules which can be combined to generate specialized and highly optimized systems fulfilling specific requirements [Eis97]. The goals are to (a) decrease the conceptual gap between program code and domain concepts (known as achieving high *intentionality*), (b) achieve high reusability and adaptability, (c) simplify managing many variants of a component, and (d) increase efficiency (both in space and execution time) [CEG+98].

To meet these goals, GP deploys several principles [CEG+98]:

- *Separation of concerns*: This term, coined by Dijkstra [Dij76], refers to the importance of dealing with one important issue at a time. To avoid program code which deals with many issues simultaneously, generative programming aims to separate each issue into a distinct set of code. These pieces of code are combined to generate a needed component.

- *Parameterization of differences*: As in generic programming, parameterization allows us to compactly represent families of components (i.e. components with many commonalities).

- *Analysis and modeling of dependencies and interactions*: Not all parameter value combinations are usually valid, and the values of some parameters may imply the values of some other parameters. These dependencies are referred to as *horizontal configuration knowledge*, since they occur between parameters at one level of abstraction.

- *Separating problem space from solution space*: The problem space consists of the domain-specific abstractions that application programmers would like to interact with, whereas the solution space contains implementation components (e.g. generic components). Both spaces have different structures and thus we map between them with *vertical configuration knowledge*. The term *vertical* refers to interaction between parameters of two different abstraction levels. Both horizontal and vertical configuration knowledge are used for automatic configuration.

- *Eliminating overhead and performing domain-specific optimizations*: By generating components statically (at compile time), much of the overhead due to unused code, run-time checks, and unnecessary levels of indirection may be eliminated. Complicated domain-specific optimizations may also be performed (for example, loop transformations for scientific codes).

1.5 Generative Programming and Related Paradigms

There are three other programming paradigms which have similar goals to Generative Programming:

- Generic programming,

- Domain-Specific Languages (DSLs), and

- Aspect-Oriented Programming (AOP).

Generative Programming is broader in scope than these approaches, but uses important ideas from each [CEG+98]:

Generic Programming may be summarized as “reuse through parameterization.” Generic programming allows components which are extensively customizable, yet retain the efficiency of statically configured code. This technique can eliminate dependencies between types and algorithms that are conceptually not necessary. For example, iterators allow generic algorithms which work efficiently on both dense and sparse matrices [SL98a]. However, generic programming limits code generation to substituting concrete types for generic type parameters and welding together pre-existing fragments of code in a fixed pattern. *Generative* programming is more general since it provides automatic configuration of generic components from abstract specifications and for a more powerful parameterization.

Domain-Specific Languages (DSLs) provide specialized language features that increase the abstraction level for a particular problem domain; they allow users to work closely with domain concepts (i.e. they are higly intentional), but at the cost of language generality. Domain-specific languages range from widely-used languages for numerical and symbolic computation (e.g., Mathematica) to less well-known languages for telephone switches and financial calculations (to name just a few). DSLs are able to perform domain-specific optimizations and error checking. On the other hand, DSLs typically lack support for generic programming.

Aspect-Oriented Programming Most current programming methods and notations concentrate on finding and composing functional units, which are usually expressed as objects, modules, and procedures. However, several properties such as error handling and synchronization cannot be expressed using current (e.g. OO) notations and languages in a cleanly localized way. Instead, they are expressed by small code fragments scattered throughout several functional components. Aspect-Oriented Programming (AOP) [KLM+97] decomposes problems into functional units and *aspects* (such as error handling and synchronization). In an AOP system, components and aspects are *woven* together to obtain a system implementation that contains an intertwined mixture of aspects and components (i.e. *tangled code*). Weaving can be performed at compile time (e.g. using a compiler or a preprocessor) or at runtime (e.g. using dynamic reflection). In any case, weaving requires some form of *metaprogramming*[6] (see Section 7.6.2). Generative programming has a larger scope since it includes automatic configuration and generic techniques, and provides new ways of interacting with the compiler and development environment.

Putting It All Together: Generative Programming The concept of generative programming encompasses the techniques of the previous three approaches, as well as some additional techniques to achieve the goals listed in Section 1.4:

- DSL techniques are used to improve intentionality of program code, and to enable domain-specific optimizations and error checking. Metaprogramming allows us to implement the necessary language extensions.[7]
- AOP techniques are used to achieve *separation of concerns* by isolating aspects from functional components. Metaprogramming allows us to weave aspects and components together.
- Generic Programming techniques are used to parameterize over types, and iterators are used to separate out data storage and traversal aspects.
- Configuration knowledge is used to map between the problem space and solution space. Different parts of the configuration knowledge can be used at different times in different contexts (e.g. compile time or runtime or both). The implementation of automatic configuration often requires metaprogramming.

1.6 Generative Analysis and Design

Generative Programming focuses on designing and implementing reusable software for generating specific systems rather than developing each of the specific systems from scratch. Therefore, the scope of generative analysis and design are families of systems and not single systems. This requirement is satisfied by Domain Engineering. Part of Domain Engineering is Domain Analysis, which represents a systematic approach to identifying the scope, the features, and the variation points of the reusable software based on the analysis of existing applications, stakeholders, and other sources. Domain Analysis allows us to identify not only the immediately relevant features, but also the potentially relevant ones as early as possible. The knowledge of the planned and potential features is a prerequisite for arriving at a robust design capable to scale up.

---

[6] Metaprogramming involves writing programs whose parts are related by the "about" relationship, i.e. some parts are about some other parts. An example of a metaprogram is a program which manipulates other programs as data, e.g. a template metaprogram, a compiler, or a preprocessor (see Section 8.1). Another example are programs implementing the abstractions of a programming language in a reflective way, e.g. metaclasses in Smalltalk (the latter implement the behavior of classes). An example of metaprogramming in Smalltalk is given in Section 7.4.7.

[7] By a language extension we mean capabilities extending the expressive power of a programming language which are traditionally not being packaged in conventional libraries, e.g. domain-specific optimizations, domain-specific error checking, syntax extensions, etc. (see Section 9.4.1).

Furthermore, Domain Analysis helps us to identify the dependencies between variation points. For example, selecting a multi-threaded execution mode will require activating synchronization code in various components of a system or selecting some storage format may require selecting specialized processing algorithms. This kind of explicit configuration knowledge allows us to implement automatic configuration and to design easy-to-use and scalable configuration interfaces, e.g. interfaces based on specialized languages (so-called *domain-specific languages*) or application builders (e.g. GUI builders).

None of the current OOA/D methods address the above-mentioned issues of multi-system-scope development. On the other hand, they provide effective system modeling techniques. Thus, the integration of Domain Engineering and OOA/D methods is a logical next step.

From the viewpoint of OOA/D methods, the most important contribution of Domain Engineering is feature modeling, a technique for analyzing and capturing the common and the variable features of systems in a system family and their interdependencies. The results of feature modeling are captured in a feature model, which is an important extension of the usual set of models used in OOA/D.

We propose methods which

- combine aspects of Domain Engineering, OOA/D, and AOP and
- are specialized for different categories of domains

to be most appropriate for Generative Programming. As an example of such a method, we develop a Domain Engineering Method for Reusable Algorithmic Libraries (DEMRAL), which is appropriate for building algorithmic libraries, e.g. libraries of numerical codes, image processing libraries, and container libraries.

1.7 Generative Implementation Technologies

As noted in the previous section, Generative Programming requires metaprogramming for weaving and automatic configuration. Supporting domain-specific notations may require syntactic extensions. Libraries of domain abstraction based on Generative Programming ideas thus need both implementation code and *metacode* which can implement syntax extensions, perform code generation, and apply domain-specific optimizations. We refer to such libraries as *active libraries* [CEG+98]:

> Active libraries *are not passive collections of routines or objects, as are traditional libraries, but take an active role in generating code. Active libraries provide abstractions and can optimize those abstractions themselves. They may generate components, specialize algorithms, optimize code, automatically configure and tune themselves for a target machine, and check source code for correctness. They may also describe themselves to and extend tools such as compilers, profilers, debuggers, domain-specific editors, etc.*

This perspective forces us to redefine the conventional interaction between compilers, libraries, and applications. Active Libraries may be viewed as knowledgeable agents, which interact with each other to produce concrete components. Such agents need infrastructure supporting communication between them, code generation and transformation, and interaction with the programmer.

Active Libraries require languages and techniques which open up the programming environment. Implementation technologies for active libraries include the following [CEG+98]:

Extensible compilation and metalevel processing systems In metalevel processing systems, library writers are given the ability to directly manipulate language constructs. They can analyze and transform syntax trees, and generate new source code at compile time. The MPC++ metalevel architecture system [IHS+96] provides this capability for the C++ language. MPC++ even allows library developers to extend the syntax of the language in certain ways (for example, adding new keywords). Other examples of metalevel processing systems are Open C++ [Chi95], Magik [Eng97], and Xroma [CEG+98]. An important differentiating factor is whether the metalevel processing system is implemented as a pre-processor, an open compiler, or an extensible programming environment (e.g. Intentional Programming; see Section 6.4.3).

Program Specialization Researchers in Partial Evaluation have developed an extensive theory and literature of code generation. An important discovery was that the concept of *generating extensions* [Ers78] unifies a very wide class of apparently different program generators. This has the big advantage that program generators can be implemented with uniform techniques, including diverse applications such as parsing, translation, theorem proving, and pattern matching. Through partial evaluation, components which handle variability at run-time can be automatically transformed into *component generators* (or generating extensions in the terminology of the field, e.g. [DGT96]) which handle variability at compile-time. This has the potential to avoid the need for library developers to work with complex meta-level processing systems in some cases. Automatic tools for turning a general component into a component generator (i.e. generating extension) now exist for various programming languages such as Prolog, Scheme, and C (see [JGS93]).

Multi-level languages Another important concept from partial evaluation is that of two-level (or more generally, multi-level) languages. Two-level languages contain static code (which is evaluated at compile-time) and dynamic code (which is compiled, and later executed at run-time). Multi-level languages [GJ97] can provide a simpler approach to writing program generators (e.g., the Catacomb system [SG97]).

The C++ language includes some compile-time processing abilities quite by accident, as a byproduct of template instantiation. Nested templates allow compile-time data structures to be created and manipulated, encoded as types; this is the basis of the *expression templates technique* [Vel95a]. The *template metaprogram technique* [Vel95b] exploits the template mechanism to perform arbitrary computations at compile time; these “metaprograms” can perform code generation by selectively inlining code as the “metaprogram” is executed. This technique has proven a powerful way to write code generators for C++. In this context, we can view C++ as a two-level language.

Runtime code generation (RTCG) RTCG systems allow libraries to generate customized code at run-time. This makes it possible to perform optimizations which depend on information not available until run-time, for example, the structure of a sparse matrix or the number of processors in a parallel application. Examples of such systems which generate native code are `C (Tick-C) [EHK96, PEK97] and Fabius [LL95]. Runtime code modification can also be achieved using dynamic reflection facilities available in languages such as Smalltalk and CLOS. The latter languages also provide their own definitions in the form of extendible libraries to programmers.

1.8 Outline

The thesis consists of several chapters surveying areas related to Generative Programming, a chapter describing the newly proposed Domain Engineering Method for Reusable Algorithmic Libraries (DEMRAL) and a chapter featuring a comprehensive case study which involves the application of DEMRAL to matrix computations. The chapters are organized into four parts:

In the following, we provide a brief summary of each chapter.

Part I *Setting the Stage*

Chapter 1 What is this Thesis About? (The chapter you are just reading)

Chapter 1 introduces Generative Programming and gives an overview of this work.

Problems of current software development technologies. Definition of Generative Programming. Relationship of Generative Programming to other paradigms. Analysis and design methods and implementation technologies for Generative Programming. Outline and contributions.

Chapter 2 Conceptual Modeling

The purpose of this chapter is to show the connection between cognitive science and the modeling of software. It discusses the limitations of the classical modeling and the need to model variability more adequately. It introduces concepts and features, which are central to Domain Engineering.

Concepts and features. Theories of concepts: the classical, the probabilistic, and the exemplar view. Subjectivity of concepts. Generalization and specialization. Abstraction and concretization.

Part II *Analysis and Design Methods and Techniques*

Chapter 3 Domain Engineering

Chapter 3 provides an introduction to Domain Engineering (DE) and a survey of existing DE methods.

Definition of DE. Basic concepts of DE. Survey of DE methods.

Chapter 4 Domain Engineering and Object-Oriented Analysis and Design

Chapter 4 motivates the integration of DE and OOA/D methods and surveys existing work on this topic.

Relationship between DE and OOA/D. Approaches to integrating DE and OOA/D methods. Survey of DE-and-OOA/D integration work.

Chapter 5 Feature Modeling

Chapter 5 provides an in-depth treatment of feature modeling – the key modeling technique in Generative Programming. The presented approach integrates concepts from OO Programming and Aspect-Oriented Programming.

Role of feature modeling. Extended feature diagrams notation. Contents of feature models. Relationship between feature diagrams and UML class diagrams. Process of feature modeling.

Part III ***Implementation Technologies***

Chapter 6 Generators

Chapter 6 discusses existing approaches to generation, one of the key implementation technologies for Generative Programming.

Basic concepts of generators. Compositional and transformational approaches. Transformation systems. Survey of selected approaches to generation (includes C++ idioms for implementing GenVoca architectures and a description of Intentional Programming).

Chapter 7 Aspect-Oriented Decomposition and Composition

Chapter 7 gives an up-to-date account of Aspect-Oriented Programming (AOP), a paradigm promoting a better separation of concerns in programming. AOP is covered by Generative Programming.

Definition of AOP. Survey of AOP work. Relationship to Domain Engineering. Aspect-oriented composition mechanisms (with C++ examples and a Subject-Oriented Programming example). Smalltalk implementation of an aspect-oriented library for thread synchronization. Language extensions and active libraries as implementation technologies for AOP.

Chapter 8 Static Metaprogramming in C++

Chapter 8 discusses template metaprogramming, a C++ programming technique for implementing program generators. This technique is used in the case study.

Basic concepts of template metaprogramming. Writing template metafunctions and control structures for metaprograms. List processing using template metaprograms. Techniques for avoiding partial template specializations. Implementing code generators using template metaprogramming.

Part IV ***Putting It All Together: DEMRAL and the Matrix Case Study***

Chapter 9 Domain Engineering Method for Reusable Algorithmic Libraries (DEMRAL)

Chapter 9 describes DEMRAL, a new Domain Engineering method for developing algorithmic libraries. This method exemplifies the idea of Domain Engineering methods using OO and AOP techniques and specialized for a particular category of domains.

Applicability of DEMRAL. DEMRAL Domain Analysis (including feature starter sets for abstract data types and algorithms). DEMRAL Domain Design and Implementation (including the concepts of configuration and expression domain-specific languages).

Chapter 10 Case Study: Generative Matrix Computation Library (GMCL)

Chapter 10 contains a comprehensive case study involving the application of DEMRAL to the domain of matrix computations. The result is an implementation of a generative matrix computation library in C++ and in IP and a comparison of both implementation technologies.

Domain Analysis of the Domain of Matrix Computation Libraries (including the analysis of existing libraries and application areas and the feature modeling of matrices). Domain Design (including a full specification of a generative matrix component). Domain Implementation (including the detailed description of the matrix component implementation using generative C++ techniques, a summary of the implementation using Intentional Programming, and a comparison of both approaches).

Chapter 11 Conclusions and Outlook

1.9 Contributions

The contributions of this work can be summarized as follows:

1. contributions to feature modeling:

   1.1. precise formulation of the FODA feature diagram notation (Section 5.4.1);

   1.2. extension of the FODA feature diagram notation with or-features (Section 5.4.1.4);

   1.3. new feature modeling concepts such as normalized feature diagrams (Section 5.4.1.5), classification of variation points (Table 8), properties of variation points (homogeneous vs. inhomogeneous, singular vs. nonsingular, simultaneous vs. non-simultaneous; see Section 5.4.1.7);

   1.4. extended list of information categories contained in feature models (see Sections 5.4.2, 5.4.3, and 5.4.4);

   1.5. study of the relationship between feature diagrams and UML class diagrams (Section 5.5);

   1.6. the concept of feature starter set (Section 5.8.1);

   1.7. feature modeling process including concepts from Aspect-Oriented Programming (Section 5.8);

2. contributions to generators and static metaprogramming in C++:

   2.1. use of configuration repositories for propagating types from upper layers to lower layers of a GenVoca model (Section 6.4.2.4)[8];

   2.2. the metacontrol structure IF<> (Section 8.2);

   2.3. list processing using template metaprogramming (Section 8.4) and the implementation of a Lisp interpreter as a template metaprogram (Section 8.12);

   2.4. techniques for avoiding partial template specialization in template metaprograms (Section 8.5);

   2.5. the concept of a configuration generator based on template metaprogramming and configuration repositories (Section 8.7);

   2.6. combination of expression generators based on expression templates with configuration generators (Section 8.8);

3. contributions to Aspect-Oriented Programming:

   3.1. extension of the Cool-approach to synchronization with the concept of ports and reconfiguration and the implementation in Smalltalk (see Section 7.5.2);

   3.2. investigation of the relationship between AOP and DE (Section 7.2.4);

   3.3. some minor contributions such as kinds of join points (Section 7.4.1.1), requirements on composition mechanisms (Section 7.4.1), and use of active libraries as an implementation technology for AOP (Section 7.6.4);

4. contributions of DEMRAL:

[8] The idea of implementing GenVoca models using parameterized inheritance and nested classes described in Section 6.4.2.4 was developed independently from [SB98a].

4.1. new method illustrating the concept of a Domain Engineering method containing OO and AOP ideas and specialized for a particular category of domains (Chapter 9);

4.2. the concepts of a *configuration domain-specific language* (configuration DSL), configuration knowledge, and *implementation components configuration language* (ICCL) (Section 9.4.2);

4.3. the interaction between a configuration DSL and an expression DSL (Section 9.4.4);

4.4. feature starter sets for abstract data types (Section 9.3.2.2.1) and for algorithms (Section 9.3.2.2.2);

5. contributions of the Matrix Case Study:

5.1. to our knowledge, the most comprehensive and best documented publicly-available case study of Domain Engineering available today (Chapter 10);

5.2. demonstration of the DEMRAL method;

5.3. demonstration (Section 10.3.1) and evaluation (Section 10.3.1.8) of the generative C++ techniques;

5.4. comparison between the generative C++ approach and the Intentional Programming approach for the development of algorithmic libraries (Section 10.3.2);

Another contribution of this work is a set of comprehensive and up-to-date surveys on DE (Chapter 3), DE-and-OOA/D integration work (Chapter 4), generator approaches (Chapter 6), and AOP (Chapter 7). As of writing, no other surveys on the second, and the last topic are known to the author.

1.10 References

[Ber90] L. Berlin. When Objects Collide: Experiences With Reusing Multiple Class Hierarchies. In *OOPSLA/ECOOP '90 Conference Proceedings*, *SIGPLAN Notices*, vol. 25, no. 10, October 1990, pp. 181-193

[Big94] T. Biggerstaff. The Library Scaling Problem and the Limits of Concrete Component Reuse. In *Proceedings of the 3rd International Conference on Software Reuse*, W.B. Frakes (Ed.), IEEE Computer Society Press, 1994, pp. 102-109

[Big97] T. J. Biggerstaff. A Perspective of Generative Reuse. Technical Report MSR-TR-97-26, Microsoft Corporation, Redmond, Washington, 1997

[CEG+98] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative Programming and Active libraries. Draft submitted for publication, 1998

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'95), ACM SIGPLAN Notices*, vol. 30, no. 10, 1995, pp. 285-299, http://www.softlab.is.tsukuba.ac.jp/~chiba/openc++.html

[CW] ControlWORKS, a machine and process control software package for semiconductor process equipment by Texas Instruments, Inc.; see product data sheets at http://www.ti.com/control/

[DGT96] O. Danvy, R. Glück, and P. Thiemann, (Eds.). *Partial Evaluation*. LNCS 1024, Springer-Verlag, 1996

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976

[EHK96] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A Language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of POPL'96*, 1996, pp. 131-144

[Eis97] U. Eisenecker. Generative Programming (GP) with C++. In *Proceedings of Modular Programming Languages (JMLC'97, Linz, Austria, March 1997)*, H. Mössenböck, (Ed.), Springer-Verlag, Heidelberg 1997, pp. 351-365, see http://home.t-online.de/home/Ulrich.Eisenecker/

[Eng97] D. R. Engler. Incorporating application semantics and control into compilation. In *Proceedings USENIX Conference on Domain-Specific Languages (DSL'97)*, 1997

[Ers78] A. P. Ershov. On the essence of compilation. In *Formal Description of Programming Concepts*, E. J. Neuhold, (Ed.), North-Holland, 1978, pp. 391-420

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[GJ97] R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. In *Lisp and Symbolic Computation*, vol. 10, no. 2, 1997, pp. 113-158

[IHS+96] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota. Design and Implementation of Metalevel Architecture in C++ – MPC++ approach. In *Proceedings of Reflection'96*, 1996

[JGS93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings ECOOP'97 — Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 1997*, Mehmet Aksit and Satoshi Matsuoka (Eds.), LNCS 1241, Springer-Verlag, 1997, also see http://www.parc.xerox.com/aop

[LL94] M. Leone and P. Lee. Lightweight run-time code generation. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 97-106. Also Technical Report 94/9, Department of Computer Science, University of Melbourne, June 1994

[PEK97] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings PLDI'97*, 1997

[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995

[Ros95] L. Rosenstein. MacApp: First Commercially Successful Framework. In *Object-Oriented Application Frameworks*, T. Lewis et al., Manning Publications Co., 1995, pp. 111-136

[SB98a] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of the 12th European Conference Object-Oriented Programming (ECOOP'98)*, E. Jul, (Ed.), LNCS 1445, Springer-Verlag, 1998, pp. 550-570

[SG97] J. Stichnoth and T. Gross. Code composition as an implementation language for compilers. In *Proceedings USENIX Conference on Domain-Specific Languages (DSL'97)*, 1997

[Sim98] C. Simonyi. Intentional Programming. Keynote address at the Smalltalk and Java in Industry and Education Conference (STJA'98), Erfurt, October 7, 1998

[SL98a] J. G. Siek and A. Lumsdaine. The Matrix Template Library: A Unifying Framework for Numerical Linear Algebra. In *Proceedings of the ECOOP'98 Workshop on Parallel Object-Oriented Computing (POOSC'98)*, 1998, see http://www.lsc.nd.edu/

[Vel95a] T. Veldhuizen. Using C++ template metaprograms. In *C++ Report*, vol. 7, no. 4, May 1995, pp. 36-43, see http://monet.uwaterloo.ca/blitz/

[Vel95b] T. Veldhuizen. Expression Templates. In *C++ Report*, vol. 7 no. 5, June 1995, pp. 26-31, see http://monet.uwaterloo.ca/blitz/

[Web95] B. Webster. *Pitfalls of Object-Oriented Development: A Guide to the Wary and the Enthusiastic*. M&T Books, New York, New York, 1995

Chapter 2 Conceptual Modeling

2.1 What Are Concepts?

One fundamental question has occupied psychologists, philosophers, cognitive and other scientists for long time: What is the language of mind? Do we think using a natural language, such as Polish or German? Although we have the ability to process natural language, it seems that there must be some internal mental representation other than natural language. We clearly sense the distinction between having a thought and putting it into words. Defining even a most simple concept, such as *house* or *dog*, seems to be the more difficult the more precise we try to be. We find ourselves retrieving more and more facts about the concepts to be defined realizing how complex they are. This phenomenon also supports the view that there must be some mental representation other than natural language. Cognitive psychologists refer to this representation as *propositional representation*.

Propositional representation

There has been numerous experiments providing evidence supporting this theory. A classic example is the experiment by Sachs [Sac67]. In the experiment subjects listened to paragraphs on various topics. The reading of the paragraphs was interrupted and subjects were given a number of sentences. Their task was to choose the sentence which had occurred in the previous paragraph. In addition to the correct sentence, the candidate sentences included other sentences which sounded similar to the correct one but had a different meaning as well as sentences which had the same meaning but used different wording. The finding of the experiment was that subjects rarely picked a sentence with the wrong meaning but often thought that they had heard one of the sentences using different wording. A reasonable explanation of this finding is that subjects translated the text they had listened to into their propositional representation and then translated it back into natural language in the second part of the experiment. The fact that the propositional representation can be translated into a number for equivalent sentences accounts for subject's confusion.

One of the most important properties of human mind is the fact that people do not store all information about the objects they encounter. Smith and Medin note that [SM81, p. 1] "If we perceived each entity as unique, we would be overwhelmed by the sheer diversity of what we experience and unable to remember more than a minute fraction of what we encounter. And if each individual entity needed a distinct name, our language would be staggeringly complex and communication virtually impossible." Fortunately, we have the ability to recognize new objects as instances of *concepts* we already know — whereby a concept stands for the knowledge about objects having certain properties. The task of recognizing an object as an instance of a concept is referred to as *categorization* or *classification*. For this reason, in the context of categorization, concepts are often called *categories* or *classes*. This chapter focuses on categorization, but it is important to note that categorization is not the only process involving concepts: the processes of acquiring and evolving concepts are equally important.

Concepts, categories, and classes

Concepts are central to the nature of the propositional representation. By studying concepts we can certainly learn more about how knowledge is not only represented but also processed in the mind.

You may ask now what all of this has to do with programming. Well, programming is about conceptual modeling. We build conceptual models in our heads to solve problems in our everyday life. In programming, we build models that we can run on a machine. By learning about how concepts are represented in the mind, we should be able to improve our means to represent concepts externally, i.e. in software. In this chapter, you will see that the current object-oriented paradigm is based on a very simplistic view of the world, namely the classical view. In the end, you will realize that adequate implementations of concepts have to cover enormous amounts of variability. This observation will provide the motivation for all the following chapters.

2.2 Theories of Concepts

The origins of the study of categorization and concepts date back to Aristotle, the great Greek philosopher, who is also seen as the father of the so-called *classical view* of categorization. According to [SM81], Hull's 1920 monograph on concept attainment [Hul20] initiates a period of modern and intensive research on a theory of concepts. Contemporary work on concepts includes philosophically oriented studies of language (e.g. [Fod75]), linguistic studies (e.g. [Bol75]), psycholinguistics (e.g. [CC77]), and studies in the area of cognitive psychology (e.g. [And90]). Thus, concepts have been studied in multiple disciplines: philosophy, linguistics, psychology, cognitive science and, more recently, also computer science (esp. artificial intelligence).

In the course of this research, three major views about the nature of concepts emerged:

- the *classical* view,
- the *probabilistic* view (also referred to as the *prototype* view) and
- the *exemplar* view.

Before discussing these three views, we first introduce some basic terminology.

2.2.1 Terminology

In our discussion, we will distinguish between

- *mathematical* and
- *natural* concepts.

Examples of mathematical concepts are numbers, geometric figures, matrices, etc. The most important property of a mathematical concept is that it has a precise definition. By natural concepts we mean concepts used in our everyday communication with natural language, e.g. *dog*, *table*, *furniture*, etc. Later we will see that in most cases the definition of natural concepts is problematic.[9]

We describe concepts by listing their properties. According to Smith and Medin [SM81], there are three major types of properties of concepts:

- *features*,
- *dimensions*, and

[9] Of course, there are also other categories of concepts e.g. object concepts (e.g. *dog*, *table*), abstract concepts (e.g. *love*, *brilliance*), scientific concepts (e.g. *gravity*, *electromagnetic waves*), etc.

- *holistic properties*.

Features

We use features to represent qualitative properties of concepts. Table 1 gives some examples of features. For example, the features of a chicken are *animate*, *feathered*, and *pecks*. Thus, each concept is described by a list of features.

concept:	**Robin**	**Chicken**	**Collie**	**Daisy**
features:	animate feathered flies red breast	animate feathered pecks	animate furry brown-gray	inanimate stem white

Table 1 *Examples of features (adapted from [SM81, p. 14])*

Dimensions

Dimensions are usually used to express quantitative properties such as *size* or *weight*. The value range of a dimension can be continuous (e.g. a real number) or discrete (e.g. *small*, *middle-size*, and *large* for the dimension *size*). Typically, there is the requirement that the values of a dimension are ordered. If we drop this requirement, we get a “weak notion of dimensions”, which is sometimes referred to as *attributes*. In contrast to featural descrriptions, we use just one set of dimensions to represent a number of concepts: each concept is

concept:	**Robin**	**Chicken**	**Collie**
features:	animacy: *animate* size: S_R ferocity: F_R	animacy: *animate* size: C_{Ch} ferocity: F_{Ch}	animacy: *animate* size: S_C ferocity: F_C

Table 2 *Examples of dimensions (S and F stand for some appropriate value; adapted from [SM81, p. 63])*

represented by a tuple of values (one value for each dimension). An example of a dimensional description of concepts is given in Table 2.

Holistic vs. component properties

Features and dimensions are also referred to as *component properties* since a feature or a dimension does not constitute a complete description of a concept [Gar78]. The counterpart of component properties are *holistic properties*. A holistic description of a concept consists of only one property, i.e. a holistic property. A holistic property can be thought of as a concept template.

We will discuss a number of important issues regarding features and dimensions in Section 2.3. But first, we focus our attention on the three views of concepts.

2.2.2 The Classical View

Necessary and sufficient properties

Until the mid-seventies, the categorization research was dominated by the *classical view* of categorization (also called *definitional theory*). According to the classical view, any concept (or category) can be defined by listing a number of *necessary* and *sufficient* properties which an object must possess in order to be an instance of the concept. For example, the concept of a square can be defined using the following four features [SM81]:

1. closed figure

2. four sides

3. sides equal in length, and

4. equal angles.

All four features are jointly sufficient and each of them is singly necessary in order to define a square. Thus, the essence of the classical view is that a concept can be precisely defined using a single *summary description* (e.g. four features in our example). This summary description is the result of an abstraction and generalization process (see Section 2.3.7) which takes a number of concrete concept instances as its input. This summary description is the *essence* of a concept and corresponds to what software developers refer to as "the right abstraction". A precise summary description can be given for mathematical concepts since they are defined precisely. Unfortunately, this is usually not the case for natural concepts. For example, how would you define the concept of a *game*? This classic example by Wittgenstein [Wit53] illustrates the problems of defining natural concepts [SM81, p. 30]:

Summary description

Criticism of the classical view

> *"What is a necessary feature of the concept of games? It cannot be competition between teams, or even the stipulation that there must be at least two individuals involved, for solitaire is a game that has neither feature. Similarly, a game cannot be defined as something that must have a winner, for the child's game of ring-around-a-rosy has no such feature. Or let us try a more abstract feature – say that anything is a game if it provides amusement or diversion. Football is clearly a game, but it is doubtful that professional football players consider their Sunday endeavors as amusing or diverting. And even if they do, and if amusement is a necessary feature of a game, that alone cannot be sufficient, for whistling can also be an amusement and no one would consider it a game."*[10]

During the seventies and the eighties, the criticism of the classical view intensified. Some of the then-identified problems of this view include [SM81]:

- *Failure to specify defining features*: Natural concepts for which – as in the case of a *game* – no defining (i.e. necessary and sufficient) features have been found are abundant.

- *Existence of disjunctive concepts*: If the feature sets of any two instances of a concept are disjunctive, the concept is referred to as a *totally disjunctive concept*. If a concept is not totally disjunctive and if the feature sets of any two of its instances are only partially overlapping, the concept is called *partially disjunctive* (see Figure 2). Both partially and totally disjunctive concepts violate the requirement of the classical view that a concept is defined by a *single* set of sufficient and necessary features. According to Rosch et al. [RMG+76], superordinate concepts (i.e. very general concepts such as *furniture*) are often disjunctive.

Totally and partially disjunctive concepts

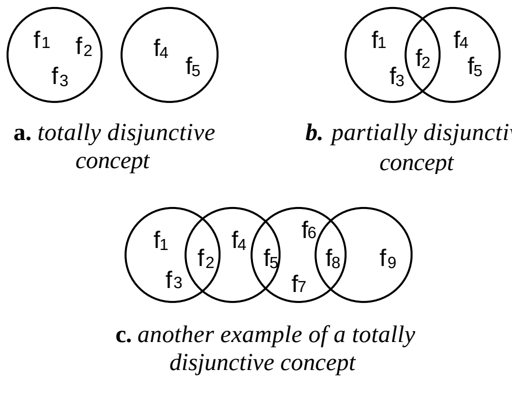

f1
f2
f3
f4
f5
a. totally disjunctive concept
f1
f2
f3
f4
f5
b. partially disjunctive concept
f1
f2
f3
f4
f5
f6
f7
f8
f9
c. another example of a totally disjunctive concept

Figure 2 *Disjunctive concepts (f stands for a feature; each circle represents a set of features describing some instances of a concept)*

[10] As an exercise, try to define the concept of a table. You will quickly realize that it is as hopeless as defining the concept of a game.

- *Simple typicality effects*: Smith and Medin note that [SM81, p. 33]: "People find it a natural task to rate the various subsets or members of a concept with respect to how typical or representative each one is of a concept." This finding cannot be reconciled with the classical view.

Non-necessary features

- *Use of non-necessary features in categorization*: As documented by Hampton [Ham79], people often use *non-necessary features* (i.e. features which only some instances of a concept have) in the categorization of objects. This suggests that nonnecessary features play an important role in defining concepts.

Nesting of concept's defining features in subsets

In addition to the single summary description consisting of sufficient and necessary properties, the classical view has one more assumption which is as follows [SM81, p. 24]:

> *"If concept X is a subset of a concept Y, the defining features of Y are nested in those of X."*

For example, consider the concept of *rectangles* and *squares*. All instances of the concept of squares are clearly instances of the concept of rectangles. Also, the defining features of rectangles are nested in those of squares (note that this requires the notion of nesting to include logical subsumption).

This assumption also turns out to be problematic. It is – in many cases – too strong and it has to be weakened in order to reflect reality more adequately. As Smith and Medin note [SM81, p. 29], many people, when asked: "Is tomato a fruit?", are unsure of whether this particular subset relation holds. People often even change their mind over time about whether a particular subset relation holds or not and there always seem to be exceptions violating assumed subset relations. The classical view is unable to account for these effects.

The criticism of the classical view discussed in this section eventually lead to the development of the probabilistic and exemplar views. Some of the most influential work supporting this criticism is that by Rosch and Mervis (e.g. [Ros73, Ros75, Ros77, RM75, RMG+76]).

2.2.3 The Probabilistic View

Semantic networks

In the probabilistic view, each concept is described – just as in the classical view – by a list of properties, i.e. we also have a single summary description of a concept. The main difference between the classical view and the probabilistic view is that in the probabilistic view each feature has a likelihood associated with it. For example, one could associate the likelihood of 0.9 with the feature *flies* of the concept of a *bird* in order to indicate that most (but not all) birds fly.

At odds with the term "probabilistic view", the number associated with each feature is usually not a probability value. It is rather a weight whose value could be calculated depending on many factors, e.g. the probability that the feature is true of an instance of a concept, the degree to which the feature distinguishes the concept from other concepts, and the past usefulness or frequency of the feature in perception and reasoning [SM81].

Semantic networks

Spreading activation

An example of a probabilistic representation of the concepts *vegetable* and *green bean* is shown in Figure 3. This kind of network representation is also referred to as a *propositional* or *semantic network* (please note that, in general, a semantic network need not be probabilistic). A possible classification procedure based on this representation is so-called *spreading activation* (proposed by [CL75]). For example, in order to check if green bean is a vegetable, the corresponding nodes in Figure 3 are activated (imagine that the net in Figure 3 is just a part of a larger network). The activation spreads along all the paths starting from each of the two nodes. The amount of activation depends on the weights. Some of the activated paths will intersect forming a pathway connecting the two nodes. The amount of activation in the pathways will be used to decide if *green bean* is a *vegetable*.

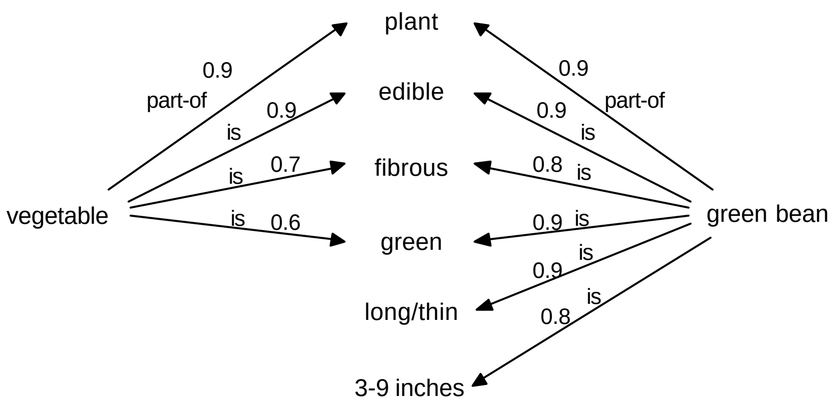


Figure 3 *A small part of a probabilistic semantic network (adapted from [SWC+95, p. 91])*

2.2.4 The Exemplar View

In the exemplar view, a concept is defined by its exemplars (i.e. representative instances) rather than by an abstract summary. Exemplars can be specific concept instances as well as subsets. For example, the representation of the concept *bird* could include the subset concept *robin* and the specific instance *"Fluffy"* (see Figure 4). The exemplar representation is especially well suited for representing disjunctive concepts (see Section 2.2.2).

In the case of an exemplar representation, new objects are categorized based on their similarity to the stored exemplars. One problem in this context is how to efficiently store and process a large number of exemplars. A possible approach is to deploy a *connectionist architecture*, i.e. a massively parallel architecture consisting of a large number of simple, networked processing units, such as a *neural network* (see [Kru92]). In a neural network, for example, a new exemplar is stored by adjusting the weights associated with the connections between neurons. This adjustment can be accomplished using the *back-propagation algorithm* (see e.g. [HN90]).

Connectionist approaches

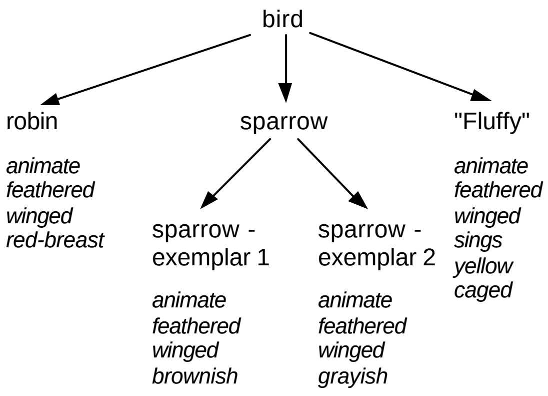


Figure 4 *An exemplar representation (adapted from [SM81, p. 145])*

2.2.5 Summary of the Three Views

All three views have been summarized in Figure 5. Both the probabilistic and the exemplar view do not suffer from the specific problems of the classical view listed in Section 2.2.2. This state of affairs does not invalidate the classical view, however. Certain problems can be adequately represented and solved using the classical view. Since concepts capture some relevant properties of objects in a given context, a classical representation might be feasible in some specific context at hand. In such cases, there is no need to use the more complicated probabilistic or exemplar views. As Winston notes [Win92]: “Once a problem is described using appropriate representation, the problem is almost solved.” Thus, for each view there are problems which are best solved using that view. Some newer knowledge-based models seek to combine the different representations (e.g. multi-paradigm expert systems [Flo95]). There is also evidence that human mind deploys both abstract and exemplar representations and all three views will have to be reflected in a comprehensive theory of concepts.

One important lesson follows from this presentation, however: Natural concepts, which we deal with on daily basis and try to model in software, are inherently complex. The classical view works best for mathematical concepts. It might also be adequate for creating simplified models of natural concepts. However, it usually breaks if we try to cover the diversity of natural concepts more adequately. This finding is particularly relevant to software reuse whose goal is to provide generic solutions that work in many contexts.

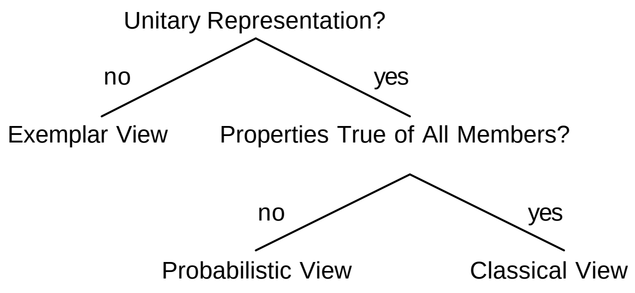


Figure 5 *Three views of concepts (from [SM81, p. 4])*

2.3 Important Issues Concerning Concepts

So far we have covered some basic terminology as well as the three major views of concepts. In the following seven sections we will focus on some important issues concerning concepts. These issues are concept stability, concept core, information contents of features, relationships between features, quality of features, the relationship between features and dimensions and between abstraction and generalization.

2.3.1 Stability of Concepts

Any natural concept, even the simplest, involves an enormous amount of knowledge. Stillings et al. note [SWC+95, p. 95]:

> *“Your knowledge about cucumbers, for example, might include tactile information (‘those tiny bumps with little spines growing out of them’), the picking size for several varieties, when and how to plant, type of machinery and labor needed for farm harvest, how to test for bitterness, smell when rotting, use in several of the world’s cuisines, next-door neighbors hate them, waxy grocery store surface an unnatural phenomenon, and so on.”*

Intra- and inter-personal concept stability

It is clear that one person’s knowledge about a concept evolves over time. Thus, we talk about temporal *concept stability* within the mind of a person or *intra-personal concept stability*. If we further consider the fact that the knowledge which different persons associate with a concept varies, we can also speak about *inter-personal concept stability*.[11]

[11] In [SM81, p. 10], intra-personal concept stability is referred to as “within-individual stability” and inter-personal concept stability as “across-individual stability”.

The knowledge associated with a concept also depends on the context. Before explaining this idea, we first need to consider some aspects of the basic architecture of the human mind. Roughly speaking, the human mind has two kinds of storage: the *long-term memory* and the *working* (or *short-term*) *memory* (see [New90] for a thorough presentation of various theories of cognition).[12] The long-term memory is where all the knowledge which we maintain over long periods of time (minutes to years) is stored. The knowledge in the long-term memory is not directly accessible to thought processes. The working memory, on the other hand, contains the knowledge which is directly accessible. However, the working memory is limited both in terms of time and space. Knowledge in the working memory needs to be "refreshed" every few seconds to in order to stay there. The space limitation is described by the famous "rule of seven, plus or minus two" by Miller [Mil56], which states that a person can only remember about seven, plus or minus two, items at a time. Because of this limitation of the working memory, only these aspects of a concept will be brought into the working memory, which are considered relevant to solving the problem at hand. A possible characterization of this idea is to think of concepts as actually being assembled "on-the-fly" as they are needed. This effect tells us that the content of a concept depends on the context in which the concept is currently used. We will refer to this effect as *cognitive subjectivity*.

Long-term and working memory

Cognitive subjectivity

2.3.2 Concept Core

For a graphical figure to be an instance of a square, it must have the following four defining properties: *closed figure*, *four sides*, *sides equal in length*, and *equal angles*. These properties are common to all instances of the concept *square*. They are not only common to all squares – they are *essential* for a square to be a square. Squares have other properties, e.g. certain *size*, which are non-essential. The essential properties of a concept constitute its *core*.

Essential properties

Concepts are used in the inference process during problem solving. Once we assume, for example, that a certain object is a square, we can infer the four essential properties of squareness from this assumption.

Concepts are used to solve different problems in different contexts. Not all properties are relevant in all contexts, but the more contexts a property is relevant in the more essential the property is. Necessary properties which, by definition, hold for all instances of an object are very likely to be essential since one can relay on the fact that they hold and they are usually used in many contexts. Therefore, we assert that defining properties are essential. But this statement describes essentiality only in the context of the classical view, which assumes that concepts have their defining properties. In general, this view is too restrictive.

If we subscribe to the probabilistic view, from the fact that an object is an instance of a concept, we can infer that it has some property only with a certain probability. Such inference is still useful and used by people. And the more situations the non-necessary property can be used in, the more essential it is. Thus, an essential property does not need to be common to all instances of a concept. For example, the property *flies* is an essential property of *birds*, but it does not hold for all its instances.

Non-necessary but essential properties

2.3.3 Informational Contents of Features

The simplest and natural way of describing a concept is by listing its properties. For example, for the concept *daisy* we listed *inanimate*, *stem*, and *white* (see Table 1). By listing *stem* as its feature, we mean more specifically that a *daisy* has a *stem* as its part. We can represent this fact by drawing an arrow from *daisy* to *stem* and annotating this arrow with *part-of*. This is exactly how we produced the semantic net in Figure 3. Thus, a feature can be represented as another concept plus its relationship to the concept which is being described. But a feature can also be represented using more than one concept (and more than one relationship) as in the case of *red*

[12] In [New90, p. 30] Newell notes that the current understanding of these two kinds of storage is that the working memory is a part of the long-term memory rather than a separate memory requiring transfer. This view aligns well with the idea of spreading activation discussed in Section 2.2.3.

breast (see Table 1), i.e. *robin* $\xleftarrow{\textit{part-of}}$ *breast* $\xrightarrow{\textit{is}}$ *red*. If we would like to reveal more semantics of *red* and *breast*, we would have to grow this graph bigger and bigger by adding new concepts and relationships. Relationships are themselves concepts. For example, the properties of *part-of* can be modeled by another semantic network.

Concepts vs. features

We can view concepts as chunks of knowledge. Features are also chunks of knowledge used to build up a larger chunk of knowledge, i.e. the concept they describe. It becomes immediately clear that the distinction between a concept and a feature is a relative one and determined by the focus of the description. If we would like to know the contents of a feature, we would move our focus on that feature and break it down piece by piece, as we did it with the initial concept.

Why seem features so natural for describing concepts?

Features allow as to describe concepts by using few items or chunks. As we remember, the working memory of the human mind can store only a limited number of chunks at a time — seven or so (see Section 2.3.1). Therefore it is convenient to list *inanimate*, *stem*, and *white* for a daisy and not to instantly think of all the detail that these features imply. These details can be retrieved from the long-term memory as needed. Chunking, i.e. organizing information in units and doing this in a recursive way, is believed to be the basic property of human memory (see [New90]). It tells us that the human mind internally employs modularity and that any models we build have to be modular in order to be understandable.

2.3.4 Feature Composition and Relationships Between Features

Relationships between features are particularly interesting in the context of feature composition. Feature composition can be regarded as a means of creating concepts or concept instances. In this context, relationships between features are manifested through *constraints on feature combinations* and *translations between features*:

- *Constraints on feature combinations*: In general, features cannot be freely combined since certain feature combinations may lead to a contradiction. For example, a matrix cannot be non-square and diagonal at the same time since diagonal implies square.

- *Translations between features*: Certain features (or feature combinations) may imply some other features. In our matrix example, if a matrix is diagonal then it is also square.

2.3.5 Quality of Features

The quality of features has to be judged in the context of the tasks they are used for. However, three general feature qualities can be given here:

- *Primitiveness*: Features make apparent relationships (e.g. differences and similarities) between concepts. A feature is primitive if it does not have to be decomposed in order to show some relevant differences among concept instances.

- *Generality*: A feature is more general if it applies to a larger number of concepts. A set of features is general, if it describes a large number of concepts with a minimal number of features.

- *Independency*: The fewer constraints on feature combinations apply to a set of features the larger number of concepts can be described by combining the features.

These are structural qualities which tell us how “economical” a set of features is in describing relevant concepts.

2.3.6 Features versus Dimensions

Featural descriptions use different numbers of features per concept. Therefore it is easy to introduce new features and add them to a description of a concept. In the dimensional approach, each concept is represented using all dimensions. This requirement makes the dimensional approach slightly less flexible: adding new dimensions to a description model requires all concept descriptions to be updated and adding a value to the description of one concept requires a new dimension for this value. Sometimes the special pseudo-value *not applicable* has to be used in order to indicate that a dimension makes no sense for certain concepts. The featural approach does not suffer from these problems.

In a featural description, two concepts may be described using incomparable sets of features. On the other hand, the dimensional approach makes the similarities and differences between concepts more explicit than the featural approach since each dimension corresponds to a comparison criterion. Dimensions can also be seen as an organization mechanism for their values. This makes the dimensional approach easier to use than a flat feature set. Finally, the dimensional approach guides the concept description (or synthesis) process by requiring to choose a value for each dimension.

Clearly some kind of combination of both description styles would be useful. Indeed, *feature diagrams* described later in Chapter 5 combine both styles.

2.3.7 Abstraction and Generalization

Both in the classical and the probabilistic view, a concept is defined through an abstract summary description. This abstract description is usually the result of an abstraction and generalization process, which takes a number of sample objects (i.e. exemplars) as its input.

Abstraction

Abstraction involves the extraction of properties of an object according to some focus: only those properties are selected which are relevant with respect to the focus (e.g. a certain class of problems). Thus, abstraction is an information filtration process which reduces the initial amount of information to be used in problem solving.

Generalization

Generalization is an inductive process of collecting information about a number of objects and presenting this information in a single description. Generalization usually results in the increase of information. The construction of a generalized description of a number of instances can also lead to a larger description than each individual object description. For example, in the first step, a number of object descriptions could be lumped together (which corresponds to the exemplar view). In the second step, the compound description could be restated in a more declarative form and, as a result, become more concise (no information is lost in this process, i.e. all original objects could be reproduced from this description). However, in most cases, generalization and abstractions are combined when describing a set of objects. This combination produces an abstract and generalized description which is even more concise since the information not relevant to the abstraction focus is removed.

Abstraction, concretization, generalization, and specialization are operations on concepts:

- An existing description of a set of objects can be further *generalized* by adding new objects to the set and modifying the description to take account of these new objects.

- An existing description of a set of objects can be further *abstracted* using a new focus and filtering away all parts of the description which are not relevant with respect to the new focus.

Concretization (refinement) and specialization

- The inverse operation to abstraction is *concretization* (also called *refinement*). Concretization results in the increase of detail per object.

- The inverse operation to generalization is *specialization*. Specializing the description of a set of objects involves the reduction of the set to a subset.

The relationship between the input and the output concepts of each operation gives rise to the abstraction, concretization, generalization, and specialization relationships.

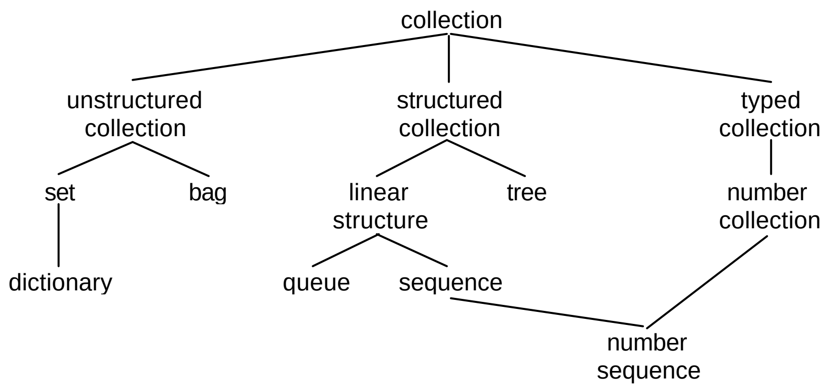


Figure 6 *Partial hierarchy of data concepts (adapted from [Nav96, p. 38])*

Generalization and abstraction are often confused, even in popular software engineering textbooks ([Nav96] mentions some of them). The confusion can be contributed to the fact that a typical hierarchy of data concepts, such as the one in Figure 6, can be interpreted as an abstraction hierarchy or a generalization hierarchy or both. Indeed, the relationships in Figure 6 represent both abstraction and generalization relationships. For example, *collection* is a generalization of *unstructured collection* since the set of all unstructured collections is a subset of the set of all collections (in a given universe of collections). At the same time, *collection* is an abstraction of *unstructured collection* since *collection* abstracts away the property *unstructured* of *unstructured collection*. Furthermore, the abstraction relationships in Figure 6 use multiple abstraction criteria (e.g. structuring and element type). An example of a hierarchy of sets using one abstraction criterion, specifically the *type of representation*, is shown in Figure 7. In any case, the hierarchies in Figure 6 and Figure 7 involve both abstraction and generalization.

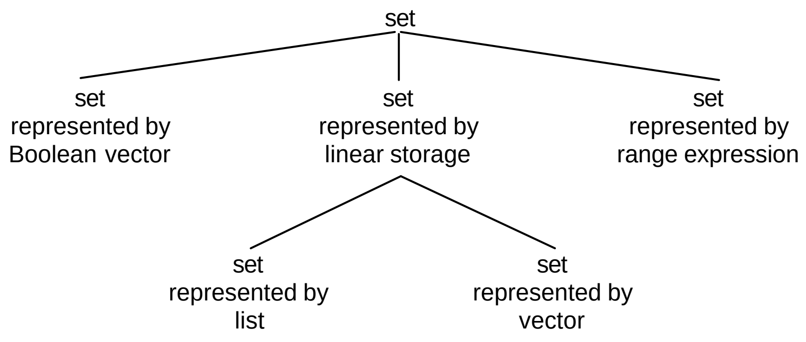

set
set represented by Boolean vector
set represented by linear storage
set represented by range expression
set represented by list
set represented by vector

Figure 7 *Partial abstraction hierarchy of data concept* set *(adapted from [Nav96, p.39])*

A thorough treatment of the topic of generalization and abstraction in the context of computer science can be found in [Nav96].

2.4 Conceptual Modeling, Object-Orientation, and Software Reuse

Concepts can be regarded as natural modeling elements since they represent a theory about knowledge organization in the human mind. The relationship between concepts and object-orientation (specifically the classical object model) is apparent: concepts correspond to classes. The major difference is that object-orientation makes more specific assumptions about objects: they have *state* and *behavior* and collaborate through *interactions*.

Based on the presentation of the three views of concepts we can draw the following conclusions:

- The classical view of concepts is well-suited for representing well-defined concepts which are defined by a set of necessary and sufficient properties. In the classical object model, this view is adequately modeled by classes. A class represents a unitary and exact description of all its instances.

- Natural concepts are modeled more adequately using probabilistic and exemplar representations. In software engineering, concepts such as customer and bank account are examples of natural concepts. The probabilistic and the exemplar views allow us to represent the great structural variety of instances of natural concepts. This structural variety is related to the fact that natural concepts are used in a large number of different contexts – each requiring different structures. In the classical object model, the structural variety of a concept can be expressed only in indirect ways, e.g. encoded in the state space of an object or as an often large and complicated inheritance hierarchy. It would be clearly desirable to have a means of explicitly and concisely representing concepts including a convenient mechanism for expressing their variants. This critique aligns well with the famous “critique of pure objects” by Harrison and Ossher [HO93], which points out the inadequate handling of subjectivity and context dependency of objects by the classical object model. The need for an adequate support for modeling concept variations is particularly important in the context of reusable software.

- Featural and dimensional descriptions represent a convenient model for representing the variability of concepts. In this context, the issues concerning concepts, features, and dimensions discussed in Section 2.3 (e.g. essentiality of features, relationships between features, features vs. dimensions) become relevant.

- The probabilistic and the exemplar representation models are well suited for implementing component retrieval mechanisms since they allow us to capture the relationships between natural language terms (i.e. words used to index reusable components).

2.5 Suggested Readings

The book by Stillings et al. [SWC+95] represents a modern and comprehensive treatment of cognitive science. For an excellent treatment of the three views and a comprehensive survey of theories of concepts see [SM81]. The book by Newell [New90] – one of the AI classics – provides a survey of theories of cognition. The topic of abstraction and generalization in the context of computer science is discussed in [Nav96].

2.6 References

[And90] J. R. Anderson. *The architecture of cognition*. Harward University Press, Cambridge, Massachusetts, 1990

[Bol75] D. L. Bolinger. *Aspects of language*. Second edition, Harcourt Brace Jovanovich, New York, 1975

[CC77] H. H. Clark and E. V. Clark. *Psychology and language*. Harcourt Brace Jovanovich, New York, 1977

[CL75] A. Collins and E.F. Loftus. A spreading activation theory of semantic processing. In *Psychological Review*, no. 82, 1997, pp. 407-428

[Flo95] T. Flor. Multi-Paradigmensystem OOXSDAR VISIS: Ein Beitrag zur Entwicklung von Expertensystemen der zweiten Generation. PhD dissertation, Technical University of Ilmenau, 1995

[Fod75] J. A. Fodor. *The language of thought*. Crowell, New York, 1975

[Gar78] W. R. Garner. Aspects of a stimulus: Features, dimensions, and configurations. In *Cognition and categorization*, E. Rosch and B. B. Lloyd , (Eds.), Erlbaum, Hillsdale, New Jersey, 1978

[Ham79] J. A. Hampton. Polymorphous concepts in semantic memory. In Journal of Verbal Learning and Verbal Behavior, no. 18, 1979, pp. 441-461

[HN90] R. Hecht-Nielsen. *Neurocomputing*. Addison-Wesley, Reading, MA, 1990

[HO93] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of OOPSLA'93*, 1993, pp. 411-428

[Hul20] C. L. Hull. Quantitative aspects of the evolution of concepts. In *Psychological monographs*, Whole no. 123, 1920

[Kru92] J.K. Kruschke. ALCOVE: An exemplar-based connectionist model of category learning. In *Psychological Review*, no. 99, 1992, pp. 22-44

[Mil56] G. A. Miller. The magic number seven, plus or minus two: Some limits of our capacity for processing information. In *Psychological Review*, no. 63, 1956, pp. 81-97

[Nav96] P. Návrat. A closer look at programming expertise: critical survey of some methodological issues. In *Information and Software Technology*, no. 38, 1996, pp. 37-46

[New90] A. Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1990

[RM75] E. Rosch, and C. B. Mervis. Family resemblances: Studies in the internal structure of categories. In *Cognitive Psychology*, no. 7, 1975, 573-605

[RMG+76] E. Rosch, C. B. Mervis, W. Gray, D. Johnson, and P. Boyes-Braem. Basic objects in natural categories. In *Cognitive Psychology*, no. 3, 1976, pp. 382-439

[Ros73] E. Rosch. On the internal structure of perceptual and semantic categories. In *Cognitive development and the acquisition of language*, T.E. Moore (Ed.), Academic Press, New York, 1973, pp. 111-144

[Ros75] E. Rosch. Cognitive representations of semantic categories. In *Journal of Experimental Psychology: General*, 104, 1975, pp. 192-233

[Ros77] E. Rosch. Human categorization. In *Studies in cross-cultural psychology*, Vol. 1, N. Warren (Ed.), Academic Press, New York, 1977, pp. 1-49

[Sac67] J. S. Sachs. Recognition memory for syntactic and semantic aspects of connected discourse. In *Perception and Psychophysics*, no. 2, 1967, pp. 437-442

[SM81] E.E. Smith and D.L. Medin. *Categories and concepts*. Harvard University Press, Cambridge, Massachusetts, 1981

[SWC+95] N. A. Stillings, S. E. Weisler, C. H. Chase, M. H. Feinstein, J. L. Garfield and E. L. Rissland. *Cognitive Science: An Introduction*. Second Edition, The MIT Press, Cambridge, Massachusetts, 1995

[Win92] P. H. Winston. *Artificial Intelligence*. Third edition, Addison-Wesley, Reading, Massachusetts, 1992

[Wit53] L. Wittgenstein. *Philosophical investigations*. G. E. M. Anscombe. Blackwell, Oxford, 1953

Part II

ANALYSIS AND DESIGN METHODS AND TECHNIQUES

Chapter 3 Domain Engineering

3.1 What Is Domain Engineering?

Most software systems can be classified according to the business area and the kind of tasks they support, e.g. airline reservation systems, medical record systems, portfolio management systems, order processing systems, inventory management systems, etc. Similarly, we can also classify *parts* of software systems according to their functionality, e.g. database systems, synchronization packages, workflow systems, GUI libraries, numerical code libraries, etc. We refer to areas organized around classes of systems or parts of systems as *domains*.[13]

Domain Engineering

Obviously, specific systems or components within a domain share many characteristics since they also share many requirements. Therefore, an organization which has built a number of systems or components in a particular domain can take advantage of the acquired knowledge when building subsequent systems or components in the same domain. By capturing the acquired domain knowledge in the form of reusable assets and by reusing these assets in the development of new products, the organization will be able to deliver the new products in a shorter time and at a lower cost. *Domain Engineering* is a systematic approach to achieving this goal.

Domain Engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e. reusable workproducts), as well as providing an adequate means for reusing these assets (i.e. retrieval, qualification, dissemination, adaptation, assembly, etc.) when building new systems.

Domain Analysis, Domain Design, and Domain Implementation

Domain Engineering encompasses three main process components[14] *Domain Analysis*, *Domain Design*, and *Domain Implementation*. The main purpose of each of these components is given in Table 3.

[13] We give a precise definition of a domain in Section 3.6.1.

[14] Most of the current Domain Engineering methods still refer to the process components as *phases*. Following a recent trend in the software development methods field, we do not refer to analysis, design, and implementation as *phases* since the term *phase* implies a rigid, waterfall-style succession of engineering steps. Modern process models, such as the Rational Objectory Process, consider analysis, design, and implementation as *process components*. These are independent of the time dimension, which is itself divided into phases (see Section 4.5.1). In this newer terminology, however, phases indicate the maturity of the project over time. *Important note*: In order to be consistent with the original literature, the descriptions of the Domain Engineering methods in Section 3.7 use the term *phase* in its older meaning (i.e. to denote process components).

The results of Domain Engineering are reused during *Application Engineering*, i.e. the process of building a particular system in the domain (see Figure 8).

Application Engineering

Domain Engineering process component	**Main purpose**
Domain Analysis	defining a set of reusable requirements for the systems in the domain
Domain Design	establishing a common architecture for the systems in the domain
Domain Implementation	implementing the reusable assets, e.g. reusable components, domain-specific languages, generators, and a reuse infrastructure

Table 3 *Three Main Process Components of Domain Engineering*

Table 3 makes the distinction between the conventional software engineering and Domain Engineering clear: while the conventional software engineering concentrates on satisfying the requirements for a *single* system, Domain Engineering concentrates on providing *reusable* solutions for *families* of systems. By putting the qualifier “domain” in front of analysis, design, and implementation, we emphasize exactly this *family orientation* of the Domain Engineering process components.

Indeed, if you take a look at the intentions of most of the current software engineering methods (including object-oriented analysis and design methods), you will realize that these methods aim at the development of “*this specific* system for *this specific* customer and for *this specific* context.” We refer to such methods *software system engineering methods*.

Software system engineering methods

Domain Engineering, on the other hand, aims at the development of reusable software, e.g. a generic system from which you can instantiate concrete systems or components to be reused in different systems. Thus, Domain Engineering has to take into account different sets of customers (including potential ones) and usage contexts. We say that Domain Engineering addresses *multi-system scope* development.

Multi-system scope development

Domain Engineering can be applied to a variety of problems, such as development of domain-specific frameworks, component libraries, domain-specific languages, and generators. The Domain Analysis process subcomponent of Domain Engineering, in particular, can also be applied to non-software-system-specific domains. For example, it has been used to prepare surveys, e.g. a survey of Architecture Description Languages [Cle96, CK95].

At the beginning of this section, we said that there are domains of systems and domains of parts of systems (i.e. *subsystems*). The first kind of domains is referred to as *vertical domains* (e.g. domain of medical record systems, domain of portfolio management systems, etc.) and the second kind is referred to as *horizontal domains* (e.g. database systems, numerical code libraries, financial components library, etc.). The product of Domain Engineering applied to a vertical domain is reusable software which we can instantiate to yield any concrete system in the domain. For example, we could produce a *system framework* (i.e. reusable system architecture plus components) covering the scope of a entire vertical domain. On the other hand, applying Domain Engineering to a horizontal domain yields reusable subsystems, i.e. *components*. We will come back to the notion of vertical and horizontal domains in Section 3.6.2.

In our terminology, a component is a *reusable* piece of software which is used to build more complex software. However, as already indicated, components are not the only workproducts of Domain Engineering. Other workproducts include reusable requirements, analysis and design models, architectures, patterns, generators, domain-specific languages, frameworks, etc. In general, we refer to any reusable workproduct as a *reusable asset*.

Components and other reusable assets

3.2 Domain Engineering and Related Approaches

Domain Engineering addresses the following two aspects:

- *Engineering of reusable software*: Domain Engineering is used to produce reusable software.

- *Knowledge management*: Domain Engineering should not be a "one-shot" activity. Instead, it should be a continuous process whose main goal is to maintain and update the knowledge in the domain of interest based on experience, scope broadening, and new trends and insights (see [Sim91] and [Ara89]).

Organizational Memory, Design Rationale, and Experience Factory

Current Domain Engineering methods concentrate on the first aspect and do not support knowledge evolution. The knowledge management aspect is addressed more adequately in the work on *Organizational Memory* [Con97, Buc97], *Design Rationale* [MC96], and *Experience Factory* [BCR94]. These three approaches have much in common with Domain Engineering, although they all come from different directions and each of them has a different focus:

- Domain Engineering concentrates on delivering reusable software assets.

- Organizational Memory concentrates on providing a common medium and an organized storage for the informal communication among a group of designers.

- Design Rationale research is concerned with developing effective methods and representations for capturing, maintaining and reusing records of the issues and trade-offs considered by designers during design and the ultimate reasons for the design decisions they make.

- Experience Factory provides a means for documenting the experience collected during past projects. It primarily concentrates on conducting mostly quantitative measurements and the analysis of the results.

As the research in these four areas advances, the overlap between them becomes larger. We expect that future work on Domain Engineering will address the knowledge management aspect to a larger degree (e.g. [Bai97]). In this chapter, however, we exclusively focus on the "engineering reusable software" aspect of Domain Engineering.

3.3 Domain Analysis

The purpose of *Domain Analysis* is to

- select and define the domain of focus and

- collect relevant domain information and integrate it into a coherent *domain model*.

The sources of domain information include existing systems in the domain, domain experts, system handbooks, textbooks, prototyping, experiments, already known requirements on future systems, etc.

It is important to note that Domain Analysis does not only involve recording the existing domain knowledge. The systematic organization of the existing knowledge enables and encourages us to actually extend it in creative ways. Thus, Domain Analysis is a *creative* activity.

Domain model: commonalities, variabilities, and dependencies

A *domain model* is an explicit representation of the *common* and the *variable* properties of the systems in a domain and the *dependencies* between the variable properties. In general, a domain model consists of the following components:

- *Domain definition*: A domain definition defines the scope of a domain and characterizes its contents by giving examples of systems in a domain, counterexamples (i.e. systems outside the domain), and generic rules of inclusion or exclusion (e.g. "Any system having the capability X belongs to the domain."). *Domain Definition*

- *Domain lexicon*: A domain lexicon defines the domain vocabulary. *Domain Lexicon*

- *Concept models*: Concept models describe the concepts in a domain expressed in some appropriate modeling formalism (e.g. object diagrams, interaction and state-transition diagrams, or entity-relationship and data-flow diagrams). *Concept model*

- *Feature models*: Feature models define a set of reusable and configurable requirements for specifying the systems in a domain. Such requirements are generally referred to as *features*. A feature model prescribes which feature combinations are meaningful: It represents the configuration aspect of the reusable software. We discuss feature models in Chapter 5.4 in great detail. *Feature model*

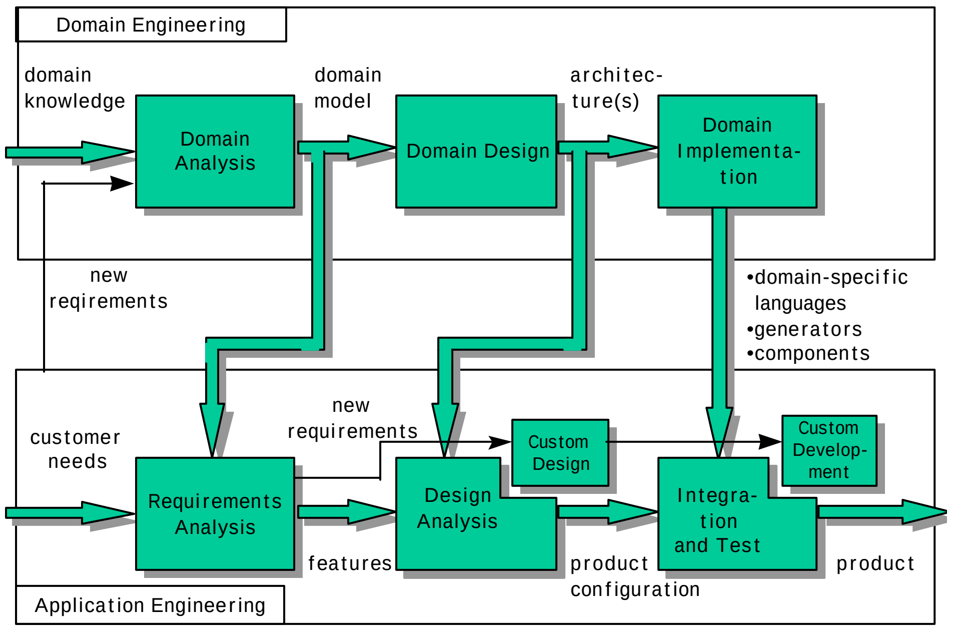


Figure 8 *Software development based on Domain Engineering (adapted from [MBSE97])*

Domain Analysis generally involves the following activities:

- *Domain planning, identification, and scoping*: planning of the resources for performing domain analysis, identifying the domain of interest, and defining the scope of the domain; *Domain planning, identification, and scoping*

- *Domain modeling*: developing the domain model. *Domain modeling*

Table 4 gives you a more detailed list of Domain Analysis activities. This list was compiled by Arango [Ara94] based on the study of eight different Domain Analysis methods.

Domain Analysis major process components	**Domain Analysis activities**
Domain characterization *(domain planning and scoping)*	*Select domain* Perform business analysis and risk analysis in order to determine which domain meets the business objectives of the organization.
	Domain description Define the boundary and the contents of the domain.
	Data source identification Identify the sources of domain knowledge.
	Inventory preparation Create inventory of data sources.
Data collection *(domain modeling)*	*Abstract recovery* Recover abstractions
	Knowledge elicitation Elicit knowledge from experts
	Literature review
	Analysis of context and scenarios
Data analysis *(domain modeling)*	*Identification of entities, operations, and relationships*
	Modularization Use some appropriate modeling technique, e.g. object-oriented analysis or function and data decomposition. Identify design decisions.
	Analysis of similarity Analyze similarities between entities, activities, events, relationships, structures, etc.
	Analysis of variations Analyze variations between entities, activities, events, relationships, structures, etc.
	Analysis of combinations Analyze combinations suggesting typical structural or behavioral patterns.
	Trade-off analysis Analyze trade-offs that suggest possible decompositions of modules and architectures to satisfy incompatible sets of requirements found in the domain.
Taxonomic classification *(domain modeling)*	*Clustering* Cluster descriptions.
	Abstraction Abstract descriptions.
	Classification Classify descriptions.
	Generalization Generalize descriptions.
	Vocabulary construction
Evaluation	Evaluate the domain model.

Table 4 *Common Domain Analysis process by Arango [Ara94]*

3.4 Domain Design and Domain Implementation

Software architecture

The purpose of *Domain Design* is to develop an *architecture* for the systems in the domain. Shaw and Garlan define software architecture as follows [SG96]:

> *"Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among these components. Such a system may in turn be used as a (composite) element in a larger system design."*

Buschmann et al. offer another definition of software architecture [BMR+96]:

> *A software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and nonfunctional properties of a software system. The software architecture of a system is an artifact. It is the result of the software development activity.*

Just as the architecture of a building is usually represented using different views (e.g. static view, dynamic view, specification of materials, etc.), the adequate description of a software architecture also requires multiple views. For example, the 4+1 View Model of software architecture popularized by the Rational methodologist Philippe Kruchten consists of a logical view (class, interaction, collaboration, and state diagrams), a process view (process diagrams), a physical view (package diagrams), a deployment view (deployment diagrams), plus a use case model (see Figure 17).

The elements and their connection patterns in a software architecture are selected to satisfy the requirements on the system (or the systems) described by the architecture. When developing a software architecture, we have to consider not only functional requirements, but also nonfuctional requirements such as performance, robustness, failure tolerance, throughput, adaptability, extendibility, reusability, etc. Indeed, one of the purposes of software architecture is to be able to quickly tell how the software satisfies the requirements. Eriksson and Penker [EP98] say that "architecture should serve as a map for the developers, revealing how the system is constructed and where specific functions or concepts are located."

Architectural patterns

Certain recurring arrangements of elements have proven to be particularly useful in many designs. We refer to these arrangements as *architectural patterns.* Each architectural pattern aims at satisfying a different set of requirements. Buschman et al. have compiled a (partial) list of architectural patterns (see [BMR+96] for a detailed description of these patterns):

- *Layers pattern*: An arrangement into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

- *Pipes and filters pattern*: An arrangement that processes a stream of data, where a number of processing steps are encapsulated in filter components. Data is passed through pipes between adjacent filters, and the filters can be recombined to build related systems or system behavior.

- *Blackboard pattern*: An arrangement where several specialized subsystems assemble their knowledge to build a partial or approximate solution to a problem for which no deterministic solution strategy is known.

- *Broker pattern*: An arrangement where decoupled components interact by remote service invocations. A broker component is responsible for coordinating communication and for transmitting results and exceptions.

- *Model-view-controller pattern*: A decomposition of an interactive system into three components: A model containing the core functionality and data, one or more views displaying information to the user, and one or more controllers that handle user input. A change-propagation mechanism ensures consistency between user interface and model.

- *Microkernel pattern*: An arrangement that separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

It is important to note that real architectures are usually based on more than one of these and other patterns at the same time. Different patterns may be applied in different parts, views, and at different levels of an architecture.

The architectural design of a system is a high-level design: it aims at coming up with a flexible structure which satisfies all important requirements and still leaves a large degree of freedom for the implementation. The architecture of a family of systems has to be even more flexible since it must cover different sets of requirements. In particular, it has to include an explicit representation of the variability (i.e. configurability) it covers so that concrete architectures can be configured based on specific sets of requirements. One way to capture this variability is to provide configuration languages for the configurable parts of the architecture. We will see a concrete example of a configuration language in Chapter 10.

A flexible architecture is the prerequisite for enabling the evolution of a system. As a rule, we use the most stable parts to form the "skeleton" and keep the rest flexible and easy to evolve. But even the skeleton has to be sometimes modified. Depending on the amount of flexibility an architecture provides, we distinguish between generic and highly flexible architectures [SCK+96]:

Generic vs. highly flexible architectures

- *Generic architecture*: A system architecture which generally has a fixed topology but supports component plug-and-play relative to a fixed or perhaps somewhat variable set of interfaces. We can think of a generic architecture as a frame with a number of sockets where we can plug in some alternative or extension components. The components have to clearly specify their interfaces, i.e. what they expect and what they provide.

- *Highly flexible architecture*: An architecture which supports structural variation in its topology, i.e. it can be configured to yield a particular generic architecture. The notion of a highly flexible architecture is necessary since a generic architecture might not be able to capture the structural variability in a domain of highly diverse systems. In other words, a flexible architecture componentizes even the "skeleton" and allows us to configure it and to evolve it over time.

Software architecture is a relatively young field with a very active research. You will find more information on this topic in [SG96, BMR+96, Arch, BCK98].

Domain Implementation

Domain Design is followed by Domain Implementation. During *Domain Implementation* we apply appropriate technologies to implement components, generators for automatic component assembly, reuse infrastructure (i.e. component retrieval, qualification, dissemination, etc.), and application production process.[15]

3.5 Application Engineering

Application Engineering is the process of building systems based on the results of Domain Engineering (see Figure 8). During the requirements analysis for a new systems, we take advantage of the existing domain model and select the requirements (*features*) from the domain model which match customer needs. Of course, new customer requirements not found in the domain model require custom development. Finally, we assemble the application from the

[15] Some authors (e.g. [FPF96, p. 2]) divide Domain Engineering into only two parts, Domain Analysis and Domain Implementation, and regard the development of an architecture merely as an activity in the Domain Implementation.

existing reusable components and the custom-developed components according to the reusable architecture, or, ideally, let a generator do this work.

3.6 Selected Domain Engineering Concepts

In the following sections, we discuss a number of basic concepts related to Domain Engineering: *domain*, *domain scope*, *relationships between domains*, *problem space* and *solution space*, and *specialized Domain Engineering methods*.

3.6.1 Domain

The American Heritage Dictionary of the English Language gives us a very general definition of a domain [Dict]:

> *"Domain: A sphere of activity, concern, or function; a field, e.g. the domain of history."*

According to this definition, we can view a domain as a body of knowledge organized around some focus, such as a certain professional activity.

Simos et al. note that the term *domain* is used in different disciplines and communities, such as linguistics, cultural research, artificial intelligence (AI), object-oriented technology (OO), and software reuse, in somewhat different meanings [SCK+96, p. 20]. They distinguish two general usage categories of this term:

1. *domain as the "real world"*;

2. *domain as a set of system*.

The notion of *domain as the "real world"* is used in the AI and knowledge-engineering communities. For example, the guidebook on Knowledge-Based Systems Analysis and Design Support (KADS), which is a prominent method for developing knowledge-based systems, gives the following definition [TH93, p. 495]:

> *"Domain: An area of or field of specialization where human expertise is used, and a Knowledge-Based System application is proposed to be used within it."*

Domain as the "real world" encapsulates the knowledge about the problem area (e.g. *accounts*, *customers*, *deposits and withdrawals*, etc., in a *bank accounting domain*), but not about the software from this problem area. This notion of domain as the "real world" is also used in object-oriented technology. For example, the UML (Unified Modeling Language) glossary defines domain as follows [UML97a]:

> *"Domain: An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area."*

In the context of software reuse and particularly in Domain Engineering, the term *domain* encompasses not only the "real world" knowledge but also the knowledge about how to build software systems in the domain of interest. Some early definitions even equate domain to a set of systems, e.g. [KCH+90, p. 2]:

> *"Domain: A set of current and future applications which share a set of common capabilities and data."*

or [Bai92, p. 1]:

> *"Domains are families of similar systems."*

This *domain as a set of systems* view is more appropriately interpreted as the assertion that a domain encompasses the knowledge used to build a family of software systems.

It is essential to realize that a domain is defined by the consensus of its *stakeholders*, i.e. people having an interest in the domain, e.g. marketing and technical managers, programmers, end-users, and customers, and therefore it is subject to both politics and legacies.

Srinivas makes the key observation that the significance of a domain is *externally* attributed [Sri91]:

> *Nothing in the individual parts of a domain either indicates or determines the cohesion of the parts as a domain. The cohesion is external and arbitrary—a collection of entities is a domain only to an extent that it is* perceived *by a community as being useful for modeling some aspect of reality.*

Shapere explains this community-based notion of a domain as follows [Sha77] (paraphrase from [Sri91]):

> *In a given community, items of real-world information come to be associated as bodies of information or* problem domains *having the following characteristics:*
>
> - *deep or comprehensive relationships among the items of information are suspected or postulated with respect to some class of problems;*
> - *the problems are perceived to be significant by the members of the community.*

Finally, it is also important noting that the kinds of knowledge contained in a domain include both

- formal models, which can often be inconsistent among each other, e.g. different domain theories, and
- informal expertise, which is difficult or impossible to formalize (as exemplified by the problems in the area of expert systems [DD87]).

As a conclusion, we will adopt the following definition of a *domain*:

Domain: An area of knowledge

- *scoped to maximize the satisfaction of the requirements of its stakeholders,*
- *including a set of concepts and terminology understood by practitioners in that area, and*
- *including knowledge of how to build software systems (or parts of software systems) in that area.*

3.6.2 Domain Scope

Horizontal and vertical scope

There are two kinds of domain scope with respect to the software systems in a domain (see Figure 1, p. 7):

- *Horizontal scope or system category scope*: How many different systems are in the domain? For example, the domain of containers (e.g. sets, vectors, lists, maps, etc.) has a larger horizontal scope than the domain of matrices since more application need containers than matrices.
- *Vertical scope or per-system scope*: Which parts of these systems are in the domain? The vertical scope is the larger the larger parts of the systems are in the domain. For example, the vertical scope of the domain of containers is smaller than the vertical scope of the domain of portfolio management systems since containers capture only a small slice of the functionality of a portfolio management system.

Based on the per-system scope, we distinguish between the following kinds of domains [SCK+96]:

Vertical, horizontal, encapsulated, and diffused domains

- *vertical* vs. *horizontal* domains;
- *encapsulated* vs. *diffused* domains.

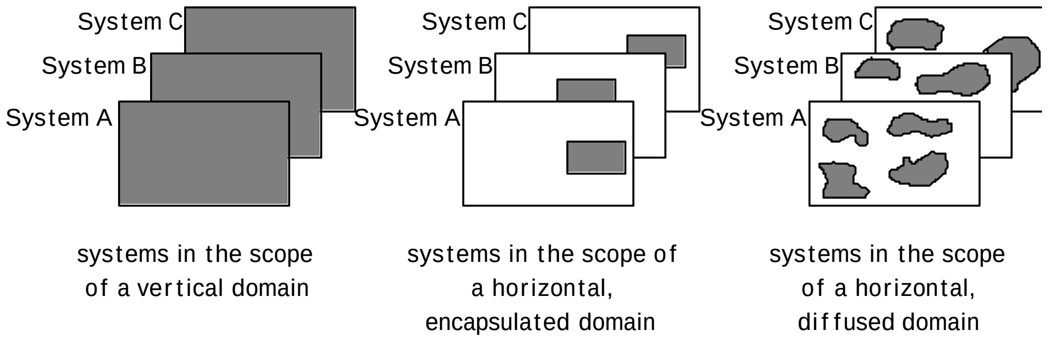

System C
System B
System A
systems in the scope of a vertical domain
System C
System B
System A
systems in the scope of a horizontal, encapsulated domain
System C
System B
System A
systems in the scope of a horizontal, diffused domain

Figure 9 *Vertical, horizontal, encapsulated, and diffused domains. (Each rectangle represents a system. The shaded areas depict system parts belonging to the domain.)*

Vertical domains contain complete systems (see Figure 9). Horizontal domains contain only parts of the systems in the domain scope. Encapsulated domains are horizontal domains where the system parts in the domain are well-localized with respect to their systems. Diffused domains are also horizontal domains, but they contain several, different parts of each system in the domain scope.[16]

Native vs. innovative domains

The scope of a domain can be determined using different strategies [SCK+96]:

1. choose a domain from the existing "*native*" domains (i.e. a domain which is already recognized in an organization);
2. define an *innovative* domain based on
   - a set of existing software systems sharing some commonalities (i.e. a family of systems) and/or
   - some marketing strategy.

The last two strategies are closely related to the following two concepts:

Product lines vs. product families

[16] Please note that, in Domain Engineering, the terms *horizontal* and *vertical* are used in a different sense than in the Object Management Architecture (OMA) defined by the Object Management Group (OMG, see www.omg.org). The OMG uses the term *vertical domain interfaces* to denote component interfaces specific to a specialized market (e.g. manufacturing, finance, telecom, transportation, etc.) and the term *horizontal facilities* (or *common facilities*) to denote generic facilities such as printing, database facilities, electronic mail facilities, etc. Thus, the OMG distinguishes between horizontal and vertical components, whereas in Domain Engineering we say that components have a horizontal nature in general since their scope does not cover whole systems but rather parts of systems. In Domain Engineering terms (see Section 6.4.1), OMG horizontal components are referred as *modeling components* (i.e. they model some general aspect such as persistency or printing) and the OMG vertical components are referred to as *application-specific components*. On the other hand, it is correct to say that modeling components have a larger horizontal scope than application-specific components.

- *Product family*: "A product family is a group of products that can be built from a common set of assets." [Wit96, p. 16] A product family is defined on the basis of similarities between the structure of its member products. A product family shares at least a common generic architecture. *Product families are scoped based on commonalities between the products.*

- *Product line*: "A product line is a group of products sharing a common, managed set of features that satisfy the specific needs of a selected market." [Wit96, p. 15] Thus, the definition of a product line is based on a marketing strategy rather than similarities between its member products. The features defined for a product line might require totally different solutions for different member products. A product line might be well served with one product family; however, it might also require more than one product family. On the other hand, a product family could be reused in more then one product line. *Product lines are scoped based on a marketing strategy.*

Unfortunately, the terms product family and product line are often used interchangeably in the literature.

We determine the scope of a domain during the domain scoping activity of Domain Analysis. The scope of a domain is influenced by several factors, such as the stability and the maturity of the candidate areas to become parts of the domain, available resources for performing Domain Engineering, and the potential for reuse of the Domain Engineering results within and outside an organization. In order to ensure a business success, we have to select a domain that strikes a healthy balance among these factors. An organization which does not have any experience with Domain Engineering should choose a small but important domain, e.g. some important aspect of most systems it builds. The resulting components and models can be reused on internal projects or sold outside the organization. After succeeding with the first domain, the organization should consider adding more and more domains to cover its product lines.

3.6.3 Relationships Between Domains

We recognize three major types of relationships between domains:

Subdomains

- A *is contained in* B: All knowledge in domain A also belongs to domain B, i.e. A is a *subdomain* of B.[17] For example, the domain of matrix packages is a subdomain of the domain of matrix computation packages since matrix computations cover both matrices and matrix computation algorithms.

Support domains

- A *uses* B: Knowledge in A references knowledge in B in a significant way, i.e. it is worthwhile to represent aspects of A in terms of B. We say that B is a *support domain* of A. For example, the storage aspect of a matrix package implemented using different containers from a container package. In other words, the domain of container packages is a support domain of the domain of matrix packages.

Analogy domains

- A *is analogous to* B [SCK+96]: There is a considerable amount of similarity between A and B; however, it is not necessarily worthwhile to express one domain in terms of the other. We say that A is an *analogy domain* of B. For example, the domain of numerical array packages is an analogy domain of the domain of matrix packages. They are both at a similar level of abstraction (in contrast to the more fundamental domain of containers, which could be a support domain for the domain of numerical array and the domain matrix packages) and clearly have different focuses (see Section 10.1.1.2.6). Yet still there is a considerable amount of similarity between them and studying one domain may provide useful insights into the other one.

[17] In [SCK+96] B is referred to as *generalization* of A and A as *specialization* of B.

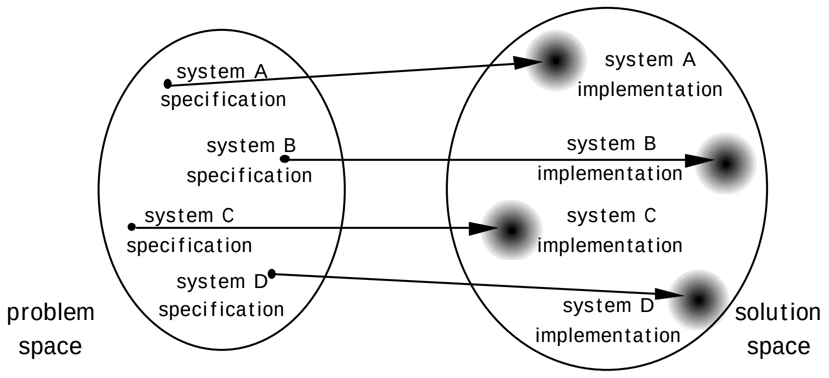


Figure 10 *Problem and solution space*

3.6.4 Problem and Solution Space

The set of all valid system specifications in a domain (e.g. valid feature combinations) is referred to as the *problem space* and the set of all concrete systems in the domain is referred as to as the *solution space* (see Figure 10). One of the goals of Domain Engineering is to produce components, generators, production processes, etc., which automate the mapping between the system specifications and the concrete systems.

A problem space contains the domain concepts that application programmers wold like to interact with when specifying systems, whereas the solution space contains the implementation concepts. There is natural a tension between these two spaces because of their different goals: The domain concepts have a structure that allows direct and intentional expression of problems. On the other hand, when we design the implementation concepts, we strive for small, atomic components that can be combined in as many ways as possible. We want to avoid any code duplication by factoring out similar code sections into small, (parameterized) components. This is potentially at odds with the structure of the problem space since not all of these small components should be visible to the application programmer. There is a number of other issues to consider when we design both spaces. We discuss them in Section 9.4.3.

The overall structure of the solution space is referred to as the *target architecture*. For example, the target architecture of the generative matrix computation library described in Chapter 10 is a special form of a layered architecture referred to as the GenVoca architecture (see Section 6.4.2). The target architecture defines the framework for the integration of the implementation components.

Domain-specific languages

The system specifications in the problem space are usually expressed using a number of *domain-specific languages* (DSLs), i.e. languages specialized for the direct and declarative expression of system requirements in a given domain. These languages define the domain concepts. We discuss the advantages of DSLs in Section 7.6.1 and the issues concerning their design and implementation in Section 9.4.1.

3.6.5 Specialized Methods

Different kinds of systems require different modeling techniques. For example, most important aspects of interactive systems are captured by use cases and scenarios. On the other hand, large data-centric applications are sometimes more appropriately organized around entity-relationship diagrams or object diagrams. Additional, special properties such as real-time support, distribution, and high availability and fault tolerance require specialized modeling techniques. Thus, different categories of domains will require different specialized domain engineering methods, i.e. methods deploying specialized notations and processes. We will discuss this issue in Chapter 4. In Chapter 9, we present DEMRAL, a specialized Domain Engineering method for developing reusable algorithmic libraries.

3.7 Survey of Domain Analysis and Domain Engineering Methods

There is a large number of Domain Analysis and Domain Engineering methods. Two of them deserve special attention since they belong to the most mature and best documented (including case studies) methods currently available: *Feature-Oriented Domain Analysis* and *Organization Domain Modeling*. We describe them in Sections 3.7.1 and 3.7.2. Sections 3.7.3 through 3.7.8 contain short descriptions of twelve other Domain Engineering methods or approaches. Each of them has made important contributions to some aspects of Domain Engineering (such as conceptual clustering, rationale capture, formal approaches, etc.).

Two surveys of Domain Analysis methods have been published to date: [WP92] and the more comprehensive [Ara94]. Compared to these surveys, the following sections also reflect the newest development in the field of Domain Engineering.

Please note that, in order to be consistent with the original descriptions of the Domain Engineering methods in the literature, the survey uses the term *phase* in its older meaning, i.e. to denote *process components* (cf. footnote on page 33).

3.7.1 Feature-Oriented Domain Analysis (FODA)

FODA is a Domain Analysis method developed at the Software Engineering Institute (SEI). The method is described in [KCH+90]. Tool support for FODA is outlined in [Kru93] and a comprehensive example of applying FODA to the *Army Movement Control Domain* is described in [CSJ+92, PC91]. A number of other military projects to use FODA are listed in [CARDS94, p. F.2]. FODA has also been applied in the area of telecommunication systems, e.g. [Zal96, VAM+98].

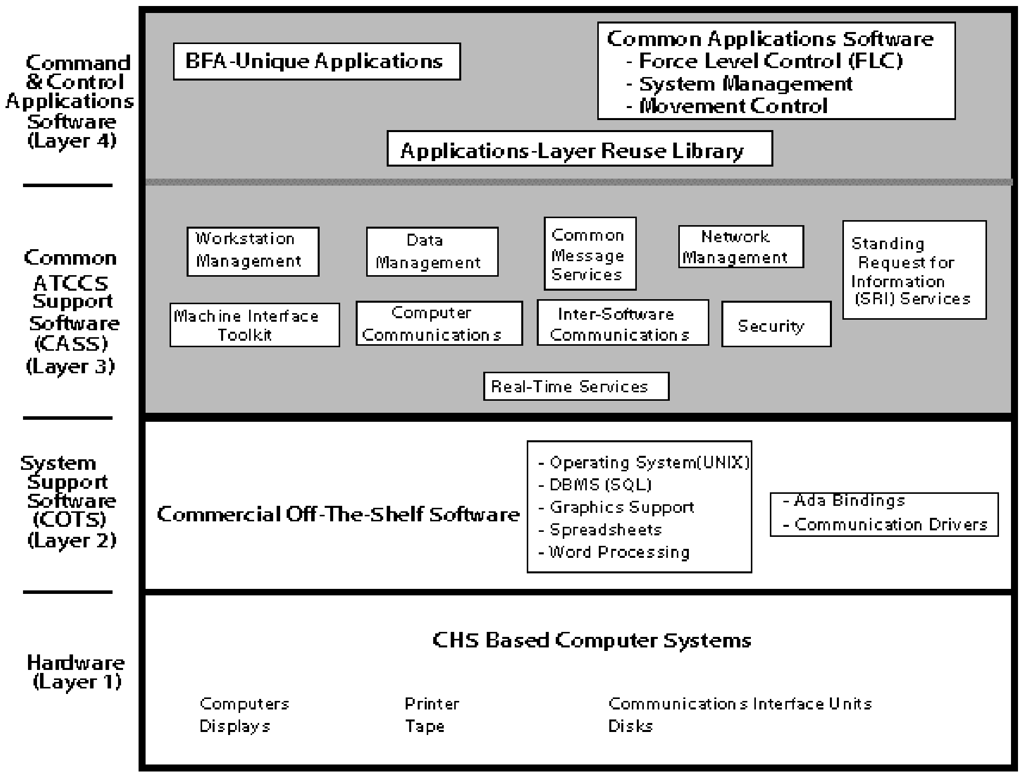

Command & Control Applications Software (Layer 4)
BFA-Unique Applications
Common Applications Software
- Force Level Control (FLC)
- System Management
- Movement Control
Applications-Layer Reuse Library
Common ATCCS Support Software (CASS) (Layer 3)
Workstation Management
Data Management
Common Message Services
Network Management
Standing Request for Information (SRI) Services
Machine Interface Toolkit
Computer Communications
Inter-Software Communications
Security
Real-Time Services
System Support Software (COTS) (Layer 2)
Commercial Off-The-Shelf Software
- Operating System(UNIX)
- DBMS (SQL)
- Graphics Support
- Spreadsheets
- Word Processing
- Ada Bindings
- Communication Drivers
Hardware (Layer 1)
CHS Based Computer Systems
Computers
Displays
Printer
Tape
Communications Interface Units
Disks

Figure 11 *Example of a FODA structure diagram: The structure diagram of the Army Movement Control Domain (from [PC91])*

3.7.1.1 FODA Process

The FODA process consists of two phases [MBSE97]:[18]

1. *Context Analysis*: The purpose of Context Analysis is to define the boundaries of the domain to be analyzed.

2. *Domain Modeling*: The purpose of Domain Modeling is to produce a domain model.

We describe these phases in the following two subsections.

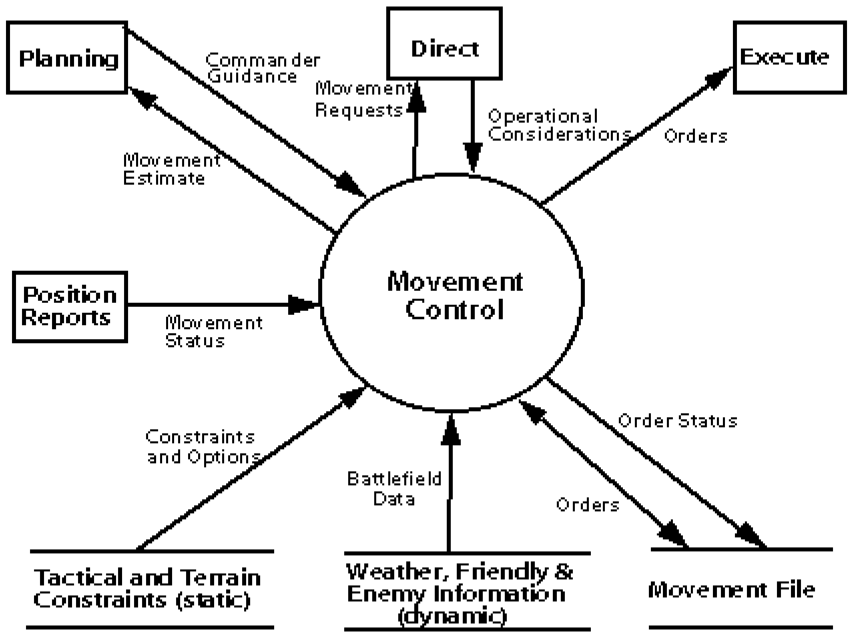


Figure 12 *Example of a FODA context diagram: The context diagram of the Army Movement Control Domain (from [PC91]). A FODA context diagram is a typical data-flow diagram: "The arrows represent the information received or generated by the movement control domain. The closed boxes represent the set of sources and sinks of information. The open-ended boxes represent the databases that the movement control domain must interact with." [MBSE97]*

3.7.1.1.1 Context Analysis

The FODA Context Analysis defines the scope of a domain that is likely to yield useful domain products.[19] In this phase, the relationships between the domain of focus and other domains or entities are also established and analyzed for variability. The results of the context analysis along with factors such as availability of domain expertise and project constraints are used to limit the scope of the domain [MBSE97]. The results of the Context Analysis are the *context model* which includes a *structure diagram* (see Figure 11) and a *context diagram* (see Figure 12).

[18] Originally, FODA contained a third phase called *Architectural Modeling* (see [KCH+90]). This phase is no longer part of FODA, but instead it was converted into the *Domain Design* phase, which follows FODA in the overall framework of *Model-Based Software Engineering* (see Section 3.7.1.3).

[19] The FODA Context Analysis corresponds to the domain planning and domain scoping activities defined in Section 3.3.

3.7.1.1.2 Domain Modeling

During the FODA Domain Modeling phase the main commonalities and variabilities between the applications in the domain are identified and modeled. This phase involves the following steps [MBSE97]:

1. *Information Analysis*: The main purpose of Information Analysis is to capture domain knowledge in the form of domain entities and the relationships between them. The particular modeling technique used in this phase could be semantic networks, entity-relationship modeling, or object-oriented modeling. The result of Information Analysis is the *information model,* which corresponds to the concept model mentioned in Section 3.3.

2. *Features Analysis*: “Features Analysis captures a customer’s or end-user’s understanding of the general capabilities of applications in a domain. For a domain, the commonalities and differences among related systems of interest are designated as *features* and are depicted in the *features model*.”[20] [MBSE97]

3. *Operational Analysis*: Operational Analysis yields the *operational model* which represents how the application works by capturing the relationships between the objects in the information model and the features in the features model.

Another important product of this phase is a *domain dictionary* which defines all the terminology used in the domain (including textual definitions of the features and entities in the domain).

3.7.1.2 The Concept of Features

In FODA, features are the properties of a system which *directly affect* end-users[21]:

“Feature: A prominent and user-visible aspect, quality, or characteristic of a software system or systems.” [KCH+90, p. 3]

For example, “when a person buys an automobile a decision must be made about which transmission feature (e.g. *automatic* or *manual*) the car will have.” [KCH+90, p. 35] Thus, FODA features can be viewed as features in the sense of Conceptual Modeling (see Section 2.2) with the additional requirement of directly affecting the end-user.

Two definitions of feature

In general there are two definitions of features found in Domain Engineering literature:

1. a end-user-visible characteristic of a system, i.e. the FODA definition, or

2. a distinguishable characteristic of a concept (e.g. system, component, etc.) that is relevant to some stakeholder of the concept. The latter definition is used in the context of ODM (see Section 3.7.2) and Capture (see Section 3.7.4) and is fully compatible with the understanding of features in Conceptual Modeling.

We prefer the latter definition since it is more general and covers the important case of software components.

Mandatory, alternative, and optional features

The features of a software system are documented in a *features model*. An important part of this model is the *features diagram.* An example of a simple features diagram of an automobile is shown in Figure 13. This example also illustrates three types of features[22]:

[20] The FODA term “features model” is equivalent to the term “feature model” defined in Section 3.3.

[21] A user may be a human user or another system with which applications in a domain typically interact.

1. *mandatory features*, which each application in the domain must have, e.g. all cars have a *transmission*;

2. *alternative features*, of which an application can posses only one at a time, e.g. *manual* or *automatic* transmission;

3. *optional features*, which an application may or may not have, e.g. *air conditioning*.

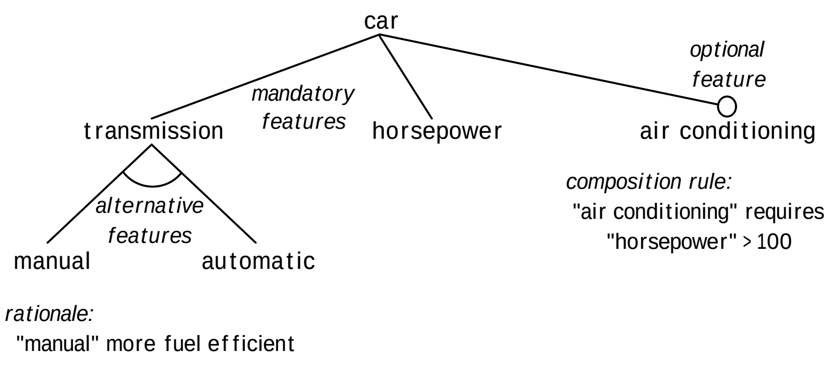


Figure 13 *Example showing features of a car (from [KCH+90, p. 36]). Alternative features are indicated by an arc and optional features by an empty circle.*

The features diagram has the form of a tree in which the root represents the concept being described and the remaining nodes denote features. The relationships are *consists-of* relationships denoting, for example, that the description of a *transmission* consists of the descriptions of *manual* and *automatic* transmissions.

The FODA-style of featural description subsumes both the featural and the dimensional descriptions from the classical conceptual modeling, which we discussed in Sections 2.2.1 and 2.3.6. This is illustrated in Figure 14.

[22] Strictly speaking, we have to distinguish between direct features of a concept and subfeatues of features. Direct features of an application may be mandatory, alternative, or optional with respect to all applications within the domain. A subfeature may be mandatory, alternative, or optional with respect to only the applications which also have its parent feature. We explain this idea in Chapter 5.4.1.

Feature interdependencies are captured using *composition rules* (see Figure 13). FODA utilizes

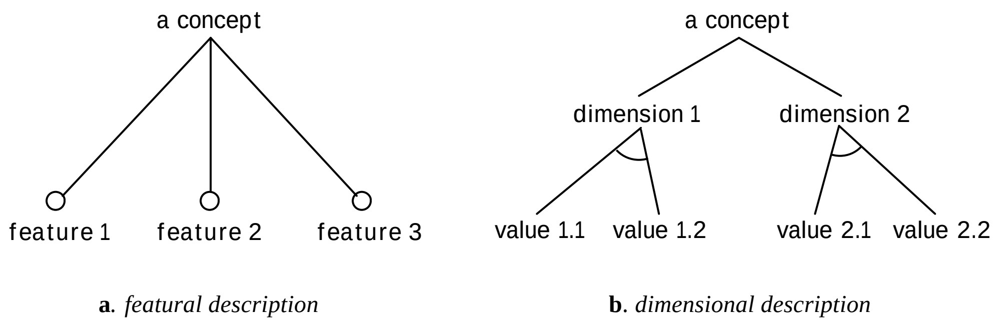


Figure 14 *Representation of featural and dimensional descriptions using FODA feature notation*

two types of composition rules:

Composition rules

1. *requires rules*: Requires rules capture implications between features, e.g. "*air conditioning* requires *horsepower* greater than 100" (see Figure 13).

2. *mutually-exclusive-with rules*: These rules model constraints on feature combinations. An example of such a rule is "*manual* mutually exclusive with *automatic*". However, this rule is not needed in our example since *manual* and *automatic* are alternative features. In general, mutually-exclusive-with rules allow us to exclude combinations of features where each feature may be seated in quite different locations in the feature hierarchy.

Rationale

We can also annotate features with *rationales*. A rationale documents the reasons or trade-offs for choosing or not choosing a particular feature. For example, manual transmission is more *fuel efficient* than automatic one. Rationales are necessary since, in practice, not all issues pertaining to the feature model can be represented formally as composition rules (due to the complexity involved or limited representation means). Theoretically, *fuel efficient* in Figure 13 could be modeled as a feature. In this case, the dependency between *manual* and *fuel efficient* could be represented as the following composition rule: *fuel efficient* requires *manual*. However, one quickly recognizes that the dependency between *fuel efficient* and *manual* is far more complex. First, we would need some measure of *fuel efficiency* and, second, *fuel efficiency* is influenced by many more factors than just the type of car transmission. The problem becomes similar to the problem of representing human expertise in expert systems [DD87]. Thus, stating the rationale informally allows us to avoid dealing with this complexity. In general, rationale refers to factors that are outside of the considered model.

Two definitions of rationale

The usage of the term rationale in the Domain Engineering literature is inconsistent. There are roughly two definitions of this term:

1. the trade-offs for choosing or not choosing a particular feature, i.e. the FODA definition (this notion is similar to the forces section in the description of a design pattern [GHJV95]);

2. the particular reason for choosing a specific feature after considering a number of trade-offs (this would correspond to recording the information about which forces were directly responsible for arriving at the decision made). The latter definition is used in Capture (Section 3.7.4) and in ODM (Section 3.7.2). This definition is motivated by the work on *design rationale capture* [Shu91, Bai97], the goal of which is to record the reason for selecting a particular design alternative by a (not necessarily software) designer during the design of a specific system.

Based on the purpose of a feature, the FODA features model distinguishes between *context*, *representation*, and *operational features* [MBSE97]:[23]

1. *Context features* "are those which describe the overall mission or usage patterns of an application. Context features would also represent such issues as performance requirements, accuracy, and time synchronization that would affect the operations." [MBSE97]

2. *Representation features* "are those features that describe how information is viewed by a user or produced for another application (i.e., what sort of input and output capabilities are available)." [MBSE97]

3. *Operational features* "are those features that describe the active functions carried out (i.e., what the application does)." [MBSE97]

Of course, other types of features are also possible. For example, Bailin proposes the following feature types: *operational, interface, functional, performance, development methodology, design,* and *implementation features* [Bai92].

Finally, FODA features are classified according to their binding time into *compile-time*, *activation-time*, and *runtime features* [KCH+90]:

1. *Compile-time features* are "features that result in different packaging of the software and, therefore, should be processed at compile time. Examples of this class of features are those that result in different applications (of the same family), or those that are not expected to change once decided. It is better to process this class of features at compile time for efficiency reasons (time and space)."

2. *Activation-time features* (or *load-time features*) are those "features that are selected at the beginning of execution but remain stable during the execution. [...] Software is generalized (e.g. table-driven software) for these features, and instantiation is done by providing values at the start of each execution."

3. *Runtime features* are those "features that can be changed interactively or automatically during execution. Menu-driven software is an example of implementing runtime features."

Binding time, binding location, and binding site

The FODA classification of features according to binding time is incomplete. There are also other times, e.g. linking time, or first-call time (e.g. when a method is called the first time; this time is relevant for just-in-time compilation [Kic97]). In general, feature *binding time* can be classified according to the specific times in the life cycle of a software system. Some specific products could have their specific times (e.g. debugging time, customization time, testing time, or, for example, the time when something relevant takes place during the use of the system, e.g. emergency time, etc.). Also, when a component is used in more than one location in a system, the allowed component features could depend on this *location*. Furthermore, binding could depend on the context or setting in which the system is used. For this reason, Simos et al. introduced the term *binding site* ([SCK+96]) which covers all these cases (i.e. binding time and context). We will discuss this concept in Section 5.4.4.3 in more detail.

The features model describes the problem space in a concise way: "The features model is the chief means of communication between the customers and the developers of new applications. The features are meaningful to the end-users and can assist the requirements analysts in the derivation of a system specification that will provide the desired capabilities. The features model provides them with a complete and consistent view of the domain." [MBSE97]

To summarize, a FODA features model consists of the following four key elements:

[23] The original FODA description in [KCH+90] uses a slightly different categorization; it distinguishes between *functional*, *operational*, and *presentation* features.

1. *features diagram*, i.e. a representation of a hierarchical decomposition of features including the indication whether or not each feature is mandatory, alternative, or optional;

2. *feature definitions* for all features including the indication of whether each feature is bound at compile time, activation time, or at runtime (or other times);

3. *composition rules* for features;

4. *rationale* for features indicating the trade-offs.

We will come back to this topic in Chapter 5 where we define a more comprehensive representation of feature models.

3.7.1.3 FODA and Model-Based Software Engineering

FODA is a part of *Model-Based Software Engineering (MBSE)*, a comprehensive approach to family-oriented software engineering based on Domain Engineering, being developed by SEI (see [MBSE97] and [Wit94]).[24] MBSE deploys a typical family-oriented process architecture consisting of two processes: Domain Engineering and Application Engineering (see Figure 8). The Domain Engineering process, in turn, consists of Domain Analysis, Domain Design, and Domain Implementation, where FODA takes the place of Domain Analysis.

3.7.2 Organization Domain Modeling (ODM)

ODM is a domain engineering method developed by Mark Simos of Synquiry Ltd. (formerly Organon Motives Inc.). The origins of ODM date back to Simos's work on the knowledge-based reuse support environment *Reuse Library Framework (RLF)* [Uni88]. Since then ODM has been used and refined on a number projects, most notably the STARS project (see Section 3.8), and other projects involving organizations such as Hewlett-Packard Company, Lockheed Martin (formerly Loral Defense Systems-East and Unisys Government Systems Group), Rolls-Royce, and Logicon [SCK+96]. During its evolution, ODM assimilated many ideas from other domain engineering approaches as well as work in non-software disciplines such as organization redesign and workplace ethnography [SCK+96]. The current version 2.0 of ODM is described in [SCK+96], a comprehensive guidebook comprising almost five hundred pages. This guidebook replaces the original ODM description in [SC93].

Unique aspects of ODM

Some of the unique aspects of ODM include

- *Focus on stakeholders and settings*: Any domain concepts and features defined during ODM have explicit traceability links to their stakeholders and relevant contexts (i.e. settings). In addition, ODM introduces the notion of a *grounded* abstraction, i.e. abstraction based on stakeholder analysis and setting analysis, as opposed to the "right" abstraction (a term used in numerous textbooks on software design), which is based on intuition.

- *Types of domains*: ODM distinguishes between horizontal vs. vertical, encapsulated vs. diffused, and native vs. innovative domains (see Sections 3.6.1 and 3.6.2).

- *More general notion of feature*: ODM uses a more general notion of feature than FODA (see Section 3.7.1.2). An ODM feature does not have to be end-user visible; instead, it is defined as a difference between two concepts (or variants of a concept) that "makes a significant difference" to some stakeholder. ODM features directly correspond to the notion of features discussed in Chapter 2.

- *Binding site*: In FODA, a feature can be bound at compile, start, or runtime (see Section 3.7.1.2). ODM goes beyond that and introduces the notion of *binding site*, which allows for

[24] FODA was conceived before the work on MBSE started.

a broader and finer classification of binding times and contexts depending on domain-specific needs. We discuss this idea in Section 5.4.4.3

- *Analysis of feature combinations*: ODM includes explicit activities aimed towards improving the quality of features, such as feature clustering (i.e. co-occurrence of features), as well as the building of a closure of feature combinations (i.e. enumerating all valid feature combinations). The latter can lead to the discovery of innovative system configurations which have not been considered before.

- *Conceptual modeling*: ODM uses a very general modeling terminology similar to that introduced in Chapter 2. Therefore, ODM can be specialized for use with any specific system modeling techniques and notations, such as object-oriented analysis and design (OOA/D) methods and notations or structured methods. We discuss this topic in Chapter 4. Also, in Chapter 9, we present a specialization of ODM for developing algorithmic libraries.

- *Concept starter sets*: ODM does not prescribe any particular concept categories to look for during modeling. While other methods specifically concentrate on some concept categories such as objects, functions, algorithms, data structures, etc., ODM uses *concept starter sets* consisting of different combinations of concept categories to jumpstart modeling in different domains.

- *Scoping of the asset base*: ODM does not require the implementation of the full domain model. There is an explicit ODM task, the goal of which is to determine the part of the domain model to be implemented based on project and stakeholder priorities.

- *Flexible architecture*: ODM postulates the need for a *flexible architecture* since a *generic architecture* is not sufficient for domains with a very high degree of variability (see Section 3.4).

- *Tailorable process*: ODM does not commit itself to any particular system modeling and engineering method, or any market analysis, or any stakeholder analysis method. For the same reason, the user of ODM has to provide these methods, select appropriate notations and tools (e.g. feature notation, object-oriented modeling, etc.), and also invest the effort of integrating them into ODM.

The following section gives a brief overview of the ODM process.

3.7.2.1 The ODM Process

The ODM process—as described in [SCK+96]—is an extremely elaborate and detailed process. It consists of three main phases:

1. *Plan Domain*: This is the domain scoping and planning phase (Section 3.3) corresponding to Context Analysis in FODA (Section 3.7.1.1.1).

2. *Model Domain*: In this phase the *domain model* is produced. It corresponds to Domain Modeling in FODA (3.7.1.1.2).

3. *Engineer Asset Base*: The main activities of this phase are to produce the architecture for the systems in the domain and to implement the reusable assets.

Plan Domain and Model Domain clearly correspond to a typical Domain Analysis. Engineer Asset Base corresponds to Domain Design and Domain Implementation.

Each of the three ODM phases consists of three sub-phases and each sub-phase is further divided into three tasks. The complete ODM process is shown in Figure 15.

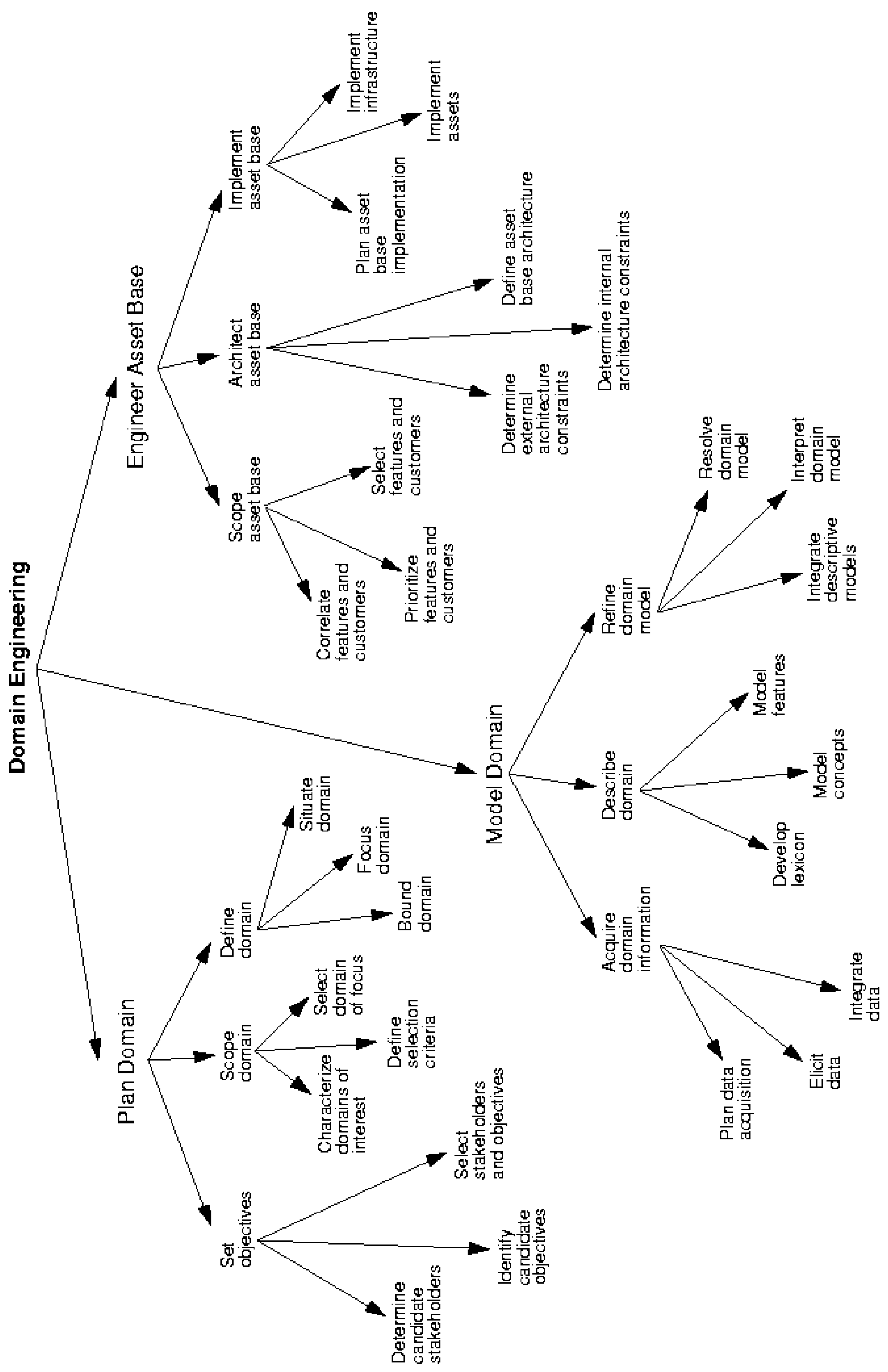

Domain Engineering
Plan Domain
Set objectives
Determine candidate stakeholders
Identify candidate objectives
Select stakeholders and objectives
Scope domain
Characterize domains of interest
Define selection criteria
Select domain of focus
Define domain
Bound domain
Focus domain
Situate domain
Model Domain
Acquire domain information
Plan data acquisition
Elicit data
Integrate data
Describe domain
Develop lexicon
Model concepts
Model features
Refine domain model
Integrate descriptive models
Interpret domain model
Resolve domain model
Engineer Asset Base
Scope asset base
Correlate features and customers
Prioritize features and customers
Select features and customers
Architect asset base
Determine external architecture constraints
Determine internal architecture constraints
Define asset base architecture
Implement asset base
Plan asset base implementation
Implement assets
Implement infrastructure

Figure 15 *Phases of the ODM process (from [SCK+96, p. 40])*

The ODM phases and sub-phases are described in Table 5.

ODM Phase	ODM Sub-Phase	Performed Tasks[25]
Plan Domain	Set objectives	• determine the stakeholders (i.e. any parties related to the project), e.g. *end-users, customers, managers, third-party suppliers, domain experts, programmers, subcontractors* • analyze stakeholders' objectives and project objectives • select stakeholders and objectives from the candidates
	Scope domain	• scope the domain based on the objectives (issues include choosing between *vertical* vs. *horizontal*, *encapsulated* vs. *diffused*, *native* vs. *innovative* domains)
	Define domain	• define the domain boundary by giving examples of systems in the domain, counterexamples (i.e. systems outside the domain), as well as generic rules defining what is in the domain and what not • identify the main features of systems in the domain and the usage settings (e.g. development, maintenance, customization contexts) for the systems • analyze the relationships between the domain of focus and other domains
Model Domain	Acquire domain information	• plan the domain information acquisition task • collect domain information from domain experts, by reverse-engineering existing systems, literature studies, prototyping, etc. • integrate the collected data, e.g. by pre-sorting the key domain terms, identifying the most important system features
	Describe domain	• develop a lexicon of domain terms • model the semantics of the key domain concepts • model the variability of concepts by identifying and representing their features
	Refine domain	• integrate the models produced so far into an overall consistent model • model the rationale for variability, i.e. the trade-offs for using or not using certain features • improve the quality of features by clustering and experimenting with innovative feature combinations
Engineer Asset Base	Scope asset base	• correlate identified features and customers • prioritize features and customers • based on the priorities, select the portion of the modeled functionality for implementation
	Architect asset base	• determine external architecture constraints (e.g. external interfaces and the allocation of features to the external interfaces) • determine internal architecture constraints (e.g. internal interfaces, allocation of groups of related features to internal interfaces) • define asset base architecture based on these constraints
	Implement asset base	• plan asset base implementation (e.g. selection of tools, languages, and other implementation strategies) • implement assets • implement infrastructure (e.g. domain-specific extensions to general infrastructures, asset retrieval mechanisms, asset qualification mechanisms)

Table 5 *Description of ODM phases and sub-phases*

3.7.3 Draco

Draco is an approach to Domain Engineering as well as an environment based on transformation technology. Draco was developed by James Neighbors in his Ph.D. work [Nei80] to be the first Domain Engineering approach. Furthermore, the main ideas introduced by Draco include *domain-specific languages* and *components as sets of transformations*. This section gives a brief overview of Draco. A more detailed discussion is given in Section 6.4.1.

The main idea of Draco is to organize software construction knowledge into a number of related domains. Each Draco domain encapsulates the needs and requirements and different implementations of a collection of similar systems. Specifically, a Draco domain contains the following elements ([Nei84, Nei89]):

- *Formal domain language (also referred to as "surface" language)*: The domain language is used to describe certain aspects of a system. The domain language is implemented by a parser and a pretty printer. The internal form of parsed code is a *parse tree*. The term *domain language* is equivalent to the term *domain-specific language* introduced in Section 3.6.4.

- *Set of optimization transformations*: These transformations represent rules of exchange of equivalent program fragments in the domain language and are useful for performing optimizations on the parse tree.

- *Set of transformational components*: Each component consists of one or more *refinement transformations* capable of translating the objects and operations of the source domain language into one or more target domain languages of other, underlying domains. There is one component for each object and operation in the domain. Thus, components implement a program in the source domain language in terms of the target domains. Draco refers to the underlying target domains as *refinements* of the source domain. As a result, the construction knowledge in Draco is organized into domains connected by *refinement* relationships.

- *Domain-specific procedures*: Domain-specific procedures are used whenever a set of transformations can be performed algorithmically. They are usually applied to perform tasks such as generating new code in the source domain language or analyzing programs in the source language.

- *Transformation tactics and strategies (also called optimization application scripts)*: Tactics are domain-independent and strategies are domain-dependent rules helping to determine when to apply which refinement. Optimizations, refinements, procedures, tactics, and strategies are organized into metaprograms (i.e. programs generating other programs).

It is important to note that, in Draco, a system is represented by many domain languages simultaneously.

The results of applying Draco to the domain of real-time applications and the domain of processing standardized tests are described in [Sun83] and [Gon81], respectively.

3.7.4 Capture

Capture, formerly known as KAPTUR (see [Bai92, Bai93]), is an approach and a commercial tool for capturing, organizing, maintaining, and representing domain knowledge. Capture was developed by Sidney Bailin of CTA Inc. (currently with Knowledge Evolution Inc.).

The Capture tool is a hypertext-based tool allowing the user to navigate among assets (e.g. architectures and components). The assets are documented using informal text and various diagrams, such as entity-relationship diagrams. The assets are annotated by their *distinctive features*, which document important design and implementation decisions. Features are themselves annotated with *trade-offs* that were considered and *rationale* for the particular decision made.[26] [Bai92]

3.7.5 Domain Analysis and Reuse Environment (DARE)

DARE is both a Domain Analysis method and a tool suite supporting the method [FPF96]. DARE was developed by William Frakes (Software Engineering Guild) and Rubén Prieto-Díaz (Reuse Inc.) and represents a commercial product.

Clusters and facets

The DARE tool suite includes lexical analysis tools for extracting domain vocabulary from system descriptions, program code, and other sources of domain knowledge. One of the most important tools is the *conceptual clustering* tool, which clusters words according to their conceptual similarity. The clusters are further manually refined into *facets*, which are main categories of words and phrases that fall in the domain [FPF96]. The idea of using facets to describe and organize systems and components in a domain has its roots in the application of library science techniques, such as faceted classification, to component retrieval [Pri85, Pri87, PF87, Pri91a, Pri91b, OPB92].

The main workproducts of DARE include a *facet table*, *feature table*, *system architecture*, and *domain lexicon* and are organized into a *domain book*. The DARE tool suite includes appropriate tools for creating and viewing these workproducts.

3.7.6 Domain-Specific Software Architecture (DSSA) Approach

The *DSSA* approach to Domain Engineering was developed under the *Advanced Research Project Agency's (ARPA) DSSA Program* (see [Hay94, TTC95]). The DSSA approach emphasizes the central role of the concept of *software architecture* in Domain Engineering. The overall structure of the DSSA process is compatible with the generic process structure described in Sections 3.3 through 3.5 (see [CT93, TC92] for descriptions of the DSSA process). The main workproducts of the DSSA process include the following [Tra95]:

1. *Domain Model*: The DSSA Domain Model corresponds to the concept model in Section 3.3 (i.e. concept model in ODM or information model in FODA) rather than a full domain model.

2. *Reference Requirements*: The DSSA Reference Requirements are equivalent to the feature model in Section 3.3. Each *reference requirement* (or *feature* in the terminology of Section 3.3) is either mandatory, optional, or alternative. The DSSA Reference Requirements include both functional and non-functional requirements.[27]

3. *Reference Architecture*: A DSSA Reference Architecture is an architecture for a family of systems consisting mainly of an *architecture model*, *configuration decision tree* (which is similar to the FODA features diagram in Section 3.7.1.2), *design record* (i.e. description of the components), and *constraints* and *rationale* (the latter two correspond to configuration rules and rationale in FODA in Section 3.7.1.2).

The need to formally represent the components of an architecture and their interrelationships led to the development of so-called *Architecture Description Languages* or *ADLs*. The concept of ADLs is described in [Cle96, Arch, SG96].

The DSSA approach has been applied to the avionics domain under the *Avionics Domain Application Generation Environment (ADAGE)* project involving Loral Federal Systems and other contractors (see [ADAGE]). As a result of this effort, a set of tools and other products supporting the DSSA process have been developed, including the following [HT94]:

- *DOMAIN*: a hypermedia-based Domain Analysis and requirements capture environment;

- *MEGEN*: an application generator based on module expressions;

- *LILEANA*: an ADL based on the ADA annotation language ANNA [LHK87] and the module interconnection language LIL [Gog83] (LILEANA is described in [Tra93, GT96]).

Other DSSA program efforts resulted in the development of other Domain Engineering tools and products (see [HT94] for more details), most notably the ADLs *ArTek* (developed by Teknowledge [THE+94]), *ControlH* and *MetaH* (developed by Honeywell [BEJV93]), and *Rapide* (developed at Stanford University [LKA+95]).

3.7.7 Algebraic Approach

The *algebraic approach* to Domain Engineering was proposed by Yellamraju Srinivas in [Sri91] (see [Smi96, SJ95] for more recent work). This section gives a brief overview of this approach. A more detailed description follows in Section 6.4.4.

The main idea of this approach is to formalize domain knowledge in the form of a network of related *algebraic specifications* (also referred to as *theories*). An algebraic specification defines a *language* and constrains its possible meanings through *axioms* and *inference rules*. Algebraic specifications can be related using *specification morphisms*. Specification morphisms define translations between specification languages that preserve the *theorems* (i.e. all statements which can be derived from the axioms using the inference rules). Thus, in the algebraic approach, the domain model is represented as a number of formal languages including translations between them. From this description, it is apparent that the algebraic approach and the Draco approach (Section 3.7.3) are closely related.[28] In fact, the only difference is that the algebraic approach is based on the algebraic specification theory (e.g. [LEW96]) and the category theory (e.g. [BW85]). Similarly to Draco, the algebraic approach lends itself well to implementation based on transformations. The inference rules of a specification correspond to the optimization transformations of Draco, and the specification morphisms correspond to refinement transformations.

First success reports on the practical use of the algebraic approach include the application of the transformation-based system *KIDS* (*Kestrel Interactive Development System*, see [Smi90]) in the domain of *transportation scheduling* by the Kestrel Institute. According to [SPW95], the scheduler generated from a formal domain model using KIDS is over 20 times faster than the standard, hand-coded system deployed by the customer. This proves the viability of the algebraic approach in narrow, well-defined domains. A successor system to KIDS is SPECWARE [SJ95], which is explicitly based on category theory (i.e. it uses category theory concepts both in its design and user interface).

3.7.8 Other Approaches

Other approaches to Domain Engineering include the following:

- *SYNTHESIS*: SYNTHESIS [SPC93] is a Domain Engineering method developed by the Software Productivity Consortium in the early nineties. The structure of the SYNTHESIS process is principally consistent with the generic process structure described in Sections 3.3 through 3.5 (although it uses a slightly different terminology). A unique aspect of SYNTHESIS is the tailorability of its process according to the levels of the *Reuse Capability Model* [SPC92]. This tailorability allows an organization to control the impact of the reuse process installation on its own structures and processes.

- *Family-Oriented Abstraction, Specification, and Translation (FASST)*: FASST is a Domain Engineering method developed by David Weiss et al. at Lucent Technologies Bell Laboratories [Wei96]. FASST has been greatly influenced by the work on SYNTHESIS (Weiss was one of the developers of SYNTHESIS).

- *Defense Information Systems Agency's Domain Analysis and Design Process (DISA DA/DP)*: DISA DA/DP [DISA93] is similar to MBSE (Section 3.7.1.3) and ODM (Section 3.7.2). However, it only includes Domain Analysis and Domain Design. DISA DA/DP uses the object-oriented Coad-Yourdon notation [CY90].

- *Joint Integrated Avionics Working Group (JIAWG) Object-Oriented Domain Analysis Method (JODA)*: JODA [Hol93] is a Domain Analysis method similar to FODA (see Section

3.7.1; however, JODA does not include a feature model) and is based on the object-oriented Coad-Yourdon notation and analysis method [CY90].

- *Gomaa*: [Gom92] describes an early object-oriented Domain Engineering method developed by Hassan Gomaa. An environment supporting the method is set out in [GKS+94].
- *Reusable Ada Products for Information Systems Development (RAPID)*: RAPID is a Domain Analysis approach developed by Vitaletti and Guerrieri [VG90], utilizing a similar process to the afore-presented Domain Engineering methods.
- *Intelligent Design Aid (IDeA)*: IDeA is a design environment supporting Domain Analysis and Domain Design [Lub91]. IDeA was developed by Mitchell Lubars. The unique aspect of IDeA is its iterative approach to Domain Analysis, whereby specific problems are analyzed one at a time and each analysis potentially leads to an update of the domain model.[29]

Since the main concepts and ideas of Domain Engineering have already been illustrated based on the methods presented in previous sections, we refrain from describing the approaches mentioned in this section in more detail.

3.8 Historical Notes

The idea of Domain Engineering can be traced back to the work on program families by Dijkstra [Dij70] and Parnas [Par76]. Parnas defines *program family* as follows [Par76, p. 1]:

> *"We consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members."*

The term *Domain Analysis* was first defined by Neighbors in his Ph.D. work on Draco [Nei80, pp. xv-xvi] as

> *"the activity of identifying objects and operations of a class of similar systems in a particular problem domain."*

Major efforts aimed at developing Domain Analysis methods (including SEI's FODA and the work by Prieto-Diaz et al. at the Software Productivity Consortium) followed in the late eighties. A comprehensive bibliography of work related to Domain Engineering from the period 1983-1990 can be found in [HNC+90].

A large share of the work on Domain Engineering was sponsored by the U.S. Department of Defense research programs related to software reuse including *Software Technology for Adaptable, Reliable Systems (STARS)* [STARS94], *Comprehensive Approach to Reusable Defense Software (CARDS)* [CARDS], and DSSA (Section 3.7.6).

Domain Engineering methods such as MBSE, ODM 2.0, and FASST (Sections 3.7.1.3, 3.7.2, 3.7.8) can be classified as second generation methods. The most recent trend in the field is to integrate Domain Engineering and OOA/D methods (see Chapter 4).

A partial genealogy of Domain Engineering methods is shown in Figure 16.

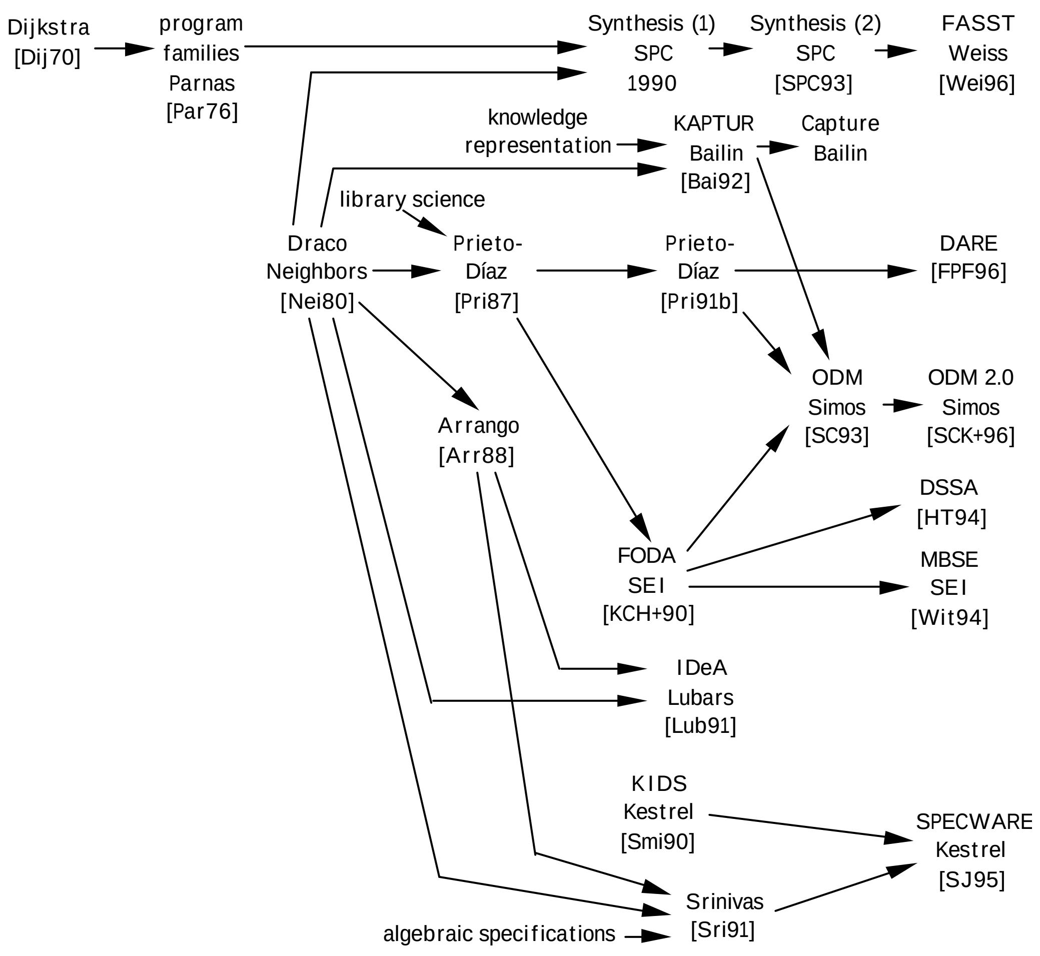


Figure 16 *Partial genealogy of Domain Engineering (based on [FP96])*

3.9 Discussion

Domain Engineering represents a valuable approach to software reuse and multi-system-scope engineering. Table 6 compares conventional software engineering and Domain Engineering based on their workproducts. Another important difference is the split of software engineering into *engineering for reuse* (Domain Engineering) and *engineering with reuse* (Application Engineering).

Domain Engineering moves the focus from code reuse to reuse of analysis and design models. It also provides us with a useful terminology for talking about reuse-based software engineering.

Software Engineering	**Domain Engineering**
Requirements Analysis ↷ requirements for one system	*Domain Analysis* ↷ reusable requirements for a class of systems
System Design ↷ design of one system	*Domain Design* ↷ reusable design for a class of systems
System Implementation ↷ implemented system	*Domain Implementation* ↷ reusable components, infrastructure, and production process

Table 6 *Comparison between conventional software engineering and Domain Engineering*

Based on the discussion of the various Domain Engineering methods, we arrive at the following conclusions:

1. The described methods and approaches are quite similar regarding the process. They use slightly different terminology and different groupings of activities, but they are, to a large degree, compatible with the generic process described in Sections 3.3 through 3.5. This process is also well exemplified by MBSE and ODM (Sections 3.7.1 and 3.7.2).

2. However, as the overall process framework remains quite stable, significant variations regarding the concrete modeling techniques and notations, approaches to software architecture, and component implementation techniques are possible. In particular, questions regarding the relationship between Domain Engineering and object-oriented technology are interesting. How do OOA/D and Domain Engineering fit together? We will address this topic in Chapter 4. Furthermore, we need to look for adequate technologies for implementing domain models. We will discuss some implementation technologies in Chapter 6 and Chapter 7.

3. Some of the presented Domain Engineering approaches make specific contributions with respect to the issue of concrete modeling techniques and implementation technologies:

   - As exemplified by Draco and the algebraic approach (Sections 3.7.3 and 3.7.7), formal methods, formal domain-specific languages, and transformation systems are well suited for mature and narrow domains. More work is needed in order to investigate a broader scope of applicability of these techniques.

   - The importance of informal techniques has also been recognized, e.g. the application of hypermedia systems for recording requirements, rationale, and informal expertise (Capture, Section 3.7.4), and the utilization of lexical analysis for domain vocabulary extraction (DARE, Section 3.7.5).

4. Which modeling techniques are most appropriate depends on the kind of the domain. For example, important aspects of GUI-based applications are captured by use cases and scenarios, whereas in scientific computing, algorithms are best captured using pseudocode. Furthermore, if the applications have some special properties, such as real-time aspects or distribution aspects, we need to apply additional, specialized modeling techniques. The organization of the Domain Engineering process itself depends on the organization and its business objectives. Thus, there will not be one Domain Engineering method appropriate for all possible domains and organizations. We will rather have specialized methods for different kinds of domains with tailorable processes for different organizational needs.

5. A major problem of all the existing Domain Engineering methods is that they do not address the evolution aspect of Domain Engineering. In [Ara89], Arango emphasized that Domain Engineering is a continuous learning process, in which each new experience in building new applications based on the reusable models produced in the Domain Engineering process is fed back into the process, resulting in the adjustment of the reusable models. None of the existing methods properly addresses these issues. They rather address only one full Domain-Engineering cycle and do not explain how to organize an efficient iterative process. The aspect of learning is usually treated as part of the reuse infrastructure, i.e. the results of using an asset should be fed back into the asset base. But since the reuse infrastructure is a product of Domain Engineering, its feed-back aspect is detached from the Domain Engineering process itself.

6. The methods also do not address how and when the infrastructure and the application production process are planned, designed, and implemented (only to include an infrastructure implementation activity in Domain Implementation is clearly insufficient).

3.10 References

[ADAGE] WWW home page of the DSSA/ADAGE project at http://www.owego.com/dssa/

[Arch] WWW home page of the Architecture Resources Guide at http://www-ast.tds-gn.lmco.com/arch/guide.html

[Ara88] G. Arango. Domain Engineering for Software Reuse. Ph.D. Dissertation, Department Information and Computer Science, University of California, Irvine, California, 1988

[Ara89] G. Arango. Domain Analysis: From Art Form to Engineering Discipline. In *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 3, May 1989, pp. 152-159

[Ara94] G. Arango. Domain Analysis Methods. In *Software Reusability,* Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, 1994, pp. 17-49

[Bai92] S. Bailin. KAPTUR: A Tool for the Preservation and Use of Engineering Legacy. CTA Inc., Rockville, Maryland, 1992, http://www-ast.tds-gn.lmco.com/arch/kaptur.html

[Bai93] S. Bailin. Domain Analysis with KAPTUR. In *Tutorials of TRI-Ada'93*, Vol. I, ACM, New York, New York, September 1993

[Bai97] S. Bailin. Applying Multi-Media to the Reuse of Design Knowledge. In the *Proceedings of the Eighth Annual Workshop on Software Reuse*, 1997, http://www.umcs.maine.edu/~ftp/wisr/wisr8/papers.html

[BC96] L. Brownsword and P. Clements. A Case Study in Successful Product Line Development. Technical Report, SEI-96-TR-016, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, October, 1996 , http://www.sei.cmu.edu

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998

[BCR94] V. Basili, G. Caldiera and D. Rombach, The Experience Factory. In *Encyclopedia of Software Engineering*, Wiley, 1994, ftp://ftp.cs.umd.edu/pub/sel/papers/fact.ps.Z

[BEJV93] P. Binns, M. Englehart, M. Jackson, and S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation, and Control, Honeywell Technology Center, 1993, http://www-ast.tds-gn.lmco.com/arch/dssa.html

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. Wiley, Chichester, UK, 1996

[BP89] T. Biggerstaff and A. Perlis. *Software Reusability. Volume I: Concepts and Models*. ACM Press, Frontier Series, Addison-Wesley, Reading, 1989

[Buc97] S. Buckingham Shum. Negotiating the Construction and Reconstruction of Organisational Memories. In *Journal of Universal Computer Science*, Special Issue on IT for Knowledge Management, vol. 3, no. 8, 1997, pp. 899-928, http://www.iicm.edu/jucs_3_8/negotiating_the_construction_and/, also see http://kmi.open.ac.uk/~simonb/DR.html

[BW85] M. Barr and C. Wells. Toposes, triples and theories. Grundlagen der mathematischen Wissenschaften, vol. 278, Springer-Verlag, New York, New York, 1985

[CARDS] WWW home page of the Comprehensive Approach to Reusable Defense Software (CARDS) Program at http://www.cards.com

[CARDS94] Software Technology For Adaptable, Reliable Systems (STARS). Domain Engineering Methods and Tools Handbook: Volume I — Methods: Comprehensive Approach to Reusable Defense Software (CARDS). STARS Informal Technical Report, STARS-VC-K017R1/001/00, December 31, 1994, http://nsdir.cards.com/libraries/HTML/CARDS_documents.html

[Cle96] P. Clements. A Survey of Architecture Description Languages. In *Proceedings of Eighth International Workshop on Software Specification and Design*, Paderborn, Germany, March 1996

[CK95] P. Clements and P. Kogut. Features of Architecture Description Languages. In *Proceedings of the 7th Annual Software Technology Conference*, Salt Lake City UT, April 1995

[Con97] E. Conklin. Designing Organizational Memory: Preserving Intellectual Assets in a Knowledge Economy. Technical Note, Group Decision Support Systems, http://www.gdss.com/DOM.htm, 1997

[CSJ+92] S. Cohen, J. Stanley, S. Peterson, and R. Krut. Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain. Technical Report, CMU/SEI-91-TR-28, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992, http://www.sei.cmu.edu

[CT93] L. Coglianese and W. Tracz. Architecture-Based Development Guidelines for Avionics Software. Version 2.1, Technical Report, ADAGE-IBM-92-03, 1993

[CY90] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990

[DD87] H. Dreyfus and S. Dreyfus. *Künstliche Intelligenz: von den Grenzen der Denkmaschine und dem Wert der Intuition*. Rowohlt, rororo Computer 8144, Reinbek, 1987

[Dict] *The American Heritage Dictionary of the English Language*. Third Edition, Houghton Mifflin Company, 1992

[Dij70] E. Dijkstra. Structured Programming. In *Software Engineering Techniques*, J. Buxton and B. Randell, (Eds.), NATO Scientific Affairs Division, Brussels, Belgium, 1979, pp. 84-87

[DISA93] DISA/CIM Software Reuse Program. Domain Analysis and Design Process, Version 1. Technical Report 1222-04-210/30.1, DISA Center for Information Management, Arlington Virginia, March 1993

[DP98] P. Devanbu and J. Poulin, (Eds.). *Proceedings of the Fifth International Conference on Software Reuse (Victoria, Canada, June 1998)*. IEEE Computer Society Press, 1998

[EP98] H.-E. Eriksson and M. Penker. *UML Toolkit*. John Wiley & Sons, 1998

[FP96] B. Frakes and R. Prieto-Díaz. Introduction to Domain Analysis and Domain Engineering. Tutorial Notes, The Fourth International Conference on Software Reuse, Orlando, Florida, April 23-26, 1996

[FPF96] W. Frakes, R. Prieto-Díaz, and Christopher Fox. DARE: Domain Analysis and Reuse Environment. Draft submitted for publication, April 7, 1996

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[GKS+94] H. Gomaa, L. Kerschberg, V. Sugumaran, C. Bosch, and I. Tavakoli. A Prototype Domain Modeling Environment for Reusable Software Architectures. In *Proceedings of the Third International Conference on Software Reuse*, Rio de Janeiro, Brazil, W. Frakes (Ed.), IEEE Computer Society Press, Los Alamitos, California, 1994, pp. 74-83

[Gog83] J. Goguen. LIL – A Library Interconnection Language. In Report on Program Libraries Workshop, SRI International, Menlo Park, California, October 1983, pp. 12-51

[Gom92] H. Gomaa. An Object-Oriented Domain Analysis and Modeling Method for Software Reuse. In *Proceedings of the Hawaii International Conference on System Sciences*, Hawaii, January 1992

[Gon81] L. Gonzales. A domain language for processing standardized tests. Master's thesis, Department of Information and Computer Science, University of California, 1981

[GT96] J. Goguen and W. Tracz. An Implementation-Oriented Semantics for Module Composition. Draft available from [ADAGE], 1996

[Hay94] F. Hayes-Roth. Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program. Version 1.01, Informal Technical Report, Teknowledge Federal Systems, February 4, 1994, available from [ADAGE]

[HNC+90] J. Hess, W. Novak, P. Carroll, S. Cohen, R. Holibaugh, K. Kang, and A. Peterson. A Domain Analysis Bibliography. Technical Report, CMU/SEI-90-SR-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1990. Reprinted in [PA91], pp. 258-259. Also available from http://www.sei.cmu.edu

[Hol93] R. Holibaugh. Joint Integrated Avionics Working Group (JIAWG) Object-Oriented Domain Analysis Method (JODA). Version 1.3, Technical Report, CMU/SEI-92-SR-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1993, http://www.sei.cmu.edu

[HT94] F. Hayes-Roth and W. Tracz. DSSA Tool Requirements For Key Process Functions. Version 2.0, Technical Report, ADAGE-IBM-93-13B, October 24, 1994, available from [ADAGE]

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990

[Kru93] R. Krut. Integrating 001 Tool Support into the Feature-Oriented Domain Analysis Methodology. Technical Report, CMU/SEI-93-TR-11, ESC-TR-93-188, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1993, http://www.sei.cmu.edu

[Kic97] G. Kiczales. Verbal Excerpt from the ECOOP'97 tutorial on "Designing High-Performance Reusable Code", Jyväskylä, Finland, 1997

[LEW96] J. Loeckx, H. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley & Teubner, 1996

[LHK87] D. Luckham, F. von Henke, B. Krieg-Brückner, and O. Owe. *Anna: A Language For Annotating Ada Programms. Language Reference Manual*. Lecture Notes in Computer Science, no. 260, Springer-Verlag, 1987

[LKA+95] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. In *IEEE Transaction on Software Engineering*, vol. 21, no. 4, April 1995, pp. 336-355

[Lub91] M. Lubars. Domain Analysis and Domain Engineering in IDeA. In [PA91], pp. 163-178

[MBSE97] Software Engineering Institute. Model-Based Software Engineering. WWW pages, URL: http://www.sei.cmu.edu/technology/mbse/, 1997 (viewed)

[MC96] T. Moran and J. Carroll, (Eds.). *Design Rationale: Concepts, techniques, and use*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1996

[Nei80] J. Neighbors. Software construction using components. Ph.D. dissertation, (Tech. Rep. TR-160), Department Information and Computer Science, University of California, Irvine, 1980

[Nei84] J. Neighbors. The Draco Approach to Construction Software from Reusable Components. In *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, September 1984, pp. 564-573

[Nei89] J. Neighbors. Draco: A Method for Engineering Reusable Software Systems. In [BP89], pp. 295-319

[OPB92] E. Ostertag, R. Prieto-Díaz, and C. Braun. Computing Similarity in a Reuse Library System: An AI-Based Approach. In *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 3, July 1992, pp. 205-228

[PA91] R. Prieto-Diaz and G. Arango (Eds.). *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, California, 1991

[Par76] D. Parnas. On the design and development of program families. In *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, 1976, pp. 1-9

[PC91] S. Peterson and S. Cohen. A Context Analysis of the Movement Control Domain for the Army Tactical Command and Control System. Technical Report, CMU/SEI-91-SR-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1991

[PF87] R. Prieto-Díaz and P. Freeman. Classifying Software for Reusability. In *IEEE Software*, January 1987, pp. 6-16

[Pri85] R. Prieto-Díaz. A Software Classification Scheme. Ph.D. Dissertation, Department of Information and Computer Science, University of California, Irvine, 1985

[Pri87] R. Prieto-Díaz. Domain Analysis For Reusability. In Proceedings of COMPSAC'87, 1987, pp. 23-29 and reprinted in [PA91], pp. 63-69

[Pri91a] R. Prieto-Díaz. Implementing Faceted Classification for Software Reuse. In *Communications of the ACM*, vol. 34, no. 5, May 1991, pp. 88-97

[Pri91b] R. Prieto-Díaz. Reuse Library Process Model. Technical Report, IBM STARS 03041-002, Electronic Systems Division, Air Force Systems Command, USAF, Hanscom Air Force Base, Hanscom, Massachusetts, July, 1991

[SC93] M. Simos and R.E. Creps. Organization Domain Modeling (ODM), Vol. I - Conceptual Foundations, Process and Workproduct Descriptions. Version 0.5, Unisys STARS Technical Report No. STARS-UC-05156/024/00, STARS Technology Center, Arlington, Virginia, 1993

[SCK+96] M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang. Organization Domain Modeling (ODM) Guidebook, Version 2.0. Informal Technical Report for STARS, STARS-VC-A025/001/00, June 14, 1996, available from http://direct.asset.com

[Sha77] D. Shapere. Scientific Theories and Their Domains. In *The Structure of Scientific Theories*, F. Suppe (Ed.), University of Illinois Press, 1977, pp. 519-565

[Shu91] S. Shum, Cognitive Dimensions of Design Rationale. In *People and Computers VI: Proceedings of HCI'91*, D. Diaper and N. Hammond, (Eds.), Cambridge University Press, Cambridge, 1991, pp. 331-344, http://kmi.open.ac.uk/~simonb/DR.html

[Sim91] M. Simos. The Growing of an Organon: A Hybrid Knowledge-Based Technology for Software Reuse. In [PA91], pp. 204-221

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on a Emerging Discipline*. Prentice-Hall, 1996

[SJ95] Y. Srinivas and R. Jüllig. Specware^{TM}: Formal Support for Composing Software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, (Ed.), Lecture Notes in Computer Science, vol. 947, Springer-Verlag, Berlin, 1995, also http://www.kestrel.edu/

[Smi90] D. Smith. KIDS: A Semiautomatic Program Development System. In *IEEE Transactions on Software Engineering*, vol. 16, no. 9, September 1990, pp. 1024-1043

[Smi96] D. Smith. Toward a Classification Approach to Design. In *Proceedings of Algebraic Methodology & Software Technology, AMAST'96*, Munich, Germany, July 1996, M. Wirsing and M. Nivat (Eds.), LCNS 1101, Springer, 1996, pp. 62-84

[SPC92] Software Productivity Consortium. Reuse Adoption Guidebook. Technical Report, SPC-92051-CMC, Software Productivity Consortium, Herndon, Virginia, 1992, http://www.asset.com

[SPC93] Software Productivity Consortium. Reuse-Driven Software Processes Guidebook. Version 02.00.03, Technical Report, SPC-92019-CMC, Software Productivity Consortium, Herndon, Virginia, November 1993, http://www.asset.com

[SPW95] D. Smith, E. Parra, and S. Westfold. Synthesis of High-Performance Transportation Schedulers. Technical Report, KES.U.1, Kestrel Institute, February 1995, http://www.kestrel.edu/

[Sri91] Y. Srinivas. Algebraic specification for domains. In [PA91], pp. 90-124

[STARS94] Software Technology For Adaptable, Reliable Systems (STARS). Army STARS Demonstration Project Experience Report. STARS Informal Technical Report, STARS-VC-A011R/002/01, November 30, 1994

[Sun83] S. Sundfor. Draco domain analysis for real time application: The analysis. Technical Report, RTP 015, Department of Information and Computer Science, University of California, Irvine, 1983

[TC92] W. Tracz and L. Coglianese. DSSA Engineering Process Guidelines. Technical Report, ADAGE-IBM-9202, IBM Federal Systems Company, December 1992

[TH93] D. Tansley and C. Hayball. *Knowledge-Based Systems Analysis an Design: A KADS Developer's Handbook*. Prentice Hall, 1993

[THE+94] A. Terry, F. Hayes-Roth, L. Erman, N. Coleman, M. Devito, G. Papanagopoulos, B. Hayes-Roth. Overview of Teknowledge's Domain-Specific Software Architecture Program. In *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 4, October 1994, pp. 68-76, see http://www.teknowledge.com/DSSA/

[Tra93] W. Tracz. Parameterized programming in LILEANA. In *Proceedings of ACM Symposium on Applied Computing, SAC'93*, February 1993, pp. 77-86

[Tra95] W. Tracz. Domain-Specific Software Architecture Pedagogical Example. In *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4, July 1995, pp. 49-62, also available from [ADAGE]]

[TTC95] R. Taylor, W. Tracz, and L. Coglianese. Software Development Using Domain-Specific Software Architectures: CDRL A011 – A Curriculum Module in the SEI Style. In *ACM SIGSOFT*

Software Engineering Notes, vol. 20, no. 5, December 1995, pp. 27-37, also available from [ADAGE]

[UML97a] Rational Software Corporation. UML (Unified Modeling Language) Glossary. Version 1.0 1, 1997, http://www.rational.com

[Uni88] Unisys. Reusability Library Framework AdaKNET and AdaTAU Design Report. Technical Report, PAO D4705-CV-880601-1, Unisys Defense Systems, System Development Group, Paoli, Pennsylvania, 1988

[VAM+98] A. D. Vici, N. Argentieri, A. Mansour, M. d'Alessandro, and J. Favaro. FODAcom: An Experience with Domain Analysis in the Italian Telecom Industry. In [DP98], pp. 166-175, see http://www.intecs.it

[VG90] W. Vitaletti and E. Guerrieri. Domain Analysis within the ISEC Rapid Center. In *Proceedings of Eighth Annual National Conference on Ada Technology*, March 1990

[Wei96] D. Weiss. Creating Domain-Specific Languages: The FAST Process. Transparencies presented at The first ACM-SIGPLAN Workshop on Domain-Specific Languages, Paris, France, January 18, 1997, http://www-sal.cs.uiuc.edu/~kamin/dsl/index.html

[Wit94] J. Withey. Implementing Model Based Software Engineering in your Organization: An Approach to Domain Engineering. Draft, Technical Report, CMU/SEI-94-TR-01, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1994

[Wit96] J. Withey. Investment Analysis of Software Assets for Product Lines. Technical Report, CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1996, http://www.sei.cmu.edu

[WP92] S. Wartik and R. Prieto-Díaz. Criteria for Comparing Domain Analysis Approaches. In *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 3, September 1992, pp. 403-431

[Zal96] N. Zalman. Making The Method Fit: An Industrial Experience in Adopting Feature-Oriented Domain Analysis (FODA). In *Proceedings of the Fourth International Conference on Software Reuse*, M. Sitaraman, (Ed.), IEEE Computer Society Press, Los Alamitos, California, 1996, pp. 233-235

Chapter 4 Domain Engineering and Object-Oriented Analysis and Design

4.1 OO Technology and Reuse

In the early days of OO, there used to be the believe that objects are reusable by their very nature and that reusable OO software simply “falls out” as a byproduct of application development (see e.g. [HC91]). Today, the OO community widely recognizes that nothing could be further from the truth. Reusable OO software has to be carefully engineered and engineering *for reuse* requires a substantial investment (see e.g. [GR95]).

So how does today’s OO technology address reuse? As Cohen and Northrop [CN98] suggest, we should answer this question from two perspectives: the problem space and the solution space perspective.

4.1.1 Solution Space

Two important areas of OO technology connected to the solution space and addressing multi-system-scope development are frameworks and design patterns. A framework embodies an abstract design for a *family* of related systems in the form of collaborating classes. Similarly, design patterns provide reusable solutions to recurring design problems across different systems. Patterns, as a documentation form, also proved useful in capturing reusable solutions in other areas such as analysis [Fow97], architecture [BMR+96], and organizational issues [Cop95].

Unfortunately, two main problem areas still remain:

- Only very few OO Analysis and Design (OOA/D) methods provide any support for the development of frameworks. Similarly, there is little systematic support in both finding and applying patterns. Since these issues are related to the analysis of the problem space, we will discuss them in the following section.

- A major problem of the current framework technology is the excessive complexity explosion and performance degradation as the generality of a framework increases. For example, by applying the classic design patterns collected in [GHJV95], we are able to add new variation points to a framework. However, this causes an excessive fragmentation of the design resulting in “many little methods and classes.” Also, framework technology heavily relies on dynamic binding, even for implementing variability between applications, in which case static binding and partial evaluation are more appropriate. This causes unnecessary performance penalties and unused code to remain in the delivered applications. Furthermore, current OO languages do not allow us to adequately separate and capture important aspects such as synchronization, remote communication, memory management, etc. All these problems also apply to strictly hierarchically composable components such as ActiveX or JavaBeans (see Section 6.4.2.8). Addressing these problems requires a combination of techniques such as new linguistic constructs, new composition mechanisms, metaprogramming capabilities, etc. We will discuss these issues in Chapter 6 and Chapter 7.

4.1.2 Problem Space

Traditional OOA/D methods such as OOSE [JCJO92], Booch [Boo94], OMT [RBP91], or even the current version 4.1 of the Rational Objectory Process [Rat98a, Rat98b], focus on developing single systems rather than families of systems.[30] Given this goal, the methods are inadequate for developing reusable software, which requires focusing on classes of systems rather than single systems.

In this context, traditional OOA/D methods have the following deficiencies (in terms of process and modeling notation):

- *No distinction between engineering for reuse and engineering with reuse*: Taking reuse into account requires splitting the OO software engineering process into engineering *for reuse* (i.e. Domain Engineering) and engineering *with reuse* (i.e. Application Engineering). OOA/D methods come closest to Application Engineering, with the important difference that Application Engineering focuses on reusing available reusable assets produced during Domain Engineering.

- *No domain scoping phase*: Since OOA/D methods focus on engineering single systems, they lack a domain scoping phase, where the target class of systems is selected. Also, OOA/D focuses on satisfying "the customer" of a single system rather than analyzing and satisfying *stakeholders* (including potential customers) of a class of systems.

- *Inadequate modeling of variability*: The only kind of variability modeled in current OOA/D is intra-application variability, e.g. variability of certain objects over time and the use of different variants of an object at different locations within an application. Domain Engineering, on the other hand, focuses on variability across different systems in a domain for different users and usage contexts. Since modeling variability is fundamental to Domain Engineering, Domain Engineering methods provide specialized notations for expressing variability.

In contrast to the traditional OOA/D methods, there are few newer methods, such as OOram [Ree96] and Catalysis [DW97], which explicitly support modeling of frameworks and the application of design patterns. The contribution of OOram to framework modeling is the recognition that the fundamental abstractions of object-oriented designs are not classes but *collaborations*. A collaboration consists of a number of *roles* communicating according to a certain pattern. Concrete object may play more than one role in one or more collaborations. Thus, classes are merely synthesized by composing collaborations. Modeling a framework as a composition of collaborations is more adequate than modeling it as a composition of classes since extending a framework usually requires the coordinated extension of more than one class.

Collaborations and roles

As of writing, OOram is the only OOA/D method which truly recognizes the need for a specialized engineering process for reuse. The method includes a domain scoping activity based on the analysis of different classes of consumers. It also includes an analysis of existing systems. We will discuss OOram in Section 4.5.2.

A general problem of all OOA/D methods is inadequate modeling of variability. Although the various modeling techniques used in OOA/D methods support variability mechanisms (e.g. inheritance and parameterization in object diagrams, composition of collaboration diagrams, etc.), OOA/D methods do not include an abstract and concise model of commonality, variability, and dependencies. There are several reasons for providing such a model:

- Since the same variability may be implemented using different variability mechanisms in different models, we need a more abstract representation of variability (cf. Sections 4.5.3 and 5.4.1.7).

- The reuser of reusable software needs an explicit and concise representation of available features and variability.

- The developer of reusable software needs to be able to answer the question: why is a certain feature or variation point included in the reusable software?

The lack of domain scoping and explicit variability modeling may cause two serious problems:

- relevant features and variation points are missing;

- many features and variation points are included but never used; this causes unnecessary complexity and cost (both development and maintenance cost).

Covering the right features and variation points requires a careful balancing between current and future needs. Thus, we need an explicit model that summarizes the features and the variation points and includes the rationale and the stakeholders for each of them. In Domain Engineering, this role is played by a *feature model* (see Sections 3.3 and 5.4). A feature model captures the reusability and configurability aspect of reusable software.

4.2 Relationship Between Domain Engineering and Object-Oriented Analysis and Design (OOA/D) Methods

As we discussed in Chapter 3, Domain Engineering focuses on engineering solutions for classes of software systems. On the other hand, current OOA/D methods focus on engineering single systems. Because of this different focus, we concluded in the previous section that current OOA/D methods are inappropriate for developing reusable software.

While Domain Engineering supports a multi-system-scope engineering process and adequate variability modeling techniques, OOA/D methods provide us with very effective system modeling techniques. Thus, Domain Engineering Methods and OOA/D methods are good candidates for integration. Indeed, this integration represents a recent focus of the Domain Engineering community [CN98].

4.3 Aspects of Integrating Domain Engineering and OOA/D Methods

When integrating development methods, we have to consider a number of specific areas that need to be integrated:

- *method goals*, e.g. Domain Engineering methods aim at supporting the development of models for classes of systems, whereas OOA/D concentrate on single systems;

- *principles*, e.g. Domain Analysis also investigates alternative implementation strategies (in order to provide the terminology and scope for further phases), whereas OOA avoids dealing with implementation issues;

- *processes*, e.g. Domain Engineering covers engineering for reuse and Application Engineering covers engineering with reuse; the distinction between engineering for reuse and with reuse is not present in most OOA/D methods;

- *models and notations*, e.g. Domain Engineering introduces new kinds of models such as feature models, whereas OOA/D provides the necessary system modeling techniques.

Integration of methods is usually a very complex task since today's methods are also complex. Additionally, it is a long and costly process since the integrated methods can only be tested and improved while being applied on real projects. A formal treatment of method integration issues can be found in [Son97].

Since OOA/D methods are currently more widely used than Domain Engineering, we first take a look at the required changes to OOA/D methods:

- *Process changes*: The required process changes include introducing separate processes for engineering for reuse (Domain Engineering) and engineering with reuse (Application Engineering). The Domain Engineering process may have a complex structure including an Application Family Engineering process and multiple horizontal Domain Engineering processes (see Section 4.4). The Application Engineering process is often quite similar to a conventional OOA/D process with the main difference that it concentrates on developing solutions in terms of the available reusable assets. The Domain Engineering processes, on the other hand, have to additionally include domain scoping and feature modeling activities.

- *Variability and dependency modeling*: Variability and dependency modeling lies at the heart of Domain Engineering. Variability is represented in different models at different stages in the development process. Variability modeling usually starts at the taxonomic level by developing the vocabulary to describe different instances of concepts. For example, we usually first talk about different kinds of banking accounts (e.g. savings account or checking account) before we build the object model of banking accounts. Feature models allow us to capture this taxonomic level and to provide a roadmap to variability in other models (e.g. object models, use case models, collaboration and interaction models, etc.). They are a necessary extension of the set of models currently used in OO software engineering. Capturing dependencies between features is also essential. It allows us to perform automatic configuration (e.g. constraint-based configuration), which relieves the reuser of some of the manual configuration work.

- *Development of a reuse infrastructure*: In addition to developing the reusable assets, we also have to develop and install a reuse infrastructure for packaging, storing, distributing, retrieving, evaluating, and integrating these assets.

In Chapter 7, we will also discuss the need of extending reuse-oriented software engineering methods with approaches for developing and using language extensions for different system aspects.

As already stated, the integration between Domain Engineering and OOA/D methods represents a recent focus of the Domain Engineering community. We can classify the integration efforts into three categories:

- *Upgrading older Domain Engineering methods*: Older Domain Engineering methods, such as FODA (see Section 3.7.1), used techniques of structured analysis and design as their system engineering method. Recent work concentrates on replacing these older techniques with newer OOA/D techniques, e.g. work on FODAcom [VAM+98], an OO upgrade of FODA specialized for the telecom domain.

- *Specializing customizable Domain Engineering methods*: Newer Domain Engineering methods such as ODM (see Section 3.7.2) treat system engineering methods as their parameters. Thus, before applying a parameterized Domain Engineering method, it needs to be specialized for some concrete system engineering method. However, such specialization may represent a substantial effort. As noted in [Sim97], ODM has been specialized for the use with the OOA/D method Fusion [CAB+94]. Another example of such an effort is Domain Engineering Method for Reusable Algorithmic Libraries (DEMRAL), which is described in Chapter 9.[31]

- *Extending existing OOA/D methods*: The third approach is to extend one of the existing OOA/D methods with the concepts of Domain Engineering. An example of such effort is Reuse-driven Software Engineering Business (RSEB) [JGJ97], which is based on the OO modeling notation UML [Rat98c], and the OO Software Engineering (OOSE) method

[JCJO92]. Also, the version 4.1 of Rational Objectory Process [Rat98a, Rat98b] (the *de facto* standard UML-based OO engineering process) includes some Domain Engineering concepts. However, these minor extensions in version 4.1 do not remove the inadequacy of this version for engineering reusable software. As stated, upgrading an OO system engineering method for Domain Engineering requires substantial changes in its process architecture. Additionally, the modeling notations have to be extended for modeling variability (e.g. feature diagrams and variation points).

- *Second generation integration*: Finally, FeatuRSEB [GFA98] is an example of the integration of two methods which already combine Domain Engineering and OOA/D concepts: FODAcom and the RSEB method, which we mentioned in the previous bullets. One of the weaknesses in the original description of RSEB was the lack of variability modeling using feature models. The integration with FODAcom concepts addresses this problem. On the other hand RSEB has a stronger OO focus than FODAcom. Thus, both methods profit from this integration.

We describe these integration efforts in the rest of this chapter.

4.4 Horizontal vs. Vertical Methods

In Section 3.6.2, we introduced the notions of vertical and horizontal domains. Horizontal domains encompass only one system part, e.g. GUIs, database systems, middleware, matrix computation libraries, container libraries, frameworks of financial objects, etc. Vertical domains, on the other hand, cover complete systems, e.g. flight reservation systems, medical information systems, CAD systems, etc. Obviously, different kinds of domains require different Domain Engineering methods. Organizations specializing in one or more horizontal domains would use specialized methods for these domains. We refer to such Domain Engineering methods as *horizontal*. An example of a horizontal Domain Engineering method is DEMRAL (see Section 4.5.5). In the case of vertical domains, we have to develop and maintain the overall reusable architecture for the entire system scope and apply the specialized horizontal methods in order to develop reusable models of the subsystems. We refer to a Domain Engineering method covering a vertical domain as a *vertical* Domain Engineering method. An example of a vertical engineering method is RSEB, which we discuss in Section 4.5.3. In general, we want to develop modular Domain Engineering methods, so that they can be configured to suit the specific needs of different organizations. One way to achieving this goal is to have a vertical method to call different specialized horizontal methods for different subsystems.

There may be significant differences between domains and thus between Domain Engineering methods. One of the major methodical differences is the modeling style:

- *Interaction style*: The main aspect of the interaction style is the interaction between entities, e.g. interaction between components, call graphs, message flows, event flows. Interaction can be modeled using use cases, collaborations, and interaction diagrams.

- *Algorithmic style*: The main aspect of the algorithmic style are algorithms performing complex computations on abstract data types.[32] Algorithms can be specified using pseudocode or some specialized specification notations.

- *Data-centric style*: The main aspect of the data-centric style is the structure of the data e.g. in database modeling. The structure of the data can be modeled using entity-relationship or object diagrams.

- *Data-flow style*: The main aspect of the data-flow style is data flow, e.g. in pipes-and-filters architectures in signal processing. Data flow can be specified using data-flow diagrams.

Of course, we often need to capture all of these fundamental aspects, namely interaction, algorithms, data structures, and data flow, for a single system part. However, it is also often the case that one of these aspects plays a dominant, unifying role. For example, most business applications have the interactive nature since they usually have an interactive GUI and are organized as interacting components in a distributed and open environment. Furthermore, the interaction aspect is also dominant in large technical systems (e.g. CAD systems). Indeed, the interaction aspect plays an important role in all large systems since the subsystems of large systems are glued by interaction implemented using procedure call, message passing, event notification, etc. This is also the reason why most of the modern OOA/D methods are use case and scenario centric, e.g. Rational Objectory Process (see Section 4.5.1). Similarly, the vertical Domain Engineering method RSEB (see Section 4.5.3) aimed at developing large systems is also use case centric. On the other hand, some specialized horizontal Domain Engineering methods may not be use case centric, e.g. DEMRAL, which is specialized for algorithmic libraries. As noted above, a vertical Domain Engineering method calls specialized horizontal Domain Engineering methods. Thus, it is possible that use cases are applied at the system level, but not at each subsystem level.

Moreover, each domain may have some special properties requiring special modeling techniques, e.g. real-time support, distribution, concurrency, etc. Thus, Domain Engineering methods have to support a variety of specialized modeling techniques for these different aspects.

4.5 Selected Methods

4.5.1 Rational Objectory Process 4.1

Rational Objectory Process 4.1 is a *de facto* standard UML-based OO software system engineering process marketed by Rational Software Corporation [Rat98a, Rat98b]. The process originated from the OO Software Engineering method by Jacobson et al. [JCJO92]. Objectory's goal is "to ensure the production of high-quality software, meeting the needs of its end-users, within a predictable schedule and budget." Objectory is an *iterative* and *use-case-centric* process, which is a prerequisite for the successful development of large software systems. Use cases are used as the integrating elements of the whole system under construction and across all development phases and models.[33] This view has been propagated in the 4+1 View Model of software architecture by Kruchten (see Figure 17).

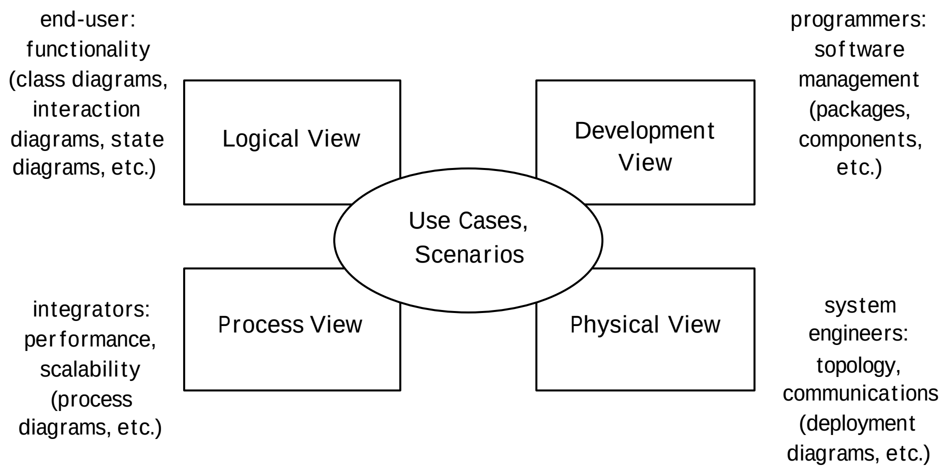

end-user: functionality (class diagrams, interaction diagrams, state diagrams, etc.)
Logical View
Development View
programmers: software management (packages, components, etc.)
Use Cases, Scenarios
integrators: performance, scalability (process diagrams, etc.)
Process View
Physical View
system engineers: topology, communications (deployment diagrams, etc.)

Figure 17 *The 4+1 View Model of the architecture of a software-intensive system (adapted from [Kru95])*

The Objectory process is organized along two dimensions:

- *time dimension* representing the life-cycle aspect of the process over time;
- *process components dimension* grouping activities logically by nature.

The time dimension is organized into cycles, phases, and iterations separated by milestones. A cycle is one complete pass through all phases. Process components are described in terms of activities, workflows organizing the activities, produced artifacts, and workers. The overview of the process is shown in Figure 18.

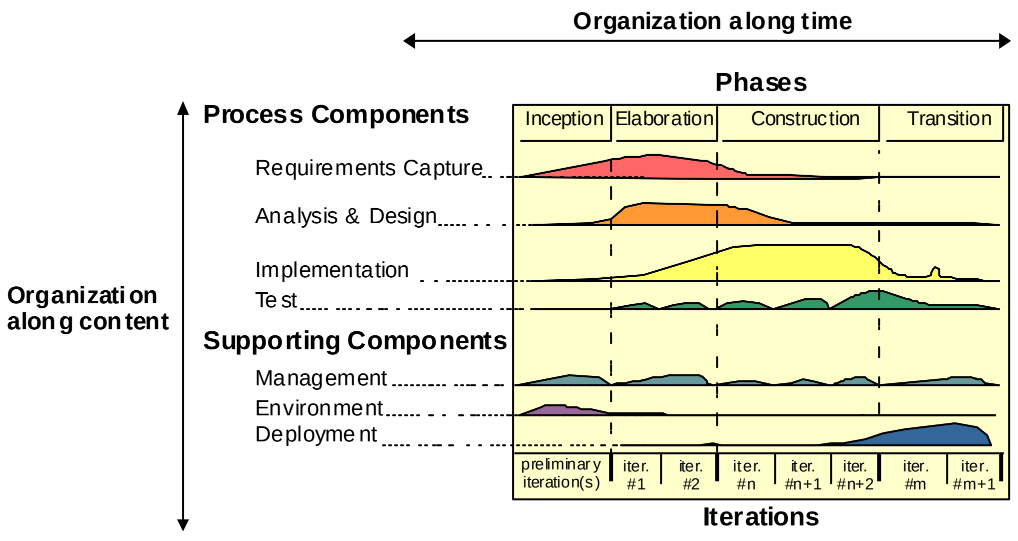


Figure 18 *Objectory process phases and components (from [Rat98a])*

By its definition, Objectory focuses on the development of a single system. Thus, it is inadequate for engineering reusable software due to the deficiencies we discussed in previous sections:

- no distinction between engineering for reuse and engineering with reuse;
- no domain scoping and multi-system-scope stakeholder analysis;
- no feature analysis activities;
- no feature models.

In the description of the activity *Capture a Common Vocabulary* (which is part of the *Requirements Capture* process component [Rat98b]), the authors of Objectory acknowledge the value of variability modeling by merely noting that "you can capture the vocabulary in a domain model." However, the documentation does not state what techniques should be used and how the domain model is to be integrated with the other artifacts produced in the process. Furthermore, Appendix A in the method description discusses the rationale for variant modeling and some variant modeling concepts on a few pages. However, the authors note that currently variant modeling is not part of the process. They further state that "developing variants, or families of systems, affects all process components" and that "modeling variants and variability is an area in which the Rational Objectory Process will improve and expand in the future."

4.5.2 OOram

OOram [Ree96] is a generic framework for creating a variety of OO methodologies based on role modeling. OOram has been developed by Trygve Reenskaug from Norway and it traces its history back to the 1970ies. For example, the early OOram ideas were applied in the Smalltalk Model-View-Controller paradigm, which was developed by Goldberg and Reenskaug at Xerox Palo Alto Research Center in 1997 [Ree96].

The central idea behind OOram is *role modeling*. Role modeling concentrates on *collaborations* of objects and the *roles* the objects play in collaborations rather than the classes of objects. Classes are merely synthesized by composing the roles their instances play in different collaborations. The role model is represented by a number of views, most notably the *collaboration view* showing the relationships between roles and the *scenario view* concentrating on the interactions. Although the original description in [Ree98] introduces its own notation, we can also use the UML notation for representing these views. In particular, collaboration diagrams can be used for the collaboration view and interaction diagrams for the scenario view.

While other OO methods, e.g. Objectory, also embrace the idea of scenario analysis, they concentrate on deriving concrete object classes from scenarios (e.g. start with use cases, derive collaboration and interaction diagrams, and finally the classes). In OOram, on the other hand, there is a focus on the *composition* of collaborations. New collaborations can be added to existing compositions as needed and the participating classes are updated accordingly. Thus, in OOram, collaborations are the primary building blocks, not classes.

Frameworks represent one possible target technology for OOram since framework designs are naturally represented as compositions of collaborations. Role modeling also has a strong relationship to design patterns since some collaborations are instances of design patterns. The relationship between role modeling, frameworks, and design patterns is currently an area of active research (e.g. [Rie97]). We will revisit this topic in Section 6.4.2.7.

OOram refrains from prescribing a comprehensive and elaborate process. Since a specific process has to be optimized to a specific combination of product, people, and work environments, OOram rather provides a framework of guidelines backed with a set of comprehensive case studies. In [Ree96], Reenskaug et al. note: “Many managers dream of the ultimate work process that will ensure satisfactory solutions from every project. We believe that this dream is not only futile: it can even be harmful.”

OOram distinguishes between three kinds of processes [Ree96]:

1. “The model creation process focuses on how to create a model or some other manifestation of thoughts for a certain phenomenon. Examples include creating a role model, performing role model synthesis, and creating object specifications.”

2. “The system development process covers the typical software life cycle, from specifying users needs, to the installation and maintenance of the system that meets these needs.”

3. “The reusable assets building process is the least mature software engineering process, but we expect it will be an essential contributor to future productivity and quality. Our focus is on the continuous production of several, closely related systems, in which we build on a continuously evolving set of reusable components. Creating a system mainly involves configuring and reusing robust and proven components, and possibly adding a few new components to complete the system.”

Thus, the need for a dedicated engineering-for-reuse process is one of the fundamental principles of OOram.

Reenskaug et al. also make the very important observation that the creation of reusable objects shares a lot properties with product development whose life cycle can be divided into five phases [Ree96]:

1. *Market analysis*: “The developer must understand the needs of the potential users, and balance these needs against the costs of alternative solutions. The developer must also understand the potential users’ working conditions to make the reusable component practically applicable.”

2. *Product development*: “The reusable component must be designed, implemented, and tested in one or more prototype applications.”

3. *Product packaging*: “Documentation is an important part of a packaged reusable component. The documentation includes work processes for the application of the component, installation procedures, and technical information.”

4. *Marketing*: “The users of the reusable component must be informed and persuaded to apply it.”

5. *Application*: “The reusable component must be applied, and must help its users to increase the quality of their products and reduce their expenditure of time and money.”

One of the four case studies in [Ree96] describes the process of developing an OO framework. The outline of this process is as follows:

- Step 1: Identify Consumers and Consumer Needs
- Step 2: Perform a Cost-benefit Analysis
- Step 3: Reverse Engineering of Existing Programs
- Step 4: Specify the New Framework
- Step 5: Document the Framework as Patterns Describing how to Solve Problems
- Step 6: Describe the Framework’s Design and Implementation
- Step 7: Inform the Consumer Community

Steps 1-3 represent some form of Domain Analysis with domain scoping, stakeholder analysis, and analysis of existing applications. Unfortunately, OOram does not include feature analysis and feature models. We will discuss the importance of the latter two in Chapter 5. In Step 5, the approach advocates the standard technique of documenting frameworks using patterns (see [Joh92, MCK97]).

4.5.3 Reuse-driven Software Engineering Business (RSEB)

RSEB [JGJ97] is a reuse- and object-oriented software engineering method based on the UML notation, the OO Software Engineering (OOSE) method by Jacobson et al. [JCJO92] (which is an early version of the Rational Objectory Process), and the OO Business Process Reengineering [JEJ94]. The method has been developed based on the experience from Hewlett-Packard Inc. (M. Griss) and Rational Software Corporation (I. Jacobson and P. Jonsson, formerly Objectory AB). It has been designed to facilitate both the development of reusable object-oriented software and software reuse. Similarly as Objectory, RSEB is an iterative and use-case-centric method.

Let us first introduce some RSEB terminology [JGJ97]:

- *Application system*: An application system in RSEB corresponds to a *software system* as we use it throughout this text. RSEB authors note that “we use the term application system instead of the looser term application because we want to stress that application systems are software system products and are defined by system models.”

- *Component*: “A component is a type, class, or any other workproduct (e.g. use case, analysis, design, or implementation model element) that has been specifically engineered to be reusable.” Thus, the RSEB definition of component corresponds to the term *reusable asset*, which we use throughout this text.

- *Component system*: A component system is a system product that offers a set of reusable features. Component systems are more generic, reusable, and specializable than application systems, but on the other hand require more effort to engineer. Examples of component systems are reusable GUI frameworks, reusable mathematical libraries, or more sophisticated component systems from which complete application systems can be generated. If we analyze a class of application systems and decompose it into generic subsystems, component systems provide reusable solutions for the subsystems.

RSEB has separate processes for engineering for reuse (i.e. Domain Engineering) and engineering with reuse (i.e. Application Engineering). Domain Engineering in RSEB consists of two processes [JGJ97]:

- *Application Family Engineering* process develops and maintains the overall layered system architecture.

- *Component System Engineering* process develops component systems for the different parts of the application system with focus on building and packaging robust, extendible, and flexible components.

The role of Application Engineering (i.e. the process of building concrete systems based on reusable assets) in RSEB plays *Application System Engineering*.

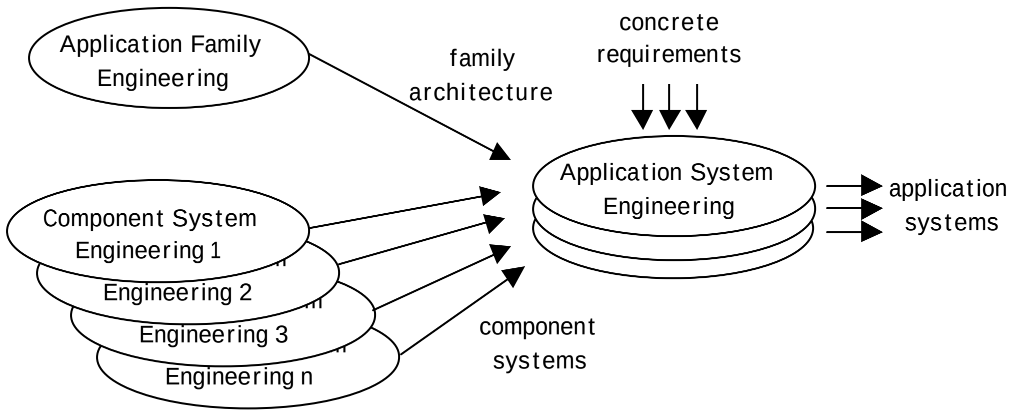


Figure 19 *Flow of artifacts in RSEB with specialized Component System Engineering processes*

The split of Domain Engineering into Application Family Engineering and Component System Engineering is based on a clear separation of foci: engineering the overall architecture versus engineering reusable solutions for the subsystems. The RSEB book describes a generic Component System Engineering process based on the general OOSE process components. This is a good start. However, as we noted in Section 4.4, there will be rather different specialized horizontal Domain Engineering methods since different subsystems may require different modeling styles and modeling techniques. Thus, the Application Family Engineering process may call different specialized horizontal Domain Engineering methods, e.g. DEMRAL (Section 4.5.5), as its Component System Engineering methods for different subsystems (see Figure 19).

Variation points

RSEB explicitly focuses on modeling variability. At the abstract level, the notion of *variation points* is introduced as an extension of the UML notation. A variation point “identifies one or more locations at which the variation will occur.” [JGJ97] A variation point is shown as a solid dot on a modeling element, e.g. a use case or a component (see Figure 20). For example, the component Account has the variation point {Account Overdrawn} and two variant components Deduct Fee and Overdraft Not Allowed are associated with this point using simple UML associations. Similarly, the use case Withdraw Money has also the variation point {Account Overdrawn}. Two variant use cases Deduct Fee and Overdraft Not Allowed are associated with this point using the <<extends>> relationship. Each variant use case describes what should happen if an account is overdrawn. We will extend the notion of variation points with different categories of variation in Section 5.4.1.7.

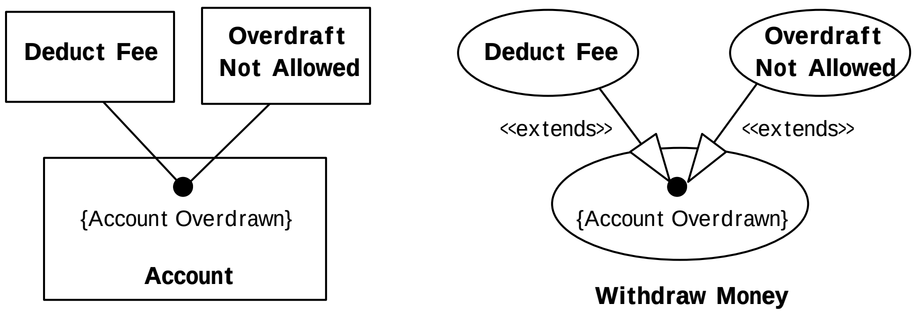


Figure 20 *Variation points in components and use cases (adapted from [JGJ97])*

Variability mechanisms

Variation points are implemented in more concrete models using different *variability mechanisms*. Indeed, any model used in the process should support some kind of variability mechanism. Examples of variability mechanisms are summarized in Table 7.

An important contribution of RSEB is variability modeling in use cases. The RSEB book lists a number of important reasons for variability in use case models:

- varying user or system interfaces;
- different entity types referenced, e.g. account in withdrawal may be a checking or a join account;
- alternative and optional functionality;
- varying constraints and business rules;
- error detection;
- performance and scalability differences.

The variability mechanisms available in use case modeling include use case templates, macros and parameters, use case inheritance (i.e. *uses* relationship), and use case extensions [JGJ97].

Mechanism	**Type of variation point**	**Type of variant**	**Use particularly when**
inheritance	virtual operation	subclass or subtype	specializing and adding selected operations, while keeping others
extensions	extension point	extension	it must be possible to attach several variants at each variation point at the same time
uses	use point	use case	reusing abstract use case to create a specialized use case
configuration	configuration item slot	configuration item	choosing alternative functions and implementations
parameters	parameter	bound parameter	there are several small variation points for each variable feature
template instantiation	template parameter	template instance	doing type adaptation or selecting alternative pieces of code
generation	parameter or language script	bound parameter or expression	doing large-scale creation of one or more types or classes from a problem-specific language

Table 7 *Some variability mechanisms (from [JGJ97])*

A key idea in RSEB is to maintain explicit traceability links connecting representations of variability throughout all models, i.e. variability present in use cases can be traced to variability in the analysis, design and implementation object models. In UML, the traceability links are modeled using the <<trace>> dependency relationship.

As stated, RSEB is based on the OOSE method and thus each of the three processes Application Family Engineering, Component System Engineering, and Application System Engineering have the five main OOSE process components:

- requirements capture
- robustness analysis
- design
- implementation
- testing

Additionally, Component System Engineering and Application System Engineering have a sixth process component, namly *packaging*, whose purpose is packaging the component system (documenting, archiving for later retrieval, etc.) or packaging the application system (documenting, developing installation scripts, etc.), respectively.

Unfortunately, despite the RSEB focus on variability, the process components of Application Family Engineering and Component System Engineering do not include Domain-Analysis-like activities of domain scoping and feature modeling.

Application family vs. component-oriented scoping

The purpose of domain scoping in Application Family Engineering would be to scope the application family or the product line with respect to classes of current and potential stakeholders. This kind of scoping is referred to as *application family scoping*. We already discussed issues concerning application family scoping in the context of product lines and families in Section 3.6.2.

The kind of scoping required for Component System Engineering is *component-oriented* one. The issues here include the following:

- reusability of the component system across multiple product lines or families within an organization;
- marketing opportunities of a component system outside an organization;
- technical issues such as the nature of the domain and the required modeling techniques;
- organizational issues such as staffing and concurrent engineering.

Component-oriented scoping corresponds to the decomposition of the applications in an application family into generic subsystems and thus represents an important aspect of the application family architecture development.

Another shortcoming of RSEB is the lack of feature models. In RSEB, variability is expressed at the highest level in the form of variation points (especially in use cases), which are then implemented in other models using various variability mechanisms. However, Griss, et al. report in [GFA98] that these modeling techniques turned out to be insufficient in practice. They summarize their experience in applying a purely use case driven approach in the telecom domain as follows:

> *"In the telecom sector, lack of explicit feature representation can be especially problematic, even in the presence of use case modeling. First, use case models do not explicitly reveal many of the implementation or technical features prevalent in telecom systems, such as 'switch types.' Second, telecom services can require very large numbers of use cases for their descriptions; and when the use cases are parameterized with RSEB variation points describing many extensions, alternatives, and options, the domain engineer can easily lose his way when architecting new systems. Reusers can easily get confused about which features and use cases to use for which application systems."*

This experience prompted them to devise FeatuRSEB [GFA98], which addresses all these problems by extending RSEB with explicit domain scoping and feature modeling activities and with feature models. We describe this new method in the following section.

4.5.4 FeatuRSEB

FeatuRSEB [GFA98] is a result of integrating FODAcom [VAM+98], an object-oriented adaptation of FODA for the telecom domain, into RSEB in a cooperation between Hewlett-Packard Inc. and Intecs Sistemi.

FeatuRSEB extends RSEB in two important ways:

- Application Family Engineering and Component System Engineering are extended with explicit domain scoping, domain planning, and feature modeling activities.
- Feature models are used as the primary representation of commonalities, variabilities, and dependencies.

Both Objectory and RSEB subscribe to the 4+1 Model View of architecture by Kruchten (see Figure 17) of developing several models, plus one that ties them all together. While, in Objectory and in RSEB, the role of this unifying "+1" model plays the use case model, the "+1" model in FeatuRSEB is the feature model. Griss et al. note that the feature model serves "as a concise synthesis of the variability and commonality represented in the other RSEB models, especially the use case model." [GFA98] In other words, FeatuRSEB is *feature model centric*. They further state that, as a FeatuRSEB principle, "not everything that *could* be a feature *should* be a feature. Feature descriptions need to be robust and expressive. Features are used primarily to discriminate between *choices*, not to describe functionality in great detail; such detail is left to the use case or object models."

Griss et al. also describe the important relationship between use case models and feature models: A use case model captures the system requirements from the user perspective (i.e. "operational requirements"), whereas the feature model organizes requirements from the reuser perspective based on commonality, variability, and dependency analysis.

The feature models in FeatuRSEB consist, like their FODA counterparts, of feature diagrams annotated with constraints, binding time, category, rationale, etc.

The feature modeling steps in FeatuRSEB are the following [GFA98]:

1. Merge individual exemplar use case models into a domain use case model (known as the Family Use case Model in the RSEB). Using variation points capture and express the differences. Keep track of the originating exemplars using «trace».

2. Create an initial feature model with *functional features* derived from the domain use case model (typically using use case names as a starting point for feature names).

3. Create the RSEB analysis object model, augmenting the feature model with *architectural features*. These features relate to system structure and configuration rather than to specific function.

4. Create the RSEB design model, augmenting the feature model with *implementation features*.

Thus, FeatuRSEB distinguishes between functional, architectural, and implementation features.

Griss et al. also propose an implementation of the feature diagram notation in UML and give some requirements on tool support for feature models. We will discuss both topics later in Sections 5.5 and 5.7.

4.5.5 Domain Engineering Method for Reusable Algorithmic Libraries (DEMRAL)

DEMRAL is a Domain Engineering method for developing algorithmic libraries, e.g. numerical libraries, container libraries, image processing libraries, image recognition libraries, speech recognition libraries, graph computation libraries, etc. Thus, it is a horizontal method. The main abstractions in algorithmic domains are abstract data types (ADTs) and algorithms. Excellent performance, effective representation of a myriad of ADT and algorithm variants, achieving high adaptability and reusability, and providing an abstract interface are the main design goals in DEMRAL. The method has been created as a specialization of ODM (Section 3.7.2).

A fundamental aspect of DEMRAL is feature modeling. Indeed, feature modeling is the driving force in DEMRAL. DEMRAL involves creating a high-level feature model of the domain of focus and feature models of each concept in the domain. DEMRAL also integrates concepts from Aspect-Oriented Programming such as aspectual decomposition and the application of domain-specific languages (DSLs) for expressing different aspects (see Chapter 7).

An important concept in DEMRAL are *configuration DSLs*. A configuration DSL is used to configure a component. It allows different levels of control, so that clients may specify their needs at the appropriate level of detail. DEMRAL provides an approach for deriving configuration DSLs from feature models.

DEMRAL also gives advise on implementing domain models using OO techniques and metaprogramming (e.g. generators, transformation systems, or built-in metaprogramming capabilities of programming languages).

DEMRAL is described in Chapter 9.

4.6 References

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. Wiley, Chichester, UK, 1996

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Second edition, Benjamin/Cummings, Redwood City, California, 1994

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, Englewoods Cliffs, New Jersey, 1994

[CN98] S. Cohen and L. M. Northrop. Object-Oriented Technology and Domain Analysis. In [DP98], pp. 86-93

[Cop95] J. O. Coplien. A Generative Development-Process Pattern Language. In *Pattern Languages of Program Design*, J. O. Coplien and D. C. Schmidt, (Eds.), Addison-Wesley, 1995, pp. 183-239

[DP98] P. Devanbu and J. Poulin, (Eds.). *Proceedings of the Fifth International Conference on Software Reuse (Victoria, Canada, June 1998)*. IEEE Computer Society Press, 1998

[DW97] D. F. D'Souza and A. C. Wills. *Objects, Components and Frameworks with UML*. Addison Wesley Longman, 1997, see http://www.iconcomp.com/catalysis/

[EP98] H.-E. Eriksson and M. Penker. *UML Toolkit*. John Wiley & Sons, 1998

[Fow96] M. Fowler. A Survey of Object-Oriented Analysis and Design Methods. Tutorial notes, presented at OOPSLA'96, 1996

[Fow97] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997

[GFA98] M. L. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In [DP98], pp. 76-85, see http://www.intecs.it

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[GR95] A. Goldberg and K. Rubin. *Succeeding with Objects: Decision Frameworks for Project Management*. Addison-Wesley, 1995

[HC91] J. Hooper and R. Chester. *Software reuse: guidelines and methods*. Plenum Press, 1991

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley, Workingham, England, 1992

[JEJ94] I. Jacobson, M. Ericsson, and A. Jacobson. *The Object Advantage - Business Process Reengineering with Object Technology*. Addison-Wesley, Menlo Park, California, 1994

[JGJ98] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley Longman, May 1997

[Joh92] R. E. Johnson. Documenting Frameworks using Patterns. In *Proceedings of the Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'92)*, ACM SIGPLAN Notices, vol. 27, no. 10, October 1992, pages 63-76

[Kru95] P. Kruchten. The 4+1 View Model of Architecture. In *IEEE Software*, vol. 12, no. 6, November 1995, pp. 42-50

[MCK97] M. Meusel, K. Czarnecki, and W. Köpf. A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and Hypertext. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, M. Aksit and S. Matsuoka (Eds.), LNCS 1241, Springer-Verlag, 1997, pp. 496-510

[Rat98a] Rational Software Corporation. Rational Objectory Process 4.1 — Your UML Process. White paper, 1998, see http://www.rational.com

[Rat98b] Rational Software Corporation. Rational Objectory Process 4.1. Commercially available process description, see http://www.rational.com

[Rat98c] Rational Software Corporation. Unified Modeling Language (UML), version 1.1. Documentation set available from http://www.rational.com/uml/documentation.html

[Ree96] T. Reenskaug with P. Wold and O.A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning, 1996

[Rie97] D. Riehle. Composite Design Patterns. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97), ACM SIGPLAN Notices*, vol. 32, no. 10, October 1997, pp. 218-228

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewoods Cliffs, New Jersey, 1991

[Sim97] M. Simos. Organization Domain Modeling and OO Analysis and Design: Distinctions, Integration, New Directions. In *Proceedings 3rd STJA Conference (Smalltalk and Java in Industry and Education, Erfurt, September 1997)*, Technical University of Ilmenau, 1997, pp. 126-132, see http://nero.prakinf.tu-ilmenau.de/~czarn/generate/stja97/

[Son97] X. Song. Systematic Integration of Design Methods. In *IEEE Software*, March/April 1997, pp. 107-117

[VAM+98] A. D. Vici, N. Argentieri, A. Mansour, M. d'Alessandro, and J. Favaro. FODAcom: An Experience with Domain Analysis in the Italian Telecom Industry. In [DP98], pp. 166-175, see http://www.intecs.it

[Weg97] P. Wegner. Why Interaction Is More Powerful Than Algorithms. In *Communications of the ACM*, May 1997, see http://www.cs.brown.edu/people/pw/

[Weg98] P. Wegner. Interactive Foundations of Computing. Final draft, February 1998, to appear in *Theoretical Computer Science*, see http://www.cs.brown.edu/people/pw/

Chapter 5 Feature Modeling

5.1 Motivation

The conclusion of the previous chapter was that feature modeling constitutes the major contribution of Domain Engineering with respect to the conventional software system engineering. Thus, we shall discuss feature modeling in more detail in this chapter.

Feature modeling is particularly important if we engineer for reuse. The reason is that reusable software contains inherently more variability than concrete applications and feature modeling is the key technique for identifying and capturing variability.

As discussed in Section 4.1.2, feature modeling helps us to avoid two serious problems:

- relevant features and variation points are not included in the reusable software;
- many features and variation points are included but never used and thus cause unnecessary complexity, development cost, and maintenance cost.

Finally, the feature models produced during feature modeling provide us with an abstract (since implementation independent), concise, and explicit representation of the variability present in the software.

5.2 Features Revisited

We discussed the notion of features in the last three chapters. We saw that different methods and approaches used slightly different interpretations of features. Before we move our attention to feature modeling, we first summarize what we have said about features so far.

Following the conceptual modeling perspective (Chapter 2) and the ODM perspective (Section 3.7.2), a feature is an important property of a concept. Features allow us to express the commonalities and differences between concept instances. They are fundamental to formulating concise descriptions of concepts with large degrees of variation among their instances. Organized in feature diagrams (Section 3.7.1.2), they express the configurability aspect of concepts.

A feature should have a concise and descriptive name — much as in the case of a design pattern. The name enriches the vocabulary for describing concepts and instances in a domain. By organizing features into feature diagrams, we actually build taxonomies.

Features are primarily used in order to discriminate between instances (and thus between choices). In this context, the quality of a feature is related to properties such as its primitiveness, generality, and independency. We discussed these concepts in Section 2.3.5.

In the context of Domain Engineering, features represent reusable, configurable requirements and each feature has to make a difference to someone, e.g. a stakeholder or a client program. For example, when we build an order processing system, one of the features of the pricing component could be *aging pricing strategy*, i.e. you pay less for older merchandise. This pricing strategy might be particularly interesting to stores selling perishable goods.

Features may occur at any level, e.g. high-level system requirements, architectural level, subsystem and component level, implementation-construct level (e.g. object or procedure level).

Modeling the semantic content of features usually requires some additional modeling formalism, e.g. object diagrams, interaction diagrams, state diagrams, synchronization constraints, etc. Thus, feature models are usually just one out of many other kinds of models describing a piece of reusable software.

5.3 Feature Modeling

Feature modeling and feature model

Feature modeling is the activity of modeling the common and the variable properties of concepts and their interdependencies and organizing them into a coherent model referred to as a feature model.

By concepts, we mean any elements and structures in the domain of interest. We already discussed the notion of concepts and conceptual modeling in Chapter 2 in detail. But let us make a few remarks about concepts.

Many OO enthusiasts do not distinguish between concepts and OO classes. For them, “everything is an object”. Of course, this is a quite naive or even profane view of the world.

There is an obvious similarity between OO classes and concepts: OO classes represent a generic description of a set of objects. Similarly, concepts represent a generic description of a set of concept instances. So what is the difference between concepts and OO classes? In order to answer this question, we have to move to the instance level. Objects, i.e. the instances of OO classes, have some predefined semantic properties: they have state, exhibit some well-defined behavior, and have a unique identity (see [Boo94]). Instances of concepts, on the other hand, do not have any predefined semantics. They could be anything. This difference is shown in Figure 21.

We can think of concepts as “reference points” in the brain for classifying phenomena. A concepts stands for a class of phenomena. Of course, it is important to give names to relevant concepts, so that we can talk about them without having to list all their properties (which, in most cases, as we have seen in Chapter 2, is impossible anyway). The conclusion of Chapter 2 also was that concepts are inherently subjective: their information contents depends not only on the person, but also on time, context, and other factors.

So why do we define feature modeling around concepts and not classes of objects? The reason is that we may want to model features of any elements and structures of a domain, not just objects. It should be possible to describe variability of use cases, OO classes, functions, procedures, etc., not just OO classes. This way, we can use feature modeling together with various other modeling techniques, e.g. use case modeling, class modeling, specifications of functions, procedures, etc.

Feature modeling may play different roles in different Domain Engineering methods. e.g.:

- producing a feature model representing the configurability aspect of reusable software as in, e.g., FeatuRSEB (Section 4.5.4);

- the driving modeling activity, e.g., all modeling in DEMRAL (Chapter 9) starts as feature modeling and other modeling techniques are “called” out of feature modeling.

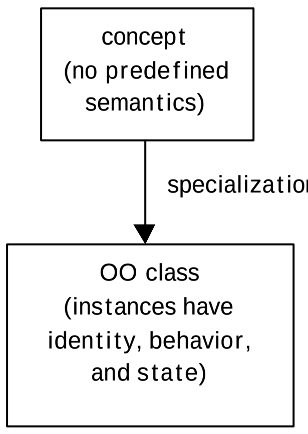

concept
(no predefined semantics)
specialization
OO class
(instances have identity, behavior, and state)

Figure 21 *Difference between a concept and an OO class*

We refer to the latter as *feature-driven modeling*. Feature-driven modeling is useful whenever variability constitutes the main aspect of the modeled software.

There are three important aspects of feature modeling:

- *"Micro-cycle" of feature modeling*: What are the basic steps in feature modeling? How do we identify features? We investigate these questions in Section 5.8.1.

- *Integration in the overall software engineering process*: The micro-cycle of feature modeling may be integrated into the overall software engineering process in different ways. For example, in FeatuRSEB, it accompanies all other modeling activities (see feature modeling steps in FeatuRSEB in Section 3.7.1.1). In DEMRAL, as noted above, it constitutes the main modeling activity. Another aspect is the relation between feature models and other models produced during development. As already discussed, we need to maintain traceability links showing the connections between variability representations in different models.

- *Content aspect of feature modeling*: Features have some semantic content and feature modeling may focus on different kinds of content at a time, e.g. functional aspects, data-flow aspects, interaction aspect, synchronization aspects, etc. Different model representations are used for different contents. This issue is connected to decomposition techniques, which we discuss in Section 5.8.2.

It is important to note that feature modeling is a creative activity. It is much more than just a simple rehash of the features of existing systems and the available domain knowledge. New features and new knowledge is created during feature modeling. For example, one technique used in feature modeling is the analysis of combinations of variable features, which may lead to the discovery of innovative feature combinations and new features. The systematic organization of existing knowledge allows us to invent new, useful features and feature combinations more easily.

In some Domain Engineering methods, feature modeling is referred to as feature analysis. However, we prefer the term feature modeling since it emphasizes the creative aspect of this activity.

Before we answer the question of how to perform feature modeling in Section 5.8, we first describe the elements of feature models.

5.4 Feature Models

A *feature model* represents the common and the variable features of concept instances and the dependencies between the variable features. Feature models are created during feature modeling.

A feature model represents the *intention* of a concept, whereas the set of instances it describes is referred to as the *extension* of the concept.

A feature model consists of a feature diagram and some additional information such as short semantic description of each feature, rationale for each feature, stakeholders and client programs interested in each feature, examples of systems with a given feature, constraints, default dependency rules, availability sites (i.e. where, when, and to whom a feature is available), binding sites (i.e. where, when, and who is able to bind a feature), binding modes (e.g. dynamic or static binding), open/closed attributes (i.e. whether new subfeatures are expected), and priorities (i.e. how important is a feature). We discuss all these items in Section 5.4.2.

From a feature diagram of a concept, we can derive *featural descriptions* of the individual instances of the concept. A featural description of an instance is a *set of features*.[34] Two feature diagrams are *equivalent*, if the set of all instance descriptions derivable from the first diagram is equal to the set of all instance description derivable from the other diagram.

We describe feature diagrams in the following sections. All the additional information contained in a feature model is described in Section 5.4.2.

5.4.1 Feature Diagrams

In this section, we introduce a slightly modified and extended version of the FODA feature diagram notation (cf. Section 3.7.1.2) as well as some useful vocabulary for talking about feature diagrams.

A feature diagram consists of a set of nodes, a set of directed edges, and a set of edge decorations. The nodes and the edges form a tree.[35] The edge decorations are drawn as arcs connecting subsets or all of edges originating from the same node (see e.g. Figure 25). Effectively, edge decorations define a partitioning of the subnodes of a node (i.e. they divide the subnodes in a number of disjoint subsets).

The root of a feature diagram represents a concept. We refer to it as the *concept node*.[36] The remaining nodes in a feature diagram represent features and we refer to them as *feature nodes*. Throughout the text, we usually leave out the word "node" and simply say "feature" instead of "feature node" and "concept" instead of "concept node".

The parent node of a feature node is either the concept node, or another feature node. Consider Figure 22, which shows a feature diagram with the three features f_1, f_2 and f_3 of the concept C, where the parent of f_1 is C, the parent of f_2 is f_1, and the parent of f_3 is f_2. Given these relationships, we say that *(i)* f_1 is a *direct feature* of C, *(ii)* f_2 and f_3 are *indirect features* of C, *(iii)* f_2 is a *direct subfeature* of f_1, and *(iv)* f_3 is an *indirect subfeature* of f_1.

As in FODA, we distinguish between *mandatory*, *alternative*, and *optional features*. In addition to these feature types, we also introduce *or-features*. Furthermore, optionality can be combined with alternative features and with or-features resulting in the two additional feature types *optional alternative features* and *optional or-features*. However, as we see later, the optional or-feature type is equivalent to the optional feature type and thus it is redundant.

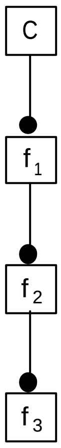


Figure 22 *Feature diagram with three features*

A description of an instance of a concept is a set of nodes which always includes the concept node and some or all of the feature nodes. A valid description of an instance is derived from a feature diagram by adding the concept node to the feature set and by traversing the diagram starting at the root and, depending on the type of the visited node, including it in the set or not. The node types and the criteria for the inclusion of a feature in an instance description are defined in the following sections.

5.4.1.1 Mandatory Features

A *mandatory feature* is included in the description of a concept instance if and only if its parent is included in the description of the instance. Thus, for example, if the parent of a mandatory feature is optional and not included in the instance description, the mandatory feature cannot be part of the description. Please remember that the concept node of a feature diagram is always included in any instance description derived from the diagram.

A mandatory feature node is pointed to by a *simple edge* (as opposed to an *arc-decorated edge*) ending with a filled circle as in Figure 23. Features f_1, f_2, f_3, and f_4 are mandatory features of concept *C*. According to Figure 23, every instance of concept *C* has features f_1 and f_2, and every instance of *C* which has f_1 also has f_3 and f_4. Thus, effectively, every instance of *C* has f_3 and f_4. Finally, we conclude that every instance of *C* can be described by the feature set $\{C, f_1, f_2, f_3, f_4\}$.

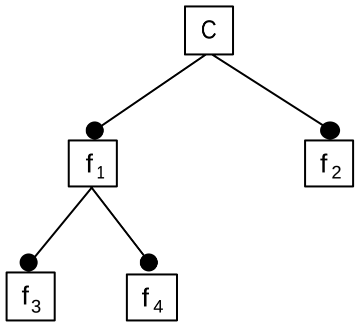


Figure 23 *Example of a feature diagram with mandatory features*

5.4.1.2 Optional Features

An *optional feature* may be included in the description of a concept instance if and only if its parent is included in the description. In other words, if the parent is included, the optional feature may be included or not, and if the parent is not included, the optional feature cannot be included.

An optional feature node is pointed to by a simple edge ending with an empty circle as in Figure 24. Features f_1, f_2, and f_3 are optional features of concept *C*. According to Figure 24, an instance of *C* might have one of the following descriptions: *{C}*, *{C, f_1}*, *{C, f_1, f_3}*, *{C, f_2}*, *{C, f_1, f_2}*, or *{C, f_1, f_3, f_2}*.

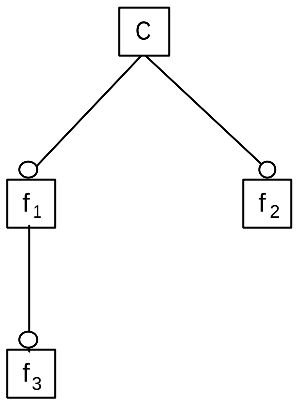


Figure 24 *Example of a feature diagram with optional features*

5.4.1.3 Alternative Features

A concept may have one or more sets of direct *alternative features*. Similarly, a feature may have one or more sets of direct *alternative subfeatures*. If the parent of a set of alternative features is included in the description of a concept instance, then exactly one feature from this set of alternative features is included in the description, otherwise none.

The nodes of a set of alternative features are pointed to by edges connected by an arc. For example, in Figure 25, *C* has two sets of alternative features: one set with f_1 and f_2 and another set with f_3, f_4, and f_5. From this diagram, we can derive the following instance descriptions: *{C, f_1, f_3}*, *{C, f_1, f_4}*, *{C, f_1, f_5}*, *{C, f_2, f_3}*, *{C, f_2, f_4}*, or *{C, f_2, f_5}*.

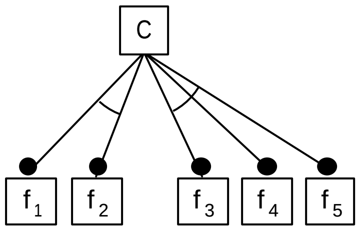


Figure 25 *Example of a feature diagram with two sets of alternative features*

A feature (or concept) with a single set of direct alternative subfeatures (or features) and no other direct subfeatures (or features) is referred to as a *dimension* (e.g. f_1 in Figure 26). We also found it useful to broaden the notion of dimensions to include features (concepts) with a single set of direct alternative subfeatures (features) and one or more direct mandatory subfeatures (features). According to this broader definition, f_2 in Figure 26 is also a dimension, but f_3 is not.[37]

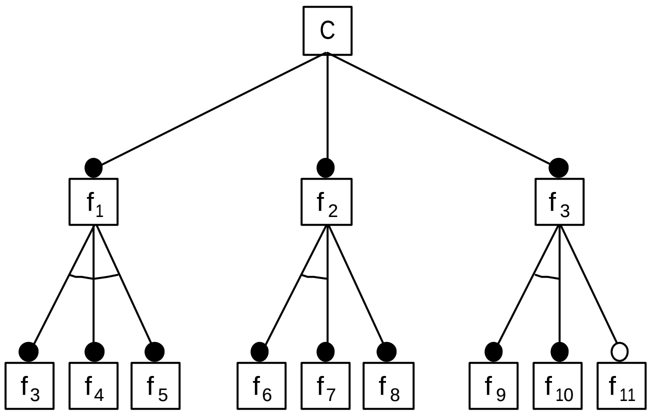


Figure 26 *Example of a feature diagram with two dimensions*

Dimensions can also be alternative, i.e. we have *alternative dimensions* (see Figure 27).

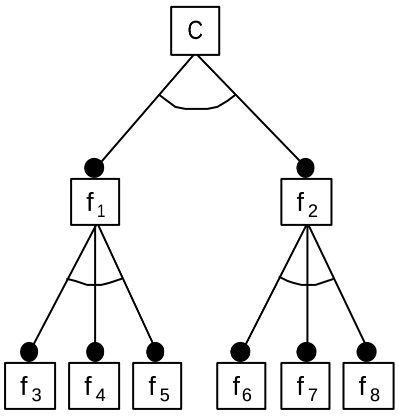


Figure 27 *Example of a feature diagram with two alternative dimensions*

Similarly, we can have *optional dimensions* (see Figure 28).

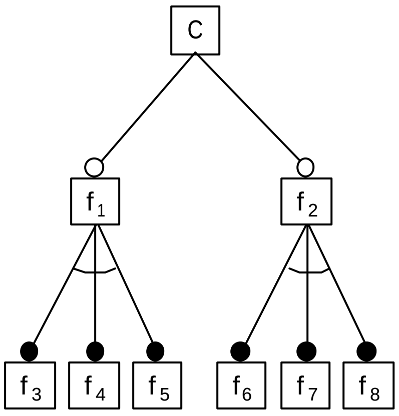


Figure 28 *Example of a feature diagram with two optional dimensions*

An alternative feature can also be optional, as f_1 in Figure 29. However, as we will see in Section 5.4.1.5, during the normalization of a feature diagram, all features of a set of alternative features containing one or more optional alternative features are replaced by optional alternative features (see Figure 32).

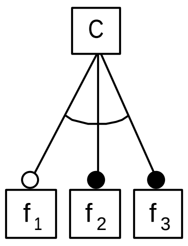


Figure 29 *Example of a feature diagram with one optional alternative feature*

5.4.1.4 Or-Features

A concept may have one or more sets of direct *or-features*. Similarly, a feature may have one or more sets of direct *or-subfeatures*. If the parent of a set of or-features is included in the description of a concept instance, then any non-empty subset from the set of or-features is included in the description, otherwise none.

The nodes of a set of or-features are pointed to by edges connected by a filled arc. For example, in Figure 30, *C* has two sets of or-features: one set with f_1 and f_2 and another set with f_3, f_4, and f_5. A total of $(2^2-1)*(2^3-1)$, i.e. 21, different instance descriptions may be derived from this diagram.

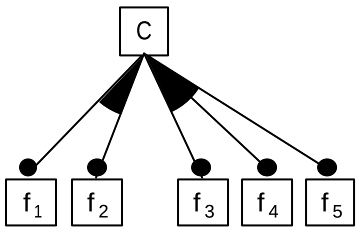


Figure 30 *Example of a feature diagram with two sets of or-features*

An or-feature can also be optional, e.g. f_1 in Figure 31. However, as we will see in Section 5.4.1.5, during the normalization of a feature diagram, all or-features of a set of or-features containing one or more optional or-features are replaced by optional features (see Figure 33).

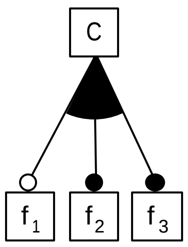


Figure 31 *Example of a feature diagram with an optional or-feature*

5.4.1.5 Normalized Feature Diagrams

A node in a feature diagram can have mandatory feature subnodes (e.g. f_1 in Figure 23), optional feature subnodes (e.g. f_1 in Figure 24), alternative feature subnodes (e.g. f_1 in Figure 25), optional alternative feature subnodes (e.g. f_1 in Figure 29), or-feature subnodes (e.g. f_1 in Figure 30), and optional or-feature subnodes (e.g. f_1 in Figure 31). Thus, in addition to the mandatory, alternative, optional, and or-feature nodes, we also have *optional alternative feature nodes* and *optional or-feature nodes*. Now we take a closer look at the latter two kinds of features.

If one or more of the features in a set of alternative features is optional, this has the same effect as if all the alternative features in this set were optional. This is illustrated in Figure 32.

Similarly, if one or more of the features in a set of or-features is optional, this has the same effect as if all the features in this set were optional or-features. Furthermore, all of the optional or-features can be replaced by optional features. Therefore, if one or more features in a set of or-features is optional, we can replace all these features by optional features. This is illustrated in Figure 33. We conclude that the category of optional or-features is redundant since it is equivalent to optional features.

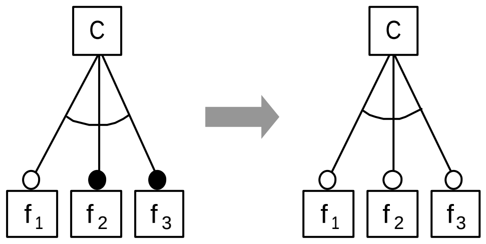


Figure 32 *Feature diagram with one optional alternative feature is normalized into a diagram with three optional alternative features*

Normalized feature diagrams

Any feature diagram can be transformed into a feature diagram which does not have any optional or-features and whose sets of alternative features may contain either only alternative features or only alternative optional features. The transformation can be accomplished by replacing the non-optional alternative features of a set of alternative features containing one or more optional alternative features by optional alternative features and by replacing all optional or-features by optional features. The resulting feature diagram is referred to as a *normalized feature diagram* and is equivalent to the original feature diagram.

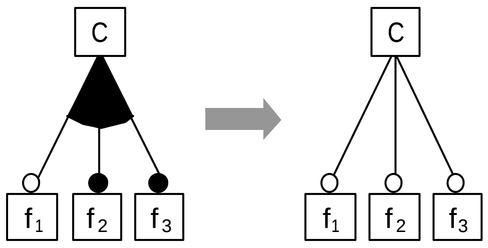


Figure 33 *Feature diagram with one optional or-feature is normalized into a diagram with three optional features*

Subnode categories: mandatory, optional, alternative, optional alternative, and or-feature nodes

A feature node in a normalized feature diagram can be classified according to its *subnode category* as either *mandatory*, or *optional*, or *alternative*, or *optional alternative*, or *or-feature node*.

Furthermore, the set of subnodes of any node of a normalized feature diagram can be partitioned into the following disjoint (possibly empty) sets: one set of mandatory feature nodes, one set of optional feature nodes, one or more sets of alternative feature nodes, one or more sets of optional alternative feature nodes, and one or more sets of or-feature nodes. This partitioning is referred to as *subnode partitioning* and each of the resulting sets is called a *subnode partition* of the parent node.

Subnode partitioning

5.4.1.6 Expressing Commonality in Feature Diagrams

Feature diagrams allow us to represent concepts in a way that makes the commonalities and variabilities among their instances explicit. We first take a look at representing commonality.

Depending on the focus, there are two types of commonalities. If we focus on the concept node, we might ask the question: What features are common to all instances of the concept? On the other hand, if we focus on a particular feature node, we might ask: What features are common to all instances of the concept that have that particular feature? In order to answer these questions, we introduce the notions of *common features* and *common subfeatures*.

Common features

A *common feature* of a concept is a feature present in all instances of a concept. All mandatory features whose parent is the concept are common features. Also, all mandatory features whose parents are common are themselves common features. Thus, a feature is a common feature of a concept if the feature is mandatory and there is a path of mandatory features connecting the feature and the concept. For example, in Figure 23, f_1, f_2, f_3, and f_4 are common features.

Common subfeatures

A *common subfeature* of the feature *f* is a (direct or indirect) subfeature of *f* which is present in all instances of a concept which also have *f*. Thus, all direct mandatory subfeatures of *f* are common subfeatures of *f*. Also, a subfeature of *f* is common if it is mandatory and there is a path of mandatory features connecting the subfeature and *f*.

5.4.1.7 Expressing Variability in Feature Diagrams

Variable features

Variability in feature diagrams is expressed using optional, alternative, optional alternative, and or-features. We refer to these features as *variable features*.

Variation points

The nodes to which variable features are attached are referred to as *variation points*.[38] More formally, a variation point is a feature (or concept) which has at least one direct variable subfeature (or feature).

Dimensions and extension points

Different important types of variation points such as dimensions and extension points are defined in Table 8.

Homogeneous vs. inhomogeneous variation points

Moreover, we distinguish between *homogeneous* vs. *inhomogeneous variation points*. A variation point is homogeneous if all its direct subfeatures (or features) belong to the same node subcategory (i.e. they are all optional, or all alternative, etc.), otherwise it is inhomogeneous.

Singular vs. nonsingular variation points

We can further categorize variation points according to whether they allow us to include at most one direct subfeature (or feature) in the description of an instance, or more than one direct subfeature (or feature). The first category is referred to as *singular variation points* and it includes dimensions, dimensions with optional features, and extension points with exactly one optional feature. The second category is referred to as *nonsingular variation points* and it includes extension points with more than one optional feature and extension points with or-features.

Mutually exclusive features

In a similar way as we generalized mandatory features to common features, we can also generalize alternative features to *mutually exclusive features*: Two features in a feature diagram of a concept are mutually exclusive, if none of the instances of the concept have both features at the same time. Given a tree-shaped feature diagram, two features are mutually exclusive if the direct parent or any of the indirect parents of the first feature is in the same set of alternative features as the direct parent or any of the indirect parents of the other feature.

Simultaneous vs. non-simultaneous variation points

Finally, we also distinguish between simultaneous and non-simultaneous variation points. Two variation points of a feature diagram without edge decorations are *simultaneous* if and only if they are both not mutually exclusive and not related by the (direct or indirect) parentship relation. Analogously, two variation points are *non-simultaneous* if and only if they are either mutually exclusive or related by the direct or indirect parentship relation (i.e. one is a parent of the other one).

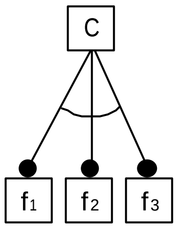
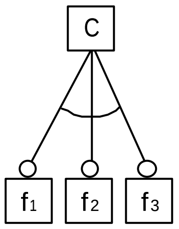
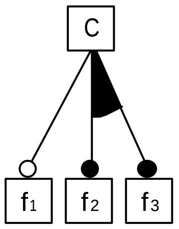
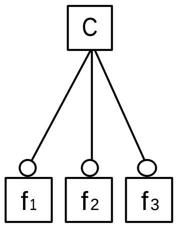
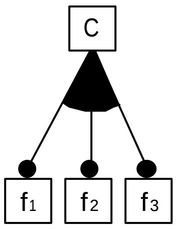
Type of variation point	Definition	Example
dimension	feature (or concept) whose all direct subfeatures (or features) are alternative features	
dimension with optional features	feature (or concept) whose all direct subfeatures (or features) are alternative optional features	
extension point	feature (or concept) which has at least one direct optional subfeature (or feature) or at least one set of direct or-subfeatures (or or-features).	
extension point with optional features	feature (or concept) whose all direct subfeatures (or features) are optional features	
extension point with or-features	feature (or concept) whose all direct subfeatures are or-features	

Table 8 *Types of variation points*

5.4.1.8 Feature Diagrams Without Edge Decorations[39]

In this section, we present *feature diagrams without edge decorations*, a feature diagram notation, which does not have any edge decorations (i.e. empty arcs and filled arcs). Feature diagrams without edge decorations do not contain inhomogeneous variation points. Any feature diagram in the notation introduced previously can be converted into an equivalent feature diagram without edge decorations.

The feature diagram notation presented so far (i.e. feature diagrams with edge decorations) is more concise than the notation without edge decorations. Therefore, we will use feature diagrams with edge decorations in domain analysis. The reason for introducing the equivalent feature diagrams without edge decorations is that they have a simpler structure. For example, they do not contain any inhomogeneous variation points. Because of this simpler structure, they allow an easier analysis.

Node Category	Icon
concept	●
parent of mandatory features	
leaf	
parent of optional features (i.e. extension point with optional features)	○
parent of alternative features (i.e. dimension)	◉
parent of optional alternative features (i.e. dimension with optional features)	◎
parent of or-features (i.e. extension point with or-features)	◒

Table 9 *Parent node categories occurring in feature diagrams without edge decorations and their icons*

A feature diagram without edge decorations is a tree (see Figure 35). Each node of the tree belongs to one of the seven *parent node categories* listed in Table 9. The root of a diagram belongs to the category *concept* and is the only element of this category. Nodes without outgoing edges (except, if applicable, the root) belong to the category *leaf.* Nodes with direct mandatory subfeatures only belong to the category *parent of mandatory features*. The remaining four categories are homogeneous variation points.

Now, we give the rules for deriving a valid description of an instance of a concept based on its feature diagram without edge decorations. We assume that the diagram is a tree and the descriptions are sets of features. We have the following inclusion rules:

1. The concept node and its direct features are always included in the description.

2. If a node of the category “parent of mandatory features” is included in the description, all its subnodes are also included in the description.

3. If a node of the category “parent of optional features” is included in the description, any (possibly empty) subset or all of its subnodes are also included in the description.

4. If a node of the category “parent of alternative features” is included in the description, exactly one of its subnodes is also included in the description.

5. If a node of the category “parent of optional alternative features” is included in the description, none or exactly one of its subnodes is also included in the description.

6. If a node of the category “parent of or-features” is included in the description, any non-empty subset or all of its subnodes are also included in the description.

Given a feature diagram in the notation with edge decorations, we can convert it into an equivalent feature diagram without edge decorations by performing the following transformations:

1. Normalize the feature diagram (see previous Section 5.4.1.5).

2. Record the subnode partitions for each of the nodes in the normalized feature diagram.

3. Drop the edge decorations, i.e. the arcs and the filled arcs.

4. Insert intermediate feature nodes between the concept node and its subnodes. The intermediate nodes are inserted one per subnode partition, so that the feature nodes in one subnode partition have one common intermediate node as their parent. The partitions recorded in step 2 have to be updated by inserting the newly created nodes into the appropriate partitions of mandatory feature nodes.

5. Assign the parent node categories based on the subnode categories determined by the partitioning recorded in steps 2 and 4 as follows: The root is concept, a parent node of mandatory feature nodes is "parent of mandatory features", a parent node of optional feature nodes is "parent of optional features", a parent node of alternative feature nodes is "parent of alternative features", a parent node of optional alternative feature nodes is "parent of optional alternative features", a parent node of or-feature nodes is "parent of or-features", the leaf nodes are leaf.

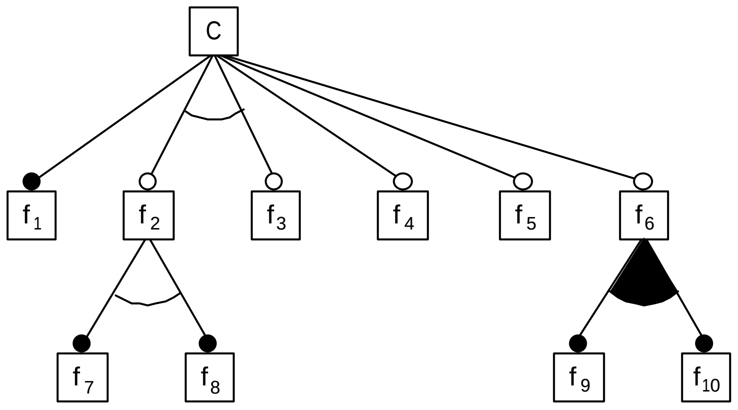


Figure 34 *Example of a feature diagram with edge decorations. This diagram is equivalent to the feature diagram in Figure 35.*

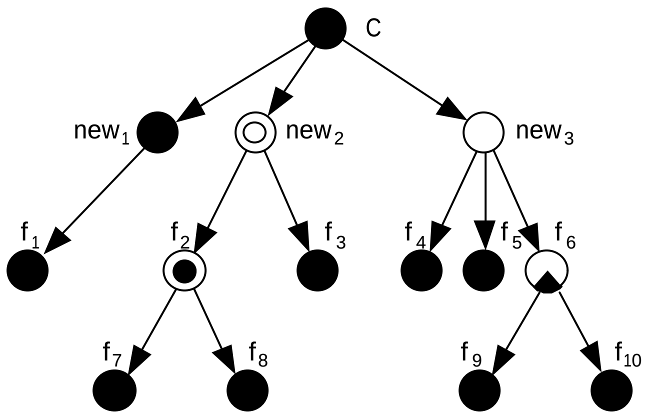


Figure 35 *Example of a feature diagram without edge decorations. This diagram is equivalent to the feature diagram in Figure 34.*

This transformation produces a feature diagram without edge decorations which is equivalent to the original feature diagram with edge decorations. They are equivalent in the sense that the set of instance descriptions which can be derived from the latter diagram is equal to the set of instance descriptions which can be derived from the first diagram after removing the new intermediate feature nodes from the descriptions in the latter set.

An example of a feature diagram without edge decorations is shown in Figure 35. This diagram is equivalent to the diagram in Figure 34. The icons representing the nodes of a feature diagram without edge decorations are explained in Table 9.

It is worth noting that nodes in feature diagrams with edge decorations are classified according to their subnode categories, whereas nodes in feature diagrams without edge decorations are classified according to their parent node categories. Furthermore, subnode categories are determined based on edge decorations and the optionality attribute of a node, whereas parent node categories may be directly stored as an attribute of a node.

5.4.2 Other Information Associated with Feature Diagrams

A complete feature model consists of a feature diagram and other information associated with it including the following:

- *Semantic description*: Each feature should have at least a short description describing its semantics. It may also be useful to attach some models in appropriate formalisms (e.g. an interaction diagram, pseudo code, equations, etc.). Eventually, there will be traceability links to other models implementing this feature. We may also assign categories to features, e.g. categories indicating the aspect a given feature belongs to.

- *Rationale*: A feature should have a note explaining why the feature is included in the model. Also each variable feature should be annotated with conditions and recommendations when it should be used in an application.

- *Stakeholders and client programs*: It is useful to annotate each feature with stakeholders (e.g. users, customers, developers, managers, etc.) who are interested in the feature and, in the case of a component, with the client programs (or examples of such) which need this feature.

- *Exemplar systems*: It is useful to annotate a feature with existing systems (i.e. exemplar systems) which implement this feature, if known.

- *Constraints and default dependency rules*: Constraints record required dependencies between variable features, possibly over multiple feature diagrams. Two important kinds of constraints are *mutual-exclusion constraints* (i.e. constraints describing illegal combinations of variable features) and *requires constraints* (i.e. constraints describing which features require the presence of which other features). *Default dependency rules* suggest default values for unspecified parameters based on other parameters.[40] We distinguish between *horizontal* and *vertical* constraints and default dependency rules. Horizontal constraints and default dependency rules describe dependencies between features of a similar level of abstraction (e.g. constraints within one feature diagram are usually horizontal features), whereas vertical constraints and default dependency rules map high-level specification features onto implementation features. Constraints and default dependency rules allow us to provide an automatic (e.g. constraint-based) configuration, which relieves the user or the reuser of much manual configuration work.

- *Availability sites, binding sites, and binding mode*: Availability site describes when, where, and to whom a variable feature is available and binding site describes when, where, and by whom a feature may be bound. Binding mode determines whether a feature is statically, changeably, or dynamically bound. We discuss availability site, binding site, and binding mode in Section 5.4.4.

- *Open/closed attribute*: It is useful to mark variation points as open if new direct variable subfeatures (or features) are expected. For example, the element type of a matrix could have integer, long, float, double, and complex as alternative subfeatures. We would mark it *open* to indicate that, if needed, other number types may be added. By marking a variation point as *closed*, we indicate that no other direct variable subfeatures (or features) are expected.

- *Priorities*: Priorities may be assigned to features in order to record their relevance to the project. We discuss priorities in the following section.

5.4.3 Assigning Priorities to Variable Features

It is often useful to annotate variable features with priorities. This might be so in the following situations:

- *Domain scoping and definition*: Priorities can be used in order to record the typicality rates of variable features of a system based on the analysis of existing exemplar systems and the target application areas for the system. The priorities may also be adjusted according to the goals of the stakeholders to record the relevance of each variable feature for the project. For example, the definition of the *domain of matrix computations libraries* involves the construction of a feature diagram of the concept *matrix computations library* (see Figure 137). The features of a matrix computations library are determined based on the analysis of existing matrix computations libraries and the application areas of matrix computations libraries. Finally, the feature diagram is annotated with priorities reflecting the importance for a matrix computations library to provide certain features in order to "be a matrix computations library". For example, every matrix computations library has to provide dense matrices or sparse matrices or both; however, dense matrices are more commonly implemented than sparse matrices. This type of concept definition corresponds to the probabilistic view of concepts discussed in Section 2.2.3.

- *Feature modeling*: The variable features of feature diagrams produced during feature modeling can also be annotated with priorities in order to record their relevance for the project and to help to decide which features to implement first. For example, one dimension in the feature diagram of a matrix is its *shape*, e.g. *rectangular*, *triangular*, and *Toeplitz*. We could assign a lower priority to Toeplitz than to rectangular or triangular since Toeplitz is "more exotic" than the other two and it can also be represented using the rectangular shape.

- *Implementation scoping*: The first phase of domain design is implementation scoping, whose purpose is to determine which features will be implemented first. This decision is based on the priorities assigned to variable features in feature models.

The assigned priorities may change over the course of a project. Thus, we have to update the feature models accordingly.

We do not prescribe any specific schema for assigning priorities. However, please note that priorities can potentially conflict with the constraints of a feature model since constraints define dependencies between features. Thus, for example, given the features f_1 and f_2, if there is a constraint requiring the inclusion f_2 whenever f_1 is included, the priority of f_2 has to be at least as high as the priority of f_1.

5.4.4 Availability Sites, Binding Sites, and Binding Modes

Availability site describes when, where, and to whom a variable feature is available. An available variable feature has to be bound before it can be used. Binding site describes when, where, and by whom a feature may be bound (and unbound, if applicable). Binding mode determines whether a feature is statically, changeably, or dynamically bound.

Before describing availability sites, binding sites, and binding modes in more detail, we first introduce the concept of sites.

5.4.4.1 Sites

A *site* defines the when, where, and who for a domain. Each domain may have its own site model, which may be arbitrarily complex. A simple site model might consist of a number of predefined times, usage contexts (e.g. different usage contexts of a system and/or usage locations of a component within a system), and a stakeholder model (e.g. users, customers, developers, etc.).

Based on a generic product lifecycle, important times include construction time, compile time, debug time, load time, runtime, and post runtime. However, sometimes we might want to define sites relative to product-specific workflows, use cases, etc.

5.4.4.2 Availability Sites

The variability available in a system or a component usually depends on its current site. We can model this by annotating each variable feature with its availability sites, i.e. the sites at which the feature is available for selection. For example, using availability site annotations, we can specify which items of a menu will be shown to whom, at what time, and in which context.

Please note that availability sites may potentially conflict with feature constraints. Thus, for example, given the features f_1 and f_2, if there is a constraint requiring the inclusion f_2 whenever f_1 is included, the set of availability sites of f_1 has to be a subset of the availability sites of f_2.

5.4.4.3 Binding Sites and Binding Modes

An available variable feature has to be first bound before it can be used. Binding corresponds to selecting a variable feature — just like selecting an item from a menu of options.

Binding sites

We usually control binding by annotating variation points with binding sites, i.e. sites at which the variable subfeatures of a variation point may be bound; however, if more control is needed, we can also annotate the variable features themselves.

Binding mode

In addition to annotating a variation point or a variable feature with binding sites, we can also annotate them with a *binding mode*. We have the following binding modes.

- *Static binding*: A statically bound feature cannot be rebound.
- *Changeable binding*: A changeably bound feature can be rebound.
- *Dynamic binding*: In this mode, a feature is automatically bound before use and unbound after use. Dynamic binding is useful in cases where we have to switch features at a high frequency.

5.4.4.4 Releationship Between Optimizations and Availability Sites, Binding Sites, and Binding Modes

The knowledge of binding site may be used to reduce the memory footprint of an application. For example, certain features my be needed only for certain application versions. Obviously, we do not want to link unused features to an application.[41] This can be achieved by applying technologies such as configuration management, pre-processors, generators, and static configuration using parameterized classes (e.g. C++ templates). If different features are needed at different times during runtime, we may want to use dynamic linking. Binding site modeling is especially relevant for dynamic and distributed architectures supported by Java technology, where we have to pay a special attention to network bandwidths and the widely differing resources available to different users.

Binding mode tells us something about the stability of a configuration and we can use this knowledge to optimize for execution speed. For example, if a feature is bound statically, we know that the feature cannot be rebound and we can optimize away any dispatching code, indirection levels, etc. In the case a feature should be bound statically at compile time, we can use implementation techniques such as static method binding, parameterized classes, static metaprogramming, partial evaluation at compile time, etc. In the case of changeable binding, it is useful to collect statistics such as frequencies of rebounds and average time between rebounds for different features. Based on these statistics, we can decide whether to apply runtime optimizations for certain features. Finally, if we need maximum flexibility, we have to use dynamic binding. Implementation techniques for dynamic feature binding include dynamic method binding, flags, dispatch tables, interpreters, and dynamic reflection.

5.5 Relationship Between Feature Diagrams and Other Modeling Notations and Implementation Techniques

Feature diagrams allow us to express variability at an abstract level. As stated, the variability specified by a feature diagram is implemented in analysis, design, and implementation models using different variability mechanisms. We already discussed some variability mechanisms in Table 7. For example, variability mechanisms for use cases include parameters, templates, extends relationships, and uses relationships. Variability mechanisms for class diagrams include inheritance, parameterization, dynamic binding, and cardinality ranges.

The following example illustrates the point that feature diagrams express variability at a more abstract level than class diagrams. Figure 36 shows a feature diagram of a simple car. The car consists of a car body, transmission, and an engine. The transmission may be either manual or automatic. Furthermore, the car may have a gasoline engine or an electric engine, or both. Finally, the car may pull a trailer.

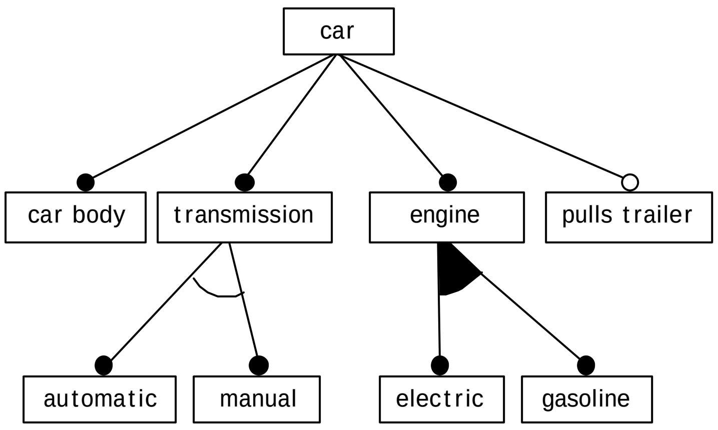


Figure 36 *Feature diagram of a simple car*

Figure 37 shows one possible implementation of our simple car as a UML class diagram. The car is represented as the parameterized class Car, where the transmission parameter implements the transmission dimension. CarBody is connected to Car by a part-of relationship. The optional Trailer is connected to Car by the association pulls. The cardinality range 0..1 expresses optionality. Finally, cars with different engine combinations are implemented using inheritance (classes ElectricCar, GasolineCar, and ElectricGasolineCar). If there were additional constraints between features in the feature model, we could have implement them as UML constraints (e.g. using the UML Object Constraint Language [Rat98b]).

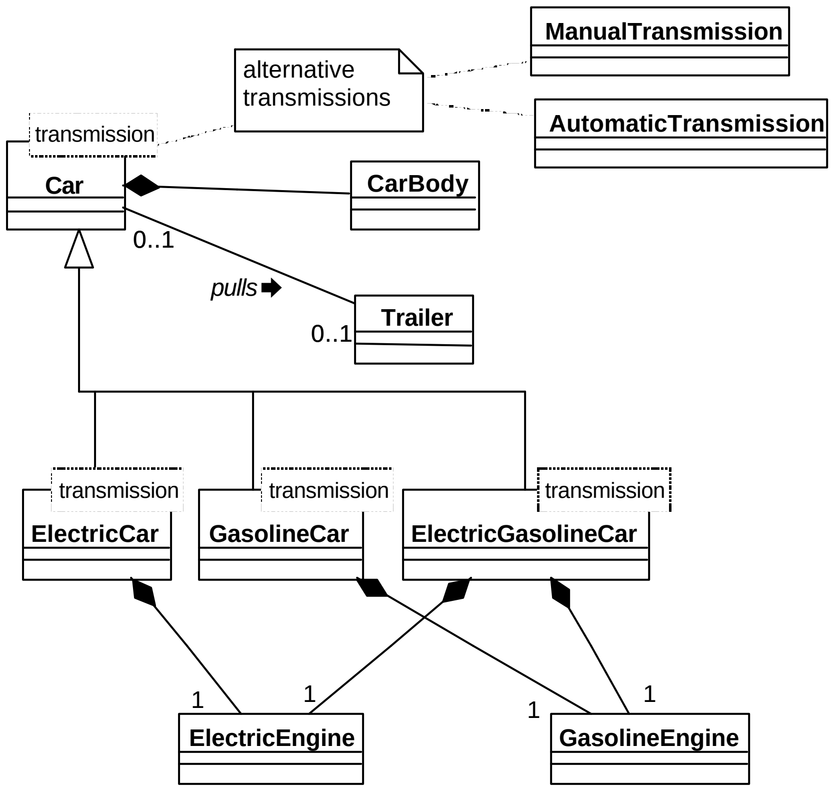


Figure 37 *One possible implementation of the simple car from Figure 36 using a UML class diagram*

Obviously, the implementation in Figure 37 is just one of many possible implementations. For example, we could use dynamic parameterization (see Section 5.5.5) to parameterize transmission instead of static parameterization. Another possibility would be to enlarge the inheritance hierarchy to accommodate different transmission and engine combinations. Thus, we see that the feature diagram in Figure 36 represents the variability of our car abstractly, i.e. without committing to any particular variability mechanism.

In the following five section, we investigate a number of important variability mechanisms available in current OO programming languages, namely single inheritance, multiple inheritance, parameterized inheritance, static parameterization, and dynamic parameterization. In Chapter 7, we will see that these variability mechanisms cause unnecessary complexity when used for parameterizing certain features. In particular, they have a difficult job parameterizing *crosscutting* aspects. What this means and what other techniques can be used instead is discussed in Chapter 7.[42]

5.5.1 Single Inheritance

Single inheritance may be used as a static, compile time variability mechanism.[43] It is well suited for implementing statically bound, non-simultaneous, singular variation points. For example, Figure 39 shows the implementation of the dimension from Figure 38. Each subclass may add some attributes and methods specific to it, e.g. ΔEmployee indicates the attributes and methods specific to Employee.

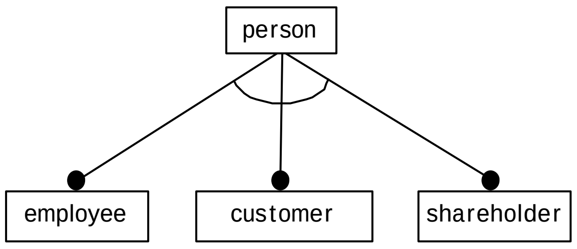


Figure 38 *Example of a dimension*

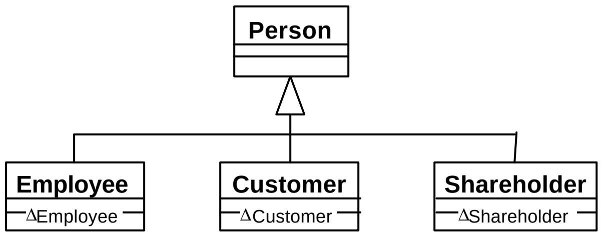


Figure 39 *Implementation of a dimension as an inheritance hierarchy*

We can also implement an nonsingular variation point using single inheritance; however, we will have to implement some features more than once. For example, Figure 41 shows the implementation of an extension point with or-features from Figure 40. Please note that ΔShareholder is implemented in three classes (Shareholder, ShareholderCustomer, and ShareholderEmployeeCustomer) and ΔEmployee in other three classes (Employee, EmployeeCustomer, and EmployeeShareholder).

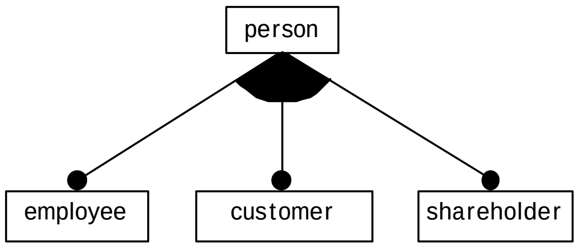


Figure 40 *Example of an extension point with or-features*

Similarly, if we were to implement a feature diagram containing two or more simultaneous variation points using single inheritance, the resulting inheritance hierarchy will also contain duplicate feature implementations. This is illustrated in Figure 42 and Figure 43.

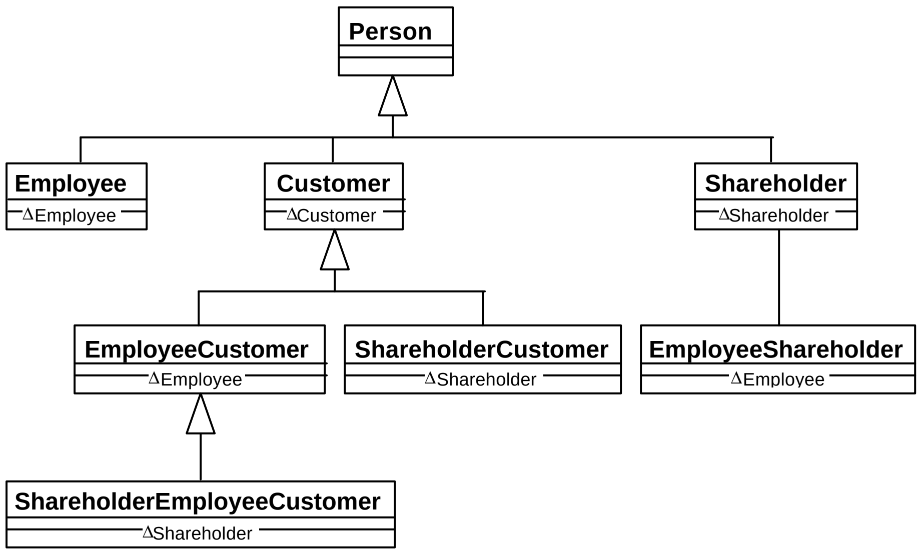


Figure 41 *Implementation of an extension point with or-features as a single inheritance hierarchy*

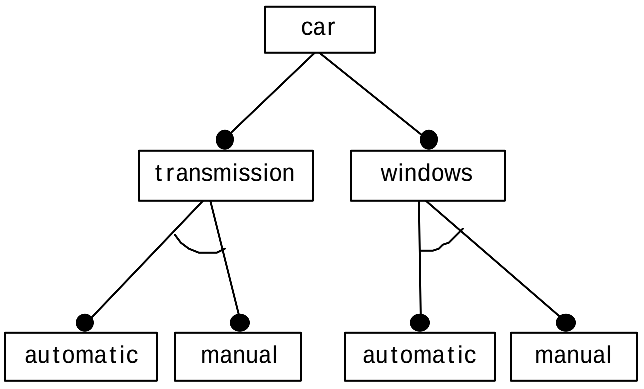


Figure 42 *Feature diagram with two simultaneous dimensions*

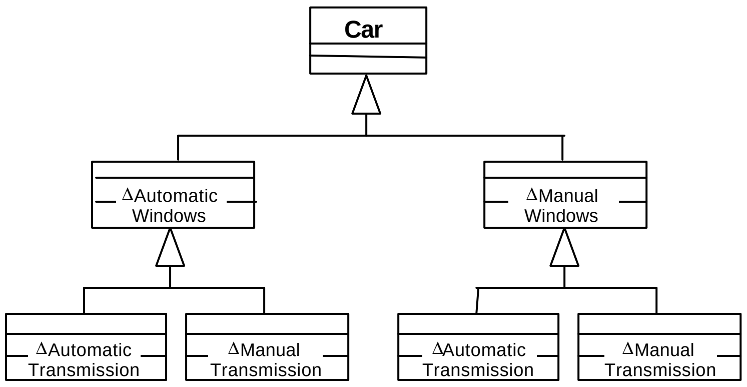


Figure 43 *Implementation of two simultaneous dimensions as a single inheritance hierarchy (the names of the subclasses are not shown)*

In general, a feature diagram can be implemented as a single inheritance hierarchy without feature duplication if and only if the diagram contains either (i) no variation points, or (ii) exactly one variation point which is singular, or (iii) more than one variation point, all of which are singular and non-simultaneous. A feature diagram cannot be implemented as a single inheritance hierarchy without feature duplication if and only if the diagram contains at least one nonsingular variation point or at least two simultaneous singular variation points.

In cases where the use of single inheritance causes feature duplication, we should consider using other variability mechanisms, such as multiple inheritance, parameterized inheritance, static parameterization, or dynamic parameterization.

5.5.2 Multiple Inheritance

The extension point with or-features from Figure 40 may also be implemented as a multiple inheritance hierarchy. This is shown in Figure 44. Please note that we do not have to duplicate features. The classes ΔEmployee, ΔCustomer, and ΔShareholder are referred to as mixins. Unfortunately, the multiple inheritance hierarchy has more complicated relationships than the single inheritance hierarchy in Figure 41. A more flexible solution is to use parameterized inheritance (see the following section).

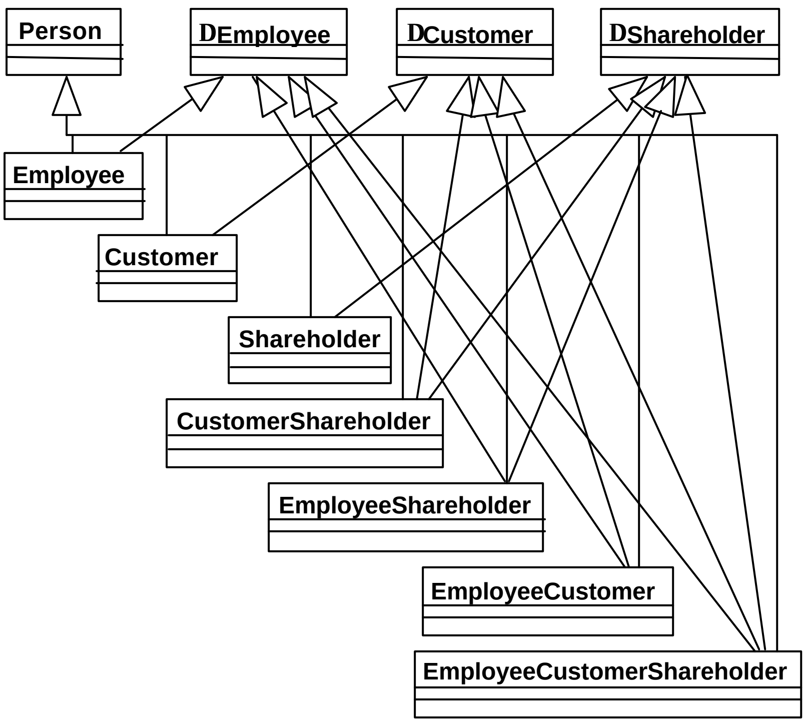


Figure 44 *Implementation of an extension point with three or-features as a multiple inheritance hierarchy*

5.5.3 Parameterized Inheritance

C++ allows us to parameterize the superclass of a class. We refer to this language feature as *parameterized inheritance*. Parameterized inheritance represents an attractive alternative to multiple inheritance for implementing statically bound extension points. We will use parameterized inheritance in Sections 6.4.2.4, 8.7, and 10.3.1 extensively. An implementation of the extension point with three or-features from Figure 40 using parameterized inheritance is shown in Figure 45. The mixin classes are parameterized with their superclasses. We can compose them to implement any of the six relevant composite classes. In C++, we could implement Employee as follows:

```
template<class superclass>
class Employee : public superclass
```

```
{
  //employee members...
};
```

We would implement Customer and Shareholder in an analogous way. Given Person and the mixin classes, ShareholderCustomerEmployee (shown in Figure 45) can be defined as follows:

```
Shareholder<Customer<Employee<Person> > >
```

and ShareholderEmployee:

```
Shareholder<Employee <Person> >
```

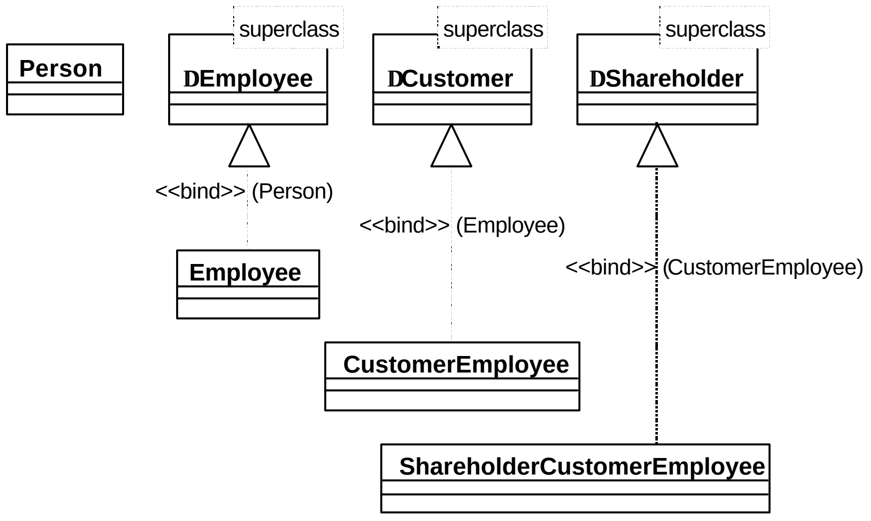


Figure 45 *Implementation of an extension point with three or-features using parameterized inheritance (the classes Customer, Shareholder, and ShareholderEmployee and ShareholderCustomer are not shown)*

5.5.4 Static Parameterization

Parameterized classes (e.g. templates in C++) are well suited for implementing statically bound, simultaneous and non-simultaneous dimensions. Figure 46 shows the implementation of two simultaneous dimensions from Figure 42 using static parameterization. Since UML does not provide any notation for associating parameters with candidate parameter value classes, we use UML *notes* for this purpose (e.g. alternative transmissions).

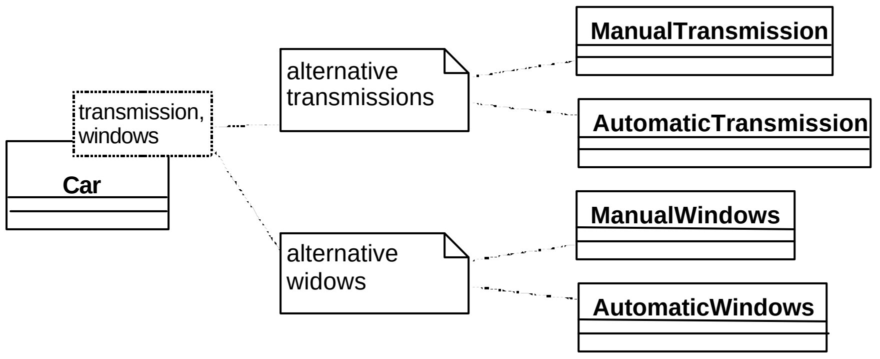


Figure 46 *Implementation of two simultaneous dimensions using static parameters*

5.5.5 Dynamic Parameterization

If we use dynamic method binding, a variable can hold objects of different classes at runtime. We refer to this mechanism as *dynamic parameterization*. In C++, the classes of the varying objects must have a common superclass which declares a virtual interface. This is shown in Figure 47. The class diagram implements the feature diagram with two simultaneous dimensions from Figure 42. In a Smalltalk implementation, the classes Transmission and Windows are not needed. In a Java implementation, we would implement Transmission and Windows as interfaces.

Dynamic parameterization should be used only if dynamic binding is required (or if changeable binding is required and no other appropriate technology is available).[44] Otherwise, we should use static parameterization since it avoids binding overhead and the inclusion of unused alternative features in the application.

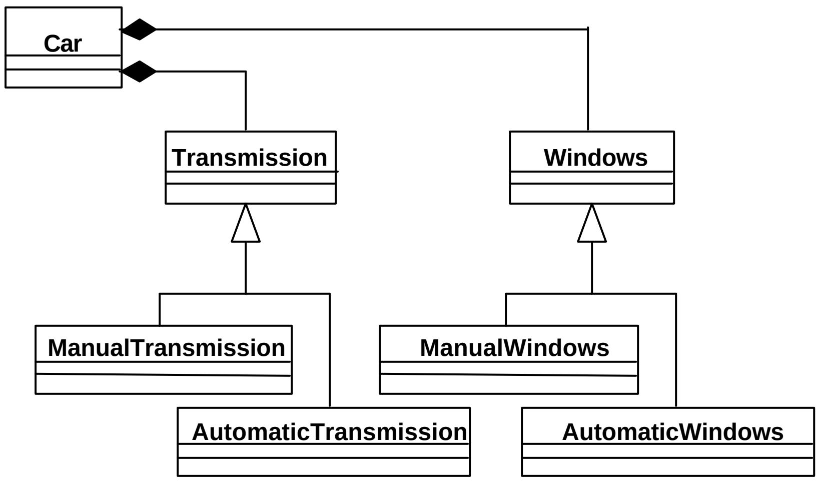


Figure 47 *Implementation of two simultaneous dimensions using dynamic parameterization*

5.6 Implementing Constraints

Feature models contain not only mandatory and variable features, but also dependencies between variable features. These dependencies are expressed in the form of constraints and default dependency rules. Constraints specify valid and invalid feature combinations. Default dependency rules suggest default values for unspecified parameters based on other parameters.

Constraints and default dependency rules allow us to implement automatic configuration. For example, in addition to our feature diagram of a car (see Figure 36), we could also have an extra feature diagram defining the three high-level alternative features of a car: *limousine*, *standard*, and *economy*. Furthermore, we could have the following vertical default dependency rules relating the three high level features and the variable detail features from Figure 36:

- limousine implies automatic transmission and electric and gasoline engines;
- standard implies automatic transmission and gasoline engine;
- economy implies manual transmission and gasoline engine.

Given these default dependency rules, we can specify a car with all extras as follows: *limousine and pulls a trailer*.

Thus, just as we need variability mechanisms in order to implement feature diagrams in other models, we also need means of implementing constraints.

If we use the UML for analysis and design models, we can express constraints using the UML Object Constraint Language [Rat98b]. For the concrete implementation of constraints, however, we have several possibilities. Configuration constraints at the file level can be managed by a configuration management system. Configuration constraints at the class and object level are best implemented as a part of the reusable software. In the case of dynamic configuration, we would implement them simply by writing runtime configuration code (e.g. the code configuring the “hot spots” of an OO framework at runtime). In the case of static configuration (e.g. configuring statically parameterized classes), we need static metaprogramming. Static metaprogramming allows us to write metacode which is executed by the compiler. Thus, we write static configuration code as static metacode. This configuration metacode and the parameterized classes to be configured constitute one program. When we send this program to the compiler, the compiler configures the classes by executing the configuration metacode and then compiles the configured classes, all in one compiler run. We will discuss static metaprogramming in C++ in Chapter 8. If our programming language does not support static metaprogramming, we could still use a preprocessor. However, the problem with this approach is that a preprocessor usually does not have access to the programming language level (e.g. it cannot read the values of static variables, access class metainformation, etc.). Finally, we could also implement a dedicated generator.

Simple constraints can be implemented directly as imperative code. However, if we have to manage a large number of complicated constraints, it may be necessary to use a constraint solver engine.

5.7 Tool Support for Feature Models

As of writing, feature models are not supported by the commercially available and widely used CASE tools. In order to adequately support feature modeling, a CASE tool should

- support the feature diagram notation,
- help to manage all the additional information required by feature models, and
- allow us to hyperlink feature models with other models (e.g. linking semantic descriptions of features with other models, traceability links to other models, etc.).

An additional useful feature would be a constraint management facility (including consistency checking) for complex feature models. As noted in [GFA98], some kind of integration with configuration management would also be useful.

Given the growing acceptance of the UML in the software industry, it would be certainly useful to extend UML with the feature diagram notation.

Griss et al. describe in [GFA98] an approach for implementing the feature diagram notation using the predefined UML modeling elements. They implement features as classes with the stereotype <<feature>>. Furthermore, they use an *optionality attribute* to indicate whether a feature is optional or not and introduce the special node type called “variation point”, which corresponds to a dimension in our terminology. Features are related using the *composed_of relationship* and, in the case of dimensions, the *alternative* relationship. Finally, they have a *binding time flag* indicating whether a dimension is bound at use time (e.g. runtime) or at reuse time (e.g. compile or construction time).

Thus, the approach in [GFA98] does not distinguish between different kinds of variation points (e.g. different kinds of dimensions, extension points, and inhomogeneous variation points; cf. Section 5.4.1.7). Second, it does not allow inhomogeneous variation points with alternative subfeatures in the diagram (since there is a special dimension node type). It also does not support or-features. Furthermore, it does not distinguish between availability sites, binding sites, and binding mode.

A very nice feature of the approach by Griss et al. is the possibility of expanding and collapsing features in the diagram. A collapsed feature is represented by an icon. In the expanded version, on the other hand, some of the additional information (e.g. feature category, semantic description, etc.) can be directly viewed and edited.

From our experience, we find it useful to be able to draw all kinds of inhomogeneous variation points in a feature diagram since, in some situations, they allow us to create more concise and natural diagrams. One possibility to extend the approach by Griss et al. to handle inhomogeneous variation points would be to use constraints between relationships. In UML, we can draw a dashed line connecting a number of associations and annotate it with a constraint that refers to all these associations. Thus, we could use such dashed lines annotated with OR or with XOR in order to represent edge decorations. Indeed, the OR-annotated dashed line connecting associations is already predefined in UML. Associations connected by such a dashed line are called *or-associations*. Unfortunately, or-associations have actually XOR-semantics.

To our taste, the above approach for implementing a feature diagram notation based on stereotyped classes is an instance of “diagram hacking”. In UML, stereotypes are used to define new modeling elements based on existing ones such that the properties of the base elements are inherited. However, as we discussed in Chapter 2 and in Section 5.3, concepts and features are not classes (although some of them may be implemented as classes). While creating a stereotype for concepts and features derived from classes allows us to inherit some useful properties, we also inherit undesired properties, e.g. class properties such as being able to have attributes and methods. We find this approach quite confusing.

A more adequate approach is to extend the UML metamodel with concepts and features. Of course, we can make the UML feature diagrams look exactly the way as defined in this chapter. Unfortunately, as of writing, only very few CASE tools support editing their own metamodels.

Due to this inadequate support for feature modeling by current CASE tools, we maintained our feature models for the matrix computations library described in Chapter 10 in a word processor.

5.8 Process of Feature Modeling

So far we have discussed how to represent feature models. In the rest of this chapter, we describe how to perform feature modeling.

5.8.1 How to Find Features?

In this section, we address the following three important questions:

- What are the sources of features?
- What are the strategies for identifying features?
- What are the general steps in feature modeling?

Sources of features include the following:

- existing and potential stakeholders,
- domain experts and domain literature,
- existing systems,
- pre-existing models (e.g. use-case models, object models, etc.), and
- models created during development.

Strategies for identifying features include both top-down and bottom-up strategies:

- Look for important domain terminology that implies variability, e.g. checking account vs. savings account, diagonal vs. lower triangular matrix, thread-safe vs. not thread-save component, etc. It is important to keep in mind that anything users or client programs might want to control about a concept is a feature. Thus, during feature modeling, we not only document functional features, e.g. operations such as addition and multiplication of matrices, but also implementation features, e.g. algorithm variants for implementing the operations, various optimizations, alternative implementation techniques, etc. In this aspect, Domain Analysis is different from classical software analysis, where the usual rule is to avoid analyzing any implementation issues.

- Examine domain concepts for different sources of variability, e.g. different stakeholders, client programs, settings, contexts, environments, subjective perspectives, aspects, etc. In other words, investigate what different sets of requirements these variability sources may postulate for different domain concepts (we discuss these issues in Section 5.8.2);

- Use *feature starter sets* to start the analysis. A feature starter set consists of a set of modeling perspectives for modeling concepts. Some modeling perspectives combinations are more appropriate for a given domain than for another. For example, for abstract data types in algorithmic domains, we use a starter set containing modeling perspectives such as attributes, operations, synchronization, memory management, optimizations, etc. (see Section 9.3.2.2). Other domains and concepts may require investigating other perspectives, such as distribution, security, transactions, historization, etc. Starter sets may also contain examples of features, sources of features, etc. Thus, starter sets are reusable resources capturing modeling experience. As stated, there will be different starter sets for different categories of domains. Ideally, feature starter sets are updated by each new project. We discuss feature starter sets in Section 5.8.2.

Feature starter set

- Look for features at any point in the development. As discussed before, we have high-level system requirements features, architectural features, subsystem and component features, and implementation features. Thus, we have to maintain and update feature models during the entire development cycle. We may identify all kinds of features by investigating variability in use case, analysis, design, and implementation models.

- We found it useful to identify more features that we initially intend to implement. This strategy allows us to “leave some room for growth”. Although we will not be able to identify all features that may be relevant in future, it is a big gain if we identify some of them. At some point in domain design, there should be an extra scoping activity where we actually decide which features to implement. Therefore, we should record priorities of variable features when we first document them. By having documented potential features, we will be able to develop more robust client and configuration interfaces, even if not all of the features will be implemented at first. We will see a concrete example of applying this strategy in Chapter 10.

Feature modeling is a continuos, iterative process with the following steps:

1. Record similarities between instances, i.e. common features, e.g. all accounts have an account number.

2. Record differences between instances, i.e. variable features, e.g. some accounts are checking accounts and some are savings accounts.

3. Organize features in feature diagrams. Feature diagrams allow us to organize features into hierarchies and to classify them as mandatory, alternative, optional, or-, and optional alternative features. We discussed feature diagrams in Section 5.4.1 in detail.

4. Analyze feature combinations and interactions. We may find certain combinations to be invalid (*mutual-exclusion constraints*), e.g. a collection cannot be *unordered* and *sorted* at the same time. We may discover dependencies allowing us to deduce the presence of some features from the presence of others (*requires constraints*), e.g. if a collection is *sorted*, it is also *ordered*. We may also find innovative combinations, which we did not thought of previously. Furthermore, when we investigate the relationship of two features, we may discover other features. For example, when we analyze different combinations of matrix shapes (e.g. rectangular, diagonal, triangular, etc.) and matrix element containers (e.g. array, vector, list, etc.), we realize that even for the same combination of shape and container, different layouts for storing the elements in the container are possible (e.g. a rectangular matrix can be stored in a two-dimensional array row- or column-wise). Therefore, we

introduce the new feature *storage format*. Similarly, by investigating the relationship between matrix operations and matrix shapes, we will realize that various *optimizations* of the operations due to different shapes are possible. We say that new features may emerge from the *interaction* of other features.

5. Record all the additional information regarding features such as short semantic descriptions, rationale for each feature, stakeholders and client programs interested in each feature, examples of systems with a given feature, constraints, availability sites, binding sites, binding modes, open/closed attributes, and priorities. All these concepts are explained in Section 5.4.2.

We refer to these steps as the "micro-cycle" of feature modeling since they are usually executed in small, quick cycles. To give you a concrete example of how to perform feature modeling, we describe how we actually came up with the feature models during the development of the matrix computations library described in Chapter 10:

> *Start with steps 1 and 2 in the form of a brainstorming session by writing down as many features as you can. Then try to cluster them and organize them into feature hierarchies while identifying the kinds of variability involved (i.e. alternative, optional, etc.). Finally, refine the feature diagrams by checking different combinations of the variable features, adding new features, and writing down additional constraints. Maintain and update the initial feature models during the rest of the development cycle. You may also start new diagrams at any point during the development.*

As the feature modeling process progresses, some new features may be recognized as special cases of old ones and other new features may subsume other old features. For example, we may have already documented the matrix shapes square, diagonal, lower triangular, upper triangular, bidiagonal, and tridiagonal. Later, we recognize that the shape *band* subsumes all these shapes since each of these special shapes can be described as the band shape with certain lower and upper bandwidth. However, this does not imply that we should replace the special shapes by the band shape in the feature model. The feature model should rather include all these shapes as well as the relationships between them. If there is a name for a certain property in the domain vocabulary, this usually indicates the relevance of this property. Obviously, properties which play an important role in the domain (e.g. certain shapes which are more common than others and/or are important for optimizing certain algorithms) should have unique names in the model.

Section 9.4.3 describes an approach for deriving reuser- and implementer-oriented feature diagrams from the analysis feature diagrams. Such an approach is important since the implementers of features have a different focus than the reusers of features. The reusers need features which allow them to specify their needs at the most adequate level of detail (which is different for different reusers or client programs). Implementers, on the other hand, decompose their solution into elementary, reusable pieces, which they can use and re-use within the implementation and across implementations of different product lines. Some of them may be too implementation oriented to be part of the reuser-oriented feature diagrams.

During development, we have to maintain traceability links from the feature diagrams to other models and update the domain dictionary whenever we introduce new features.

A feature diagram and the additional information constitute a feature model. We discussed feature models in Section 5.4.

5.8.2 Role of Variability in Modeling

Now that we know how to perform feature modeling, we would like to investigate how feature modeling fits into other modeling activities. However, before we address this question, we first need to introduce the principle of separation of concerns (Section 5.8.2.2) and three important decomposition techniques based on this principle (Section 5.8.2.3). We then discuss the integration of feature modeling and the decomposition techniques in Section 5.8.2.1

5.8.2.1 Separation of Concerns and Perspectives

One of the most important principles of engineering is the *principle of separation of concerns* [Dij76]. The principle acknowledges that we cannot deal with many issues at once, but rather with one at a time. It also states that important issues should be represented in programs intentionally (explicitly, declaratively) and well localized. This facilitates understandability, adaptability, reusability, and the many other good qualities of a program since intentionality and localization allow us to easily verify how a program implements our requirements.

Unfortunately, the relevant issues are usually dependent and overlapping since they all concern one common model (i.e. our program being constructed). Thus, if we try to represent all these issues explicitly and locally, we will introduce a lot of redundancies. This causes maintenance problems since we have to make sure that all the redundant representations are consistent. Also, the overall model becomes very complex since we have to maintain all the knowledge relating the different representations.

On the other hand, if we choose a less redundant representation of our solution to a problem, some issues will be well localized and some others will not. This is similar to the idea of representing a signal in the time domain and in the frequency domain (see Figure 48). In the time domain, we can explicitly see the amplitude of the signal at any time, but we cannot see the component frequencies. In the frequency domain, on the other hand, we can see the component frequencies, but we cannot see the amplitude of the whole signal at a given time. If we keep both representations, we introduce redundancy.

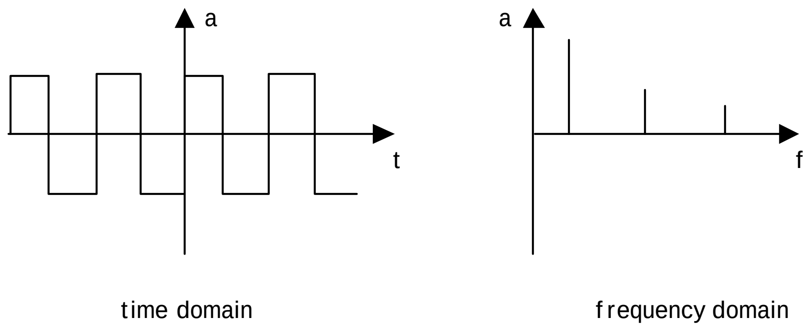


Figure 48 *Representation of a signal in the time and in the frequency domain (from [Kic98])*

Ideally, we would like to store our solution to a problem in some efficient representation, i.e. one with minimal redundancies, and have some supporting machinery allowing us to extract any perspective on the model we might need. It should be possible to make changes in the extracted model and have the machinery update the underlying model automatically for us. (Of course, as we make more and more changes, the machinery might need to transform the underlying representation into some other form to reduce the accumulated redundancies.)

Unfortunately, our ideal solution is impractical — at least given today's technology. First, we would have to model an enormous amount of formal knowledge in order to be able to extract any desired perspective on a large system automatically. Second, we would need some very efficient transformation and deductive reasoning machinery (most probably involving AI techniques) to perform the extraction. (Nevertheless, transformation systems and other tools, such as some CASE tools, already allow us to compute different perspectives on software models, e.g. extracting control and data flow.)

We have to look for more practical solution to address our problems: The purpose of today's modeling techniques is to develop models which meet the requirements (i.e. functional requirements and qualities such as performance, throughput, availability, failure safety, etc.) and, at the same time, strike a balance between

- separation of concerns (i.e. having most important issues as localized as possible),
- complexity of implementation (trying localize certain issues may also increase the overall complexity of the implementation; see Section 7.6.2),
- minimal redundancy, and
- ability to accommodate anticipated change and also unanticipated change to some degree.[45]

The result of such a balance is what we might call a "clean" and adaptable code. Different decomposition techniques help us to achieve this balance.

5.8.2.2 Decomposition Techniques

We distinguish between two important kinds of decomposition of a concept[46]:

- *Modular decomposition*: Modular decomposition involves decomposing systems into *hierarchical (i.e. modular) units* (e.g. modules, components, objects, functions, procedures, etc.). The word hierarchical indicates that a unit may contain other units, etc. The boundaries of such units are drawn in a way, such that they encapsulate some cohesive "model neighborhoods". The goal is to achieve high cohesion within the units and minimal coupling between the units.
- *Aspectual decomposition*: The main idea behind aspectual decomposition is to organize the description of a concept (e.g. a system, a domain, a component, a function, etc.) into a set of perspectives, where each perspective concerns itself with a different aspect and none of which is itself sufficient to describe the entire concept. An important property of such decomposition is that each perspective yields a model with a different structure and all of these models refer to the same concept. As a consequence, there are locations in one model that refer to locations in other models, which is referred to as *crosscutting*. Examples of aspects include interactions, algorithms, data structures, data flow, synchronization, error handling, memory management, and historization. More examples of aspects are given in Chapter 7. Most of the existing modeling techniques apply some form of aspectual decomposition. For example, the UML [Rat98a] deploys different diagrams to describe different aspects of systems (e.g. class diagrams, use cases, interaction diagrams, collaboration diagrams, activity diagrams, and state diagrams). It is important to note that even if we stay within one aspect, we still want to reduce its perceived complexity by dividing it into modular units.

The basic difference between aspects and modular units[47] is shown in Figure 49. The drawing on the left shows that modular units are cleanly encapsulated and organized into a hierarchy. The drawing on the right shows that an aspect *crosscuts* a number of modular units.

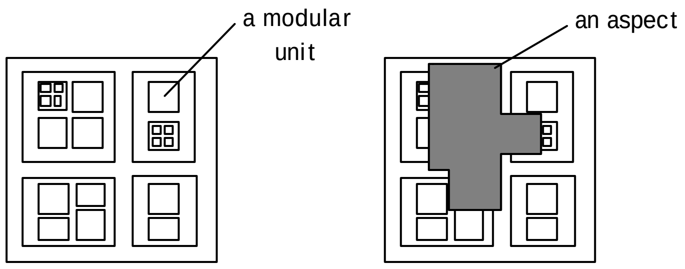


Figure 49 *Modular vs. aspectual decomposition*

Thus, the quality of being an aspect is a relative one: a model is an aspect of another model if it crosscuts its structure. The aspect shown in Figure 49 is an aspect with respect to the hierarchical structure also shown in this figure. However, at the same time, the aspect could be a modular unit of another hierarchical structure not shown in the figure.

Figure 50 shows another view on an aspect: Model A is an aspect of model C since it refers to many locations in C. For example, C could be a class implementing some abstract data structure and A could be a synchronization specification referring to the methods of the abstract data structure.

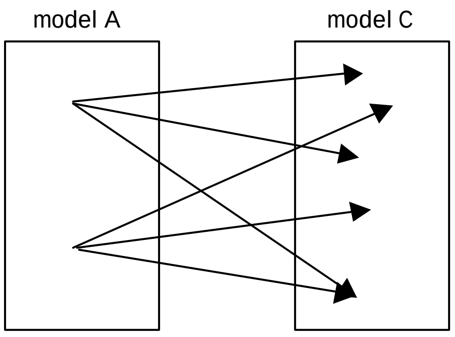


Figure 50 *Example of an aspect*

Modular decomposition allows us to do simple refinement by adding structure that never crosses the boundaries of the already established modular units. Whenever we have to add structure that crosses these boundaries, we are actually applying aspectual decomposition.

Modular decomposition and aspectual decomposition complement each other and should be used in combination. They correspond to the natural modeling strategies of humans: we deploy both investigating things from different perspectives and dividing them into hierarchies.

Aspectual decomposition is investigated in the area of Aspect-Oriented Programming (AOP), which we discuss in Chapter 7. As we stated above, aspectual decomposition is quite common in software development (e.g. the different aspects used in analysis and design methods). However, the AOP research gives aspectual decomposition some new perspectives:

- AOP encourages the introduction of new aspects rather than adhering to a small set of general aspects (as in the case of existing OOA/D methods).
- It emphasizes the need for specialized aspects and specialized combinations of aspects for different categories of domains.
- It postulates the need to support aspects not only in design models, but also in the implementation. In particular, there is the need for new composition mechanisms and aspect languages (ideally, implemented as modular language extensions; see Sections 6.4.3.1 and 7.6.3).
- It concentrates on achieving quality improvements due to aspectual decomposition, such as reduced code tangling, better understandability, maintainability, adaptability, reusability, etc.

It is important to note that aspects represent perspectives which proved to be useful in constructing past systems, i.e. they are based on experience. Thus, different systems of aspects record modeling experience in different categories of domains.

5.8.2.3 Variability in Modeling

In this section, we investigate the question how variability modeling and the different decomposition techniques fit together.

Variability plays an important role in our modern society and strategies for coping with variability constitute an essential prerequisite for success. This situation has been accurately characterized by Reenskaug [Ree96]:

> *"It is popular to claim that in our modern society, change is the only constant factor. Enterprises have to be able to adapt to a continuously changing environment.*
>
> *We believe this to be both true and misleading. It is true in the sense that an enterprise has to keep changing to adapt to changes in its environment. It is also true in the sense that a business has to change the core of its operations to accommodate the transition from manual to computer-based information processing; that it has to move its rule-based operations from people to computer; and that it has to cultivate the creativity, responsibility, and problem-solving capabilities of its human staff.*
>
> *It is misleading in the sense that you cannot change everything all the time. So the challenge to the evolving enterprise is to identify what can be kept stable and what has to be kept fluid. The stable elements of a business form an infrastructure on which it can build a light and rapidly changing superstructure."*

Obviously, software is another point in case. We have to explicitly address variability during modeling. As we have seen so far, this is even more important for reusable software.

If the things that we model contain variability, the variability will emerge in some aspects and modules somehow. In other words, the variability aspect crosscuts aspects and modules. Indeed, variability is just another aspect of the reusable software.

However, we stated that reusable software usually contains a lot of variability. Thus, we have to apply variability modeling *in coordination* with other aspectual and modular decomposition techniques. By applying them in a coordinated fashion, we make sure that decomposition decisions are influenced by variability needs. Adding variability "after the fact" – just as adding any other aspect – may cause significant code tangling.

Before we move on, let us first clarify, what we exactly mean by variability. We talk here about the variability of *how* computation is done, not about the fact that data gets modified all the time in a running program.[48] We have two kinds of variability sources:

- *internal variability sources* and
- *external variability sources*.

The internal source of variability is the evolution of state. For example, if a collection grows beyond a certain limit, we might want to switch to a different sorting algorithm.

External variability sources are more versatile, e.g. different stakeholders (including different users, developers, and customers), different client programs, different environments, usage settings, changing requirements, etc.

In previous sections, we discussed feature modeling as an effective technique for modeling variability. The question now is how we can integrate feature modeling and the other decomposition techniques.

An example of such an integration is DEMRAL (Domain Engineering Method for Algorithmic Libraries), which we describe in Chapter 9 In DEMRAL, feature modeling and the other decomposition techniques are applied in a coordinated fashion as follows:

- *Modular decomposition*: The first activity in DEMRAL Domain Modeling is identifying key concepts. DEMRAL specializes in modeling two categories of concepts: abstract data types (ADTs) and algorithms. The key ADTs and key algorithm families are later encapsulated in separate packages. New objects and algorithms may be also identified during feature modeling. In other DE methods, we will look for other kinds of concepts, such as workflows, clients, servers, agents, interactions, use cases, etc.

- *Aspectual decomposition*: Aspectual decomposition is fundamental to DEMRAL. It is performed in parallel with feature modeling. The idea is to provide so-called feature starter sets, which are basically sets of aspects suitable for algorithmic domains (see Section 9.3.2.2). The modeling proceeds by modeling each aspect separately in an iterative fashion. Other methods will define their own starter sets appropriate for other kinds of concepts.

- *Feature modeling*: Feature modeling is performed for each aspect in separation in an iterative fashion. Features relevant for the configurability aspect are organized into feature diagrams. Other features may lead to the development (or reuse) of specialized aspect languages or modular language extensions. Relationships between features of different aspects are also modeled.

- *Subjective decomposition*: Subjective decomposition is based on modeling different subjective perspectives of different stakeholders (e.g. users, developers) on a system or a domain. Subjective decomposition has been popularized by the work of Harrison and Ossher on Subject-Oriented Programming (SOP). We discuss SOP in Section 7.2.1. It is important to note that subjects may crosscut many aspects and modules of a system. Subjectivity is accounted for in DEMRAL by considering different stakeholders and annotating features by their stakeholders.

The interaction of the different decomposition techniques is shown in Figure 51.

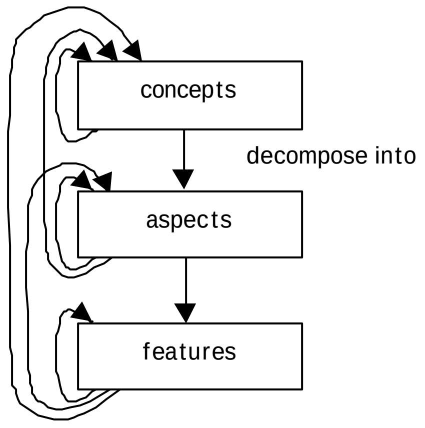


Figure 51 *Interaction of different decomposition techniques in DEMRAL*

5.9 References

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Second edition, Benjamin/Cummings, Redwood City, California, 1994

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976

[DP98] P. Devanbu and J. Poulin, (Eds.). *Proceedings of the Fifth International Conference on Software Reuse (Victoria, Canada, June 1998)*. IEEE Computer Society Press, 1998

[GFA98] M. L. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In [DP98], pp. 76-85, see http://www.intecs.it

[Höl94] U. Hölzle. Adaptive Optimization in SELF: Reconciling High Performance with Exploratory Programming. Ph.D. Thesis, Department of Computer Science, Stanford University, 1994, see http://self.stanford.edu

[JGJ98] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley Longman, May 1997

[Kic98] G. Kiczales. Aspect-Oriented Programming. Transparencies for the invited talk at the OOP'98 in Munich, see www.parc.xerox.com/aop

[Rat98a] Rational Software Corporation. Unified Modeling Language (UML), version 1.1. Documentation set available from http://www.rational.com/uml/documentation.html

[Rat98b] Rational Software Corporation. Object Constraint Language Specification. Documentation available from http://www.rational.com/uml/documentation.html

[Ree96] T. Reenskaug with P. Wold and O.A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning, 1996

Part III

IMPLEMENTATION TECHNOLOGIES

Chapter 6 Generators

6.1 Introduction to Generators

The main goal of generators is to produce software systems[49] from higher-level specifications. Generators have been studied in various research communities, most notably in

- knowledge-based software engineering or program synthesis community (as one of their primary subjects, e.g. [Bal85, RW90, Smi90, Pai94, SG96, AKR+97]),
- software reuse community (as means of implementing domain models, e.g. [Nei80, BO92, ADK+98]), and
- formal specification community (as means of implementing formal specifications, e.g. [BEH+87, HK93, SJ95]).

It is worth noting that the reuse community has traditionally had a strong focus on practical applications (e.g. [SSGRG, BT, IPH]).

Three important issues addressed by generators are

- raising the intentionality of system descriptions,
- computing an efficient implementation, and
- avoiding the library scaling problem [Big94].

As we already explained in Section 1.5, intentional descriptions directly represent the structure of a problem. Intentionality is usually achieved through domain-specific notations and we can implement such notations using generators.

Generators bridge the wide gap between the high-level, intentional system description and the executable. In particular, the implementation has to meet certain performance requirements (e.g. execution speed, response time, memory consumption, utilization of resources, etc.). The challenge is that the structure of the specification is usually very different from the structure of the implementation: there is no simple one-to-one correspondence between the concepts in the specification and the concepts in the implementation and even a slight change in the specification might require a radically different implementation. Generators, unlike conventional programming languages and libraries, perform computations at generation time which allow them to achieve this non-trivial mapping.

The library scaling problem concerns the horizontal scaling of conventional libraries (also see Section 1.3.2). If a library implements each feature combination as a concrete component in order to achieve good performance, each new feature can potentially double the number of concrete components in the library. On the other hand, factoring libraries into components corresponding to features and composing them using function or method calls avoids the exponential growth of the library size, but results in poor performance. Generators allow us to avoid the performance penalties in the latter case, i.e. we achieve both good performance and linear library scaling. As we already explained, generators separate the modeling space from the implementation space, so that these spaces can have different structures. Efficient implementations are then computed at generation time by applying domain-specific optimizations and replacing, merging, adding, and removing components.

Generators are based on domain-specific models which define the semantics of the system specification language and also contain the knowledge of how to produce efficient implementations. Two major techniques are used during generation: *composition* and *transformation*. Generators based on composition are referred to as *compositional generators* [Bat96], whereas generators primarily using transformations are called *transformation systems*.[50]

6.2 Composition vs. Transformation

There are two fundamental ways of generating instances of a given concept: composition and transformation. We will illustrate them using an abstract example. Figure 53 shows us how to produce an instance of the concept *star* by composition and by transformation. A simple star can be described by specifying the following features (see Figure 52):

- the number of arms,
- the inner radius,
- the outer radius, and
- the angle describing the position of the first arm.

In the composition model, we glue together a number of components, whereas in the transformation model, we perform a number of transformations arriving at the desired result. In order to be able to generate different stars using the composition model, we need many concrete components of different sizes and shapes. This is illustrated in Figure 54, on the left. The circles are described just by the *inner radius*. The arms, on the other hand, are described by the three parameters *inner radius*, *outer radius*, and *number of points*. When generating a star, we need to compose one circle and four copies of one arm selected from Figure 54 according to the desired parameter values. In the transformation model, on the other hand, we do not need such large gallery of components, but just four transformations (see Figure 54, on the right).

We can organize the compositional model more effectively by using *generative components* instead of concrete components. A generative component accepts an abstract description of a concrete instance and generates the instance according to the description. Thus, instead of a whole gallery of concrete circles and arms, we just need two generative components: a generative circle component and a generative arm component (see Figure 55). The generative circle component takes *radius* as its parameter, and the generative arm component has the parameters *inner radius*, *outer radius*, and *angle* (angle describes the width of section of the inner circle serving as the bottom of the arm). The components are generative and not generic since they do not necessarily expect concrete components as their parameters (e.g. concrete shapes), but rather *abstract parameters* (e.g. radius, angle, etc.). In terms of software, an abstract parameter does not have any concrete implementation directly associated with it, but the implementation has first to be computed based on many parameters. For example, a performance specification is usually an abstract parameter. Abstract parameters are mapped to concrete components through *configuration knowledge*. We will explain this idea in a moment. Components expecting concrete parameters only are referred to as *concretely parameterized components*. For example, data structures and memory allocators are the typical parameters in the generic Standard Template Library [MS96]. These are concrete parameters since they have concrete implementations associated with them.

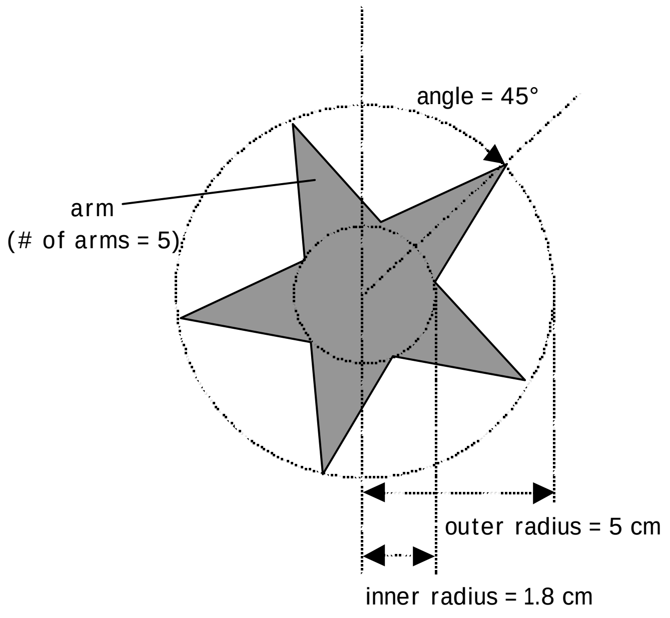


Figure 52 *Features of a simple concept of a star*

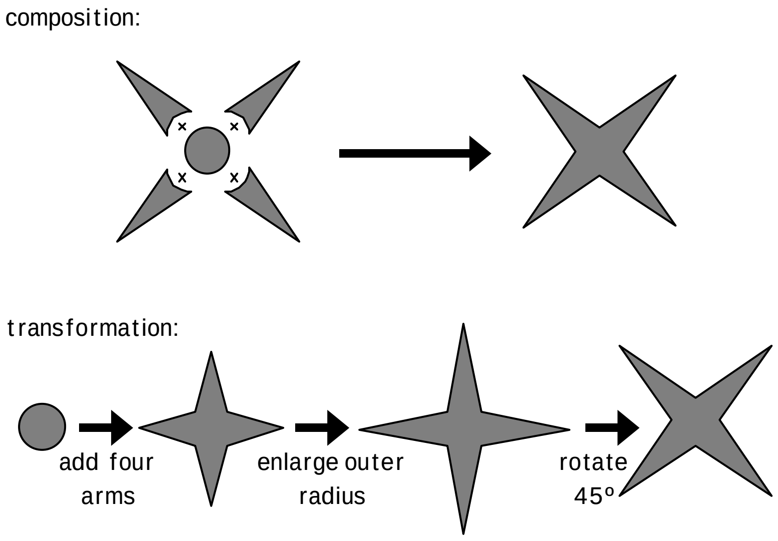


Figure 53 *Generating instances of the star concept by composition or transformation*

The component implementing a star in Figure 55 is a generative component itself. The features of the star concept became the parameters of this component. The star component uses the generative circle component and the generative arm component to generate the concrete components it needs. The parameters of the star component are mapped on to the parameters of the circle and the arm component. This mapping is done by the configuration knowledge of the star component. For example, the number of arms is used to calculate the angle parameter of the arm component: angle = 360 / number of arms. This is a *vertical constraint*, i.e. it maps between layers of abstraction. In addition to mapping between abstraction layers, configuration knowledge also contains constraints between parameters of one level, i.e. *horizontal constraints*. For example, the inner radius must be smaller than the outer radius, but grater than zero. Constraints between parameters can be used not only to verify the consistency of the parameters supplied to a component, but also to complete partial specifications by inferring unknown parameters from the known ones (whenever possible). The component assembler in

Figure 55 is a *generic* component taking the generated concrete components and assembling them into a star. The component takes also one (as we will see later — so-called horizontal) parameter, namely the *angle* describing where to attach the arms to the circle.

The generative circle and arm components can be implemented using composition or transformation technology. They could also contain other generative components, but, eventually, the model has to be mapped onto generic or concrete components or transformations, or any combination of these.

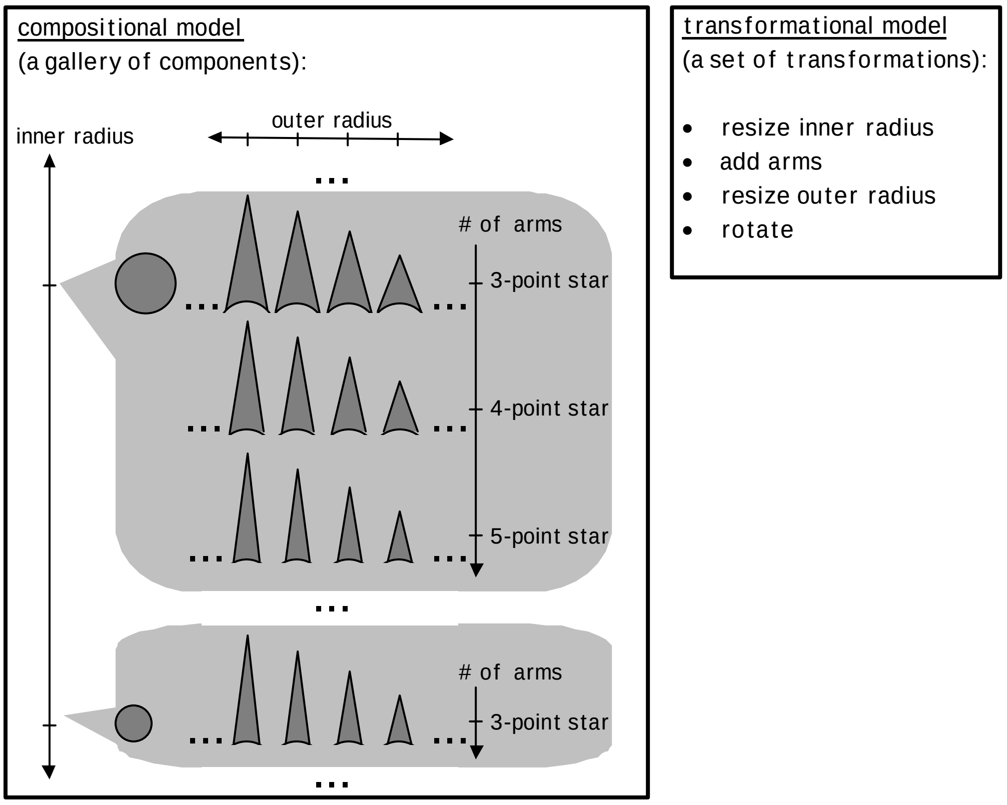


Figure 54 *Implementing the domain model of the star concept using the compositional model (on the left) and the transformational model (on the right)*

It is worth mentioning that composition can be seen as a special case of transformation. This is illustrated in Figure 56. Composition is sometimes referred to as *forward refinement* [Big97] since, during composition, we add new components and progressively approach the final result, i.e. components are never removed or replaced. In the more general transformational model, on the other hand, components can be added, replaced, removed, modified, etc., i.e. the progress is nonmonotonic. It is also important to point out that transformation, in contrast to composition, allows us to perform optimizations since they require structural changes. The best we can do with the compositional approach is to use inlining for gluing components together, but domain-specific optimization cannot be performed without transformations.

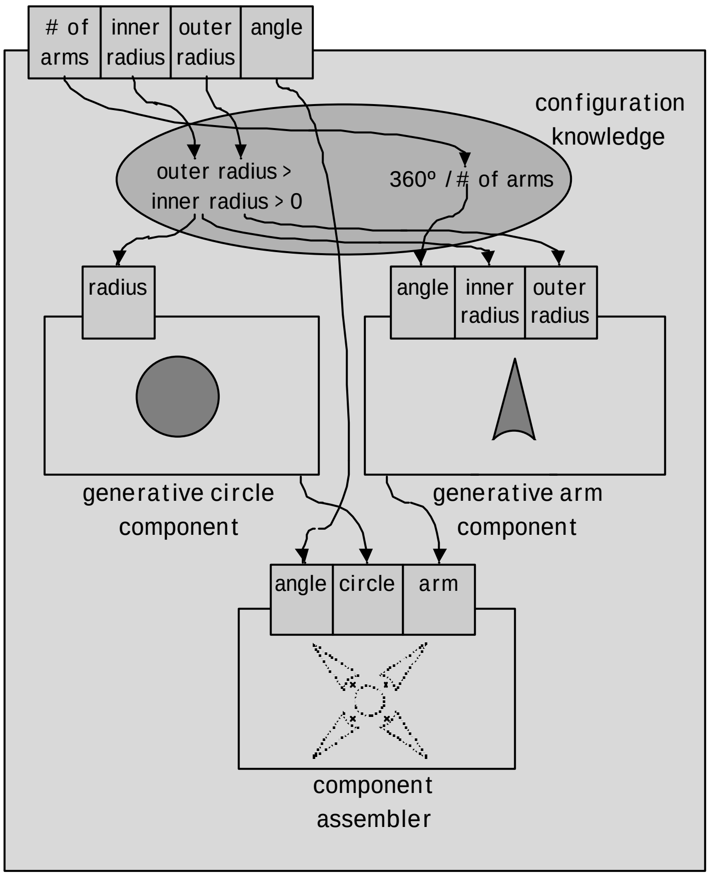


Figure 55 *Implementing the star concept using generative components*

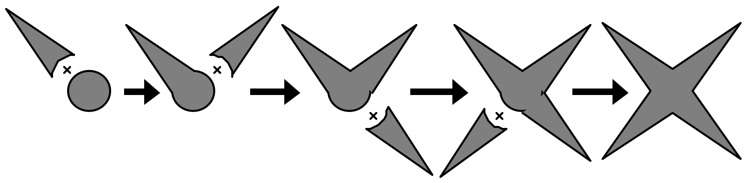


Figure 56 *Composition as a special case of transformation*

Whether a generator is primarily based on composition or on transformation, has profound consequences for its architecture. Most generators found in practice are predominantly compositional, e.g. GUI builders, visual programming environments, etc. They usually have some kind of graphical interface, where components can be connected through lines, and they do some composition consistency checking. Composition is well suited for very large components, where the time spent for the communication between the components is much smaller than the time spent for computation within the components. The composition model also allows us to wrap legacy software and use it in compositions with new components. An example of a successful environment based on these concepts is METAFrame® [SMCB96, MS97]. METAFrame® is a commercial system used in the domain of service configuration for intelligent telecommunication networks (see [SMBR95]).

Since transformation systems are less widely known than compositional generators, we will discuss some of their basic concepts in the next section (see [Fea86] for a survey of transformation systems).

6.3 Transformation Systems

A *transformation* is an automated, semantically correct (as opposed to arbitrary) modification of a program. It is usually specified as an application of a *transform* to a program at a certain location. In other words, a transform describes a generic modification of a program and a transformation is a specific instance of a transform. The best known category of transforms are *rewrite rules* (also called *transformation rules*). An example of a rewrite rule is the replacement of the division of two equal expressions by 1:

$x/x \rightarrow 1$ if $x \neq 0$

This transform replaces occurrences of x/x by 1 whenever $x \neq 0$. x/x is the *head*, 1 is the *body*, and $x \neq 0$ is the *condition* of the transform. An example of a transform which is not a rewrite rule would be a procedural transform which performs repeated modifications that cannot be represented by a single rule.

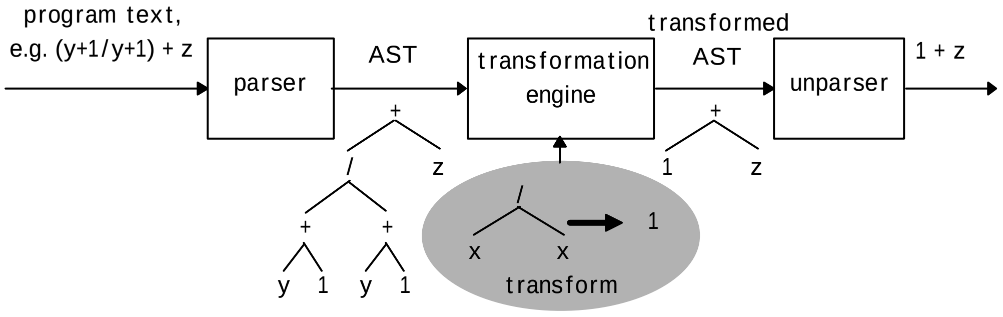


Figure 57 *Main components of a typical transformation system*

Transforms are not applied to program texts directly, but to their syntax trees. This is shown in Figure 57, which summarizes the structure of a typical transformation system. The program text to be transformed is turned into an abstract syntax tree (AST) by a parser and the *transformation engine* (also referred to as the *rewrite engine*) applies the transforms to the AST. Finally, the transformed AST is turned back into the textual representation by an *unparser* (or, alternatively, machine code could be generated by a *code generator*). The variable x in the transform in Figure 57 is referred to as a *metavariable* since it matches against whole subtrees (in our case $y+1$), i.e. parts of a program.

It is worth mentioning that the structure shown in Figure 57 resembles the structure of a compiler, which also has a parser and implements optimizations as transformations. The main difference, however, is that in a transformation system, transforms are written by the user of the system. Furthermore, it is usually also possible to provide user-defined grammars for the parser and the unparser. For this reason, transformation systems are often referred to as *open compilers*.

In general, transforms can be applied not only to trees but also to general graphs. For example, the *Reasoning5* system transforms — in addition to abstract syntax trees — data flow and control flow graphs [RS].

6.3.1 Types of Transformations

Software development involves many different types of transformations (see [Par90, BM97]). In particular, *refinements* and *optimizations* are used to implement a specification, whereas *refactorings* is used in evolution and maintenance. We will discuss these and other transformations in the following four sections (see [Par90] for a comprehensive treatment of this topic).

6.3.1.1 Refinements

A *refinement* adds implementation detail. It usually involves implementing concepts of one abstraction level in terms of concepts of a lower abstraction level. This is the main type of transformations used for implementing a specification. Examples of refinements are the following:

- *Decomposition*: Concepts of a higher level are decomposed into concepts of a lower level, e.g. an abstract data type is implemented in terms of a number of other abstract data types. The decomposition results in a structural organization of the lower level. We refer to this organization as the *horizontal structure* of one level (e.g. module structure, object structure). In general, the correspondence between the concepts of two levels may be many-to-many (esp. because of optimizations). Also, different decompositions favor different sets of goals and it is sometimes desirable to maintain overlapping, partial decompositions (each favoring certain point of view) that can be composed into one complex model. Unfortunately, traditional decomposition techniques do not support such overlapping partial decompositions well. We will discuss this topic in Chapter 7 in more detail. Refinements introduce traceability relationships between concepts of different levels. They represent the *vertical structure*.

- *Choice of representation*: As implementation details are added, we often need to choose an appropriate lower level representation for the higher level concepts, e.g. a matrix data type can be represented using arrays, vectors, hash tables, etc. The choice of a particular representation depends on the desired performance characteristics and other properties of the higher level concept (e.g. the shape of the matrix).

- *Choice of algorithm*: Operations can be implemented using different algorithms. The choice depends on the required operation properties (e.g. performance characteristics) and the properties of other concepts involved. For example, the choice of the matrix multiplication algorithm depends on the shapes of the operand matrices (see Section 10.1.2.2.1.5).

- *Specialization*: Concepts specialized for a certain context of use are often obtained by specializing more general concepts, i.e. concepts designed for more than one context by analyzing multiple concrete instances (see Section 2.3.7). For example, a parameterized module can be specialized by supplying concrete parameters. Another general specialization technique is *partial evaluation*. We will discuss this technique in the following section since it is often used for optimization purposes.

- *Concretization*: Concretization involves implementation by adding more detail to abstract concepts (see Section 2.3.7). For example, concrete classes add implementation details to abstract classes. Since concepts are often abstract and general at the same time, specialization and concretization may be combined.

6.3.1.2 Optimizations

Optimizations improve some of the performance characteristics of a program (e.g. execution speed, response time, memory consumption, consumption of other resources, etc.). Optimizations involve structural changes of the code. Two important types of such changes are the following [BM97]:

- *Interleaving*: Two or more higher-level concepts are realized in one section of the lower-level code (e.g. one module or class).

- *Delocalization*: A higher-level concept is spread throughout the whole lower-level code, i.e. it introduces details to many lower-level concepts.

Both types of structural changes make the lower-level code harder to understand and to reason about. The effects of delocalization on comprehension were studied in [LS86] and [WBM98].

Examples of optimizations are listed below [ASU86, BM97]:

- *Partial evaluation*: Partial evaluation is a technique for the specialization of a more general program for a specific context of use. In most cases, programs are specialized at compile time to be used in a specific context at runtime, but specialization at runtime is also possible. The specialization is based on the knowledge of the specific context (i.e. certain parameters are fixed). In abstract terms, partial evaluation can be thought of as partially evaluating a function based on the knowledge of some of its parameters to be constant in the special context. The evaluation involves propagating these constants throughout the program and simplifying and optimizing the code (e.g. eliminating unnecessary code, folding constants, unrolling loops, etc.). Partial evaluation is not only an important optimization technique, but also a reuse paradigm allowing us to specialize a general, reusable piece of software for the use in a specific context. In particular, it has been used to specialize scientific code (see [BW90, BF96]). An extensive treatment of partial evaluation can be found in [JGS93].

- *Finite differencing*: Finite differencing is an optimization technique involving the replacement of repeated costly computations by their less expensive differential counterparts (see [PK82]). For example, if the value of some variable X in a loop grows by some constant delta in each iteration and some other variable Y in the loop is computed by multiplying X by a constant factor, we can optimize this multiplication by realizing that Y also grows by a constant delta. Thus, instead computing Y by multiplying X by the constant factor in each iteration, we obtain the new value of Y by adding the constant Y delta to the old Y value. The Y delta can be computed outside the loop by multiplying the X delta by the constant factor.

- *Inlining*: Inlining involves the replacement of a symbol by its definition. For example, a procedure call can be replaced by the body of the procedure to avoid the calling overhead.

- *Constant folding*: Constant folding is the evaluation of expressions with known operands at compile time. It is a special case of partial evaluation.

- *Data caching/memoization*: After some often needed data has been once computed and the computation is expensive, it is worth caching this data, i.e. memoizing it, for later reuse [BM97].

- *Loop fusion*: If two loops have the same (or similar) structure and the computation in both loops can be done in parallel, then the loops can be replaced by one loop doing both computations, i.e. the loops are fused. For example, elementwise matrix operations, e.g. matrix addition involving multiple operands (e.g. A+B+C+D) can be performed using the same set of loops as required for just two operands (e.g. A+B).

- *Loop unrolling*: If the number of iterations for a loop is a small constant C known at compile time, the loop can be replaced by C inline expanded copies of its body.

- *Code motion*: Code motion is another loop optimization. It involves recognizing invariant code sections inside a loop and moving them outside the loop.

- *Common subexpression elimination*: This optimization involves recognizing and factoring out common subexpressions to reuse already computed results.

- *Dead-code elimination*: Unreachable code or unused variables can be eliminated.

Most of these optimizations (esp. the last eight) are performed by good compilers. Unfortunately, compilers can apply them at a very low abstraction level, i.e. the level of the programming language. On the other hand, optimizations are most effective at higher levels since the application of optimizations needs information which are often not available at lower levels (we will see an example of this in Section 6.4.1). Optimizations performed based on higher-level domain knowledge are referred to *as domain-specific optimizations*. Another example of optimizations which compilers usually do not perform are *global optimizations*. Global optimizations involve gathering information from remote parts of a program in order to decide how to change it at a given location. For example, the use of a certain algorithm at one location in a program could influence the selection of a different algorithm at a different location.

The use of domain-specific and global optimizations is one of the important differences between generators based on domain models and conventional compilers.

6.3.1.3 Refactoring

Refactoring transformations reorganize code at the design level. Some refactorings change the horizontal structure of one layer and other move code between layers. Refactorings are particularly interesting for evolutionary development and maintenance. Examples of refactorings include

- *Generalization*: Generalization involves building general models for a number of instances. For example, we can generalize a number of classes by factoring out common code into one parameterized class and converting the portions varying among them into parameters. Alternatively, we could organize them into a inheritance hierarchy by moving common parts up the hierarchy. Extracting the common parts represents an example of *abstraction*.

- *Simplification*: Simplifications reduce the representation size of a program (i.e. they make programs shorter, but not necessarily faster). They often improve understandability of a program.

- *Introducing new variation points*: In order to increase the horizontal scope of a model, we might want to introduce new variation points, e.g. new parameters.

These and other general classes of refactorings involve many smaller transformations which are usually dependent on the modeling paradigm and the technology used. A detailed study of refactorings for object-oriented models can be found in [Obd92].

6.3.1.4 Other Types of Transformations

Editing transformations perform various simple, rather syntactic code modifications, such as stylistic improvements, applying De Morgan to logical expressions, etc. They mainly help with editing.

In [MB97], Mehlich and Baxter also mention *jittering transformations*, whose purpose is to modify a program in order to make the transformations we discussed so far applicable.

6.3.2 Scheduling Transforms

Another issue which has to be addressed by a transformation system is when to apply which transform at which location in the syntax tree. This issue is also referred to as *scheduling transforms*. Some transforms can be scheduled procedurally, and others are scheduled by an inference process based on some domain knowledge (e.g. the configuration knowledge). In any case, the programs scheduling transforms are referred to as *metaprograms* [BP97].[51]

The implementation of a specification using transformations is best illustrated using the *derivation tree model* [Bax90, Bax92, BP97] shown in Figure 58. We start with some functional specification f_0 (at the top), e.g. the specification of a sorting problem, and some performance specification G_0 (on the left), e.g. required time complexity and implementation language. Based on the domain model of sorting and some inference, the transformation system determines that, in order to satisfy the performance goal G_0, it must satisfy the goals G_1 and G_2. Then, the system decomposes the goals G_1 and G_2 into further subgoals. The goal decomposition is carried out until a concrete transformation T_1 is applied to the initial functional specification f_0. The following transformations are determined in an analogous way until we arrive at the implementation f_n of the functional specification f_0 satisfying the performance specification G_0.

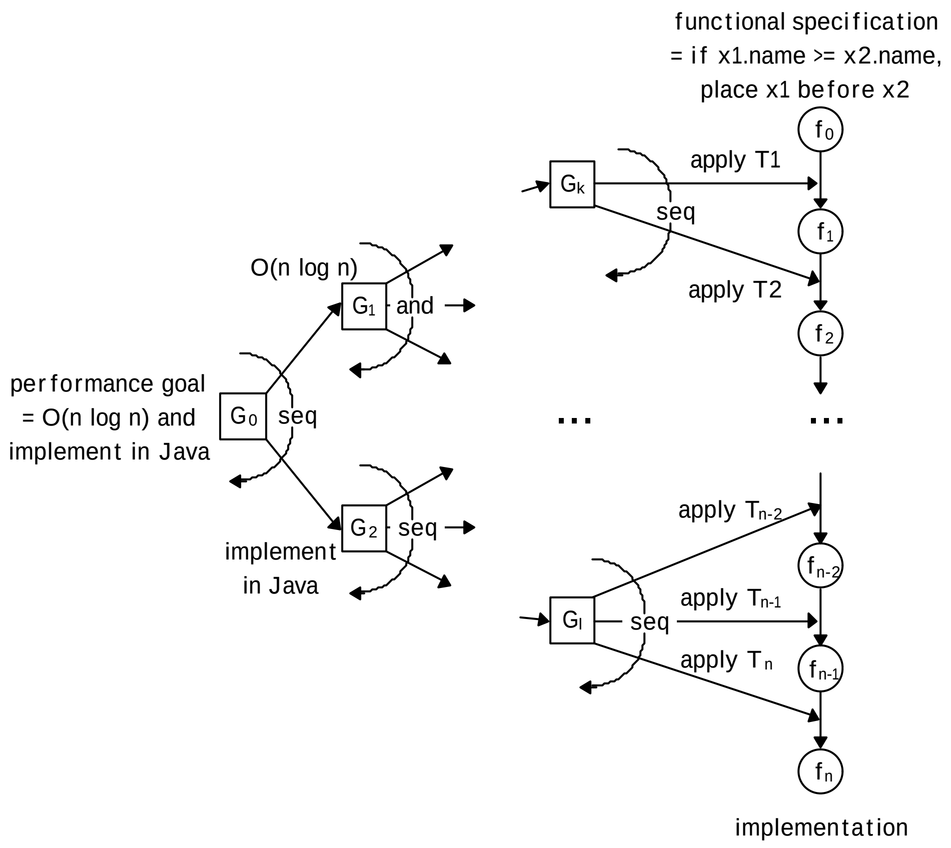


Figure 58 *Derivation tree of the transformational implementation of a functional specification (adapted from [BP97]). The middle part of the tree is not shown.*

The goal decomposition could be done procedurally, i.e. by procedural metaprograms (e.g. [Wil83]), or based on inference, i.e. planning-style metaprograms (e.g. [McC87, Bax90]), or by some mixture of both (e.g. [Nei80, MB97]). For example, G_0 could be a procedure calling G_1 and G_2, and G_1 could be decomposed in to some G_3, G_4, and G_5 goals by inference. Also the transforms can be applied by procedures or scheduled by inference. The goal decomposition can be viewed as a process of making implementation decisions and asserting new properties of the program under refinement and then making new implementation decision based on these properties, etc. In this sense, the inference is based on the fact that each decision has some pre- and post-conditions.

An important property of the derivation history model is that each step in the derivation tree can be explained, i.e. each subgoal helps to achieve its parent goal. Also, any changes to the functional specification being transformed are well defined in the form of transforms (rather than being some arbitrary manual changes). In [Bax90, Bax92, BP97, BM97, MB97], Baxter et al. describe an approach allowing us to propagate specification changes through an existing derivation tree in a way that only the affected parts of the tree need to be recomputed (this technology is being commercialized by Semantic Designs, Inc. [SD]). The derivation model clearly demonstrates us the advantages of transformation systems for software maintenance and evolution. Unfortunately, it requires a comprehensive and detailed domain model, whose development is not feasible or cost effective in all cases.

We will discuss the scheduling issue in the context of concrete systems and approaches in later sections.

6.3.3 Existing Transformation Systems and Their Applications

Transformation systems are used in various areas, e.g.

- implementing specifications (i.e. program synthesis) and metaprogramming, e.g. Draco [Nei80], TAMPR [BM84, BHW97], CIP [BEH+87], Medusa [McC88], KIDS [Smi90], TXL [CS92], Prospectra [HK93], Polya [Efr94], APTS [Pai94], SPECWARE [SJ95], SciNapse® [AKR+97], and IP [ADK+98],

- evolutionary development through refactoring and reengineering legacy systems, e.g. Reasoning5 [RS], DMS [BP97, BM97, MB97], SFAC [BF96], and Refactory [RBJ98],

- symbolic mathematical computations, e.g. Mathematica [Wol91], and

- language prototyping (i.e. generating compilers, debuggers, and often syntax-oriented editors from a specification of a language), e.g. Cornell Generator Synthesizer [RT89], CENTAUR [BCD+89], Popart [Wil91], and ASF+SDF [DHK96].

Three of these systems and approaches deserve special attention:

- Reasoning5 [RM], which is currently the only commercially available general-purpose transformation system,

- Intentional Programming [IPH], a transformation-based programming environment and platform (still under development), and

- SciNapse® [AKR+97], a commercial domain-specific transformation system synthesizing high-performance code for solving partial differential equations.

Reasoning5, formerly known as Refine [KM90], has been developed by Reasoning Systems, Inc., based on some 15-year experience in transformation technology. Reasoning5 represents programs in three forms: syntax trees, data-flow graphs, and control-flow graphs. These representations can be analyzed and manipulated by transforms expressed in a declarative language CQML (*code query and manipulation language*), which also includes high-level concepts such as sets and relations. Reasoning5 has been primarily used in the area of reengineering and maintenance, where legacy code, e.g. in COBOL, Fortran, or C, is imported using parsers into the internal representations. Transforms can be packaged into problem-specific plug-ins (possibly binary plug-ins by third party vendors). For example, a special plug-in for correcting year 2000 problems in COBOL code is available. Reasonig Systems' technology has also been used as a platform for forward engineering tools, such as the program synthesis environments KIDS and SPECWARE (they are discussed in Section 6.4.4). Reasoning5 is written in Lisp.

Intentional Programming is another commercial general-purpose transformation technology under development at Microsoft Research. Unlike Reasoning5, Intentional Programming is set up as an efficient, general-purpose programming environment. We will describe it in Section 6.4.3.

A remarkable example of a commercially available domain-specific transformation system is SciNapse® (formerly known as Sinapse [Kan93]) by SciComp, Inc. [SC]. The system synthesizes high-performance code for solving partial differential equations and has been successfully applied in various domains, e.g. wave propagation [Kan93], fluid dynamics, and financial modeling [RK97]. It accepts specifications in a high-level domain-specific mathematical notation. The refinement process in SciNapse® involves performing some domain-specific mathematical transformations, selecting an appropriate discretization method based on the analysis of the equations, selecting appropriate algorithms and data structures, generating the solution in pseudocode, optimizing at the pseudocode level, and, finally, generating C or Fortran code from the pseudocode. At each major refinement stage, so-called level summaries are generated which inform the user about the current state of the refinement and the intermediate results. The system is built on top of a planning expert system written in Mathematica, where the concepts from the problem specification are represented as objects in the knowledge base and transforms are triggered by rules.

6.4 Selected Approaches to Generation

In the following four sections, we will describe four generative approaches, which were selected to give you an overview of the area. The first approach is Draco. It is a prominent example of transformation technology. The second approach is GenVoca, which is a compositional approach with transformational elements. Next, we will describe Intentional Programming, which is an approach and a transformation-based programming environment supporting compositional and transformational generation. Finally, we give an overview of a formal approach to generation based on algebraic specifications.

6.4.1 Draco

Draco[52] is an approach to Domain Engineering based on domain-specific languages and transformation technology. The Draco approach and a prototype of a development environment, also called Draco, were developed by James Neighbors in his Ph.D. work [Nei80]. It was the first Domain Engineering approach. Since then, the original Draco ideas have been translated into commercial products, e.g. the CAPE[53] environment for prototyping and developing communication protocols (see [Bey98]). In addition to domain engineering, the new ideas introduced by Draco include domain-specific languages and components as sets of transforms.

The main idea of Draco is to organize software construction knowledge into a number of related domains. Each Draco domain encapsulates the knowledge for solving certain class of problems. There are several types of domains in Draco [Nei89, Bax96]:

- *Application domains*: Application domains encapsulate knowledge for building specific applications, e.g. avionics, banking, manufacturing, video games, etc.

- *Modeling domains*: A modeling domain is the encapsulation of the knowledge needed to produce a part of a complete application. It can be reused in the construction of applications from many different application domains. We can further subdivide this category into *application-support domains*, e.g. navigation, accounting, numerical control, etc., and *computing technology domains*, e.g. multitasking, transactions, communications, graphics, databases, user interfaces, etc. Some modeling domains can be as abstract as time, currency, or synchronization.

- *Execution domains*: Application domains are eventually refined into execution domains, i.e. *concrete target languages*, e.g. C, C++, or Java. Different types of languages could be also grouped into *abstract programming paradigms*, e.g. procedural, OO, functional, logic, etc.

Application domains are typically expressed in terms of several modeling domains and the latter in terms of execution domains. The lower level domains are also referred to as refinements of the higher level domains. An example of a number of interrelated Draco domains is shown in Figure 59. It is important to note that Draco domains need not to be organized in a strict hierarchy, i.e. cycles involving one or more domains are possible (e.g. implementation of arrays as lists and lists as arrays demonstrates a cycle).

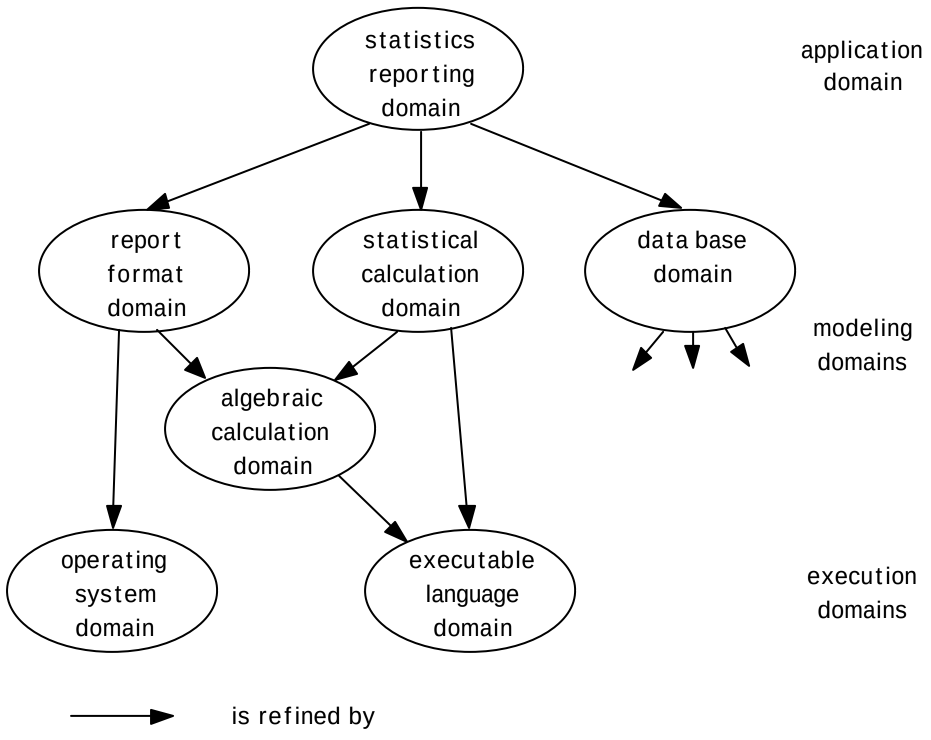


Figure 59 *Examples of interrelated Draco domains (adapted from [Nei84])*

Specifically, a Draco domain contains the following elements ([Nei84, Nei89]):

- *Formal domain language*[54] *(also referred to as "surface" language)*: The domain language is used to describe certain aspects of an application. It is implemented by a parser and a pretty printer, and the internal form of parsed code is an abstract syntax tree.

- *Optimization transforms*: These transforms represent rules of exchange of equivalent program fragments in the *same* domain language and are used for performing optimizations.[55]

- *Transformational components*: Each component consists of one or more *refinement transforms* capable of translating the objects and operations of the source domain language into one or more target domain languages of other, underlying domains. There is one component for each object and operation in the domain. Thus, transformational components implement a program in the source domain language in terms of the underlying domains.

- *Domain-specific procedures*: Domain-specific procedures are used whenever a set of transformations can be performed (i.e. scheduled) algorithmically. They are usually applied to perform tasks such as generating new code or analyzing programs in the source language. For example, we could write a procedure implementing a parser from a grammar specification.

- *Transformation tactics and strategies (also called optimization application scripts)*: Tactics are domain-independent and strategies are domain-dependent rules helping to determine when to apply which refinement. Optimizations, refinements, procedures, tactics, and strategies are effectively organized into metaprograms.

Now we illustrate each of these elements using simple examples.

Domain-specific languages are designed to allow us writing intentional and easy-to-analyze specifications. For example, a communication protocol can be nicely defined using a finite state automaton. This idea is used in CAPE, a Draco-based environment for prototyping and development of protocols by Bayfront Technologies [BT]. In CAPE, protocols are specified using the domain-specific language PDL (*protocol definition language*). The style of PDL specifications is illustrated in Figure 60. An excerpt from a larger example of a simple data transfer protocol is shown in Figure 61 (see [BT] for details).

```
example1 { InitialState = State1;
        state State1::
             Event1 -> Action1, Action2 >> State2;
             Event2 -> Action3;
        state State2::
             Event1 -> Action4;
             Event2 -> Action5, Action6 >> State1;
}
```

Figure 60 *Sample specification in PDL (from [BT])*

Draco optimization transforms operate within the same language. Examples of simple optimization transforms in the *algebraic calculation domain* (see Figure 59) are eliminating the addition of zero

ADDx0: X+0 → X

and replacing EXP(X, 2) by X*X, where EXP(A, B) raises A to the Bth power

EXPx2: EXP(X, 2) → X*X.

```
[ simple data transfer protocol in PDL with timeouts, retry, and no error recovery ]
dataxfer { InitialState = Idle;
    [ Idle state - no connections have been established ]
    state Idle::
        recv(Connect,Net) -> send(Connect,Usr), StartTimer(Tconn) >> SetupReceive;
        recv(Connect,Usr) -> send(Connect,Net), StartTimer(Tconn) >> SetupSend;
    [ SetupReceive state - network has requested a connection with the user ]
    state SetupReceive::
        timeout(Tconn) | recv(Refuse,Usr) -> send(Refuse,Net), StopTimer(Tconn) >> Idle;
        recv(Accept,Usr) -> send(Accept,Net), StopTimer(Tconn), StartTimer(Trecv) >> Receiving;
        macro EventDisc;
    [ Receiving state - user has accepted a connection request from the network, ]
    [ network to user transmission is in progress ]
    state Receiving::
        timeout(Trecv) -> macro ActionError;
        recv(Disconnect,Net) -> send(Disconnect,Usr), StopTimer(Trecv) >> Idle;
        recv(Message,Net) -> CheckMsg{
                                MsgOK -> send(Message,Usr), send(Ack,Net), RestartTimer(Trecv);
                                MsgBad -> send(NotAck,Net);
                           };
        macro EventDisc;

    [ ... ]

[ end of dataxfer ]
}
```

Figure 61 *Excerpt from the definition of a simple data transfer protocol in PDL (from [BT])*

The exponentiation function EXP(A,B) could be implemented using the transformational component shown in Figure 62. This component defines two alternative refinements of EXP(A,B): one using the binary shift method and one using the Taylor expansion. Each of these refinements has a CONDITIONS section defining when the corresponding refinement is applicable. If a refinement is selected, EXP(A,B) is replaced by its CODE section. The application of a refinement to a specification produces a new specification with new properties, which can be stated in the ASSERTIONS section, e.g. the complexity of the implementation. These assertions are attached to the resulting code as annotations.

```
COMPONENT: EXP(A,B)
    PURPOSE: exponentiation, raise A to the Bth power
    IOSPEC: A a number, B a number / a number
    DECISION:The binary shift method is O(ln2(B)) while the Taylor expansion is an adjustable number
        of terms. Note the different conditions for each method.
    REFINEMENT: binary shift method
        CONDITIONS: B an integer greater than 0
        BACKGROUND: see Knuth's Art of Computer Programming, Vol. 2, pg. 399, Algorithm A
        INSTANTIATION: FUNCTION,INLINE
        ASSERTIONS: complexity = O(ln2(B))
        CODE: SIMAL.BLOCK
         [[ POWER:=B ; NUMBER:=A ; ANSWER:=1 ;
           WHILE POWER>0 DO
             [[ IF ODD(POWER) THEN ANSWER:=ANSWER*NUMBER;
               POWER:=POWER//2 ;
               NUMBER:=NUMBER*NUMBER ]] ;
            RETURN ANSWER ]]
    END REFINEMENT

    REFINEMENT: Taylor expansion
        CONDITIONS: A greater than 0
        BACKGROUND: see VNR Math Encyclopedia, pg. 490
        INSTANTIATION: FUNCTION,INLINE
        ASSERTIONS: error = (B*ln(A))^TERMS/TERMS!
        ADJUSTMENTS: TERMS[20] - number of terms, error is approximately
                                - (B*ln(A))^TERMS/TERMS!
        CODE: SIMAL.BLOCK
         [[ SUM:=1 ; TOP:=B*LN(A) ; TERM:=1 ;
           FOR I:=1 TO TERMS DO
            [[ TERM:=(TOP/I)*TERM ;
              SUM:=SUM+TERM ]] ;
           RETURN SUM ]]
    END REFINEMENT
END COMPONENT
```

Figure 62 *Example of a transformational component implementing* EXP(A,B) *(adapted from [Nei80])*

At any point during the refinement process of a system specification, more than one refinement or optimization transforms might be applicable. This is illustrated in Figure 63 showing three alternative paths of refining EXP(X,2) into C programs. EXP(X,2) can be either directly refined using the binary shift method refinement or the Taylor expansion refinement, or it can be first transformed into X * X using our optimization EXPx2. Thus the question is: how are the appropriate transforms selected? The transforms are scheduled partly by procedures, tactics, and strategies and partly by the user.

Our example of refining EXP(X,2) illustrates an important point: Optimizations are most effective, when applied at the appropriate level of abstraction. EXPx2 simplifies EXP(X,2) into X * X No such simple optimization could achieve the same result at a lower level. The code produced by the binary shift method refinement would require an extremely complicated series of transformations in order to be reduced to X * X For the Taylor expansion code, no such transformations exist since the Taylor expansion is only an approximation of EXP(A,B). Thus, an example of a simple tactic is to try optimization transforms first before applying any refinements.

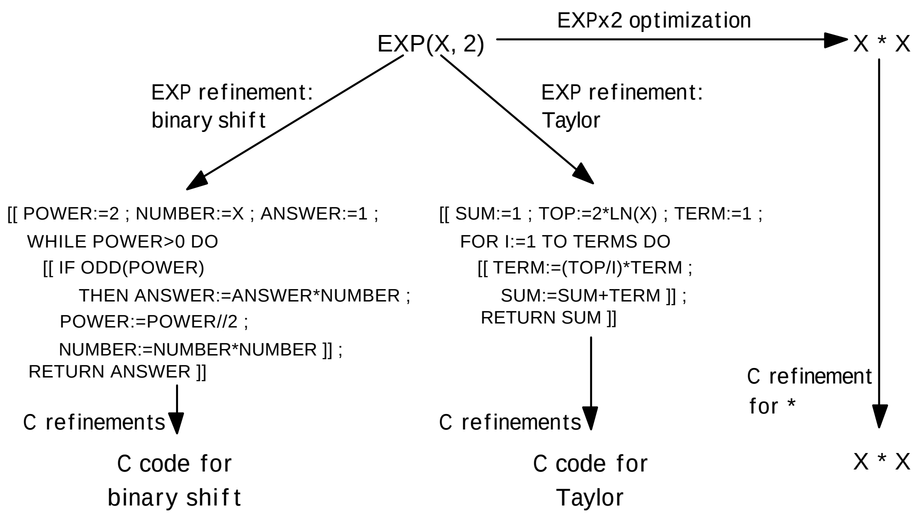


Figure 63 *Three alternative ways of refining EXP(X,2) into a C program (adapted from [Nei80]). The C code for the binary shift method and the Taylor expansion implementation are not shown.*

Procedures, tactics, and strategies allow only a partial automation of the refinement process, which is a creative process requiring user intervention. The space of all possible derivations represents an enormous search space. However, this search space can often be significantly pruned by means of AI planning and configuration techniques.

6.4.2 GenVoca

GenVoca is an approach to building software system generators based on composing object-oriented layers of abstraction, whereby layers are staked one on another. Each layer contains a number of classes in the OO sense and the layer “above” refines the layer “below” it by adding new classes, adding new methods to existing classes, etc. This model roughly corresponds to the refinement occurring in OO frameworks by means of inheritance: Existing classes of a framework can be incrementally refined by adding new layers of derived classes. However, as we will see later, there are important differences between these two models.

The GenVoca model originated in the work by Don Batory on Genesis [BBG+88], a database management system generator, and the work by O’Malley et al. on Avoca/x-kernel [HPOA89, OP92], a generator in the domain of network protocols. These two independently conceived systems shared many similarities, which lead to the formulation of the GenVoca (a name compiled from the names Genesis and Avoca) model in [BO92]. Since then, Batory and colleges have been working on refining and extending this original model and building new GenVoca generators. P1 [BSS92, SBS93] and P2 [BGT94] were extensions of the C programming languages for defining GenVoca generators; in particular, they were used to develop Predator [SBS93], a data container generator. Work on P++ [SB93, Sin96] involved equivalent extensions of C++. A mechanism for composition validation in GenVoca models was proposed in [BG96, BG97], and more recently, the originally compositional GenVoca model has been extended with transformational techniques [BCRW98]. A Java pre-compiler for implementing domain-specific language extensions and GenVoca domain models is described in [BLS98]. Other systems based on GenVoca include the distributed file system *Ficus* [HP94], the *ADAGE* generators in the domain of avionics navigation systems [BCGS95], the generator of query optimizers *Prairie* [DB95], and the implementation of the data container generator in the IP system (see Section 6.4.3) *DiSTiL* [SB97].

6.4.2.1 Example

A typical application of the GenVoca model is to organize a library of data containers according to this model (see [BSS92, SBS93, Sin96, SB97]). Data containers (or collections) belong to the most fundamental building blocks in programming. Thus, any general-purpose, industrial-strength programming language has to offer such a library. Examples of data container libraries for C++ include libg++ [Lea88], the Booch C++ Components [Boo87], and, most recently, the Standard Template Library (STL) [MS96].

Data containers are good examples of concepts exhibiting wide feature variations. For example, the Booch C++ Components offer various data structures (*bag*, *deque*, *queue*, *list*, *ring list*, *set*, *map*, *stack*, *string*, and *tree*), various memory management strategies (*bounded*, *unbounded*, *managed*, *controlled*), various synchronization schemas (*sequential*, *guarded*, *concurrent*, *multiple*), and *balking* and *priority* implementations for queues (see Table 10 for explanations). A concrete component can be described through a valid combination of these features and there are more than 200 valid combinations for the Booch C++ Components.

Data structure families in the Booch library	
Bag	unordered collection of objects, which may contain duplicates
Deque	ordered sequence of objects, where objects may be inserted or removed at either end
Queue	ordered sequence of objects, with “first-in first-out” semantics
List	a linked list of objects, which are automatically garbage collected
Ring List	ordered sequence of objects, organized in a loop
Set	unordered collection of objects, which contains no duplicates
Map	a tabular data structure which associates instances of one kind of object with instances of some other kind of object
Stack	ordered sequence of objects, with “last-in first-out” semantics
String	ordered sequence of objects, where any object may be accessed directly
Tree	a binary tree of objects, which are automatically garbage collected
Data structure features which are available to every family	
Bounded	static memory allocation (upper bound on total number of objects)
Unbounde d	dynamic memory allocation algorithm (no upper bound on total number of objects)
Managed	free objects are stored on a list for subsequent reuse
Controlled	a version of managed which operates correctly in a multi-threaded environment
Sequential	assumes a single-threaded environment
Guarded	assumes a multi-threaded environment, where mutual exclusion is explicitly performed by the user
Concurrent	assumes a multi-threaded environment, where the object ensures that all read and write accesses are serialized
Multiple	assumes a multi-threaded environment, where the object permits multiple simultaneous read access, but it serializes write access
Data structure features which are available to deques and queues only	
Balking	objects may be removed from the middle of a sequence
Priority	objects are sorted based on some priority function

Table 10 *A glossary of the Booch data structure terminology (from [Sin96])*

Implementing all feature combinations as concrete classes is clearly inefficient in terms of development and maintenance cost and scaling, i.e. adding a new feature can potentially double the number of concrete classes.[56] The Booch C++ Components library addresses this problem through a careful design avoiding code duplication by means of inheritance and templates and reducing the number of concrete classes to about 100. However, a container library of an equivalent scope but designed using the GenVoca model requires only about 30 components (i.e. GenVoca layers) and exhibits even a much lower level of code duplication [Sin96]. This is achieved through consequent interface unification and standardization and aggressive parameterization. At the same time, the library achieves an excellent performance in terms of execution speed.

6.4.2.2 GenVoca Model

In the GenVoca model, each abstract data type or feature is represented as a separate layer and concrete components (or systems) are defined by type expressions describing layer compositions. For example, the following type expression defines a *concurrent*, *unbounded*, *managed bag* which allocates the memory for its elements from the *heap* and counts the number of elements it contains:

```
bag[concurrent[size_of [unbounded[managed[heap]]]]]
```

Figure 64 shows the layered structure equivalent to this type expression. Each layer contributes classes, attributes, and methods implementing the corresponding features to the entire composition. heap implements memory allocation from the heap, managed manages the allocated memory on a free list, unbounded provides a resizable data structure based on managed, size_of adds a counter attribute and a read-size method, concurrent wraps the accessing methods in semaphore-based serialization code (we will see an implementation of size_of and concurrent later), and, finally, bag implements the necessary element management and bag operations.

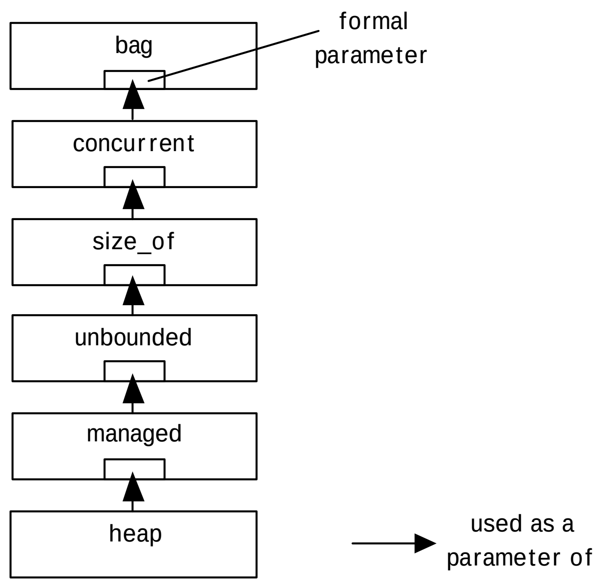


Figure 64 *Example of GenVoca layering*

Each of these layers, except heap, represents a parameterized component. For example, size_of takes a data structure as its parameter. We denote this as follows: size_of[DataStructure]. In general, a GenVoca layer may have more than one parameter. Thus, in general, GenVoca expressions have a tree structure, e.g. A[B[D, E], C[F, G]] (see Figure 65).

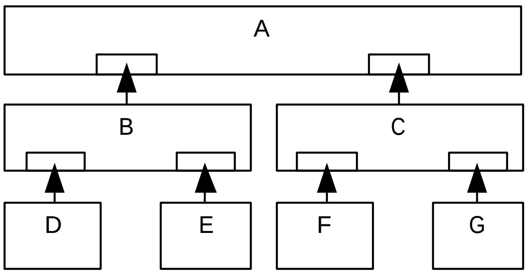


Figure 65 *Example of a tree-like GenVoca layering*

Each GenVoca layer exports a certain interface and expects its parameters (if any) to export certain interfaces. These contracts can be expressed explicitly by defining layer interfaces at a separate location and then using them in layer declarations. In GenVoca terminology, a standardized layer interface is referred to as a *realm*. A realm can be thought of as a collection of class and method signature declarations (we will see a concrete example of a realm in Section 6.4.2.3). We also say that certain layers or layer compositions "belong to a realm" if they export all the classes and methods declared in the realm (but they are allowed to export other classes and methods at the same time, too). For example, given the realms R and S, and given the layers A, B, C, D and E, we could write

```
R = {A, B[x:R]}
S= {C[x:R], D[x:S], E[x:R, y:S]}
```

This notation states that

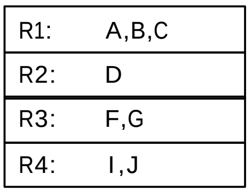
- A and B export the interface R,
- C, D, and E export the interface S,
- B, C, and E import the interface R, and
- D and E import the interface S.

Alternatively, GenVoca domain models can be represented more concisely as grammars. For our realms R and S and the layers A, B, C, D, and E, we have the following grammar:

```
R : A | B[R]
S : C[R] | D[S] | E[R, S]
```

where the vertical bar indicates an or. Please note that the layers B and D import and export the same realms. We refer to such layers as *symmetric*. An example of a symmetric layer is concurrent. Symmetric layers are important since they usually strongly increase the number of possible configurations. For example, Unix utilities are symmetric and can be combined in many orders using pipes.

In some cases, we found it useful to have differently named realms, but declaring the same interface. We can use such realms to differentiate between layers exporting the same operations in terms of signatures, but of different semantics. For example, the concurrent layer exports the same operations it imports, and the only difference is that the exported operations are serialized. If certain other layers require synchronized data structures (which is an example of a configuration constraint), we might want to define the realms SynchronizedDataStructure and DataStructure, even if they declare the same operations. Alternatively, we might just define the DataStructure realm and express the configuration constraint using extra annotations (see Section 6.4.2.5). Whichever method works best has to be decided in a specific context.



R1: A[R2] | B[R2] | C[R2]

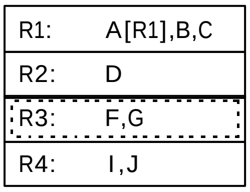
R2: D[R3]

R3: F[R4] | G[R4]

R4: I | J

Figure 66 *Example of a stacking model in graphical notation and the corresponding grammar*

An important class of GenVoca domain models can be represented using a quite intuitive, graphical notation shown in Figure 66. In this notation, each box represents a *generic layer* containing a set of alternative GenVoca layers (a generic layer corresponds to a realm). The idea is that we can obtain a concrete instance of this model by selecting exactly one layer per box. For example, according to Figure 66, B[D[F[J]]] is a valid type expression. We will refer to such GenVoca models as *stacking models* since each layer has only one parameter and instances of this model are simple stacks of layers (in general, instances of GenVoca domain models have a tree structure). More precisely, all layers of one generic layer export the same realm, e.g. R1, and also import the same realm, e.g. R2, which, in turn, is exported by all layers in the generic layer beneath it. We can further extend this model by allowing a generic layer to contain layers exporting and importing the same realm (as A[R1] in R1 in Figure 67). And finally, we can also have *optional* generic layers, i.e. one or none of the layers contained in an optional generic layer need to be selected when constructing a concrete instance. In other words, the layers contained in an optional generic layer are optional alternative layers. Optional generic layers are marked by a dashed inner box as R3 in Figure 67.



R1: A[R1] | B[R2] | C[R2]

R2: D[R3]

R3: F[R4] | G[R4] | R4

R4: I | J

Figure 67 *Example of an optional generic layer*

In addition to layer parameters (i.e. the realm-typed parameters[57]), such as x in B[x:R], layers can also have other types of parameters, such as constant parameters (e.g. maximum number of elements in bounded) and type parameters[58] (e.g. each of the data container layers require the element type as its parameter). The constant and type parameters are referred to as *horizontal parameters* and the layer parameters as *vertical parameters* (this terminology is due to Goguen and Burstall [GB80]; also see [Gog96]). The vertical parameters are instrumental in defining the vertical refinement hierarchies of layers (as in Figure 64), whereas the horizontal parameters do not affect this hierarchy, but rather provide for some variability within a single layer. We will enclose the horizontal parameters into round braces.[59] For example, B[x:R](y:type, z:int) has one vertical parameter x of type R and two horizontal parameters, namely the type parameter y and the integer constant z.

The GenVoca implementation of the Booch Components described in [Sin96] requires 18 realms and 30 layers. The main advantage of expressing the Boch Components in terms of realms and layers instead of inheritance hierarchies and parameterized classes is that layers allow us to refine multiple classes in a coordinated manner, so that the structure of the library is captured by layers more adequately. For example, a data container layer contains not only the data container class, but also a cursor class (i.e. the iterator) and an element class. All these classes are incrementally refined by each layer in a coordinated fashion as the layers are stacked up.

Next, we will take a look at some concrete examples of realm and layer definitions.

6.4.2.3 Defining Realms and Layers in P++

The adequate implementation of the GenVoca modeling concepts requires new language features, which are absent in current OO languages, such as C++ or Java. P++ is an extension of C++ implementing these new features [SB93, Sin96]. We will first explain the required language constructs using examples in P++. We will then show how to emulate some of these features in C++ in the next section.

```
template <class e>
realm DS
{
   class container
   {
      container();
      bool is_full();
      ... // other operations
   };

   class cursor
   {
      cursor (container *c);
      void advance();
      void insert(e *obj);
      void remove();
      ... // other operations
   };
};

template <class e>
realm DS_size : DS<e>
{
   class container { int read_size(); };
};
```

Figure 68 *Realm and subrealm declarations (from [BG97])*

Figure 68 shows two realm declarations. Realm DS defines the interface of the data structure used in the data container described in the previous section. DS is exported by unbounded and imported by size_of (i.e. we have size_of[x:DS]; see Figure 64). size_of augments the DS realm by adding the method read_size() to its container class. Thus, the realm exported by size_of is DS_size. concurrent also exports DS_size and bag and concurrent import DS_size (i.e. we have bag[x:DS_size] and concurrent[x:DS_size]). Please note that DS_size is declared as a subrealm of DS, i.e. it inherits all its classes and methods. Multiple inheritance between realms is also allowed. Realm inheritance is an example of pure interface inheritance as opposed to implementation inheritance.

```
template <class e, DS<e> x>
component size_of: DS_size<e>
{
   class container
   {
      friend class cursor;
      x::container lower;
      int count;

      container() { count = 0; };
      int read_size() { return count; };

      bypass_type bypass(bypass_args)
      {  return lower.bypass(bypass_args); };
   };

   class cursor
   {
      x::cursor *lower;
      container *c;

      cursor (container *k)
      {  c = k;
         lower = new x::cursor(&(c->lower)); };

      e* insert (e *element)
      {  c->count++;
         return lower->insert(element); };

      void remove()
      {  c->count--;
         lower->remove(); };

      bypass_type bypass(bypass_args)
      {  return lower->bypass(bypass_args); };
   };
};
```

```
template <class e, DS_size <e> x>
component concurrent: DS_size <e>
{
   class container
   {
      friend class cursor;
      x::container lower;
      semaphore sem;

      container() { };

      bypass_type bypass(bypass_args)
      {  bypass_type tmp;
         sem.wait();
         tmp = lower.bypass(bypass_args);
         sem.signal();
         return tmp; };
   };

   class cursor
   {
      x::cursor *lower;
      container *c;

      cursor (container *k)
      {  c = k;
         lower = new x::cursor(&(c->lower)); };

      bypass_type bypass(bypass_args)
      {  bypass_type tmp;
         sem.wait ();
         tmp = lower->bypass(bypass_args);
         sem.signal();
         return tmp; };
   };
};
```

Figure 69 *Implementation of the layers* size_of *and* concurrent *(from [BG97])*

The implementation of the layers size_of and concurrent is shown in Figure 69. The syntax resembles C++ class templates with member classes. size_of has one horizontal type parameter e (i.e. the element type) and one layer parameter x expecting a layer of the realm DS<e> and exports the realm DS_size<e>.[60] Please note that only the type of the layer parameters can be specified, i.e. layer parameters are an example of *constrained* generic parameters. size_of implements the method read_size() of container and refines the cursor methods insert() and remove() from the lower layer by adding statements incrementing or decrementing the count variable. All other container and cursor methods declared in the DS_size realm but not explicitly implemented in the size_of layer are implicitly defined by the bypass construct. bypass matches with the name of any method declared in DS_size, but not defined in size_of. bypass_type matches with the return type of such method and bypass_args with the argument list. A slightly different usage of this construct is found in the concurrent layer. The concurrent layer uses the bypass construct to wrap all the container and cursor methods from the lower layer in semaphore wait and signal statements.

An important aspect of GenVoca is the ability to propagate types up and down the layer hierarchy. The upward type propagation is achieved through accessing member types of the layer parameter, e.g. x::container in size_of, i.e. size_of accesses container of the layer beneath it. The downward type propagation occurs by means of type parameters of realms such as the type parameter of the realm DS<e> of the parameter x of size_of. Given the layer bounded (see Figure 70) implementing a bounded data structure with the type parameter e (i.e. the element type) and the integral constant parameter size (maximum capacity of the data structure), we can write the following type expression:

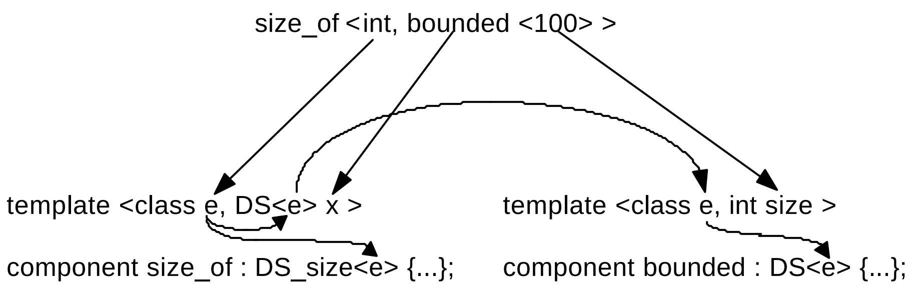
```
typedef size_of <int, bounded <100> > integer_ds;
```

According to this type expression, integer_ds exports the DS_size interface. It is interesting to note that we provided bounded only with the size parameter. How does bounded obtain the value for its type parameter e? The value for this type parameter is derived from the realm type of the parameter x of size_of, i.e. DS<e>, and, in our case, it is clearly int (see Figure 71). This type inference constitutes an example of downward type propagation. P++ has also a construct for passing types defined within one layer to the layer beneath it. This is accomplished by annotating the newly defined type with the forward keyword and passing it as a parameter to the realm type of a layer parameter, just as we passed e to DS<e> in size_of (see [Sin96] for details).

```
template <class e, int size>
component bounded : DS<e>
{
   class container
   {
      e objs[size];
      ... // other methods and attributes
   };

   class cursor
   {
      int index;
      ... // other methods and attributes
   };
};
```

Figure 70 *Implementation of the layer* unbounded *(adapted from [Sin96])*

The overall flow of information in a GenVoca hierarchy of layers is summarized in Figure 72.



size_of <int, bounded <100> >
template <class e, DS<e> x >
component size_of : DS_size<e> {...};
template <class e, int size >
component bounded : DS<e> {...};

Figure 71 *Parameter propagation in a component composition (adapted from [Sin96])*

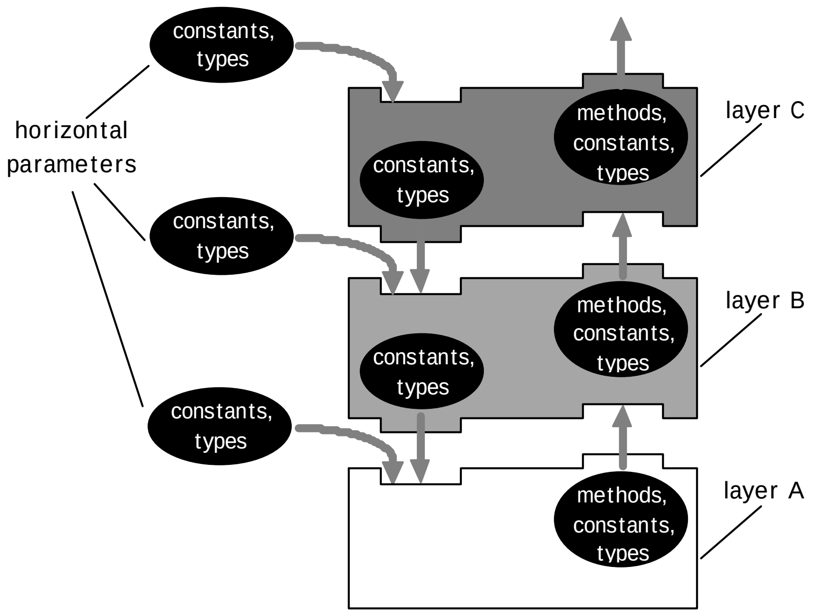


Figure 72 *Propagation of types, constants, and methods in a GenVoca hierarchy of layers (adapted from [Sin96])*

6.4.2.4 Implementing GenVoca Layers in C++

GenVoca layers can be implemented in C++ as class templates containing member classes. Unfortunately, there is no adequate idiom in C++ to implement realms since C++ templates do not support constrained type parameters, i.e. there is no way to specify the realm of a layer parameter (e.g. as DS<e> of x in size_of).[61] We will discuss the consequences of this problem later. But even without realms, implementations of GenVoca domain models in C++ are, as we will see in Chapter 10, very useful.

Figure 73 illustrates the general idea of how to implement a GenVoca layer in C++. ClassA and ClassB are defined as member classes of LayerA, a class template with the parameter LowerLayer. ClassA of LowerLayer is accessed using the scope operator :: and lower is declared to be of this type. The implementations of operationA() in ClassA of LayerA refines operationA() in ClassA of LowerLayer and the implementation of operationB() forwards the latter operation to LowerLayer. Since the operation in GenVoca layers are usually defined as inline, calling operationA() on lower in the operationA() in ClassA of LayerA does not incur any extra dispatching overhead. This way, we achieve clearly separated layers of abstraction and, at the same time, do not have to pay any performance penalties.

```
template <class LowerLayer>
class LayerA
{   public:
        class ClassA
        {   public:
                LowerLayed::ClassA lower;

                // refine operationA()
                void operationA()
                {   ... // LayerA-specific work
                    lower.operationA();
                    ... // LayerA-specific work
                };

                //forward operationB()
                void operationB()
                {   lower.operationB(); };
        };

        class ClassB
        { ... };
};
```

Figure 73 *Forwarding implementation of GenVoca layer in C++*

The implementation in Figure 73 is based on aggregation, i.e. LayerA holds references to class instances from the layer beneath it (e.g. lower). Some of the operations from the lower layer may be exported unchanged to the upper layer using forwarding methods (e.g. operationB()), some may be refined (e.g. operationA()), yet other may be simply used as supporting methods to implement some new methods of the upper layer. This kind of layer implementation corresponds to a *forwarding static wrapper* (wrapper, also referred to as a decorator, is a design pattern described in [GHJV95])[62] shown in Figure 74.

```
template <class Component>
class Wrapper
{   public:
        Component component;

        // refine operationA()
        void operationA()
        {   ... // wrapper-specific work
            component.operationA();
            ... // wrapper-specific work
        };

        //forward operationB()
        void operationB()
        {   component.operationB(); };
};
```

Figure 74 *Example of a forwarding static wrapper*

Forwarding-based implementation of a layer is quite adequate if only few or no operations exported by the lower layer are propagated unchanged up to the interface exported by the upper layer. Otherwise, one has to write many forwarding methods, which is not only tedious, but also impairs adaptability and maintainability since any changes to interface of the lower level ultimately require changes to the forwarding methods of the upper layers. In P++, this problem is addressed by the bypass construct. But what can we do in C++?

```
template <class Component>
class Wrapper : public Component
{   public:
      // refine operationA()
      void operationA()
      {   ... // wrapper-specific work
         Component::operationA();
         ... // wrapper-specific work
      };
};
```

Figure 75 *Example of an inheritance-based static wrapper*

In C++, the propagation of operations can be achieved using inheritance. Figure 75 shows an inheritance-based static wrapper. Please note that we no longer need an instance variable to hold an instance of the component and the original operation can be called using the C++ scope operator. Based on this idea, we can now rewrite the forwarding layer from Figure 73 as an inheritance-based layer as shown in Figure 76. Of course, a layer can use forwarding for some of its classes and inheritance for other classes.

```
template <class LowerLayer>
class LayerA
{   public:
      class ClassA : public LowerLayer::ClassA
      {   public:
            // refine operationA()
            void operationA()
            {   ... // LayerA-specific work
               LowerLayer::ClassA::operationA();
               ... // LayerA-specific work
            };
      };

      class ClassB : public LowerLayer::ClassB
      { ... };
};
```

Figure 76 *Inheritance-based implementation of GenVoca layer in C++*

In all the C++ examples we have seen so far, types were propagated upwards using the C++ scope operator (e.g. LowerLayer::ClassA). In general, we can use this operator to access any member types, i.e. also types defined using the typedef statement. For example, given the class BottomLayer

```
class BottomLayer
{ public:
   typedef int ElementType;
};
```

the member type ElementType can be accessed as follows:

```
BottomLayer::ElementType
```

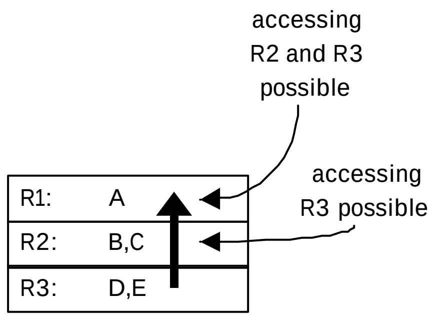
We can also use the same accessing syntax for integral constants. The required idiom is shown below:

```
class BottomLayer
{ public:
   enum { number_of_elements = 100 };
};
```

The integral constant number_of_elements is accessed as follows:

```
BottomLayer::number_of_elements
```

The idea of how to propagate types and constants upwards over multiple layers is illustrated in Figure 77. Using this idiom, we can reach out any number of layers down the hierarchy and then access the members of some layer we are interested in. Please note that we modeled layers as structs instead of classes. We do this in cases, where all the members of a layer are public. This saves us writing the keyword public:.



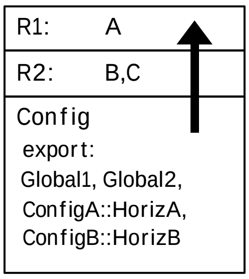
```
template <class R2>
struct A
{  typedef typename R2::R3 R3;
   ... // member classes
};

template <class R3_>
struct B
{  typedef R3_ R3;
   ... // member classes
};

template <class R3_>
struct C
{  typedef R3_ R3;
   ... // member classes
};

struct D {...};
struct E {...};
```

Figure 77 *Upward type propagation in a GenVoca model*

Upward type propagation can also be used to supply layers with global types and constants as well as with horizontal parameters. First of all, we found that mixing both vertical parameters (i.e. layer parameters) and horizontal parameters (i.e. types and constants) in the parameter declaration of a template implementing a layer obscures the structure of a GenVoca domain model. This is particularly evident in type expressions since the structure of a model instance described by an expression is determined by the values of the layer parameters, whereas the horizontal parameters just distract the reader. Second, if most of the layers of a model require the same horizontal parameter (e.g. element type in the data container example), we want to provide the parameter value at one place instead of explicitly supplying each layer with this value. A solution to both these problems is to propagate the global and horizontal parameters to all layer from some standard place.

The propagation style shown in Figure 77 has the problem that a layer explicitly asks the layer beneath it for all the types it needs or another layer above it might need. Thus, a layer has to explicitly pass a type, even if it is not interested in this type itself. This fact impairs adaptability since the need to propagate a new type might require changes to many layers. This problem can be avoided by passing an standard "envelope" containing all the layers, types, and constants to be propagated (see the struct Config in Figure 78). We refer to this envelope as a *configuration class* or, in short, a *config class*. This use of a config class is an example of a more general design pattern described in [Eis97]. Config is implemented as a so-called *traits class* [Mye95], i.e. a class aggregating a number of types and constants to be passed to a template as a parameter. The config class is passed to the leaf layers of a GenVoca domain model (i.e. layers that do not have any layer parameters) and is explicitly propagated upwards by all other layers. The config class is the only type explicitly passed between layers. All other communication between layers goes through this envelope. Thus, we can view configuration class as an implementation of a *configuration repository*, i.e. a repository of configuration information.

We usually define global types (e.g. Global1 in Figure 78) or constants (e.g. Global2) as direct members of the config class. On the other hand, all horizontal parameters of a certain layer (e.g. HorizA of A) are preferably wrapped in an extra config class specifically defined for that layer (e.g. ConfigA for A). This idiom prevents name clashes if two layers need two different horizontal parameters having the same name and is an example of the *nested config idiom* [Eis98].



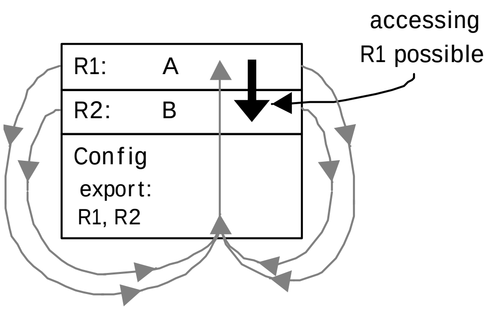
```
template <class R2>
struct A
{   // expose Config to upper layers
    typedef typename R2::Config Config;
    // retrieve types and constants from Config
    typedef typename Config::Global1 Global1;
    enum { Global2 = Config::Global2};
    typedef typename Config::ConfigA::HorizA HorizA;
    ... // Global1, Global2, and HorizA
    ... // are used in further code
};

template <class Config_>
struct B
{   // expose Config to upper layers
    typedef Config_ Config;
    // retrieve types and constants from Config
    typedef typename Config::Global1 Global1;
    enum { Global2 = Config::Global2};
    typedef typename Config::ConfigB::HorizB HorizB;
    ... // Global1, Global2, and HorizB
    ... // are used in further code
};

template <class Config>
struct B {...};

struct Config
{   typedef int Global1;
    enum { Global2 = 100 };
    struct ConfigA
    {   typedef char HorizA; };
    struct ConfigB
    {   typedef float HorizB; };
    ...
};
```

Figure 78 *Upward propagation of global and horizontal parameters*

So far we have seen a number of examples of upward type and constant propagation. But how to propagate types and constants downwards?

The downward propagation of types and constants can be accomplished in an indirect way. The main idea is to put the type expressions assembling the layers in the config class (see Figure 79). In other words, we wrap the whole layer hierarchy in our config class and propagate this class up the same hierarchy. This way, we can access any layer exported in the config class in any other layer, e.g. R1 in R2. In fact, we accomplished layer propagation in any direction. Unfortunately, there is a limitation to the downward propagation: Types propagated downwards can be only referenced in lower layers but not *used* in the C++ sense at compile time. In other words, downwards-propagated types can be used as types in variable and function parameter declarations (i.e. we can reference them), but we cannot access their member types (i.e. their structure is not defined at the time of access). This is so since the hierarchy is built up in C++ in the bottom-up direction (i.e. functionally: the arguments of a template are built before the template is built) and types propagated down to a lower layer are actually not yet defined at the time they are referenced in the lower level. However, even with this limitation, downward type propagation is still useful and we will see examples of its use in Section 8.7.

The C++ programming techniques presented in this section cover all the parameters and information flows shown in Figure 72: vertical and horizontal parameters, upward propagation of operations, types, and constants, and downward propagation of types and constants.



```
template <class R2>
struct A
{   typedef typename R2::Config Config;
    ...
};

template <class Config_>
struct B
{   typedef Config_ Config;
    typedef typename Config::R1 R1;
    ...
};

struct Config
{   // assemble the layers
    typedef B<Config> R2;
    typedef A<R2> R1;
    ...
};
```

Figure 79 *Downward type propagation*

6.4.2.5 Composition Validation

A relatively small number of GenVoca layers can be usually composed in a vast number of combinations. Whether two layers can be connected or not, depends on the compatibility of their interfaces. Layer A can be used as a parameter of layer B, if layer A exports the same realm as B imports or a subrealm of the latter. However, the realm compatibility takes into account only the compatibility of the exported and the imported signatures and not the semantic compatibility of the corresponding operations. Thus, even if a type expression is syntactically correct according to a GenVoca grammar, the system defined by this expression may still be semantically incorrect. For example, the layers corresponding to the concurrency features (see Table 10) in the GenVoca implementation of the Booch Components import and export the same realm, namely DS (see Figure 68). Assume that the GenVoca grammar describing the Booch Components contains the following productions:

```
DS_size : guarded[DS_size] | concurrent[DS_size] | multiple[DS_size] | size_of[DS] | ...
DS : bounded | unbounded[Memory] | ...
```

Given these productions, we could define the following data structure:

```
guarded[guarded[multiple[concurrent[guarded[concurrent[size_of[bounded]]]]]]]
```

This type expression is syntactically correct, but semantically a complete nonsense. Each of the layers guarded, concurrent, and multiple are intended to be used as alternatives and only one or none in one expression defining a data type. We clearly need a way to express such configuration rules. In general, the rules can be more complicated and they can span over multiple layers. For example, we could have a higher-level layer which requires that all the data structures used by this layer are synchronized, i.e. the expressions defining the data structures used by this layer have to contain the concurrent layer, but there can be several layers between the data structure expression and the higher-level layer. In other words, we need to verify the effect of using a layer at a "distance". Furthermore, a layer can affect the use of other layers not only below it, but also above it.

A model for defining such configuration constraints for GenVoca layers is presented in [BG96, BG97].[63] The model is based on the upward and downward propagation of attributes representing constraints on layers or properties of layers. There are two main types of constraints: *conditions* and *restrictions*. Conditions are constraints propagated downward and restrictions are constraints propagated upwards (see Figure 80). A given layer can be used at a certain position in a type expression if it satisfies all conditions propagated by upper layers down to this layer and all the restrictions propagated by lower layers up to this layer. We refer to such conditions as *preconditions* and to such restrictions as *prerestrictions* (see Figure 80) The use of a layer in a type expression modifies the conditions and restriction "flowing" through this layer: The conditions leaving a layer in the downward direction are referred to as *postconditions* and the restrictions leaving a layer in the upward directions are referred to as *postrestrictions*. By modifying the postrestrictions, a layer can express some requirements on the layers above it. Similarly, by modifying the postconditions, a layer can express some requirements on the layers below it.

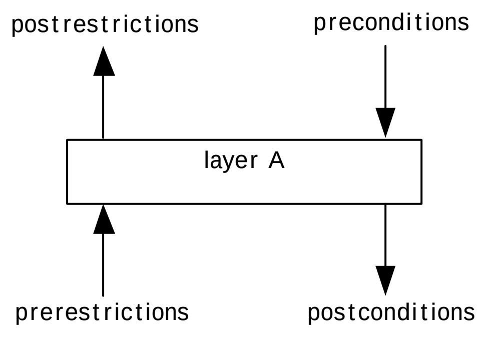


Figure 80 *Flow of conditions and restrictions through a GenVoca layer*

There are many ways of implementing conditions and restrictions. In the most simple case, layers could propagate their properties represented as attributes and check for required or undesired properties. The details of a slightly more elaborate implementation are described in [BG96].

The validation checking mechanism based constraint propagation also allows the generation of detailed warnings and error reports, which actually give specific hints of how to repair the type expression [BG96].

The composition constrains in a GenVoca domain model correspond to the configuration constraints we discussed in the context of feature diagrams (see Section 5.4.2) or to the conditions and annotations of Draco transformations (see Section 6.4.1). As we remember, constraints can be used in at least two ways: to validate a given configuration or to automatically complete a partial configuration. Validation is based on detecting constraint violations and reporting an error. Automatic completion is based on inferring property values based on constraints and a partially specified configuration. For example, given the property of a subsystem *synchronized* and the constraint that a synchronized subsystem has to use synchronized data structures, we can infer that the data structures used in this subsystem also have to be *synchronized*. This knowledge could be used to automatically insert the concurrent layer into the type expressions defining the data types, whenever the property of the subsystem is *synchronized*. We will discuss this kind of automatic configuration in the next section.

6.4.2.6 Transformations and GenVoca

The GenVoca model, as we described it so far, represents a typical example of the compositional approach to generation. Each layer represents a relatively large portion of the generated system or component. But as we stated in Section 6.2, compositions may also be viewed as transformations. From the transformational perspective, each GenVoca layer represents a large-scale refinement[64] and type expressions can be viewed as a hierarchy of refinement applications. For example, the type expression

```
concurrent[size_of [unbounded[managed[heap]]]]
```

when interpreted in a top-down order,[65] denotes the application of the concurrent refinement, then of size_of, etc. We can illustrate this refinement using the operation read_size() as an example (see Figure 81). The refinement starts with the empty read_size() operation declared in the realm DS_size. concurrent refines read_size() by inlining the highlighted code. size_of finishes the refinement by inlining direct access to counter.

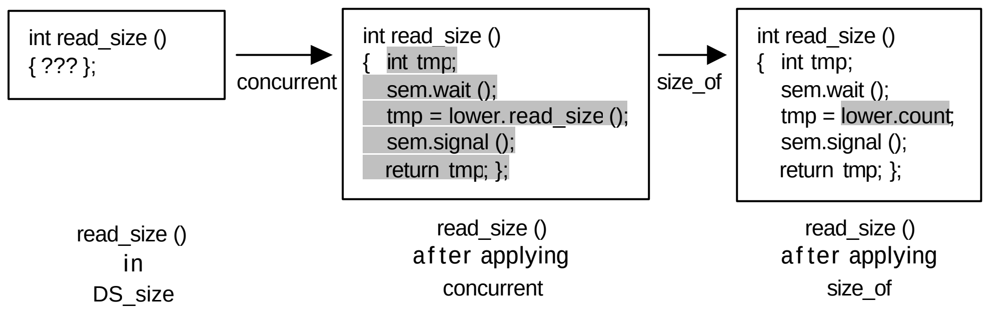


Figure 81 *Refinement of* read_size() *by successive inlining*

The transformational view of GenVoca suggests us that transformation systems certainly represent a possible implementation platform for GenVoca domain models.

Beyond this transformational interpretation of GenVoca layers, there is also an opportunity of a truly transformational extension to GenVoca: transformations on type expressions [BCRW98]. We will explain the usefulness and the concept of such transformations in the remainder of this section.

Type expressions such as

```
bag[concurrent[size_of [unbounded[managed[heap]]]]]
```

are still quite close to the implementation level since each layer represents a single concrete refinement. The layers correspond to features which were determined by factoring out commonalities and variabilities of concrete code of different systems or subsystems. The main goal of this factorization was to minimize code duplication and to be able to compose these features in as many ways as possible. All the features are concrete since each of them is directly implemented as a layer. The writer of type expressions has to know the order of the layers, the design rules, and also these layers that can be regarded as implementation details. However, the user can often provide a more abstract specification of the system or component he needs. For example, he could specify some usage profile for a data container by stating abstract features such as the calling frequency of insert, update, remove and other operations. This information could be then used to synthesize an appropriate type expression. The synthesis is complicated by the fact that different layers have different influence on the performance parameters and that there are a vast number of different type expressions for a given GenVoca domain model. Thus, exhaustive search for a solution is usually not practical and we have to use heuristic search methods. This search problem is usually addressed in transformations systems by means of inference-based scheduling of transformations deploying rules, procedures, tactics, and strategies. We have already discussed these components in the context of Draco (see Section 6.4.1). In [BCRW98], Batory et al. describe a prototype of a design wizard that is based on the cost model of various operations on data structures and applies transformations to type expressions which insert, delete, or replace layers in order to improve the performance characteristics of expressions defining data containers. As already stated, in general, the kind of knowledge needed for code synthesis includes not only constraints on component combinations but also constraints, procedures, and heuristics mapping from higher-level specifications to lower-level specifications. Until now, knowledge-based configuration has been predominantly studied in the context of AI Configuration and Planning and applied to configure physical systems.

In Chapter 10, we will present a case study, in which we actually treat a GenVoca domain model as an implementation language and we will deploy some simple metaprogramming techniques to translate specifications in a more abstract domain-specific language into GenVoca type expressions.

6.4.2.7 Class Libraries, Frameworks, and GenVoca

Class libraries represent libraries of relatively small components implemented using OO techniques and intended to be called by application programs. Examples of class libraries are container libraries such as the Booch C++ Components [Boo87] or the STL [MS96]. Frameworks, on the other hand, are reusable frames which can be customized by completing them with specialized components. OO frameworks are typically implemented as sets of cooperating classes and application programmers can customize them by integrating specialized classes into them. The user-defined classes are called by the framework and not the other way round. Examples of frameworks are ControlWORKS [CW] and Apple's MacApp [Ros95].

We have already seen how to use the GenVoca model to organize class libraries in Section 6.4.2.3. As described in [SB98a, SB98b], the GenVoca model is also useful for organizing OO frameworks. As already stated, OO frameworks are sets of cooperating classes. In fact, not all classes of a framework communicate with all other classes, but they are rather involved in smaller, well defined *collaborations*. Each class participating in a collaboration plays a certain role and this role usually involves only some of its methods. A collaboration is then defined by a set of roles and a communication protocol between the objects participating in the collaboration. One object can play different roles in different collaborations at the same time. This view of the world is the cornerstone of the analysis and design methods called *role modeling* [Ree96, KO96] and *collaboration-based design* [VHN96, Rie97, VH97]. According to this methods, concrete classes are obtained by composing the roles they play in different collaborations. Such designs are then implemented as frameworks. Unfortunately, when using the conventional framework implementation techniques (e.g. [GHJV95]), the boundaries between collaborations are lost in the implementation. However, we can use the GenVoca model to encapsulate each collaboration as a single layer. This way, we not only preserve the design structure in the implementation, but we can also conveniently configure frameworks in terms of collaborations using type expressions and easily extend the whole model with new (e.g. alternative) layers.

6.4.2.8 Components and GenVoca

Component-based software engineering is currently a rapidly developing area (see e.g. [Szy98]). The main contributions in this area in the 90ies include component technologies such as CORBA [OMG97, Sie96]), COM[66] [Ses97, Box98], ActiveX[67] [Arm97], and JavaBeans [Sun97]. The first two technologies enable the communication of independent, distributed components, which are often written in different languages and run on different platforms (at least with the first technology). ActiveX and JavaBeans define models for the components themselves by standardizing how they advertise their services, how they should be connected, delivered to the client machine, etc. Most of these technologies emphasize the binary character of the components: their services should be exposed through a binary standard.

It is interesting to note that some of the concepts found in Draco or GenVoca are part of the component models ActiveX and JavaBeans. For example, both models use attributes to describe parameterized properties of components and support the communication between components at composition time to exchange and modify these attributes. This communication enables the components to tune to each other by selecting appropriate implementation algorithms, parameters, etc., at composition time.

Unfortunately, the configuration of features within the ActiveX or JavaBeans components is usually based on dynamic binding. This might be desirable for features that have to remain reconfigurable at runtime. But features that are provided to be used in different constant configurations in different systems should be implemented using static binding and domain-specific code optimization that can actually "interweave" or "merge" the code pieces implementing various features together, as in hand-optimized code (see [Big97]). The failure to do so causes problems which can be observed in many contemporary OCX controls which are often bulky components with a large number of runtime flags. First, the configuration through runtime flags leads to inefficiencies that, even if they may be acceptable in GUIs, are prohibitive in many other contexts (e.g. automatic control, numerical computing, image processing, etc.). Moreover, runtime flags (including dynamic polymorphism) require the unused parts of the code to be present in the final component and this results in wasting memory space.

Generation models, such as GenVoca, provide a solution to this problem by allowing to optimize and regenerate a component for the specific context of use based on the settings in the customization interface. They also are orthogonal to the component models and can be integrated into the latter. For example, Batory et al. describe in [BCRW98] *ContainerStore*, an Internet-based component server for data containers. Containers can be requested from the server over the Internet by specifying the required properties and the requested component is generated and then delivered to the user. Biggerstaff refers to such component servers as a *virtual libraries* [Big97].

Generation techniques seem to be particularly useful for the implementation of the components of large software systems. As Biggerstaff notes in [Big97], conventional composition techniques such as function and method calls seem to be adequate for composing subsystems of large software systems (typically over one million lines of code) since the time spent in the subsystems is often far more longer than the time needed for the communication between them. Thus, it appears to be reasonable to compose large systems from components using conventional technologies rather than optimizing generation technologies. On the other hand, the latter are very appropriate at the component level, which has also been their usual scale of use. For example, the GenVoca model has been used to generate systems (or subsystems) up to some 100 thousands of lines of code (e.g. Genesis 70 KLOC [SB93]).

6.4.2.9 OO Languages and GenVoca

None of the current main-stream OO languages provides all the features needed to adequately implement GenVoca domain models, although C++ is powerful enough to express practical GenVoca domain models. In Section 6.4.2.4, we have seen how to implement layers, vertical and horizontal parameters, upward propagation of operations, types, and constants, and downward propagation of types and constants in C++.

Also the translations between higher level representations and GenVoca type expressions discussed in Section 6.4.2.6 can be expressed in C++ using template metaprogramming (see Chapter 8). Properties of components can be encoded as member constants or types and propagated as described in Section 6.4.2.4. In conjunction with template metaprogramming, they can also be used to select appropriate implementation algorithms, components, parameters, etc., at compile time. We will demonstrate these techniques in Chapter 8.

Since C++ does not support constrained type parameters, there is no way to implement realms. The bottom-up instantiation order of nested templates also limits the downward type propagation. Finally, although attributes needed for composition validation can be encoded and propagated as constants or types, composition validation still cannot be adequately implemented in C++ because of the inability to issue user defined warnings and error reports at compile time. In other words, we can prevent a semantically incorrect type expression from compiling, but the user will receive cryptic error reports coming from the middle of the component implementation. In our experience, the last limitation is the most severe.

Java provides even less support for implementing GenVoca models since it lacks type parameters and compile-time metaprogramming. A possible solution to this problem is provided by the *Jakarta Tool Suite* (JTS) [BLS98], which is a transformation-based pre-compiler for Java allowing us to implement language extensions and supporting compile-time metaprogramming. JTS has been used to extend Java with features for implementing GenVoca domain models.

Ideally, we would like to have a metaprogramming language allowing us to write components and to manipulate and modify them both at compile time and runtime — all in the same language. With such language we could express configuration knowledge equally usable at compile time and runtime.

6.4.3 Intentional Programming

Intentional Programming (IP) is a ground-breaking extendible programming environment under development at Microsoft Research since early nineties. In particular, IP supports the development of domain-specific languages and generators of any architecture (e.g. Draco or GenVoca), as we will see soon, in a unique way. The idea of IP was originated by Charles Simonyi,[68] who also leads its development [Sim95, Sim96, Sim97, ADK+98].

The main idea of IP is not to view computer languages as separate, fixed entities, but rather focus on the abstractions (i.e. language features) they provide and to allow the use of these abstractions in a program as needed without being confined to one language at a time. This view is very different from the conventional, computer science view, wherein a compiler is developed for one language (e.g. by specifying its grammar and the code generation actions and presenting this specifications to a compiler generator such as the Unix utility *yacc*). The main problem with this view is that real programs require abstractions from different domains, so that one domain-specific language is not sufficient [Sim97]. As a consequence of this observation, IP takes rather an incremental view: new abstractions are added to the programming system as they are needed. Of course, each new abstraction is expressed in terms of the already defined ones.

In the IP terminology, these abstractions or language features are referred to as *intentions* [Sim95, Sha98]. As programmers conceive solutions in terms of domain concepts in their brains (Chapter 2), the main purpose of intentions in IP is to represent these domain concepts directly (or *intentionally*), i.e. without any loss of information or obscure language idioms.

The IP programming environment can be viewed as a special kind of transformation system. The most unique feature of this transformation system is the lack of a parser: abstract syntax trees (ASTs) are entered and edited directly using commands. Since IP no longer represents programs as text, there is no need for a parser to construct the AST from a textual representation (which is not there). There are exciting advantages of direct editing and not having a parser, but we will discuss them a bit later. Of course, we still need parsers for importing legacy code written in textual legacy languages, such as C++ or Java, into the IP system.

Furthermore, IP uses a framework-based approach: the nodes of the AST use default and user-defined methods for displaying, editing, optimizing, transforming, and generating code. Just about any aspect of the system can be specialized using methods including debugging and source control. When new intentions (i.e. language features) are loaded into the system, they bring along all the methods implementing their behavior for these aspects.

6.4.3.1 What Is Wrong With the Fixed Programming Language View?

One could argue that there are at least two alternative ways to deal with the problem of a program requiring many domain-specific abstractions. One solution is to use a general-purpose programming language with abstraction mechanisms such as procedures or objects, which allow us to define our own libraries of domain-specific abstractions. This is the conventional and widely-practiced solution. The second solution would be to provide one comprehensive application-specific language per application type, so that the language contains all the domain-specific abstractions needed for each application type as part of this language. Unfortunately, both solutions have severe problems.

6.4.3.1.1 Problems with General-Purpose Languages and Conventional Libraries

There are four main problems with the general-purpose programming language and conventional library approach: loss of design information, code tangling, performance penalties, and no domain-specific programming support.

- *Loss of design information*: When using a general-purpose programming language, domain-specific abstractions have to be mapped on to the idioms of the programming language. The resulting code usually includes extra clutter and fails to represent the abstractions declaratively (i.e. intentionally). Even worse, some of the domain information is lost during this transformation. For example, there are many ways to implement the *singleton* pattern [GHJV95].[69] And given a particular implementation code only, it is not one hundred percent certain that the intention of the code was to implement the singleton pattern. This information could be included in a comment, but such information is lost to the compiler. On the other hand, with the domain-specific (or application-specific) language approach, one would introduce the class annotation *singleton*, which would allow us to unambiguously express the singleton intention. The loss of design information makes software evolution extremely difficult since changes are usually expressed at the higher level of abstraction. Using the general-purpose programming language and library approach, the program evolution would require code analysis to recover the intended abstractions, which, as we illustrated with the singleton concept, is an impossible task.

- *Code tangling*: Programming problems are usually analyzed from different perspectives and an adequate, intentional encoding should preserve the separation of perspectives (e.g. separating synchronization code from functional code). As we will see in Section 7.6, in most cases, achieving this separation requires domain-specific transformations, but such cannot be encapsulated in conventional libraries, unless the language supports static metaprogramming.[70] In other words, we need to put some code extending the compiler to be executed at compile time into the library, but this is usually not supported by current library technologies. Thus, when using conventional procedural or class libraries, we are forced to apply these transformations manually. Thus, we produce tangled code, which is hard to understand and to maintain. We will investigate these issues in Chapter 7 in great detail.

- *Performance penalties*: The structure of the domain-level specification does not necessarily corresponds to the structure of its efficient implementation. Unfortunately, the main property of procedures and objects is that they preserve the static structure of a program into runtime. The compiler can apply only simple optimizations since it knows only the level of the implementation language, but not the domain level. For example, as we discussed in Section 6.4.1, no compiler could possibly optimize the Taylor expansion code into `X*X`(see Figure 63). A considerable amount of domain-specific computation at compile time might be required in order to map a domain-level representation into an efficient implementation. With the general-purpose programming language and library approach, no such computation takes place (again, this would require static metaprogramming).

- *No domain-specific programming support*: Domain-specific abstractions usually require some special debugging support (e.g. debugging synchronization constraints), special display and editing support (e.g. displaying and editing pretty mathematical formulas), etc. Such support would require that libraries, in addition to the procedures and classes to be used in client programs, also contain extensions of programming environments. However, current technologies do not support such extensions.

6.4.3.1.2 Problems With Comprehensive Application-Specific Languages

Given a comprehensive application-specific language containing all language features we need, we could implement a programming environment which would provide an adequate support of the language, i.e. implementing necessary optimizations and debugging, displaying, and editing facilities. The language itself would allow us to write intentional, well separated code. In other words, we would solve all the problems mentioned in the previous section. Unfortunately, there are three major problems with the comprehensive-application-specific-language approach: parsing problem, high cost of specialized compilers and programming environments, and problems of distributing new language extensions.

- *Parsing problem*: The problem of conventional text-based languages is that it is not possible to add more and more new language features without eventually making the language unparsable. C++ is a good example of a language reaching this limit. C++ has a context-sensitive grammar, which is extremely difficult to parse. The requirement of being able to parse a language also imposes artificial constraints on the notation. For example, in C++, one has to insert an extra space between the triangular brackets closing a nested template (e.g. `foo<bar<foobaz> >`) in order to differentiate it from the right-shift operator (i.e. `>>`). The requirement of parsability represents an even more severe restriction on domain-specific notations since they are usually full of ambiguities in the sense of parsability. Even the simplest notations found in mathematical books would be impossible to be directly represented as conventional, text-based computer languages. The ambiguity of a domain-specific notation does not imply the ambiguity of the underlying representation. In fact, the ambiguity of the domain-specific notations is caused by the fact that they do not have to show all the details of the underlying representation. For example, when editing text in a WYSIWYG[71] text editor, such as Microsoft Word, one does not see whether two paragraphs were assigned the same style (e.g. *body text*) since they could have the same text properties (e.g. font size, font type, etc.). In other words, the WYSIWYG view is ambiguous with respect to the underlying representation since we cannot unambiguously determine the style of a paragraph based on its text properties. An example of an unambiguous textual representation is the $\TeX$ file format [Knu86]. However, this representation is not WYSIWYG. Other limitation of textual representation include being confined to one-dimensional representations, no pictures, no graphics, no hyperlinks, etc.

- *High cost of specialized compilers and programming environments*: The cost of developing compilers and programming environments is extremely high. On the other hand, large portion of the compiler and programming environment infrastructure can be reused across languages. Thus, the development of large numbers of separate, extremely specialized programming environments from scratch does not seem to be economical.

- *Problems of distributing new language extensions*: Finally, even if we extend a language with new features, dissemination of the new features is extremely difficult since languages are traditionally defined in terms of a fixed grammars and compilers and programming environments do not support an easy and incremental language extensibility. That is why, as Simonyi notes in [Sim97], new useful features have the chance to reach a large audience only if they are lucky enough to be part of a new, widely spread language (e.g. interfaces in Java). Such opportunities are, however, a rare occasion. On the other hand, Java is also a good example of a language that is extremely difficult to extend because of its wide use and the legacy problem that comes with it.

6.4.3.1.3 The IP Solutions

IP addresses the problems listed in the two previous sections as follows:

- Loss of design information and code tangling are avoided by providing domain-specific language extensions. In IP, language extensions are packaged into *extension libraries*, which can be loaded into the programming environment in order to extend it.

- Performance penalties are avoided by applying domain-specific optimizations, which are distributed as part of extension libraries.

- Domain-specific programming support (e.g. domain-specific debugging, editing, displaying, etc.) can also be provided as a part of the extension libraries.

- The parsing problem is solved in IP by abandoning the textual representation altogether and allowing direct and unambiguous entry of AST nodes using commands.

- Some of the high development cost of specialized compilers and programming environments is reduced by providing a common reusable programming platform (i.e. the IP system), so that only the language extensions need to be programmed. Furthermore, single new language features can be integrated into many existing larger frameworks of language features.

- Extension libraries represent a convenient and economical means for distributing language extensions.

We will discuss the IP solutions in the following section in more detail.

6.4.3.2 Source Trees

All IP programs are represented as ASTs, also referred to as *source trees* (since they represent the source of the programs). The source tree of the expression `X+Y` is shown in Figure 82. The nodes representing X and Y are referred to as the *operands* of the + node. Operands are always drawn below their parent node and we count them in the top-down direction, i.e. X is the first operand and Y is the second operand. The nodes of the source tree are also referred to as *tree elements*.

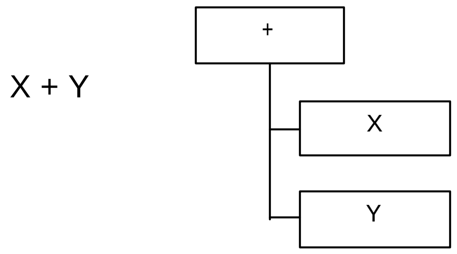


Figure 82 *Source tree of* X+Y *(the graphical notation is based on [Sha98])*

Each tree element has its declaration. We use dotted lines to point to the declarations (see Figure 83). In our example, the nodes representing +, X, and Y are really just references to their declarations. Declaration nodes are designated by putting "DCL" in front of their names. As we will later see, only the solid lines organize the nodes into trees. The dashed lines, on the other hand, connect the nodes to form a general directed graph. This is the reason why the solid lines are often referred to as *tree-like links* and the dashed lines as *graph-like links*.[72]

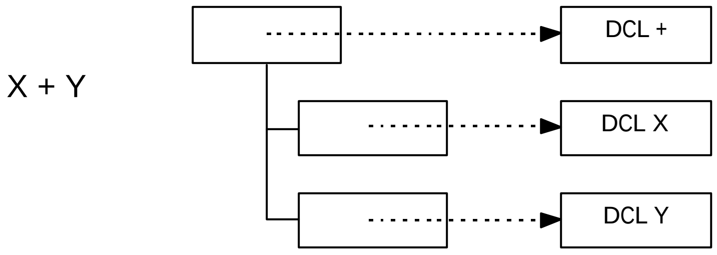


Figure 83 *Source tree of X+Y with declarations*

Declarations are trees themselves. The source tree representing the declaration int X; is shown in Figure 84. The root node of a tree representing a declaration — just as any node — also has a declaration, which, in our case is the special declaration DCL (represented as the DCL DCL node in Figure 84). In fact, the declaration of the root of any declaration is always the declaration DCL. The first operand of the of the root node is usually the *type* of the declaration. In our case, it is the integral type int.

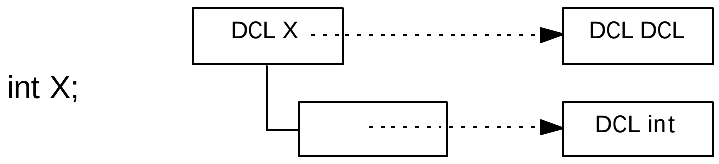


Figure 84 *Source tree of* int X;

Figure 85 shows the fully expanded source trees of a small program declaring the variables X and Y and adding them together. The three source trees enclosed in gray boxes represent the three statements of the program. They reference other trees provided by the system, such as the declaration of + and the declaration of int. There are also the three (meta-)declarations TypeVar, Type and DCL. As we already explained, DCL is the declaration of all declarations and, consequently, also the declaration of itself. Type is the type of all types (and also of itself). +, int, TypeVar, Type, and DCL are all examples of *intentions* — they define language abstractions. The nodes referencing them are their *instances*. We also refer to nodes not being *declarations* (i.e. nodes having other declarations than DCL) simply as *references*.

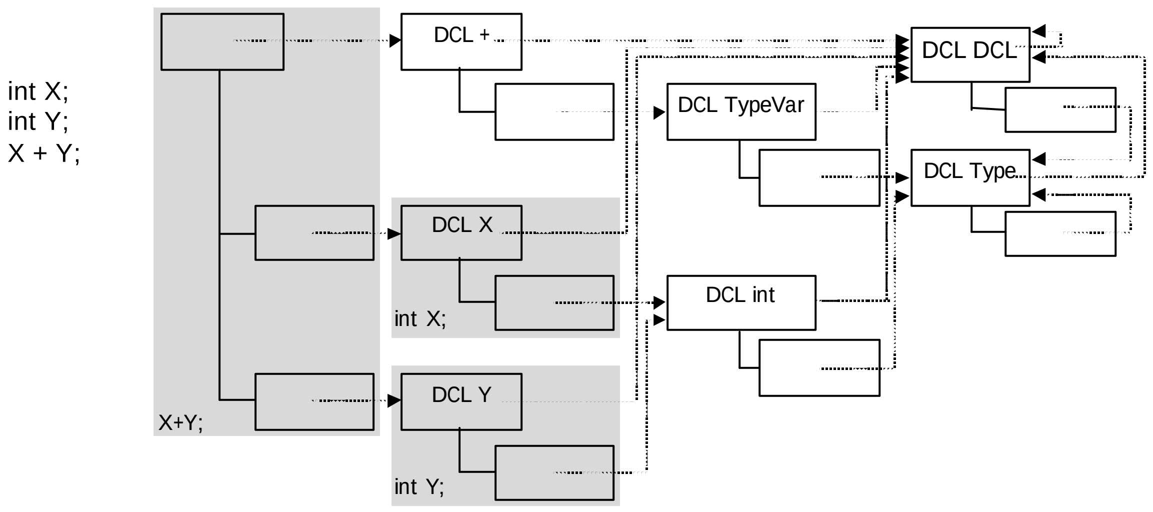


Figure 85 *Fully expanded source tree of a small program*[73]

Each tree element is implemented as a collection of fields (see Figure 86). There is the optional field *name*. For example, the name X is stored in the name field of the DCL X node. References usually do not have names. Names are exclusively used for the communication with the programmer, i.e. to be displayed on the screen or to be typed in. The reference to DCL X in the expression X+Y is a true pointer, i.e. it does not use the name. When the system displays X+Y, the name X is retrieved from DCL X The field *operator* (Figure 86) points to the declaration of the node. Since every node has a declaration, every node also has the operator field. The optional field *constant data* is for storing constant data. For example, we can use this field to store the bits representing a constant. Figure 87 shows a DCL X of type int and with the initial value 1. The value is stored in the constant data field of the second operand. The operator field of this operand points to DCL constant, which provides the interpretation for the value. A node can also contain a number of *annotations*. Annotations are somewhat similar to operands, but they are used to provide some extra information about the node. For example, we could annotate the declaration of X with const, i.e. const int X = 1;, in order to specify that the value of X should not change. Annotations are a convenient means of expressing preferences about what kind of implementation to generate, e.g. we could annotate the type *MATRIX* with *diagonal* to tell the system to generate optimized code. Finally, there are the *operand* fields pointing to the operands.

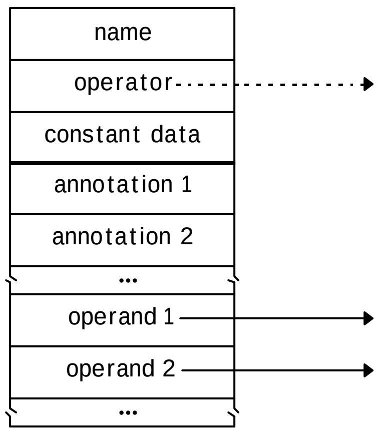


Figure 86 *Fields of a tree element*

Source trees are stored on disk as binary files. Thus, they can contain bitmaps, hyperlinks, drawings, etc.

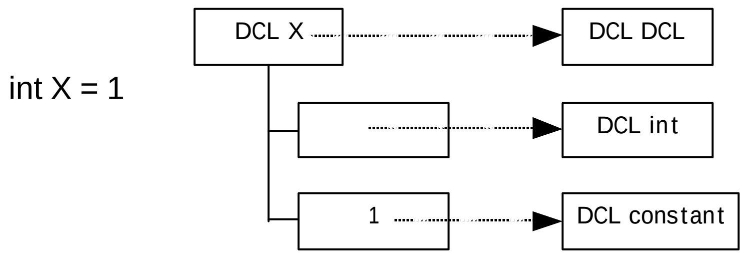


Figure 87 *Source tree of* int X = 1;

6.4.3.3 Extension Methods

As we already explained, the IP system is organized as a framework calling default and user-defined methods, where each method defines some aspect of the system, e.g. displaying, compiling, typing in intentions, etc. These methods are referred to as *extension methods* (or *Xmethods*) since, as a new extension method is defined and loaded, it extends the IP system.

Extension methods are classified according to their purpose. We have the following categories of methods [Sha98]:

- *Reduction methods*: The process of transforming source trees into lower-level trees is referred to as *reduction*. The process continues as long as the tree being reduced contains only instances of a predefined set of intentions for which machine code can be directly generated (in the phase called *code generation*). This representation is referred to as *reduced code* or *R-code*. Reduction involves the replacement of instances of intentions by instances of lower-level intentions and is performed by *reduction methods*. Reduction methods, as we explain later, are attached to intentions. Thus, the reduction methods of an intention know how to compile its instances and, effectively, they implement the semantics of the intention. The methods may gather information from the context of the intention instance being reduced, e.g. from other parts of the program, before replacing it by the appropriate code. In general, there is more than one set of R-code intentions, each defined for a different target platform, e.g. the Intel 86 family of processors or Java bytecodes. The reduction methods can usually reduce the source tree towards a different target platform, based on the context settings. However, it is possible that not all high-level intentions can be reduced to a given set of R-code intentions.

- *Rendering methods*: The display of a source tree on the screen is accomplished by the *rendering methods* attached to the intentions being referenced. The rendering methods may display the tree using true two-dimensional output (as in mathematical formulas), graphical representations, embedded bitmaps, etc. They can also display information coming from remote places, e.g. when defining the implementation of a procedure, its signature is automatically displayed based on the information contained in the procedure prototype (i.e. declaration), even if the prototype is defined in a different source tree. In other words, the signature is not being stored redundantly, but only displayed twice for better readability. In general, rendering methods would usually display just some aspects of the tree at a time, i.e. one would implement different views of the tree rather than displaying all information at once.

- *Type-in methods*: Type-in methods are called when the source tree is entered or manipulated. When a reference to an intention is entered, a special method can be defined to insert an appropriate number of place holders as operands of this instance (e.g. after typing the name of a procedure, the appropriate number of place holders for the arguments are inserted) or to do any other kind of special editing (e.g. typing in a type will replace the type by a declaration of this type). There are methods defining how to select the elements shown on the screen, the tabbing order, etc.

- *Debugging methods*: Since the reduction methods can perform arbitrary, user-defined computation to generate the executable code, there might be little or no structural correspondence between the source tree and the runtime code. This lack of correspondence renders the conventional debugging model of stepping through the source impractical. While domain-specific notations will have to provide their specialized debugging models for the client code (e.g. GUI specifications are debugged visually by examining the graphical layout or real time constraints can be highlighted when they are violated), the programmer of the reduction methods still needs some debugging facilities in order to track down bugs in the reduction process. The IP system supports the programmer in a unique way. At any time during the reduction process, a reduction method can call a snapshot method, which takes a snapshot of the current state of the tree. Later, when the compiled client code is executed, it is possible to step through the code at any of the levels defined by the snapshots. The mapping between the code positions in the executable and in any of the levels is taken care of automatically by the system.[74] However, not all values of the abstract variables of a higher level can be inspected since they might not be present in the executable (e.g. they could have been optimized away or represented by other variables). However, the programmer has the opportunity to write his own *debugging methods* which may map back the values in the executable onto the variables of each of the intermediate levels and the source level, so that they can be inspected.

- *Editing methods*: Since the source is represented as an AST, it is quite easy to perform mechanical restructuring of the source. Methods performing such restructuring are called editing methods. Some editing methods may be as simple as applying De Morgan to logical expressions or turning a number of selected instructions into a procedure and replacing them by a call to this procedure (in IP this is done with the *lift* command), and some may perform complex design-level restructuring of legacy code (i.e. refactorings).

- *Version control methods*: Version control methods allow us to define specialized protocols for resolving conflicts, when two or more developers edit the same piece of code.

The are also other methods which do not fit in any of these categories.

Extension methods (or Xmethods), as a language mechanism, correspond to methods in the OO paradigm. However, the specific implementation of methods in the IP system is unique. Extension methods have two conceptual parts: a prototype, which is basically a signature defining the arguments and their type, and multiple implementation bodies attached to different intentions. This implementation combines the best of C++ and Smalltalk methods. In C++, the definition of pure virtual methods corresponds to defining Xmethod prototypes. Derived classes may then provide specific implementations to the pure virtual methods defined in the base class. So C++ limits the implementors of methods to derived classes only. This is clearly a severely limited form of polymorphism (called *inheritance-bound polymorphism*), which, for example, lead to the introduction of interfaces in Java. In Smalltalk, on the other hand, any class can choose to provide an implementation for any message (this is called signature-bound polymorphism). Thus, a Smalltalk method is extremely generic since it works with any objects implementing the messages sent to them in the method. Unfortunately, since the matching between messages and methods in Smalltalk is based on their names, it is not clear, whether two methods implemented in different classes but having the same name implement semantically the same message or whether their equal names are just a coincidence. This is a severe problem when trying to change the name of a message or trying to understand someone else's code. Xmethods implement the Smalltalk flavor but without name ambiguities since the methods have "free-standing" (i.e. not attached to any intention) prototypes which identify them uniquely. If an intention defines an implementation body for a particular method, it references the prototype explicitly using a pointer (i.e. using a graph-like link). Compared to Java interfaces, Xmethods avoid the necessity to define many small, artificial interfaces, such as sortable, printable, clickable, etc.

Now, we will take a look at a sample Xmethod definition. As we already explained, each intention has to provide the methods for reducing its instances. In particular, in IP there is the method KtrTransformTe, which is called by the system when a tree element needs to be reduced. In other words, the system provides the prototype for this method and each intention defines its specific implementation body for this method. The prototype of KtrTransformTe is defined as follows (in the default rendering view):

```
MethodDefaultPfn(KtrTransformTeDefault) XMETHOD KTR KtrTransformTe(PVIP pvip, HTE hte);
```

This corresponds to a source tree for a declaration of the type XMETHOD and the name KtrTransformTe. Furthermore, the declaration has the annotation MethodDefaultPfn with the operand referencing DCL KtrTransformTeDefault (which is the default implementation body for the Xmethod defined elsewhere) and operands defining the Xmethod return type KTR and the arguments pvip and hte. KTR stands for *kind of transformation result*, which is an enumerated type used to report on the success of the method. PVIP is a pointer to a virtual inheritance path (explained at the end of this section) and HTE is a handle to a tree element. hte is the handle to the tree element being transformed.[75]

Assume that we are defining a new intention, namely the new type MATRIX. If we use MATRIX in a client program, the program will contain references to MATRIX and the IP system will call the KtrTransformTe method on these references when reducing the client program. Thus, we need to define the KtrTransformTe method for MATRIX. This can be done as follows:

```
DefineXmethod KtrTransformTe for (MATRIX) dxmMATRIXKtrTransformTe:
   KTR KtrTransformTe(PVIP pvip, HTE hte)
{
...// implementation statement list
}
```

The corresponding source tree is shown in Figure 88. Thus, the implementation body of the KtrTransformTe method is defined as a declaration of type DefineXmethod.[76] The reference to the method prototype (i.e. DCL KtrTransformTe) and the reference to MATRIX are the operands of the reference to DCL DefineXmethod. Please note that the section of the code implementing the method body highlighted below

```
DefineXmethod KtrTransformTe for (MATRIX) dxmMATRIXKtrTransformTe:
   KTR KtrTransformTe(PVIP pvip, HTE hte)
{
...// implementation statement list
}
```

is not represented in the tree. This section is displayed by the rendering methods and the needed information is retrieved from DCL KtrTransformTe. Also, DCL MATRIX is referenced through a list since we could define the same implementation of a method for more than one intention.

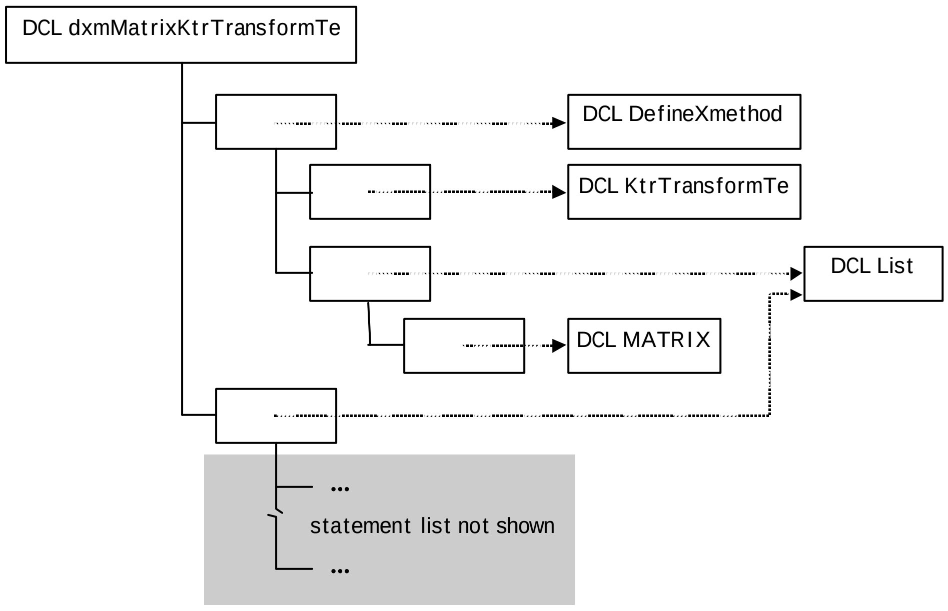


Figure 88 *Source tree of* DCL dxmMatrixKtrTransformTe[77]

Currently, all intentions for the language C are implemented in the IP system (implementation of C++ and Java intentions is underway). Thus, we can use C to implement the bodies of methods.

As we already explained, dxmMatrixKtrTransformTe will be called whenever a reference to MATRIX needs to be reduced. But in client code, MATRIX is usually used for declaring variables of type MATRIX, e.g.

```
MATRIX m;
```

Thus, client programs contain references to MATRIX, declarations of type MATRIX, and also references to declarations of type MATRIX (i.e. variables of type MATRIX) and we still need to define the transform method implementations for the latter two. The question that arises is where to attach these implementations? They have to be somehow associated with MATRIX since they must be in the server code (i.e. the code defining MATRIX). The particular solution in IP is to define so-called *virtual intentions* representing declarations of the type MATRIX and the references to such declarations. These intentions are called virtual since they are never referenced from the client code, but they represent certain patterns in the client code — in our case declarations of the type MATRIX and references to such declarations. The virtual intentions are than associated with the type MATRIX using the two special annotations DclOfThisTypeVI and RefToDclOfThisTypeVI as follows:

```
VI viDclOfTypeMATRIX;
VI viRefToDclOfTypeMATRIX;
DclOfThisTypeVI(viDclOfTypeMATRIX) RefToDclOfThisTypeVI(viRefToDclOfTypeMATRIX) Type MATRIX;
```

This code first declares the virtual intentions viDclOfTypeMATRIX and viRefToDclOfTypeMATRIX, then it declares MATRIX to be a Type, and, finally, it associates both virtual intentions with DCL MATRIX through the annotations DclOfThisTypeVI and RefToDclOfThisTypeVI.

Now, we can declare the bodies of the transform method for declarations of type MATRIX and references to such declarations:

```
DefineXmethod KtrTransformTe for (viDclOfTypeMATRIX) dxmviDclOfTypeMATRIXKtrTransformTe:
   KTR KtrTransformTe(PVIP pvip, HTE hte)
{
...// implementation statement list
}

DefineXmethod KtrTransformTe for (viRefToDclOfTypeMATRIX)
dxmviRefToDclOfTypeMATRIXKtrTransformTe:
    KTR KtrTransformTe(PVIP pvip, HTE hte)
{
...// implementation statement list
}
```

An Xmethod can be called as follows

```
KTR ktr = hte.KtrTransformTe(hte);
```

where hte is a handle to a tree element. This method call will prompt the system to look for the appropriate implementation of the method KtrTransformTe to be invoked. If hte points to a tree element which is a reference to a declaration of the type MATRIX, the implementation dxmviRefToDclOfTypeMATRIXKtrTransformTe is invoked. If the tree element is a declaration of the type MATRIX, the implementation dxmviDclOfTypeMATRIXKtrTransformTe is executed. In these two and any other cases, the search starts at the type of the declaration. If no method is found, the search is continued in the types of the declarations of the declarations, etc. If no implementation is found, the default method (i.e. KtrTransformTeDefault) is executed. Effectively, this search represents the method inheritance mechanism in IP. For each method call, the system computes a so-called *virtual inheritance path* (VIP), which is used for the search. It is also possible to call a method supplying the inheritance path explicitly [Sha98]:

```
VipFromHte(&vip, hte);
ktr = &vip.HteTransform(hte);
DestroyVip(&vip);
```

This way, it is possible to manipulate the inheritance path before calling the method and to influence the default look-up process.

6.4.3.4 Reduction

The KtrTransformTe method is called by the system during the reduction process of a tree. Thus, the system decides when and on which tree element to call this method. On the other hand, it is necessary to synchronize the reduction of instances of intentions which depend on each other. For example, if one intention needs some information from another intention, the instances of the latter should not be reduced before the first intention gets a chance to look at them. The IP system provides a special dependency mechanism for accomplishing this goal: We can define a special method for an intention, which declares all the intentions of which instances are examined or created in the transform method of the first intention. Based on these declarations, the so-called *scheduler* [ADK+98], which is a part of the system, can determine when to call the transform method and on which tree element. If the intention A declares to examine the instances of the intention B, all instances of A will be reduced before reducing any instances of B. Now, if the intention C declares to create instances of B, all instances of C will be reduced first (so all instances of A come into existence to be examined by B), then all instances of A, and finally all instances of B. Given such dependency mechanism, deadlocks of the scheduling algorithm are possible. We can visualize them best using simple diagrams, wherein *examine dependencies* are represented as dashed lines and *create dependencies* are represented as solid lines [IPD]. Any cycle in the dependency diagram represents a deadlock (see Figure 89).

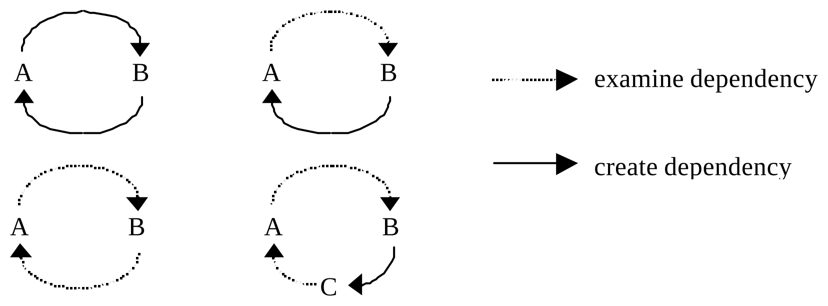


Figure 89 *Examples of deadlocks*

It turns out, that many practical designs lead to deadlocks. In general, many deadlocks are caused by the fact that intentions often do not really want to look at all the instances of another intention, but only some of them occurring in specific contexts. Unfortunately, the dependency mechanism allows us to declare dependencies on all instances of an intentions only. The challenge is that a dependency mechanism checking for instances in certain contexts (i.e. checking for certain patterns) would lead to an inefficient scheduling algorithm. However, most deadlocks can be resolved through some simple design changes (e.g. introducing extra intermediate-level intentions).

If a number of intentions wish to interact in some special way other than allowed by the standard dependency protocol, they can be declared to be members of a *guild* [ADK+98]. A guild is a special intention, whose transform method transforms its members in any fashion and order it wishes to. The guild itself has its dependencies on other intentions declared as usual. When its transform method is called, it gets the source tree containing, among others, instances of its members as a parameter and then looks for these instances and reduces them. Effectively, guilds allow us to write custom rewrite systems for some number of intentions. For example, the matrix operations +,-, and * are good candidates to be members of a guild since generating optimized code for matrix expressions requires looking for many neighboring operations at once, i.e. the intentions depend on each other and can be reduced in one method.

The body of KtrTransformTe method uses the tree editing API in order to transform the source tree. The tree editing API consists of a number of operations on tree nodes, such as accessing, adding, and deleting operands and annotations, reading and setting the operator, name, etc. (see [ADK+98] for details]). However, there is a set of rules of what is allowed and what not in the transform method (e.g. there are special limitation on deleting nodes and accessing other nodes can be done using special synchronizing questions only).

6.4.3.5 System Architecture

The IP system consists of the following components:

- *Editing and browsing tools*: The editing tools call the rendering and type-in methods for displaying and editing code. Browsing tools use the declaration links for navigation and names of intentions for string-based search.

- *Transformation engine and scheduler*: The scheduler calls the transform methods of intentions to reduce source trees into the appropriate R-code. From the R-code machine code for Intel processors (currently using the Microsoft back-end of the Visual product family) is generated or Java bytecodes.

- *Libraries of intentions*: The intentions available for programming are distributed in libraries. For example, there is a library containing all the C intentions. Loading this library allows writing C code. Other libraries, e.g. C++ intention library or libraries of domain-specific intentions, can be also loaded and the client programs may use them all at once.

- *Version control system*: The team-enabled IP version control system — as opposed to conventional text-based version control systems — works on the binary IP source files.

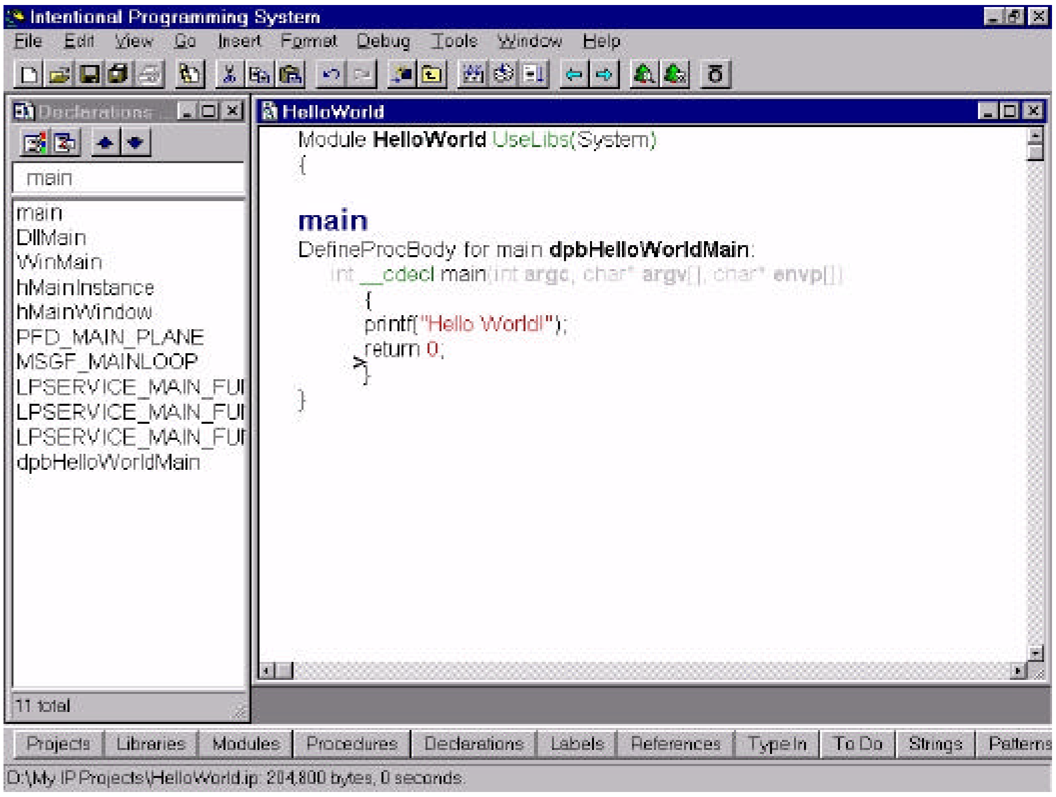
- *Parsers*: Language parsers, e.g. for C++ or Java, are available for importing legacy code into the IP system. This needs to be done only once. The code is then saved in the IP source tree file format.

The IP system was originally written in C including the C intentions. Then it was bootstrapped by compiling its C sources with the IP system in 1995 [Sim95]. This way, the system "liberated" itself from C and since then new intentions have been added, e.g. the intentions implementing Xmethods.[78] In other words, the IP is being developed using IP which, besides delivering the proof of concept, allows the IP developer team to continually improve the system based on their own immediate feedback. The size of the system is currently over one million source tree nodes.

6.4.3.6 Working With the IP Programming Environment

The IP programming environment supports all the usual programming activities: writing code, compiling, and debugging. Figure 90 shows a screenshot of a typical programming session. The editor subwindow contains a simple C "Hello World" program. The program can be stored on the disk in one binary IP source file referred to as a *project* or a *document*. In general, a program can consist from more than one document. To the right of document editor subwindow, the *declarations list tool* is shown. This tool allows the developer to search for declarations based on their names. A click on one of the displayed names opens the document containing the corresponding declaration in the current editor window. There are other browsing tools, such as the *references list tool* showing all the references to a certain declaration, *libraries list tool* enumerating all the currently opened libraries, *to-do list tool* displaying a list of to-do annotations, etc. There is also a tree inspector showing the exact tree structure of the selected code. Finally, one can jump to the declaration of a selected node by a mouse click.

The "HelloWorld" program can be compiled by simply pushing the compile button on the menu bar. This initiates the reduction process, which, if successfully completed, is followed by the generation of the executable. If there are syntax or semantic errors in the source, the error notifications are attached to the involved nodes and appear on the screen next to the erroneous position in the code in a different color. After a successful compilation, it is possible to step through the intermediate results of the reduction process, as they were recorded by the snapshot function which can be called for debugging reasons at various places in the transform methods of different intentions. The executable can be then debugged at the source level or any of the intermediate levels.


Intentional Programming System
File Edit View Go Insert Format Debug Tools Window Help
Declarations
HelloWorld
Projects Libraries Modules Procedures Declarations Labels References Type In To Do Strings Patterns

```
main
```

```
main
DllMain
WinMain
hMainInstance
hMainWindow
PFD_MAIN_PLANE
MSGF_MAINLOOP
LPSERVICE_MAIN_FUI
LPSERVICE_MAIN_FUI
LPSERVICE_MAIN_FUI
dpbHelloWorldMain
```

11 total

```
Module HelloWorld UseLibs(System)
{

main
DefineProcBody for main dpbHelloWorldMain:
    int __cdecl main(int argc, char* argv[], char* envp[])
        {
        printf("Hello World!");
        return 0;
        }
}
```

D:\My IP Projects\HelloWorld.ip: 204,800 bytes, 0 seconds.

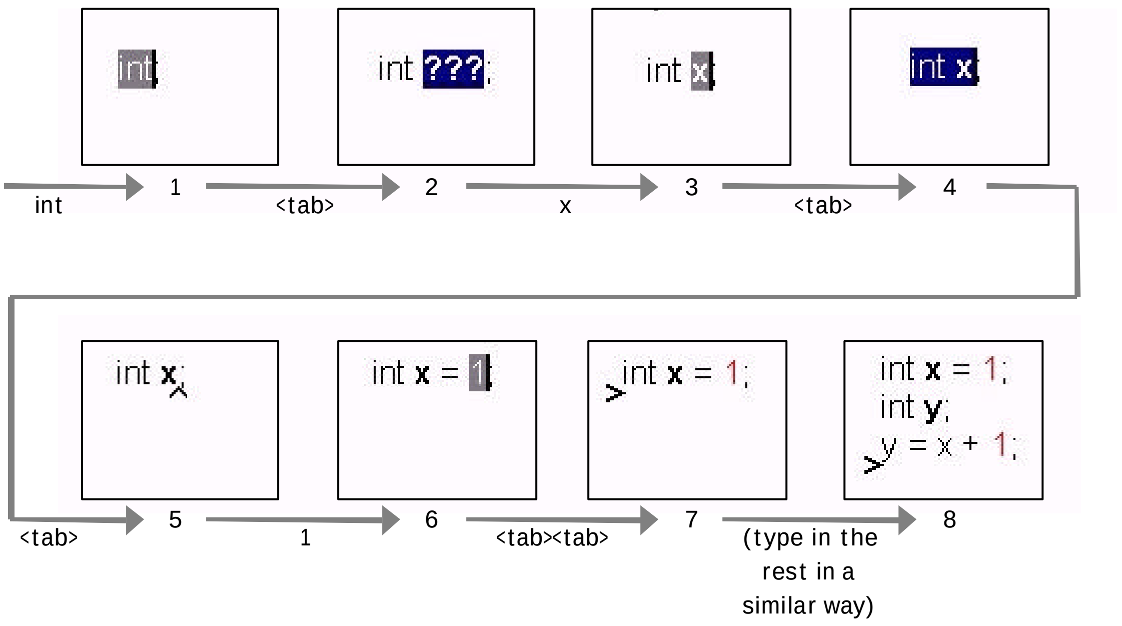
Figure 90 *Screenshot of a typical programming session with the IP system*

6.4.3.6.1 Editing

Probably the most unusual experience to the novice IP programmer is editing. This is since the programmer edits the tree directly, which is quite different from text based-editing. To give you an idea of how tree editing works, we will walk through a simple editing example. Figure 91 shows you how to type in the following simple program:

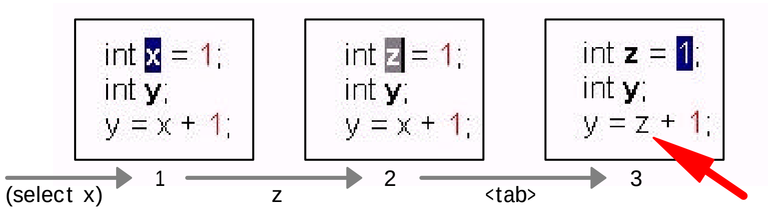
```
int x = 1;
int y;
y = x + 1;
```

Each box in Figure 91 shows you the editing screen after typing the text shown below the preceding arrow. We start with the empty screen and type in “int”. As we are typing it, the gray selection indicates that we have still not finished typing the token. We then finish typing the token by entering <tab> or <space> (the first is preferred since it automatically positions the cursor at the next reasonable type-in position). After typing <tab>, the system tries to find a binding for the token we have just typed in. In our case, the system finds that the name of the declaration of the type int matches the token. But before replacing the token with a reference to the int declaration, the system calls the type-in method of int. The method look-up actually finds a type-in method in Type, which is the type of int (see Figure 85). This method wraps int in a declaration, which results in the tree shown in Figure 84. This is so since types are usually typed in to be the types of declarations. The name of the new declaration is set to “???”. We can see the result in box 2 in Figure 91. Since we typed <tab> last, the declaration name (i.e. “???”) is now selected and we can type in the name of the declaration, in our case “x”. The result is shown in box 3. We finish typing this token with a <tab> and enter an extra <tab> to position the cursor for typing in the initializer (box 5), in our case 1. The result of entering 1 is the box 6 and the corresponding source tree after entering <tab> is in Figure 87. We did not type in the equal sign — it is displayed by the rendering method of the declaration. By entering an extra <tab>, we position the cursor behind the declaration statement in the current statement list and are ready to type in the next statement (box 7). We type in the following two statements in an analogous way. Please note that we actually explicitly enter the equal sign in the third statement since it denotes an assignment.



```
int

int ???;

int x

int x

int x;

int x = 1

int x = 1;

int x = 1;
int y;
y = x + 1;
```

Figure 91 *Typing in a simple program in IP*

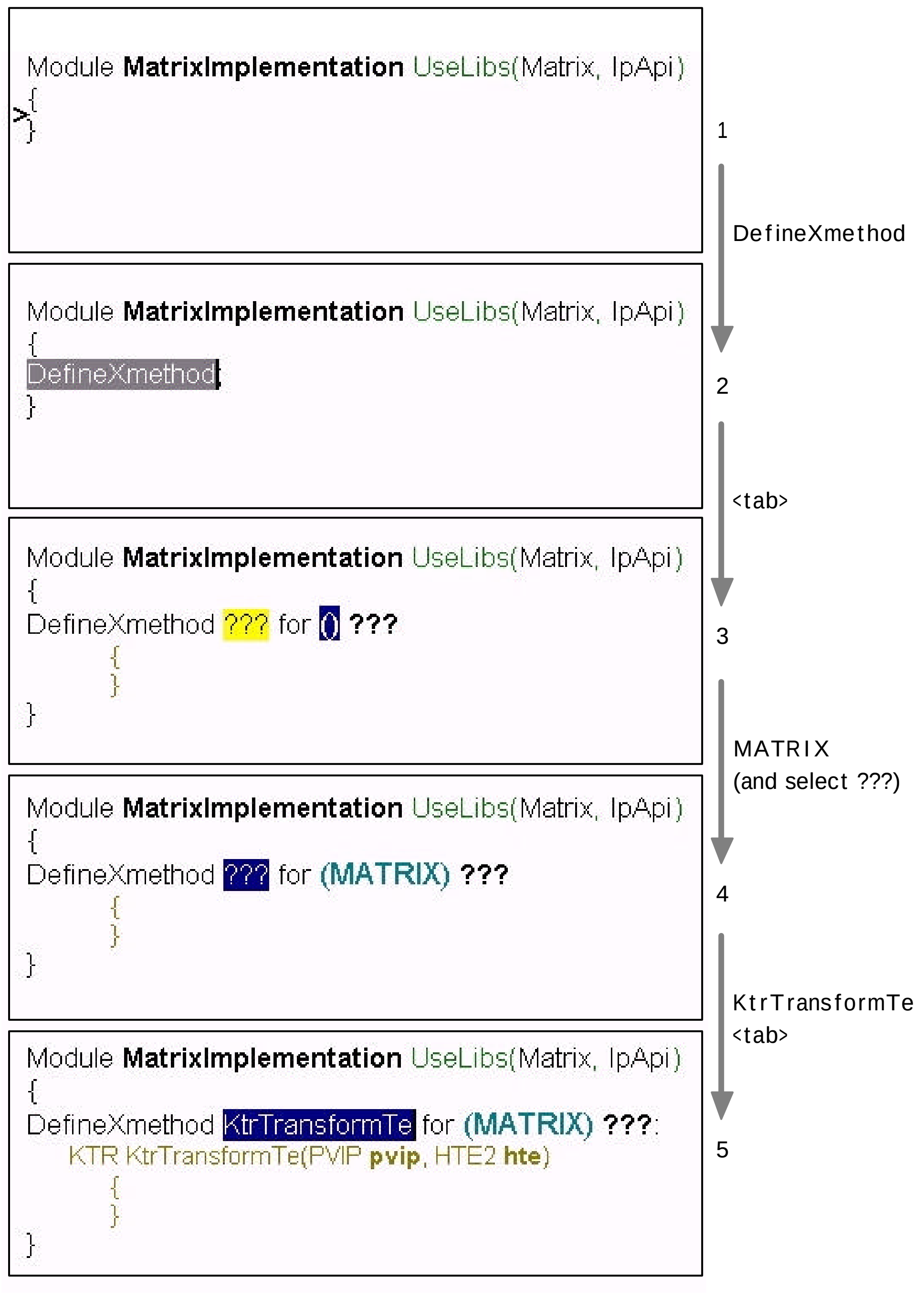


```
int x = 1;
int y;
y = x + 1;

int z = 1;
int y;
y = x + 1;

int z = 1;
int y;
y = z + 1;
```

Figure 92 *Changing the name of a declaration*

It is interesting to take a look at how to change the name of a declaration. For example, we might want to rename x to z. All we have to do in IP is to select the name of the x declaration (box 1 Figure 92) in and change it to z (box 3). Since all references to the x declaration do not store its name but retrieve it from this declaration for display, the third statement displays the correct name immediately (box 3). This simple example illustrates the power of a tree representation compared to text-based representations. Also, if we select z and then push the jump-to-declaration button, the cursor will jump to the z declaration. In fact, we could have changed the name of the y declaration to z as well. In this case, the last statement would contain two z references (i.e. z = z + 1); however, both references will still correctly point to the two different declarations as previously (we can verify this using the jump-to-declaration button). In such cases, where there are more than one declaration with the same name in the same scope, typing in the name will actually not bind the token to any of them. The token will turn yellow instead, indicating a dangling reference. We can still bind the token using a list tool listing all the candidate declarations and selecting the one we would want bind the token to.

A slightly more complex example is shown in Figure 93. This example demonstrates how to type in the transform method for the MATRIX intention, as discussed in Section 6.4.3.3. We start with an empty module and type the name of the intention DefineXmethod. The type-in method of DefineXmethod inserts all the other necessary nodes, effectively constructing most of the tree shown in Figure 88. However, the first operand of DefineXmethod points to the special *to-be-determined* declaration TBD, which is rendered as the first three question marks in the box 3. The second three question marks are the default name of the whole declaration. Next, we enter the reference to MATRIX (box 4) and then to KtrTransformTe (box 5). Please note that the signature of the method is automatically displayed. As we already explained, the rendering method retrieves this information from DCL KtrTransformTe.



```
Module MatrixImplementation UseLibs(Matrix, IpApi)
{
}
```

```
Module MatrixImplementation UseLibs(Matrix, IpApi)
{
DefineXmethod
}
```

```
Module MatrixImplementation UseLibs(Matrix, IpApi)
{
DefineXmethod ??? for () ???
        {
        }
}
```

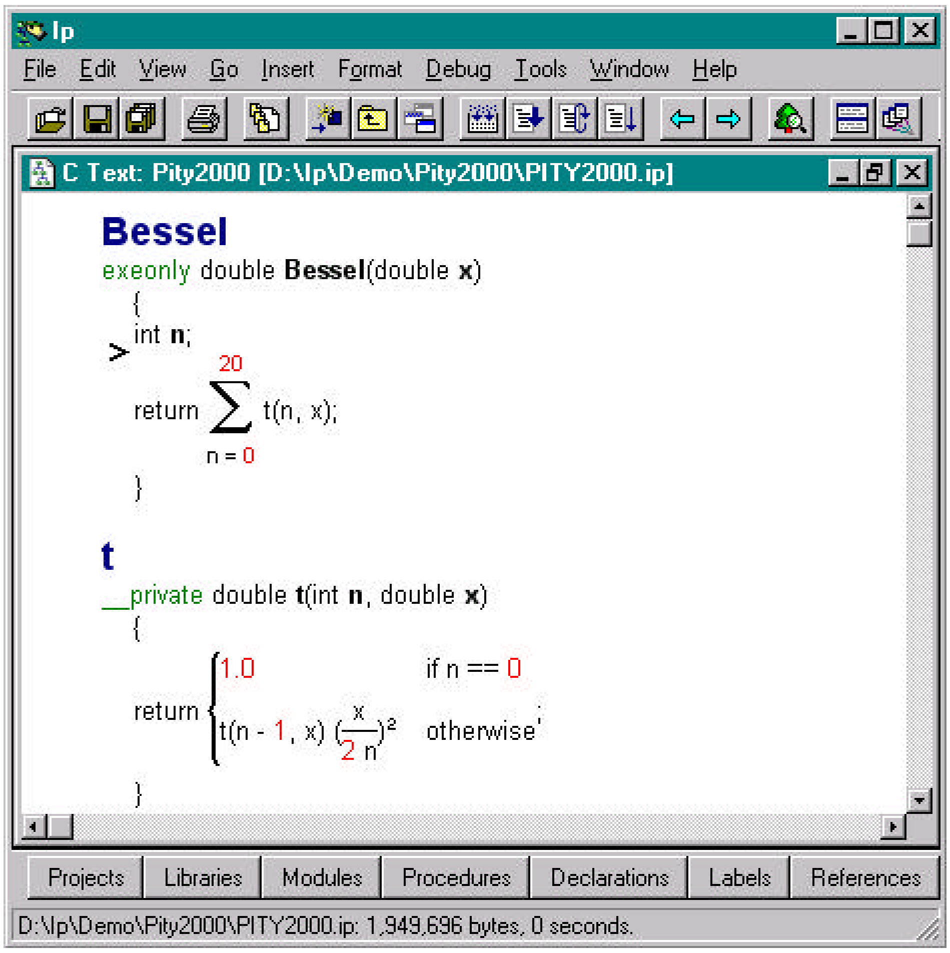
```
Module MatrixImplementation UseLibs(Matrix, IpApi)
{
DefineXmethod ??? for (MATRIX) ???
        {
        }
}
```

```
Module MatrixImplementation UseLibs(Matrix, IpApi)
{
DefineXmethod KtrTransformTe for (MATRIX) ???:
    KTR KtrTransformTe(PVIP pvip, HTE2 hte)
        {
        }
}
```

Figure 93 *Typing in a transform method*

Clicking on a token on the screen does not select its letters but the corresponding tree node or subtree. In fact, there are a number of different selection types. We can select one node (*crown selection*), or a node including all its subnodes (*tree selection*), or we can select a place between two nodes (*place selection*). It is also possible to select the token as such, and change its name (contents selection). If the token is a reference then the reference will be rebound based on the new name. If the token is a name of a declaration, the name will be simply changed (as in Figure 92). There are also other types of selections.

The IP editor is not a syntax-oriented editor, i.e. it is perfectly acceptable for the edited tree to be in an inconsistent state. For example, if we type a name which cannot be bound to any declaration, the token will turn yellow and we know that we need to fix this before attempting to compile the source. Also, as we type, the structure of the tree can be syntactically incorrect. On the other hand, through the type-in methods, as we have seen in Figure 91 and Figure 93, intentions may provide the programmer with type-in templates and suggest what to type in next. Effectively, syntax errors are quite rare and we still have the freedom to type in the tree as we wish without being forced into the syntax straightjacket at every moment.


Ip
File Edit View Go Insert Format Debug Tools Window Help
Projects Libraries Modules Procedures Declarations Labels References

C Text: Pity2000 [D:\Ip\Demo\Pity2000\PITY2000.ip]

Bessel

```
exeonly double Bessel(double x)
  {
> int n;
  return $\sum_{n=0}^{20}$ t(n, x);
  }
```

t

```
__private double t(int n, double x)
  {
  return $\begin{cases} 1.0 & \text{if } n == 0 \\ t(n-1, x)\left(\frac{x}{2n}\right)^2 & \text{otherwise} \end{cases}$;
  }
```

D:\Ip\Demo\Pity2000\PITY2000.ip: 1,949,696 bytes, 0 seconds.

Figure 94 *Example of a domain-specific notation for mathematical formulas*

The major advantage of tree editing is that we are not dealing with passive text. The intentions can even have different type-in behaviors based on the tree context or the currently active view. They can even interact with the developer through menus, dialogs, etc.

Also the tree display provides unique opportunities. First, intentions can be rendered in a truly two-dimensional way (see Figure 94). Second, we can define different views, e.g. we could have a view showing certain aspects and suppressing other aspects of the code, e.g. code implementing synchronization or error handling code. When typing in one of the aspect views, the intentions can still communicate with the other intentions which are not displayed and provide the user with the appropriate feedback (e.g. if some inconsistency between them occurs). We could also have alternative textual or graphical views, etc. The standard system has already a number of predefined views: the *default view* (as in Figure 90), the *pretty C view* (which, e.g., uses pretty mathematical operators), and the *core view*, which displays all the details of the source tree including its implementation details. The last view is provided only for debugging purposes.

6.4.3.6.2 *Extending the System with New Intentions*

New intentions (e.g. a new control structure, or the class construct, or the built-in MATRIX type) are implemented by declaring them and writing their transform, rendering, type-in, debugging, and other methods. In most cases, we need at least the transform, rendering, and the type-in methods.[79] The intentions are usually declared in a separate file, which is the interface file. The methods are then defined in one or more other files and compiled into a DLL[80]. The user of the new intentions only needs to be given the interface file and the corresponding DLL, while the developer can keep the sources of the methods. An application program would then import the interface file in order to be able to use the new intentions. When we load the application program into the IP system, all the imported interfaces are automatically loaded and their corresponding DLLs are dynamically linked to the system (i.e. the IP system is extended). Thanks to the DLLs, the system knows how to display, compile, and debug the intentions referenced in the application program. Related intentions, e.g. intentions implementing a domain-specific notation, are usually packaged in one DLL. The DLL can also contain special commands for working with the new notation (e.g. typing aids, analysis tools, etc.), which can be automatically made available on the IP tool bar after loading the DLL.

Now we will briefly review the steps performed by a typical transform method. A transform method first analyzes the context of the tree element it was called on. It checks if the structure of the subtree is correct, so it can then reduce it. It basically looks for syntax and semantic errors. If it discovers any errors, it attaches error annotations. Even if there are no errors, it is often useful to attach some other information gained in the analysis to the tree nodes, so that other transformations can take advantage of it in later phases. In general, intentions can gather information from remote corners of the program in order to do various optimizations. Then the subtree is reduced by replacing its nodes with instances of other intentions. Accessing and modifying the tree is done using the tree editing API we mentioned in Section 6.4.3.4. In addition to this low-level tree editing API, there are also some higher-level facilities, such as pattern matching functions and quote constructs. The latter allow a compact definition of trees used for matching or replacement. For example, a declaration of type MATRIX can be reduced — based on the annotation of the declaration describing the matrix — to a C array (statically allocated matrix) or to a C struct containing the number of rows and columns and a pointer to the matrix elements (for dynamically allocated matrices) or to other C data structures. The following sample declaration declares a dynamically allocated, rectangular matrix:

```
configuration(dynamic, rectangular) MATRIX m;
```

The code for reducing this declaration is shown below:

```
HTYPE htype;
if (matrix_description.fDynamic && matrix_description.fShape == rectangular)
{
   htype = ‘struct
      {
      int rows;
      int cols;
      $htypeElement* elements;
      };
} else { ... };
```

htype is a handle to a tree element which represents a type (this will be the type, which MATRIX gets reduced to). matrix_description is a struct which was created during the analysis of the annotations of the matrix declaration shown earlier. The statement inside the if-then branch assigns a tree representing a C struct with the number of rows and columns and a pointer of the matrix element type (the element type is a variable defined elsewhere). Instead of constructing this tree using the low-level tree editing API (i.e. calling the create node, set operand, set operator operations, etc.), the C code is quoted using ‘.[81] Once we have the tree representing the C data structure, we can replace the original matrix declaration being reduced by a new one of the newly created C type. Next, the system will reduce the C intentions by calling their transform methods.

6.4.3.7 Advantages of IP

The main advantages of IP stem from the way IP represents programming concepts, which is summarized in Figure 95. The structure of a concept instance (i.e. instance of an intention) is given by the structure of its source tree. Its external representation is defined by the rendering methods and its semantics by the transform methods of its intention. More precisely, the external representation consists of the rendering methods for displaying and the type-in methods for editing.

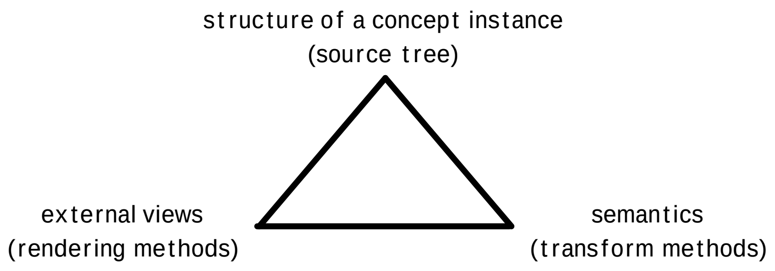


Figure 95 *Separation of structure, external representation, and semantics in IP [Sim98]*

This separation has profound consequences. If you represent a specific concept instance using a source tree, its structure represents the most invariant part of this instance. Its external view and also semantics depend on the context. It may be displayed differently at different times (e.g. editing or debugging time) or it may be displayed differently depending on the tree context (i.e. on the way it is used in a program). The generated code depends on the tree context, on the platform, etc. Given this separation, it is usually enough to modify or extend the transform methods to implement new optimizations, adapt the client code to a new context, new platform, etc., without having to modify the client code at all.

If we need to change the tree representation itself, it is easier to do this than changing a textual representation. Tree editing has many advantages over text editing, such as fewer syntax errors, active interaction with the intentions during editing, and often less typing. (e.g. short intention names, no need to retype signatures of remotely declared procedures and methods). Also, automatic refactoring and reengineering of code is easier if the source is already in the form of a resolved AST.

Rendering allows different views, special notations, graphical representations, etc., and, together with the binary source tree representation, it gives an opportunity for realizing a truly document-based programming (as in Knuth's literate programming [Knu92]). You can embed animations and hyperlinks in the comments, use pretty notations for the code, embed bitmaps (e.g. in IP you can pass a bitmap as a parameter of a procedure and the bitmap is actually shown graphically), etc. At the same time, the transform methods may perform complex computations in order to generate highly-optimized code for these programs.

As noted in [Sim97], with IP we have a major shift of focus from languages to programming abstractions, i.e. intentions. Currently, new language constructs have to look for a host language and this is quite difficult since the most popular languages are hard to extend (this would require updating all the books, existing standards, compilers, etc.). New features can only spread through new and successful languages, such as Java. Unfortunately, not only good features reach large audiences this way. If a bad feature makes into one of the widely used languages, it is difficult, or impossible, to get rid of it. The situation is very different in IP: programming abstractions become true entities with their own "life". They have to survive based on their own merits. They encapsulate the knowledge they need to be displayed, compiled, and debugged in different contexts. They can be easily distributed as extension libraries. The vision of IP is the emergence of an intention market. In such market, there will be intention vendors, who will develop new intentions and will have to make sure that these intentions cooperate as necessary. Given such a market, language abstractions are no longer looking for host languages but rather for customers [Sim97]. With all the critique of programming languages, from the IP viewpoint, existing computer languages are nevertheless important: they are sources of useful language features.

The interoperability of intentions is an important issue in IP. First, it turns out that many features can simply live in one space without any knowledge about each other since they are used in different contexts. Second, there are composition mechanisms, such as annotations, which facilitate the integration of many types of new language features. Furthermore, intentions will have to adhere to certain protocols, just as components do. And finally, given the source tree representation and methods, intentions have a lot of opportunities to find out about each other through introspection.

The research on design patterns and idioms gives a further motivation for the need of change of focus from languages to programming abstraction (see e.g. [GL98]). The pattern work attempts to classify new useful domain-specific and general programming abstractions and mechanisms. There is the conviction that programs are essentially assembled from these fundamental building blocks. On the other hand, as more such patterns are identified and documented, it becomes more difficult for any language to express them adequately. Few of these abstractions make into languages. For example, dynamic polymorphism or inheritance require implementation idioms in C, but they are part of OO languages. However, with the explosion of new abstractions hardly any language could keep up.

It is interesting to realize the analogies between components and intentions: they both are distributed to customers and then composed and they both contain code for different phases (e.g. composition time, debugging, run-time, etc.). However, there are two major differences: First, intentions are components which became the programming language themselves, whereas conventional components still need programming languages to be encoded. Second, the compilation process in IP can completely change the structure of a composition by applying domain-specific optimizations, merging components, introducing new ones, and eliminating others. On the other hand, the structure of a composition of conventional components is usually preserved into runtime.

6.4.4 Approaches Based on Algebraic Specifications

There is at least one theory which provides a theoretical foundation for many of the concepts presented in the previous sections, namely the theory of *algebraic specifications* (see [LEW96]). Algebraic specifications were primarily developed for specifying *abstract data types*. However, it turned out that they are also appropriate for formally specifying domains (see [Sri91]). Furthermore, since they also encourage the transformational implementation style (as we explain later), a number of formal transformation systems (e.g. CIP [BEH+87], SPECWARE [SJ95], and ASF+SDF [DHK96]) are based on the theory of algebraic specification.

First, we will introduce some basic concepts of algebraic specifications. We will then explain how they can be used to specify domains, and, finally, we explain how these specifications are used to generate software systems.

An algebraic specification specifies an abstract data type (ADT), i.e. a class of concrete data types. It consists of two parts: one defining the syntax of a language for talking about the instances of an ADT and the other one defining the semantics of this language. We explain this idea using a simple specification of a list (see Figure 96).

```
specification LIST:
  signature:
    types = {LIST, ELEM}
    operations = {
      . : ELEM, LIST → LIST,
      head : LIST → ELEM,
      tail : LIST → LIST
    }
  axioms:
    variables = {l : LIST, e : ELEM}
    head(e.l) = e
    tail(e.l) = l
    ...
```

Figure 96 *Algebraic Specification of a list*

The sample specification in Figure 96 consists of a *signature* and a set of *axioms*. The signature defines a simple language for defining and manipulating lists. It consists of a set of *types* (also referred to as *sorts*) and a set of *operations*. These types and operations are pure symbols. They take on meaning if we assign one set of values to each type and one function on these sets to each operation. Such assignment is referred to as an *algebra* and represents a concrete data type (i.e. data types are described as sets of values and relationships between them).

Not every assignment of sets and functions to the signature satisfies the specification. The assignments have to satisfy all the *axioms* of the specification. The axioms are the properties of all algebras described by the specification. In other words, an algebraic specification defines a set of algebras, i.e. an ADT.[82] Since axioms are statements in some appropriate logic, we can use inference to derive further properties from these axioms. Any property derivable from the axioms is referred to as a *theorem* (e.g. head(l).tail(l) = l is a theorem of LIST). The axioms can also be viewed as rewrite rules which can be applied to the expressions of the language defined by the signature. These expressions are also called *term expressions*. This view explains why algebraic specifications were used as a formal basis for numerous transformation systems (e.g. [BEH+87, SJ95, DHK96]).

Practical algebraic specifications are usually organized into networks of related smaller algebraic specifications (see [Sri91] for examples). If we consider that each of the smaller specifications defines its own language and the relationships represent refinement, inclusion, parameterization, etc., the similarity of this model to the model of Draco is striking. This is why Srinivas proposed in [Sri91] the use of algebraic specifications to specify domain models (see Section 3.7.7).

The mechanisms for organizing and composing algebraic specifications have been studied in the literature very extensively (see [BG77, Wir83, Gog86, ST88a, ST88b, GB92]). These mechanisms also lie at the heart of automatic generation of programs from specifications.

One of the basic relationships between algebraic specifications is *specification morphism* (see e.g. [LEW96]), which represents a structural relationship between two specifications. A specification morphism between a source and a target specification is defined as a *signature morphism*, i.e. a mapping between the sorts and operations of the source and target signatures, which, if used to translate theorems, ensures that the translated theorems of the source specification are also theorems of the target specification. Intuitively, given the source specification A and the target specification B, the morphism from A to B tells us that the ADT specified by A can be obtained from the ADT specified by B by “forgetting” some of the structure of the latter ADT.

Many specification composition mechanisms can be represented as specification morphisms, e.g. parameterization, inclusion, derivation, views, etc.

An example of work of using algebraic specifications and morphisms to define domain models and generating systems on this basis is the work by Douglas Smith et al. at the Kestrel Institute [SKW85, Smi90, SJ95, SJ96, KI]. An interesting facet of this work is the development of a formal theory of algorithm design called *classification approach to design* [Smi96], which can be deployed by algorithm synthesizers (e.g. [Smi90]). The theory is based on algebraic specifications and mainly two organizing structural relationships: *interpretations* and *refinements*. Interpretations define the structural correspondence between a reusable problem specification and the specification of a problem at hand. Constructing an interpretation between the reusable problem specification and the given problem specification is also referred to as *classification* since we classify the given problem as an instance of the known general problem. An interpretation from the specification A to B has the following structure: A→ A-B ← B, where the arrows represent specification morphisms and A-B is a specification called the mediator. In contrast to a simple morphism, an interpretation can define a mapping not only between types and operations, but also between types and type expressions and vice versa. Effectively, interpretation can be seen as views (or adapters).

A refinement is a morphism between two specifications, where the refined specification contains more implementation details. Effectively, a refinement defines the relationship between a reusable problem specification and its reusable solution. The reusable problem solution defines the “interface vocabulary” for the solution specification — it really is a vertical parameter of the solution specification. Thus, refinements are vertical relationships, whereas interpretations are horizontal relationships.[83]

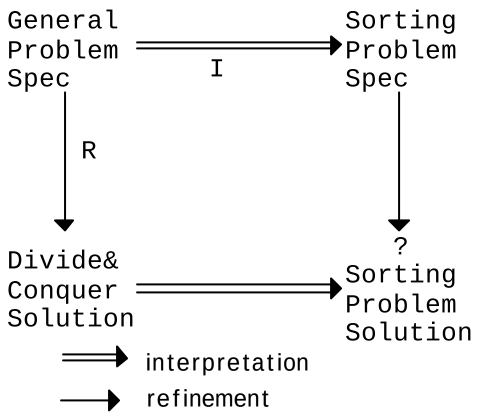


Figure 97 *Relationships between the specification of the sorting example*

As an example, we consider the problem of sorting a list of numbers. Assume that we specified this problem as the algebraic specification `SortingProblemSpec` (see Figure 97). The problem can be solved by applying the *divide and conquer* problem solution strategy. Suppose that the general problem specification `GeneralProblemSpec` and its corresponding divide and conquer solution `Divide&ConquerSolution` are stored in a library (the general problem specification defines some interface symbols such as the input and the output set for the `Divide&ConquerSolution`). Both specifications are also referred to as *design theories*. We can reuse this problem-solution pair by identifying the correspondence between the symbols in `SortingProblemSpec` and `GeneralProblemSpec`. This step defines the interpretation relationship `I`. Finding this correspondence is often not a trivial task, which, in general, cannot be automated. Different interpretations will finally lead to the derivation of different sorting algorithms, e.g. mergesort, insertion sort, or quicksort. Next, the specification `SortingProblemSolution`, which structurally satisfies both `SortingProblemSpec` and `Divide&ConquerSolution`, is determined, whereby various mathematical procedures are used. This process of interpretation and refinement is carried on untill an executable specification $\texttt{Spec}_n$ is constructed (see Figure 98). The details of the sorting example can be found in [Smi96].

$$\begin{tikzcd}
DT_1 \arrow[r, Rightarrow, "I_1"'] \arrow[d] & Spec_1 \arrow[d] \\
DT_2 \arrow[r, Rightarrow, "I_2"'] \arrow[d] & Spec_2 \arrow[d] \\
DT_3 \arrow[r, Rightarrow, "I_3"'] \arrow[d] & Spec_3 \arrow[d] \\
\vdots \arrow[d] & \vdots \arrow[d] \\
DT_n \arrow[r, Rightarrow, "I_n"'] & Spec_n
\end{tikzcd}$$

DT = design theory

Figure 98 *Ladder construction [Smi96]*

Depending on the concrete interpretations and refinements used, we can derive different sorting algorithms. The various sorting algorithms span a design space as shown in Figure 99. Also the reusable problem solving strategies used in algorithm design are organized into a refinement hierarchy (see Figure 100). Such design spaces are usually augmented with performance characteristics and design rationale to guide the selection of the appropriate design theory based on the context.

The domain of sorting is well understood today and can be easily formally described using the concepts presented above. Such formal description provides a basis for the automatic synthesis of algorithms from specifications. A prominent example of a system capable of such synthesis is *KIDS* (*Kestrel Interactive Development System*) [Smi90], which has been part of Kestrel Institute's research efforts for over 10 years. Of course, synthesizing sorting algorithms, while interesting from the theoretical viewpoint, provides little leverage for practical software development. However, the work on the transportation scheduling domain at Kestrel demonstrated that there are also practical domains which are understood well enough and stable enough to be formalized. According to [SPW95, SG96], the scheduler generated from a formal domain model using KIDS is over 20 times faster than the standard, hand-coded system deployed by the customer.

Based on the extensive experience with KIDS, a new system has been built, namely SPECWARE [SJ95, SM96], which is more systematically based on the concepts presented above than KIDS. In particular, the system is explicitly based on category theory [Gol79] which provides a theoretical foundation for working with specification morphism diagrams (such as in Figure 98). The system supports the user in the construction of interpretation and refinement relationships through various mathematical procedures (e.g., constraints propagation, unskolemization, computing pushouts, etc.).

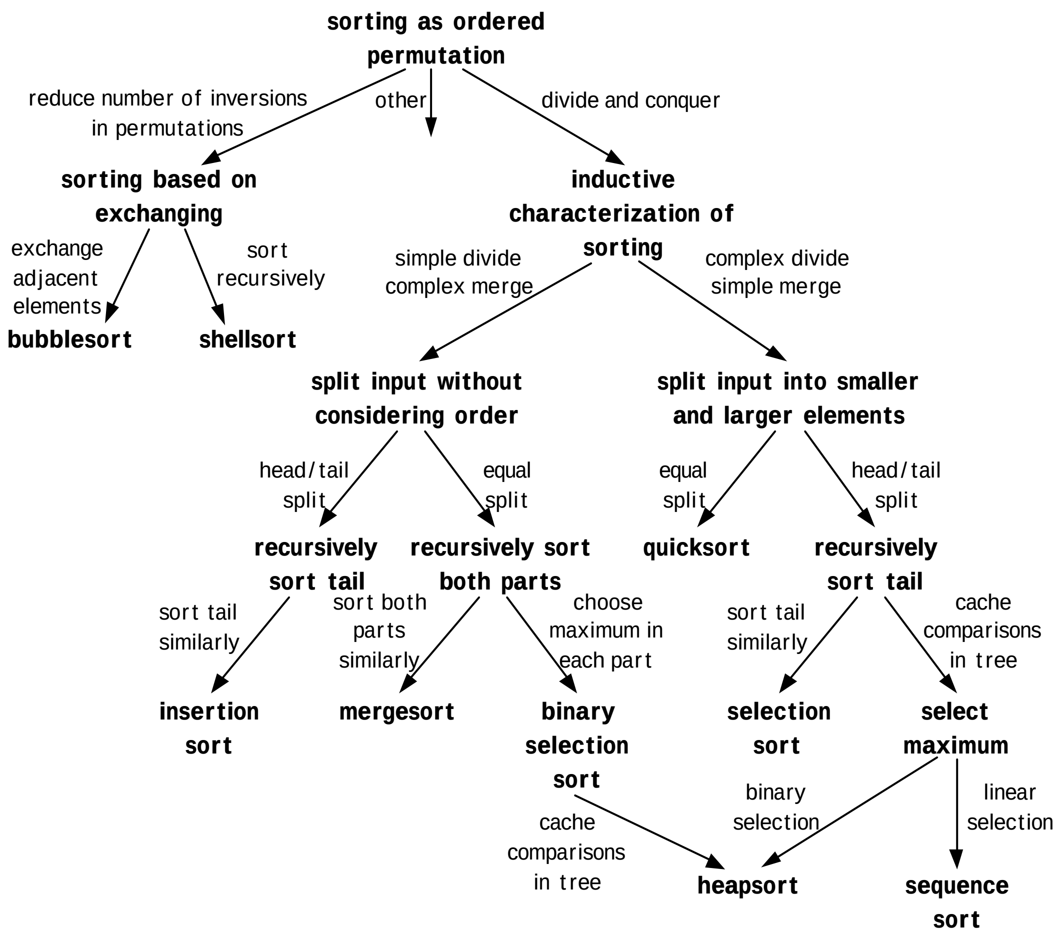

sorting as ordered permutation
reduce number of inversions in permutations
other
divide and conquer
sorting based on exchanging
exchange adjacent elements
sort recursively
bubblesort
shellsort
inductive characterization of sorting
simple divide complex merge
complex divide simple merge
split input without considering order
split input into smaller and larger elements
head/tail split
equal split
equal split
head/tail split
recursively sort tail
recursively sort both parts
quicksort
recursively sort tail
sort tail similarly
sort both parts similarly
choose maximum in each part
sort tail similarly
cache comparisons in tree
insertion sort
mergesort
binary selection sort
selection sort
select maximum
cache comparisons in tree
binary selection
linear selection
heapsort
sequence sort

Figure 99 *Design space of implementations of sorting (adapted from [Bro83])*

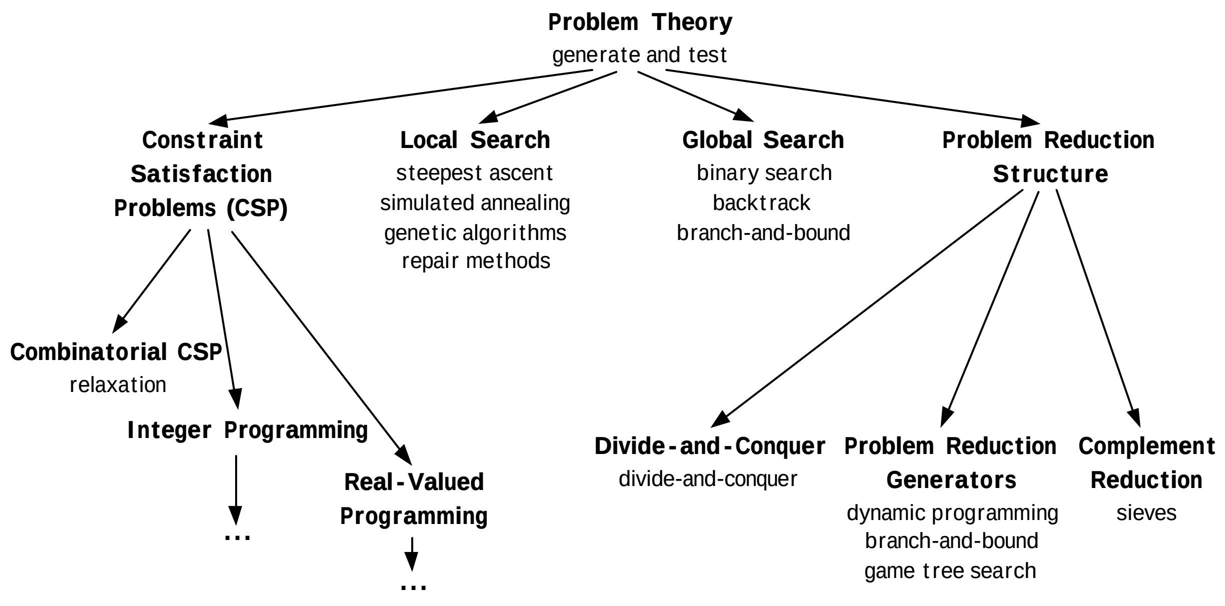


Figure 100 *Refinement hierarchy of algorithm design theories (adapted from [Smi96])*

The success in the domain of transportation scheduling demonstrates that there is clearly a potential for the practical application of these formal concepts, especially in mature, stable, and well defined domains. Unfortunately, most application domains cannot be formalized at all for reasons discussed in Chapter 2 and even if, given their complexity and instability, the formalization results would be "dead on arrival". Nonetheless, the insights gained from these theories allow us to better understand at least some theoretical aspects of software design.

6.5 References

[ADK+98] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi. Transformation in Intentional Programming. In [DP98], pp. 114-123, also [IPH]

[AKR+97] R. Akers, E. Kant, C Randall, S. Steinberg, and R. Young. SciNapse: A Problem-Solving Environment for Partial Differential Equations. In *IEEE Computational Science and Engineering*, vol. 4, no. 3, July-September 1997, pp. 32-42, also [SC]

[Arm97] T. Armstrong. *Designing and Using ActiveX Controls*. M&T Books, 1997

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986

[Bal85] R. Balzer. A 15 Year Perspective on Automatic Programming. In *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, November 1985, pp. 1257-1268

[Bat96] D. Batory. Software System Generators, Architectures, and Reuse. Tutorial Notes, Fourth International Conference on Software Reuse, April 23-26, 1996, Orlando, Florida, USA, also see [SSGRG]

[Bax90] I. Baxter. Transformational Maintenance by Reuse of Design Histories. Ph.D. Thesis, Technical Report TR 90-36, Department of Information and Computer Science, University of California at Irvine, November 1990

[Bax92] I. Baxter. Design Maintenance System. In *Communications ACM*, vol. 35, no. 4, April 1992, pp. 73-89

[Bax96] I. Baxter. Transformation Systems: Theory, Implementation, and Survey. Tutorial Notes, Fourth International Conference on Software Reuse, April 23-26, 1996, Orlando, Florida, USA

[BBG+88] D. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tuskuda, B.C. Twichell, and T.E. Wise. GENESIS: An Extensible Database Management System. In *IEEE Transactions on Software Engineering*, vol. 14, no. 11, November 1988, 1711-1730

[BCD+89] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, 1989, pp. 14-24, also http://www.inria.fr/croap/centaur/centaur.html

[BCGS95] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer. Creating Reference Architectures: An Example from Avionics. In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability*, April 28-30, 1995, Seattle, Washington, pp. 27-37, also as Technical Report ADAGE-UT-93-06C, also http://www.owego.com/dssa/ut-docs/ut-docs.html

[BCRW98] D. Batory, G. Chen, E. Robertson, and T. Wang. Design Wizards and Visual Programming Environments for Generators. In [DP98], pp. 255-267, also [SSGRG]

[BEH+87] F. L. Bauer, H Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich project CIP: Volume II, the Transformation System CIP-S*. LNCS 292, Springer Verlag, 1987

[BF96] S. Blazy and P. Facon. Interprocedural Analysis for Program Comprehension by Specialization. In *Proceedings of the 4th Workshop on Program Comprehension (March 29-31, Berlin, Germany, 1996), WPC'96*, A. Cimitile and H.A. Müller (Eds.), IEEE Computer Society Press, Los Alamitos, California, 1996, pp. 133-141

[BG77] B.R. Burstall and J.A. Goguen. Putting theories togethor to make specifications. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (Cambridge, MA, 22-25, 1977), IJCAI*, 1977, pp. 1045-1058

[BG96] D. Batory and B. J. Geraci. Validating Component Compositions in Software System Generators. In [Sit96], pp. 72-81, also extended version as Technical Report TR-95-03, Department of Computer Sciences, University of Texas at Austin, June 1995, also [SSGRG]

[BG97] D. Batory and B. J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. In *IEEE Transactions on Software Engineering*, special issue on Software Reuse, February 1997, pp. 67-82, also [SSGRG]

[BGT94] D. Batory, B. Geraci, and J. Thomas. Introductory P2 System Manual. Technical Report TR-94-26, Department of Computer Sciences, University of Texas at Austin, November 1994, see [SSGRT]

[BHW97] J. Boyle, T. Harmer, and V. Winter. The TAMPR Program Transformation System: Simplyfying the Development of Numerical Software. In *Modern Software Tools for Scientific Computing Age*, Erland et al. (Eds.), Birkhäuser Boston, 1997, pp. 353-372

[Big94] T. Biggerstaff. The Library Scaling Problem and the Limits of Concrete Component Reuse. In *Proceedings of the 3rd International Conference on Software Reuse*, W.B. Frakes (Ed.), IEEE Computer Society Press, 1994, pp. 102-109

[Big97] T. J. Biggerstaff. A Perspective of Generative Reuse. Technical Report MSR-TR-97-26, Microsoft Corporation, Redmond, Washington, 1997

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In [DP98], pp. 143-153, also [SSGRG]

[BM84] J.M. Boyle and M.N. Muralidharan. Program Reusability through Program Transformation. In *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, September 1984, pp. 574-588

[BM97] I. D. Baxter and M. Mehlich. Reverse Engineering is Reverse Forward Engineering. In *Proceedings of the Working Conference on Reverse Engineering (October 6-8, Amsterdam, The Netherlands)*, 1997, also [SD]

[BO92] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. In *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, October 1992, pp. 355-398, also [SSGRG]

[Boo87] G. Booch. *Software Components with Ada*. Benjamin/Cummings, 1987

[Box98] D. Box. *Essential COM*. Addison Wesley Longman, Inc., 1998

[BP89] T. Biggerstaff and A. Perlis. *Software Reusability. Volume I: Concepts and Models*. ACM Press, Frontier Series, Addison-Wesley, Reading, 1989

[BP97] I. D. Baxter and C. W. Pidgeon. Software Change Through Design Maintenance. In *Proceedings of International Conference on Software Maintenance (ICSM'97)*, IEEE Press, 1997, also [SD]

[Bro83] M. Broy. Program constructuin by transformation: A family tree of sorting programs. In *Computer Program Synthesis Methodologies*, A.W. Biermann and G. Guiho (Eds.), D. Reidel Publishing Company, Dordrecht, Holland, 1983, pp. 1-49

[BSS92] D. Batory, V. Singhal, and M. Sirkin. Implementing a Domain Model for Data Structures. In *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 3, September 1992, pp. 375-402, also [SSGRG]

[BT] Homepage of Bayfront Technologies, Inc., at http://www.bayfronttechnologies.com

[BW90] A. Berlin and D. Weise. Compiling Scientific Code Using Partial Evaluation. In *IEEE Computer*, December 1990, pp. 25-37

[CS92] J. Cordy and M. Shukla. Practical Metaprogramming. In *Proceedings of CASCON '92*, IBM Centre for Advances Studies Conference, Toronto, November 1992, also http://www.qucis.queensu.ca/home/cordy/ TXL-Info/index.html

[CW] ControlWORKS, a machine and process control software package for semiconductor process equipment by Texas Instruments, Inc.; see product data sheets at http://www.ti.com/control/

[DB95] D. Das and D. Batory. Prairie: A Rule Specification Framework for Query Optimizers. In *Proceedings 11th International Conference on Data Engineering*, Taipei, March 1995, also [SSGRG]

[DHK96] A. van Deursen, J. Heering, P. Klint (Eds.). *Language Prototyping: An Algebraic Specification Approach*. World Scientific Publishing, 1996, also see http://www.cwi.nl/~gipe/

[DP98] P. Devanbu and J. Poulin, (Eds.). *Proceedings of the Fifth International Conference on Software Reuse (Victoria, Canada, June 1998)*. IEEE Computer Society Press, 1998

[Efr94] S. Efremides. On Program Transformations. PhD Thesis, Cornell University, May 1994, available from http://cs-tr.cs.cornell.edu/

[Eis97] U. W. Eisenecker. Generative Programmierung und Komponenten. In *OBJEKTspektrum*, Nr. 3, Mai/Juni 1997, S. 80-84

[Eis98] U. W. Eisenecker. Konfigurationsvielfalt. In *OBJEKTspektrum*, No. 1, Januar/Februar 1998, S. 88-90

[Fea86] M. Feather. A Survey and Classification of some Program Transformation Approaches and Techniques. IFIP WG21 Working Conference on Program Specification and Transformation, Bad Toelz, Germany April 1986

[GB80] J. Goguen and R. Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical Report CSL-118, SRI Computer Science Lab, October 1980

[GB92] J. Goguen and R. Burstall. Institutions: Abstract Model Theory for Specification and Programming. In *Journal of the ACM*, vol. 39, no. 1, 1992, pp. 95-146

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1995

[GL98] J. Gil and D. H. Lorenz. Design Patterns and Language Design. In *IEEE Computer*, vol. 31, no. 3, March 1998, pp. 118-120

[Gog86] J. A. Goguen. Reusing and Interconnecting Software Components. In *IEEE Computer*, February 1986, pp. 16-28, also in [PA91], pp. 125-147

[Gog96] J. A. Goguen. Parameterized Programming and Software Architecture. In [Sit96], pp. 2-10

[Gol79] R. Goldblatt. *TOPOI: the Categorial Analysis of Logic*. Studies and Logic and the Foundations of Mathematics, vol. 98, J. Barwise et al. (Eds.), North-Holland Publishing Company, 1979

[HK93] B. Hoffmann and B. Krieg-Brückner (Eds.). *PROgram development by SPECification and TRAnsformation: Methodology - Language Family - System.* Springer Verlag, LNCS 680, 1993

[HPOA89] N. Hutchinson, L. Peterson, S. O'Malley, and M. Abbott. RPC in the x-Kernel: Evaluating New Design Technique. In *Proceedings of the Symposium on Operating System Principles*, December 1989, pp. 91-101

[HP94] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. In *ACM Transactions on Computer Systems*, vol. 12, no. 1, 1994, pp. 58-89, also as UCLA Technical Report CSD-930019, also http://fmg-www.cs.ucla.edu/fmg/summary.html

[IPD] Intentional Programming Development Team, Personal Communication, see also [IPH]

[IPH] Homepage of the Intentional Programming Project, Microsoft Research, Redmond, Washington, http://www.research.microsoft.com/research/ip/

[JGS93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993

[Kan93] E. Kant. Synthesis of Mathematical Modeling Software. In *IEEE Software*, vol.10, no. 3, May 1993, pp. 30-41, also [SC]

[KI] Publications of the Kestrel Institute, Palo Alto, California, http://www.kestrel.edu/HTML/publications.html

[Kir97] M. Kirtland. The COM+ Programming Model Makes it Easy to Write Components in Any Language. In *Microsoft Systems Journal*, December 1997, pp. 19-28, also http://www.microsoft.com/msj/1297/ complus2/complus2.htm

[KM90] G. B. Kotik and L.Z. Markosian. Program Transformation: the Key to Automating Software Maintenance and Reengineering. In *IEEE Transactions on Software Engineering*, vol. 16, no. 9, 1990, 1024-1043

[Knu86] D. E. Knuth. *The $T_EX$book*. Addison-Wesley, 1986

[Knu92] D. E. Knuth. *Literate Programming (Center for the Study of Language and Information - Lecture Notes, No 27)*. Stanford University Center for the Study, May 1992

[KO96] B. B. Kristensen and K. Østerbye. Roles: Conceptual Abstraction Theory & Practical Language Issues. In *Theory and Practice of Object Systems* (TAPOS), vol. 2, no. 3, 1996, pp. 143-160, also http://www.mip.ou.dk/~bbk/research/recent_publications.html

[Lea88] D. Lea. libg++, the GUN C++ library. In *Proceedings of the USENIX C++ Conference*, 1988

[LEW96] J. Loeckx, H.D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley & Teubner, 1996

[LS86] S. Letovsky and E. Soloway. Delocalized Plans and Program Comprehension. In IEEE Software, vol. 3, no. 3, 1986, pp. 41-49

[MB97] M. Mehlich and I.D. Baxter. Mechanical Tool Support for High Integrity Software Development. In *Proceedings of Conference on High Integrity Systems '97 (October 15-14, Albuquerque, New Mexico)*, IEEE Press, 1997, also [SD]

[McC88] R. McCartney. Synthesizing Algorithms with Performance Constraints. Ph.D. Thesis, Technical Report CS-87-28, Department of Computer Science, Brown University, 1988

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide. Addison-Wesley*, Reading, Massachusetts, 1996

[MS97] T. Margaria and B. Steffen. Coarse-grain Component-Based Software Development: The METAFrame Approach. In *Proceedings of Smalltalk and Java in Industry and Academia (STJA'97)*, Erfurt, Germany, 1997, pp. 29-34

[Mye95] N. C. Myers. Traits: a new and useful template technique. In *C++ Report*, June 1995, see http://www.cantrip.org/traits.html

[Nei80] J. Neighbors. Software construction using components. Ph.D. dissertation, (Technical Repeport TR-160), Department Information and Computer Science, University of California, Irvine, 1980, also [BT]

[Nei84] J. Neighbors. The Draco Approach to Construction Software from Reusable Components. In *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, September 1984, pp. 564-573

[Nei89] J. Neighbors. Draco: A Method for Engineering Reusable Software Systems. In [BP89], pp. 295-319

[Obd92] W.F. Obdyke. Refactoring Object-Oriented Frameworks. PhD Thesis, University of Illinois at Urbana-Champaign, 1992, see ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z

[OMG97] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Framingham, Massachusetts, 1997, also see http://www.omg.org

[OP92] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. In *ACM Transactions on Computer Systems*, vol. 10, no. 2, May 1992, pp. 110-143, also http://www.cs.arizona.edu/xkernel/bibliography.html

[PA91] R. Prieto-Diaz and G. Arrango (Eds.). *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, California, 1991

[Pai94] R. Paige. Viewing a program transformation system at work. In *Proceedings of PLILP'94*, LNCS 844, M. Hermenegildo, and J. Penjam (Eds.), Springer-Verlag, Sep. 1994, pp. 5-24, also http://cs.nyu.edu/cs/faculty/paige/research.html

[Par90] H. A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*. Texts and Monographs in Computer Science, Springer-Verlag, 1990

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. In *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, 1982, pp. 401-454

[RBJ98] D. Roberts, J. Brand, and R Johnson. A Refactoring Tool for Smalltalk. Submitted to *Theory and Practice of Object Systems*, 1998, see http://st-www.cs.uiuc.edu/users/droberts/homePage.html and http://st-www.cs.uiuc.edu/~brant/Refactory/RefactoringBrowser.html

[Ree96] T. Reenskaug with P. Wold and O.A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning, 1996

[RF92] C. Rich and Y. A. Feldman. Seven Layers of Knowledge Representation and Reasoning in Support of Software Development. In *IEEE Transactions on Software Engineering*, vol. 18, no. 6. June 1992, pp. 451-469

[Rie97] D. Riehle. Composite Design Patterns. In *Proceedings of the 1997 ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA'97), 1997, pp. 218-227

[RK97] C. Randall and E. Kant. Numerical Options Models Without Programming. *In Proceedings of the IEEE/IAFE Conference on Computational Intelligence for Financial Engineering*, New York, New York, March 23-25, 1997, pp. 15-21, also [SC]

[Ros95] L. Rosenstein. MacApp: First Commercially Successful Framework. In *Object-Oriented Application Frameworks*, T. Lewis et al., Manning Publications Co., 1995, pp. 111-136

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1989, also see http://www.grammatech.com

[RW90] C. Rich and R. C. Waters. *The Programmer's Apprentice*. ACM Press and Addison-Wesley, 1990

[SB93] V. Singhal and D. Batory. P++: A Language for Large-Scale Reusable Software Components. *In Proceedings of the 6th Annual Workshop on Software Reuse*, Owego, New York, November 1993, also [SSGRG]

[SB97] Y. Smaragdakis and D. Batory. DiSTiL: A Transformation Library for Data Structures. In *Proceedings of USENIX Conference on Domain-Specific Languages*, October 1997, also [SSGRG]

[SB98a] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of the 12th European Conference Object-Oriented Programming (ECOOP'98)*, E. Jul, (Ed.), LNCS 1445, Springer-Verlag, 1998, pp. 550-570, see [SSGR]

[SB98b] Y. Smaragdakis and D. Batory. Implementing Reusable Object-Oriented Components. In [DP98], pp. 36-45, also [SSGRG]

[SBS93] M. Sirkin, D. Batory, and V. Singhal. Software Components in a Data Structure Precompiler. In *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, MD, May 1993, pp. 437-446, also [SSGRG]

[SC] Homepage of SciComp, Inc., Austin, Texas, http://www.scicomp.com/

[SD] Homepage of Semantic Designs, Inc., Austin, Texas, http://www.semdesigns.com/

[Ses97] R. Sessions. *Com and Dcom: Microsoft's Vision for Distributed Objects*. John Wiley & Sons 1997

[SG96] D. R. Smith and C. Green Toward Practical Applications of Software Synthesis. In *Proceedings of FMSP'96, The First Workshop on Formal Methods in Software Practice*, San Diego, California, January 1996, pp. 31-39, also [KI]

[Sha98] G. Shaw. Intentional Programming. An internal online tutorial, Microsoft Corporation, 1998

[Sie96] J. Siegel (Ed.). *CORBA: Fundamentals and Programming*. John Wiley & Sons, Inc., 1996

[Sin96] V. P. Singhal. A Programming Language for Writing Domain-Specific Software System Generators. Ph.D. Thesis, Department of Computer Sciences, University of Texas at Austin, September 1996, also [SSGRG]

[Sim95] C. Simonyi. The Death of Computer Languages, The Birth of Intentional Programming. Technical Report MSR-TR-95-52, Microsoft Research, 1995, ftp://ftp.research.micro-soft.com/pub/tech-reports/Summer95/TR-95-52.doc

[Sim96] C. Simonyi. Intentional Programming — Innovation in the Legacy Age. Position paper presented at IFIP WG 2.1 meeting, June 4, 1996, also [IPH]

[Sim97] C. Simonyi. Intentional Programming. Talk given at the Usenix Conference on Domain Specific Languages, Santa Barbara, California, October 16, 1997, transparencies available from [IPH]

[Sim98] C. Simonyi, private communication, January 1998

[Sit96] M. Sitaraman (Ed.). *Proceedings of the Fourth International Conference on Software Reuse, April 23-26, Orlando, Florida*. IEEE Computer Society Press, Los Alamitos, California, 1996

[SJ95] Y.V. Srinivas and R. Jüllig. Specware^{TM}: Formal Support for Composing Software. In *Proceedings of the Conference on Mathematics of Program Construction*, Kloster Irsee, Germany, July 1995, also [KI]

[SKW85] D.R. Smith, G.B. Kotik, and S.J. Westfold. Research on Knowledge-Based Software Environments at Kestrel Institute. In *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, November 1985, pp. 1278-1295, also see [KI]

[SM91] C. Simonyi and M. Heller. The Hungarian Revolution. In *Byte*, August, 1991, pp.131-138, see also http://www.freenet.tlh.fl.us/~joeo/hungarian.html

[SM96] Y.V. Srinivas and J. L. McDonald. The Architecture of Specware, a Formal Software Development System. Technical Report KES.U.96.7, Kestrel Institute, August 1996, also [KI]

[SMBR95] B. Steffen, T. Margaria, V. Braun, and M. Reitenspieß. An Environment for the Creation of Intelligent Network Services. Invited contribution to the book *The Advanced Intelligent Network: A Comprehensive Report*, International Engineering Consortium Chicago, December 1995, pp. 287-300, also reprinted in *Annual Review of Communications*, IEC, 1996

[SMCB96] B. Steffen, T. Margaria, A. Claßen, and V. Braun. The METAFrame'95 Environment. In *Proceedings CAV'96 (July-August 1996, Brunswick, New Jersey, USA)*, LNCS1102, Springer Verlag, 1996, pp. 450-453

[Smi90] D. R. Smith. KIDS: A Semi-Automatic Program Development System. *In IEEE Transactions on Software Engineering*, Vol. 16, No. 9, September 1990, pp. 1024-1043, also [KI]

[Smi96] D.R. Smith. Toward a Classification Approach to Design. In *Proceedings of Algebraic Methodology & Software Technology, AMAST'96*, Munich, Germany, July 1996, M. Wirsing and M. Nivat (Eds.), LCNS 1101, Springer, 1996, pp. 62-84, also [KI]

[SPW95] D. Smith, E. Parra, and S. Westfold. Synthesis of High-Performance Transportation Schedulers. Technical Report, KES.U.1, Kestrel Institute, February 1995, also [KI]

[Sri91] Y. Srinivas. Algebraic specification for domains. In [PA91], pp. 90-124

[SSGRG] Homepage of the Software Systems Generator Research Group at the University of Texas at Austin http://www.cs.utexas.edu/users/schwartz

[ST88a] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. In *Information and Computation*, no. 76, 1988, pp. 165-210

[ST88b] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. In *Acta Informatica* ,vol. 25, no. 3, 1988, 233-281

[Sun97] Sun Microsystems, Inc. JavaBeans 1.01. Specification document, Mountain View, California, 1997, see http://java.sun.com/beans/docs/spec.html

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998

[VH97] M. VanHilst. Role-Oriented Programming for Software Evolution. Ph.D. Dissertation, University of Washington, Computer Science and Engineering, 1997

[VHN96] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA'96), 1996, pp. 359-369

[WBM98] R.W. Walker, E.L.A. Baniassad, and G. Murphy. Assessing Aspect-Oriented Programming and Design. In Proceedings of Workshop on Aspect-Oriented Programming at the International Conference of Software Engineering, 1998, http://www.parc.xerox.com/aop/

[Wil83] D. Wile. Program Developments: Formal Explanations of Implementations. In *Communications of the ACM*, vol. 26, no. 11, November 1983

[Wil91] D. Wile. Popart Manual. Technical Report, Information Science Institute, University of Southern Calofornia, 1991, see http://www.isi.edu/software-sciences/wile/Popart/popart.html

[Wir83] M. Wirsing. Structured algebraic specifications: A kernel language. In *Theoretical Computer Science* 42, 1986, pp. 123-249. A slight revision of his Habilitationsschrift, Technische Universität München, 1983

[Wol91] S. Wolfram. *Mathematica, A System for Doing Mathematics by Computer*. Addison-Wesley, 1991. Mathematica is sold by Wolfram Research, ftp://www.wri.com/

Chapter 7 Aspect-Oriented Decomposition and Composition

7.1 Introduction

Throughout the previous chapters, we stressed the importance of high-level, intentional specifications. But what constitutes a high-level, intentional specification? The hallmark of such specifications is that important issues concerning the system being described are dealt with in a localized way. For example, in a banking application, there will be well-localized, modular units (e.g. objects) representing accounts, customers, currency, etc.

There are numerous benefits from having an important concern of a software system being expressed well localized in a single code section. First of all, we can more easily understand how this concern is addressed in the code since we do not have to look for it in different places and discern it from other concerns. Moreover, we can more easily analyze such code, modify it, extend it, debug it, reuse it, etc.

Principle of separation of concerns

The need of dealing with one important issue at a time was named by Dijkstra as the *principle of separation of concerns* [Dij76]. Unfortunately, while the principle expresses an important quality of code and a development process, it does not tell us how to achieve it.

Generalized procedures

Separation of concerns is a fundamental engineering principle applied in analysis, design, and implementation. Most analysis and design notations, and programming languages provide constructs for organizing system descriptions as hierarchical compositions of smaller, modular units. However, as Kiczales et al. note in [KLM+97], current methods and notations concentrate on finding and composing *functional* units, which are usually expressed as objects, modules, procedures, etc. They also refer to such units as *generalized procedures* since they are called from the client code. But Kiczales et al. also state that there are other important issues, which are not well localized in functional designs. For example, system properties involving more than one functional component, such as synchronization, component interaction, persistency, etc., cannot be expressed using current (e.g. OO) notations and languages in a cleanly localized way. Instead, they are expressed by small code fragments scattered throughout several functional components.

Aspect-Oriented Programming

The latter observation lies at the heart of the new research area of *Aspect-Oriented Programming* (*AOP*) started by researchers from XEROX PARC[84] [KLM+97, AOP97, AOP98, AOP]. The goal of AOP is to provide methods and techniques for decomposing problems into a number of *functional components* as well as a number of *aspects* which crosscut functional components and then composing these components and aspects to obtain system implementations.[85]

Once we subscribe to the idea of separating aspects, we need concrete methods and techniques to achieve this separation. In particular, there are three main questions we have to address:

- *What are the important issues that need to be separated?* We already mentioned some examples such as synchronization or component interaction. What we are looking for are reusable sets of concerns to be used in the decomposition of problems. Some of the concerns will be more application-specific, e.g. defining financial products or configuring network services. Other concerns will be more general, e.g. synchronization or workflow. Some of the concerns (both application-specific and general) will be aspects. As we discussed in Section 5.8.2.1, by selecting an appropriate set of concerns for a problem, we achieve a "healthy" balance between the localization of relevant issues, complexity, and redundancy. Of course, different domains will require different sets of concerns. Throughout this chapter, we give you more examples of important concerns.

- *What composition mechanisms other than calling generalized procedures can we use?* If not all aspects can be encapsulated and composed using generalized procedures, we need to find other important composition mechanisms. The hallmark of such mechanisms is achieving loose, declarative coupling between partial descriptions. We would also like the mechanisms to support a broad spectrum of binding times and modes and also noninvasive adaptability (i.e. adaptability by composition and transformation rather than manual change).

- *How do we capture the aspects themselves?* We express aspects using some appropriate linguistic means: In the simplest cases, we can use conventional class libraries. In other cases, we might want to use specialized languages or language extensions. Each approach has its advantages and disadvantages and we will discuss them in Section 7.6.1.

AOP, while still in its definition phase, already begins to provide benefits by focusing and stimulating existing and new work addressing these three questions. We will discuss some answers to these questions in this chapter. In most cases, we will concentrate on OO methods and languages, but the AOP idea is not limited to OO.

The message of this chapter is that current methods and languages (including OO) do not allow us to cleanly encapsulate certain important design decisions. In this chapter, you will find examples of such shortcomings and also approaches to address them.

7.2 Aspect-Oriented Decomposition Approaches

There is a number of existing approaches that concentrate on encapsulating various system properties including aspects that crosscut modular units of functionality. The following three sections describe three of such approaches. These approaches extend the OO programming model to allow us the encapsulation of aspects which, in conventional OO programs, are usually implemented by slices of multiple objects (i.e. they crosscut object structures). These approaches also propose concrete composition mechanisms; however, we will postpone discussing these mechanisms until Section 7.4. In Section 7.2.4, we will discuss how Domain Engineering relates to aspect-oriented decomposition.

7.2.1 Subject-Oriented Programming

Subject-Oriented Programming (*SOP*) was proposed by Harrison and Ossher of IBM Thomas J. Watson Research Center [SOP, HO93, OKH+95, OKK+96] as an extension of the object-oriented paradigm to address the problem of handling different *subjective perspectives* on the objects to be modeled. For example, the object representing a *book* for the marketing department of a publisher would include attributes such as *subject area* or *short abstract*, whereas the manufacturing department would be interested in rather different attributes, such as *kind of paper*, *kind of binding*, etc. The usage context of an object is not the only reason for different perspectives. The approach also seeks to address the problem of integrating systems developed with a large degree of independence, e.g. the two different applications for the publisher. In this case, we have to deal with perspectives of different *development teams* on the same objects (e.g. the book object). Furthermore, the goal is to be able to add previously unforeseen extensions to an existing system in a noninvasive way. Thus, subjective perspectives may be due to different end-users, contexts, and developers.

Subjects, mixins, and composition rules

Each perspective gives rise to a so-called *subject*. A subject is a collection of classes and/or class fragments (in the sense of *mixins*[86]) related by inheritance and other relationships owned by the subject. Thus, a subject is a partial or a complete object model. Subjects can be composed using *composition rules*. There are three kinds of composition rules:

- *correspondence rules*,
- *combination rules*, and
- *correspondence-and-combination rules*.

Correspondence rules specify the correspondence, if any, between classes, methods, and attributes of objects belonging to different subjects. For example, we could use a correspondence rule to express that the book in the marketing department application is the same as the book in the manufacturing department application (even if the corresponding classes had different names). We could have further correspondence rules stating the correspondence between methods and attributes of these classes. Alternatively, it is possible to use a special correspondence rule which will establish both the correspondence of two classes and all of their members having equal names. Of course, we can override this automatic correspondence for some members by defining additional member correspondence rules. Finally, we can use combination rules to specify how the two classes are to be combined. The resulting class will include the independent methods and attributes and the corresponding ones will be combined according to the combination rules. For example, a method coming from one subject could override the corresponding methods of the other subjects or all the corresponding methods could be executed in some specified order. When combining two independently developed subjects, we would usually develop a third, so-called *glue* subject including the code necessary to combine the other two. For convenience, there are also the correspondence-and-combination rules which provide a shortcut for specifying correspondence and combination at the same time. The details of the SOP composition rules can be found in [OKH+95, OKK+96, SOP]. We will also show a simple example of a subject-oriented program in Section 7.4.5.

It is worth noting that there is a close relationship between GenVoca (Section 6.4.2) and SOP. GenVoca can be easily simulated in SOP, by implementing each GenVoca layer as a separate subject.[87] However, the SOP composition rules represent a more elaborated composition mechanism than the layer composition in GenVoca.

As of writing, a prototype support for subjects exists as an extension of IBM's VisualAge for C++ and VisualAge for Smalltalk, and an implementation for VisualAge for Java is currently under development (see [SOP]). A useful feature of the Java prototype is the possibility to specify correspondence and composition visually using a graphical user interface.

7.2.2 Composition Filters

The *Composition Filters* (*CF*) approach by Aksit et al. [AT88, Aks89, Ber94, ATB96, CF] is motivated by the difficulties of expressing any kind of message coordination in the conventional object model. For example, expressing synchronization at the interface level of an object requires some way of – in effect – injecting synchronization code into all its methods which need to be synchronized. The straightforward solution of literally inserting this code into the methods (as in Figure 108 in Section 7.4.3) results in mixing functionality and synchronization code in a way that limits reusability. For example, extending a class with new methods by subclassing may require changes in the synchronization schema and thus overriding other methods even if their functionality remains the same. This and other related problems are referred to as *inheritance anomalies* [MWY90, MY93].[88] In short, we can state that one of the problems of the conventional object model is the lack of proper mechanisms to separate functionality from the message coordination code.

Inheritance anomalies

The CF approach extends the conventional object model with a number of different so-called *message filters*. Figure 101 shows the main elements of an object in the CF model. The object consists of an *interface layer* and an *inner object* (also referred to as *kernel object*). The inner object can be thought of as a regular object defined in a conventional OO programming language, e.g. Java or C++. The interface layer contains an arbitrary number of *input* and *output message filters*. Incoming messages pass through the input filters and the outgoing through the output filters. The filters can modify the messages, e.g. by changing the message selector or the target object.[89] In effect, they can be used to redirect messages to other objects, e.g. the *internal objects*, which are encapsulated in the interface layer, or some *external objects* referenced from the interface layer, and to translate messages (by modifying the selectors of the messages). They can also discard or buffer messages or throw exceptions. Whatever action is taken depends on the type of the filter. There are a number of predefined filter types, e.g. *delegation filters* (for delegating messages), *wait filters* (for buffering messages), *error filters* (for throwing exceptions), and new filter types can be added. Whether a filter modifies a message or not may depend on the message and also on some *state conditions* defined over the state of the inner object.

Message filters

Message filtering is a very powerful technique, which allows us to implement synchronization constraints [Ber94], real-time constraints [ABSB94], atomic transactions [ABV92], precondition-style error checking [Ber94], and other aspects in a well-localized way. Indeed, any aspect that lends itself to the implementation by intercepting message sends or “wrapping” methods in *before* and *after actions* (i.e. actions executed before or after executing a method) can be adequately represented in the CF model.

Delegation

The redirecting capability can also be used to implement *delegation* and *dynamic inheritance*. In short, delegation involves redirecting some messages received by a *delegating object* to another object, called the *delegate object*, which the delegating object holds a reference on. Furthermore, we also have to make sure that when the delegate object uses the keyword *self*, it refers to the delegating object and not the delegate objects.[90] This way, the methods of the delegate objects are written as if they were methods of the delegating objects and the delegate objects can be regarded as true extensions of the delegating objects. This is quite similar to the relationship between a class and its superclass: When the superclass uses the keyword *self*, it refers to the class of the receiver, which is not necessarily this superclass.

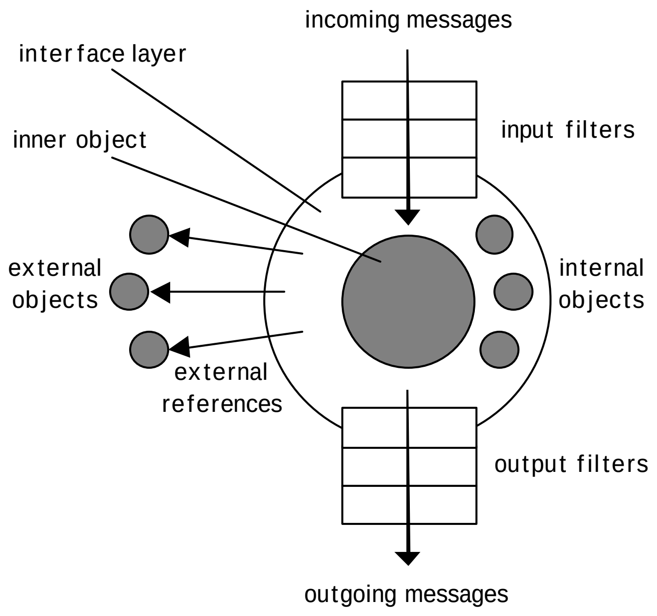


Figure 101 *Elements of an object in the CF model (adapted from [ATB96])*

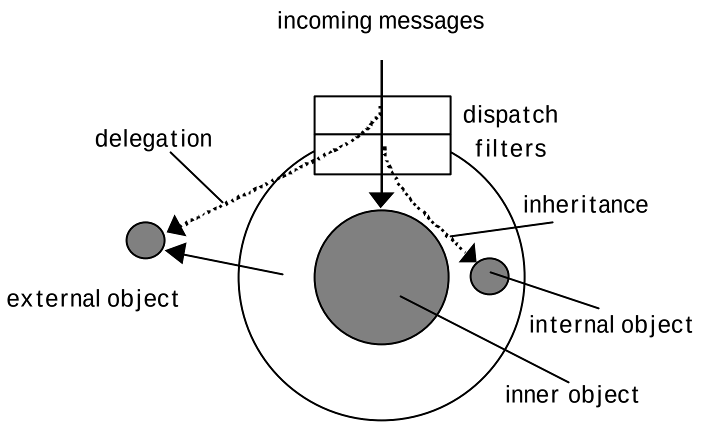


Figure 102 *Delegation and inheritance in the CF model*

In the CF model, delegation means redirecting messages to the external objects and making sure that *self* always refers to the original receiver (see Figure 102). Inheritance, on the other hand, means redirecting messages to the internal objects and making sure that *self* always refers to the original receiver. This relationship between inheritance and delegation was originally discussed in [Lie86b, Ste87].

Dynamic inheritance

A filter can also delegate a certain message to *different* internal objects based on some state conditions. Consequently, this means that the superclass of an object can change depending on its state, which is referred to as *dynamic inheritance*.

The details of the CF model can be found in [Ber94]. The model has been implemented as an extension of existing OO languages, e.g. C++ [Gla95] and Smalltalk [MD95]. Message filters can be implemented as metaobjects, so that they are present at runtime and thus can also be modified at runtime. Some filters, if necessary, can also be compiled away for performance reasons.

7.2.3 Demeter / Adaptive Programming

The original idea behind *Demeter / Adaptive Programming* was to provide a better separation between behavior and object structure in OO programs [Lie92, Lie96, Dem]. This was motivated by the observation that OO programs tend to contain a lot of small methods which do none or very little computation and call other methods passing information from one part of the object diagram[91] they operate on to other methods operating on other parts of the diagram. Trying to understand the computation in such programs involves an "endless chase" through such small methods and wondering where the "real" computation gets done. In addition to this understandability problem, the more serious flaw with such designs is that a simple change in the computation algorithm may require revisiting a large number of methods.

Law of Demeter

The large number of the small "information-passing" methods in classical OO designs is a direct consequence of the application of the *Law of Demeter* [LHR88]. Law of Demeter is an OO design principle stating that a method should only contain message sends to *self*, local instance variables, and/or method arguments.[92] In particular, we should avoid long sequences of accessing methods (e.g. `object.part1().part2().part3().part4()`), which perform some "deep" accessing into the object structure since such sequences hardwire large object structures into methods.[93] By following the Law of Demeter, we trade the "structure-hardwiring-in-large-methods" problem for the problem of having a large number of small information-passing methods. Given these two alternatives only, the latter is usually the preferred choice. Unfortunately, even when we follow the Law of Demeter, a change in the part of the object structure which is not directly involved in the computation, but needs to be traversed by the small information-passing methods, requires manual addition or modification of these methods.

In either case, we have a tight coupling between the conventional OO implementations of algorithms and the object structure: the code implementing the algorithms contains hard-coded names of classes which are not needed for the computation. For example, in order to compute the total salary paid by a company, we need to visit the Salary object of every employee in the company. Let us assume that the company consists of a number of divisions and each division contains a number of employees. One possible solution would be to implement a total-salary-computing method in the Company class, which calls a helper method of the Department class on each department object of the company, which calls a helper method of the Employee class on each employee object of a department. The latter two methods would accumulate the total salary for the departments and return it to the top-level method. Finally, the top-level method would return the total salary for the whole company. The problem with this implementation is that when we want to change the class structure, for example, by inserting *divisions* between the company and the departments, we need to implement a similar helper method in the new Division class, even if this method does not really contribute to the computation. Even if we implement the total-salary computation using the *visitor pattern* (see [GHJV95]), we still have to provide the *accept* method in *all* traversed classes as well as specialized methods (one per visited class) in the visitors. The latter need to be extended whenever new classes are added to the traversed portion of the class diagram.

Traversal strategies

A different solution proposed by Lieberherr [Lie96] involves writing behavior code against partial specifications of a class diagram instead of the whole concrete class diagram. The partial specifications mention only those classes which are really needed for the given computation. These partial specifications are referred to as *traversal strategies* (see [LP97]; in [Lie96] they are called *propagation directives*). An example of a simple traversal strategy for the total salary example is

```
from Company to Salary
```

This strategy states that the computation of the total salary involves traversing some given concrete class diagram of the company from the Company class to the Salary class. The code for the abstract method computing the total salary would state that an accumulator variable needs to be initialized when traversing Company and then passed down the company class diagram during traversal. Whenever a Salary object is visited, its value has to be added to the accumulator. Finally, when the traversal is back in Company, the total is returned. Please note that this total-salary computation specification does not mention any other classes than the ones referenced in the traversal strategy. Given this so-called *structure-shy behavior specification* (i.e. the algorithm code based on the traversal strategy) and a concrete class diagram, all the other necessary little "information-passing" methods in Department and Employee can be generated automatically. Moreover, when we extend the class graph with the Division class, no changes to the behavior specification are needed and we can simply re-generate our concrete program.

Structure-shy behavior

The term Adaptive Programming was introduced around 1991 and it covered the class structure aspect and the structure-shy behavior aspect described above [Lie98]. Later, it was extended with the synchronization aspect [LL94] and the remote invocation aspect [Lop95] (also see [Lop98]). Indeed, the work on Adaptive Programming and on the synchronization and the remote invocation aspects is one of the important roots of Aspect-Oriented Programming [Lie98]. According to a recent definition "Adaptive Programming is the special case of Aspect-Oriented Programming where one of the building blocks is expressible in terms of graphs and where the other building blocks refer to the graphs using traversal strategies. A traversal strategy is a partial specification of a class diagram pointing out a few cornerstone classes and relationships. Traversal strategies are graphs where each edge defines a regular expression specifying a traversal through a graph." [Lie98]

More recent work on Demeter [Lie97, ML98] is aimed at extending it with a fragment-based composition mechanism (cf. Section 7.4.5).

The ideas of Demeter have been integrated into OO languages such as C++ (Demeter/C++ [SHS94, Lie96]) and Java (Demeter/Java [LO97]).[94] For example, the Demeter/Java tool processes behavior specifications (wherein the algorithmic parts are expressed in Java) and class diagrams and produces Java programs. Additionally, it also provides a high-level means for instantiating complex object structures using sentences of grammars derived from class diagrams.

7.2.4 Aspect-Oriented Decomposition and Domain Engineering

Aspect languages

The relationship between Domain Engineering and aspect-oriented decomposition can be best explained using the Draco approach to Domain Engineering (see Section 6.4.1). The idea of Draco was to organize domain knowledge into a network of related domains, where each domain provides a *domain language*. In this model, applications are defined in terms of a number of high-level application and modeling domains, whereby each domain is used to describe certain aspect of the application. Similarly, the domains themselves are defined in terms of other domains. The decomposition based on domains does not necessarily adhere to functional structures. Thus, those of the domain languages explicitly aiming at capturing crosscutting aspects in a Draco model are *aspect languages* in the AOP sense.

Weaving

The transformational implementation inherent to the Draco approach is also closely related to the AOP idea. Compiling aspect-oriented programs involves processing a number of separate aspect representations (e.g. a number of aspect programs written in different languages or a number of different constructs within one language) in order to produce the resulting representation of the one common concept described by the crosscutting aspects. This compilation process is referred to as *weaving* [KLM+97] and may involve merging components, modifying them, optimizing, etc. The goal of weaving is the same as the goal of generators: to compute an efficient implementation for a high-level specification. However, it is important to note that weaving does not have to be implemented as a static generation process. It can also be realized by run-time interpretation of the aspect programs or run-time generation.[95] We will see a concrete example of dynamic weaving in Section 7.5.2.

Abstract vs. concrete features

The cornerstone of Domain Analysis is to model the common and variable system parts and their interdependencies in a class of systems. One of the techniques of Domain Analysis is to capture the commonalities and variabilities in feature models (see Section 5.4). It is important to note that features often crosscut functional designs. Just as the criteria for scoping a domain do not have to be technical ones (see Section 3.6.1), the same applies to identifying features. For example, some features could be determined based on some market analysis and thus there is a good chance that they will not fit into generalized procedures. In general, we will have some features which can be directly implemented as generalized procedures (e.g. objects), other features will be expressed using aspect languages (e.g. synchronization), yet other will be purely *abstract* and will require some configuration knowledge in order to be mapped onto the features of the two previous categories. Indeed, variability is just another aspect of reusable software. We already discussed the relationship between aspectual decomposition and feature modeling in Section 5.8.2.

Methods such as ODM or FODA (see Sections 3.7.1 and 3.7.2) provide an extremely general process for Domain Engineering. On the other hand, there are huge differences between domains found in practice. One way to deal with these differences is to introduce some categorization of domains and provide specialized Domain Engineering methods for each category. For example, domains such as data containers, image processing, matrix computations, and speech recognition, could be characterized as algorithmic domains. They can be quite adequately modeled using abstract data types (ADTs) and algorithms. Given a domain category, we can specialize an existing generic Domain Engineering method such as ODM to provide an effective support within this category. The key activities to be performed during the ODM domain analysis include domain scoping, finding the key concepts in the domain, and, finally, describing these concepts using feature models. The challenge is to jumpstart each of these activities since no concrete guidance is given by the ODM method on how to decompose problems into domains, concepts, and features. Indeed, ODM has been deliberately conceived as a generic method requiring specialization. One of the ODM modeling tools requiring specialization is a *concept starter set* (see Section 3.7.2). Concept starter sets are used to jumpstart the activity of finding the key concepts in a domain. The specialization of ODM for a certain domain category will require providing some concrete concepts categories for the concept starter set. For example, in the case of the category of algorithmic domains, the concept starter set would include concept categories such as ADTs and algorithms. In Section 5.8.1, we also introduced the analogous notion of *feature starter sets*, which help us to jumpstart feature modeling of the key concepts. For example, the feature starter set for ADTs would then include concerns such as operations, attributes, structure, synchronization, error handling, etc. The term "starter set" suggests that the items in the set are just good starting examples and that new kinds of concepts and features may emerge during analysis. Of course, some concerns in a feature starter set will be aspects in the AOP sense. We will see a concrete example of an ODM specialization in Chapter 9.

The items in a starter set correspond to concerns that we would like to separate. One of the concrete contributions of the AOP research is to identify and codify concerns for the starter sets. For example, the above-mentioned feature starter set for ADTs contains concerns which have drown a lot of attention of the AOP community. The process of codifying relevant concerns for different kinds of domains is a continuos one. We will have concrete languages for some concerns and other concerns may first just simply remain items in some domain-category-specific starter sets. The domain analysis process is aimed at developing languages (or language extensions) for the latter concerns.

7.3 How Aspects Arise

Examples of aspects

Subject composition rules, synchronization, real-time constraints, error checking[96], and structure-shy behavior are examples of aspects, which the approaches we discussed in previous section help to separate. There are many other examples of aspects, e.g. object interaction [AWB+93, BDF98], memory management [AT96], persistence, historization, security, caching polices, profiling, monitoring, testing, structure and representation of data, and domain-specific optimizations (see e.g. Sections 6.4.1 and 9.4.1). Many aspects arise together in certain kinds of systems. For example, some of the aspects of distributed systems include component interaction, synchronization, remote invocation, parameter transfer strategies, load balancing, replication, failure handling, quality of service, and distributed transactions (see [Lop97, BG98]).

But how do aspects in the AOP sense arise? Some aspects follow structures which naturally crosscut generalized procedures such as control flow or data flow. For example, synchronization, real-time constraints, and object interaction follow control flow and parameter-transfer strategies in distributed systems [Lop97] and caching strategies follow data flow. Furthermore, subjective perspectives modeled in SOP and variability modeled in Domain Analysis are often expressed in terms of incremental deltas which crosscut objects. Finally, domain-specific optimizations clearly represent a category of aspects. As we discussed in Section 6.3.1.2, optimizations involve interleaving and delocalization, i.e. some of the high-level concepts of the specification to be optimized are interleaved (i.e. merged) and some of them are distributed among other lower-level concepts. These two effects lie at the heart of the so-called *code tangling* [KLM+97] found in the optimized version, i.e. the optimized code is hard to understand and maintain. In fact, code tangling occurs whenever we try to implement aspects using generalized procedures only. In the case of optimizations, instead of writing the optimized, tangled code by hand, we should rather represent our program using the high-level, unoptimized code and a set of domain-specific optimizations (e.g. in the form of rewrite rules or some other transforms). The optimized code can then be obtained by applying the optimizations to the unoptimized code using a generator. Examples of such optimization are loop fusing and elimination of temporaries in matrix computation code. We will see an implementation of this aspect of matrix code in Sections 10.2.6 and 10.3.1.7.

Code tangling

Crosscutting

Crosscutting lies at the heart of aspects. As you remember our discussion from Section 5.8.2.2, modular units of decomposition are organized into clear hierarchies, whereas aspects crosscut such hierarchies. This is illustrated in Figure 103.

What is an aspect?

We also stated in Section 5.8.2.2 that the quality of being an aspect is a relative one: *a model is an* aspect *of another model if it crosscuts its structure.* The aspect shown in Figure 103 is an aspect with respect to the hierarchical structure also shown in this figure. However, at the same time, the aspect could be a modular unit of another hierarchical structure not shown in the figure.

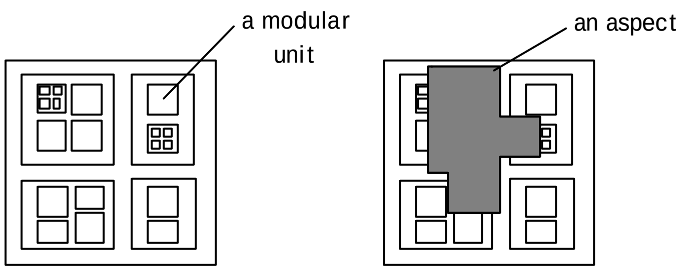


Figure 103 *Modular vs. aspectual decomposition*

Since an aspect is relative to some model, it might be possible to refractor the model so that the aspect ceases to be an aspect of the model. For example, some interaction patterns between a number of components are an aspect of the component structure since they crosscut the structure. One way to deal with this problem is to introduce a mediator component that encapsulates the interaction patterns (see *mediator pattern* in [GHJV95]). Thus, we refactored our design and turned an aspect into a component. (This pattern-based solution is not always an ideal one: in addition to simply moving the problem into a new component, it introduces extra complexity and possibly performance penalties. But, nonetheless, it is a solution.) In general, it would be foolish to think that we can "get rid" of all aspects simply by refactoring our models. Localizing some issues will always cause some other issues to become aspects (just as in our example with the signal diagram in Figure 48, Section 5.8.2.1). Real systems will always have some "inherent crosscutting" that refactoring will not be able to get rid of. Rather than trying to sweep the problem under the rug, we have to provide adequate technology to deal with the crosscutting.

It is worth noting that the code tangling problem due to crosscutting tends to occur in later phases of the conventional development process. We usually start with a clean, hierarchical functional design, then manually add various aspects (e.g. code optimizations, distribution, synchronization — indeed, all of the aspects listed at the beginning of this sections are aspect with respect to functional decompositions), and the code becomes tangled. Separating the aspects from the functionality code allows us to avoid this code tangling. Once we have identified the relevant aspects, we still need to find

- appropriate linguistic means of expressing aspects and
- efficient mechanisms for composing them.

Using generalized procedures only (e.g. objects, procedures, functions, etc.) may be appropriate for implementing the modular units of functionality, but it does not work for aspects since we still need mechanisms for dealing with crosscutting. Appropriate composition mechanisms provide a solution to this problem. We will discuss them in Section 7.4. Applying specialized linguistic means to capture aspects themselves allows us to further reduce complexity. We will discuss this idea in Section 7.5.

7.4 Composition Mechanisms

Function calls, static and dynamic parameterization, and inheritance are all examples of important composition mechanisms supported by conventional languages (see Section 5.5). However, as we stated in the previous sections, not all relevant aspects occurring in practice can be adequately composed using these mechanisms and thus we need new ones. Subject composition rules in SOP, message filters in CF, and traversal strategies in Demeter are some examples of the new composition mechanisms we have already discussed in Section 7.2 (see [NT95, CIOO96] for more examples of composition mechanisms).

In this section we will give some requirements for composition mechanisms. We will then show how these requirements can be satisfied. In the discussion, we will use a simple example of separating the synchronization aspect from the base implementation of a stack. We will walk through various stages of separation: starting with a tangled version mixing synchronization with the functionality code, then a somewhat better version using inheritance, then a cleanly separated version using the composition mechanisms of SOP. In Section 7.5, we will revisit the stack example and show you how to represent the synchronization aspect itself in a more intentional way.

7.4.1 Some Requirements

Ideally, we would like the aspect composition mechanisms to allow

- *minimal coupling* between aspects,
- different *binding times and modes* between aspects, and
- *noninvasive addition of aspects* to existing code (and thus *noninvasive adaptability* of existing code).

The following three sections explain each of these requirements.

7.4.1.1 Minimal Coupling

We require minimal coupling between aspects rather than complete separation between them since complete separation is only possible in a few trivial cases and the majority of aspects cannot be completely separated. The reason for the latter is that aspects describe different perspectives on some single model. Even the term *orthogonal* perspectives does not imply complete separation. We can back this observation with a simple example. The leftmost drawing in Figure 104 shows three orthogonal perspectives of a 3-D object. In our example, the perspectives are a rectangle and a triangle on the side planes and a circle on the bottom plane. Next, we might use our three orthogonal planes as a kind of generator, i.e. we put some 2-D figures on the planes and interpret them as perspectives of some 3-D object to be built. Even if our perspectives are orthogonal (in terms of angles), the 2-D figure cannot be chosen independently. The middle drawing in Figure 104 shows us a set of 2-D figures which are inconsistent in our model, i.e. we cannot construct an 3-D object with these figures as its orthogonal perspectives. A consistent set of 2-D figures is shown in the rightmost drawing in Figure 104.

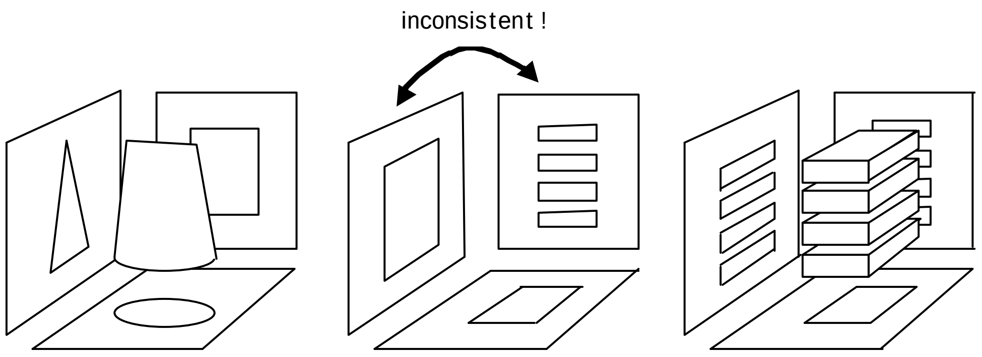


Figure 104 *Examples of consistent and inconsistent orthogonal perspectives of 3-D objects*

Demeter provides an excellent example of composition with minimal coupling (see Section 7.2.3). Conventional OO behavior code hardwires the object structure it works on by explicitly mentioning more classes and relationships than it really needs. Classes and relationships required for computation need to be mentioned, but classes and relationships used just for navigation need not. The traversal strategies in Demeter (i.e. the partial class diagram specifications) provide for a declarative, loose coupling between class structure and structure-shy behavior and also between class structure and remote communication (see [Lop98]). They can be viewed as the *aspect composition mechanism* between the class structure aspect and the behavior aspect and the remote communication aspect in Demeter.

Join points

The example of traversal strategies as a composition mechanism also illustrates the concept of *join points* [KLM+97]. Traversal strategies are part of the behavior aspect and they mention some of the classes and relationships from the class structure aspect. We say that these classes and relationships represent the join points between the behavior aspect and the class structure aspect. In the case of SOP (see Section 7.2.1), the join points between the subjects and the subject composition rules are class names, method names, and attribute names. In general, we can distinguish between three types of join points between aspects:

- Simple "by name" references to a definition of a language constructs (e.g. classes, methods, attributes, etc.). For example, an aspect could refer to the definition of a method and state that calls to this method should be logged to a file. Thus, by referring to the definition, we affect all calls to the method.

- Qualified "by name" references. Sometimes we do not want to log all calls to the method M1 but only those made within M2. In this case, the aspect would make a reference to M1 qualified with M2. Thus, qualified references allow us to refer to some points of use of a language construct.

- References to patterns.[97] Traversal strategies are an example of pattern-based coupling. A structure-shy behavior specification refers to certain regions in a class graph using traversal strategies, i.e. patterns.

Thus, if you see some constant piece of code accompanying some definition or instance of a construct or some code pattern, you can turn it into an aspect and use the construct definition or instance or the code pattern as a join point. This way, you will avoid code duplication since the constant piece of code will be stated only once.

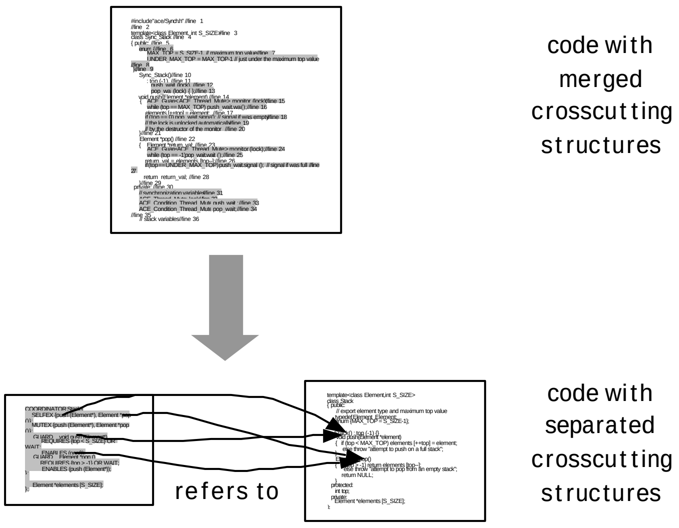

code with merged crosscutting structures
refers to
code with separated crosscutting structures

Figure 105 *Separating crosscutting structures*

The essence of separating aspects is illustrated in Figure 105. The top box displays some code whose different lines implement two different aspects (lines implementing one of the aspects are highlighted; don't try to read this code!). By separating these aspects, we get two pieces of code, each implementing one aspect. These pieces still refer to each other (or, in this particular case, one of them refers to the other one only). The points of reference are the join points. It is important to point out that, in the separated case, an aspect says something declaratively about the model it crosscuts rather than having the model make calls to the aspect. For example, in the later sections, we will see a synchronization aspect that makes statements about a stack such as "you should not push and pop elements at the same time". Thus, an aspect extends or modifies the semantics of the model it refers to. We will come back to this topic in Section 7.4.1.3.

Message join points

An important category of join points in OO are *message join points* (or *operation join points*; see [OT98]), i.e. points where operations are defined or called. As demonstrated in the CF approach (see Section 7.2.2), they can be used to couple functional designs with aspects such as concurrency, real-time constraints, atomic transactions, precondition-like error handling, and security. Other aspects amenable to coupling through message join points are profiling, monitoring, testing, caching, and historization. They can all be implemented by intercepting message sends.

7.4.1.2 Different Binding Times and Modes

As stated in Section 7.2.4, there is an important relationship between features and aspects: some features are aspects in the AOP sense. Features found during Domain Analysis are documented using feature diagrams (see Section 5.4.1), which are annotated with binding times, e.g. whether they are bound before runtime or during runtime. Indeed, we stated later in Section 5.4.4.3 that many different, even product-specific binding times are possible (the generalized concept is referred to as *binding site*). Thus, it is important that a composition mechanism supports different binding times.

We also would like to support both static and dynamic binding (i.e. different *binding modes*). Static binding means optimized and "frozen" binding, e.g. inlining, in which case rebinding is more difficult and time consuming since we possibly need to do some code regeneration. Please note that static binding does not have to happen at compile time since code regeneration or code modification can also be performed at runtime. Dynamic binding, on the other hand, leaves some indirection code between the entities to be bound. This extra code allows a quick rebind, or the binding is even computed lazily (as in the case of dynamic method binding in OO languages). We should use dynamic binding whenever rebinds are very often and static binding otherwise. It might be also useful to be able to switch between static and dynamic binding at runtime.

There are several important issues concerning binding time and binding mode. First, the composition mechanism for gluing the concrete feature implementations together needs to support different binding times and modes. Second, there is also the code which computes configurations of concrete features based on other concrete and abstract features and it should be possible to execute this code at different binding times. Finally, we want to be able to reuse as much code across different binding times as possible. In other words, we do not want to have different implementation versions of concrete features for different binding times but only one per feature (i.e. binding time should be a parameter of the composition mechanism). Similarly, we do not want different versions of configuration code for different binding times.

The latter two requirements are difficult to impossible to satisfy using conventional programming languages. For example, in C++, we have to use a complicated idiom in order to parameterize the binding mode of feature composition (see Section 7.10). We cannot satisfy the second requirement in C++ since static configuration code is written using template metaprogramming code (see Chapter 8) and dynamic configuration code is written as usual procedural C++ code. In IP (see Section 6.4.3), on the other hand, we can call the same procedure at generation and at runtime.

7.4.1.3 Noninvasive Adaptability

By *noninvasive adaptability* we mean the ability to adapt a component or an aspect without manually modifying it. This is trivial, if the component or aspect provides a parameter (or any other type of variation point) for the kind of change we would like to make. However, we would also like to be able, as far as possible, to make unplanned noninvasive adaptations for which no special hooks were foreseen in the component or the aspect.

Ideally, we want to express any change as an additive operation: we use some composition operator to add the change to the existing code. The code we want to modify is not really physically modified, but the composition expression itself states that the original code has a modified semantics now. An example of such composition operator is inheritance: we can define a new class by difference. Unfortunately, inheritance is noninvasive with respect to the superclass only. The client code has to be changed at some location(s), in order to create objects of the new derived class instead of the superclass (unless we combine inheritance with other techniques — see Section 7.4.4). Of course, we prefer composition mechanisms that are noninvasive with respect to both the component and the client code, i.e. we can modify the meaning of a component without changing its code or the client code. We will also discuss other examples of noninvasive composition mechanisms in later sections.

An example of a composition mechanism supporting this kind of noninvasive adaptability is the composition mechanism of SOP. In SOP, we can write a composition rule that composes a class and an extension to this class and publishes the new class under the same name as the original one, so that no changes in the client code are necessary. We will see an example of such composition rule in Section 7.4.5.

Noninvasive adaptability is not possible in all cases. The prerequisite is that we have some appropriate "handle" in the code of the component to be adapted. This handle does not have to be an explicit "hook" foreseen during the design. For example, using SOP, we can noninvasively override any method of a component. If a component does not have the appropriate method that we can override to achieve the desired modification, SOP will not help. Thus, the method join points go only a limited way. If our composition mechanism supports pattern join points, we can cover more complicated cases, where the locations to be modified in the component can be identified using a pattern (e.g. complex modifications within the method code). We could then use a transformation system to generate the modified code. Finally, if the only way to identify some location in the component is by stating "the spot between lines 48 and 49", we need to redesign the component.

Loose coupling is also related to the issue of adaptability. When we change one aspect, it might be necessary to change another one, which is supposed to be composed with the first one. If the coupling between the aspects is minimal, then there is less chance that the change of one aspect requires the change of the other one. For example, if we change the class structure of a conventional OO program, we often have to change a large number of methods. On the other hand, when we apply the Demeter approach, we can make more kinds of changes to the class structure without having to change the behavior aspect than in the conventional program: Changes to the parts of the class structure which are not mentioned in the traversal strategies have no effect on the behavior specification. Thus, avoiding overspecification increases adaptability.

7.4.2 Example: Synchronizing a Bounded Buffer

Imagine that you have a bounded buffer component, e.g. a stack or a queue with a limited size, and you want to synchronize access to it in a multithreaded environment. In other words, there will be clients that run in different threads and access the buffer to put elements in and take elements out. Since the buffer is a shared resource, the access of the different clients to it has to be synchronized.

Ideally, we would like to implement the buffer in one piece of code, e.g. as a class, and express the synchronization aspect in a separate piece of code. This is shown in Figure 106. The synchronization aspect would state some synchronization constraints, such as you should not put and take elements at the same time, you should wait with a put if the buffer is full, you should wait with a take if the buffer is empty, etc. This is an example of the classical "bounded buffer synchronization", which we will discuss in Section 7.4.3 in detail.

At this point, we can make two important observations: One observation is that the aspect "says something *about*" the buffer. In order to be able to say something about the buffer, it has to refer to some parts of the buffer. In our case, the synchronization aspect mentions the buffer methods put, take, is_empty, and is_full.

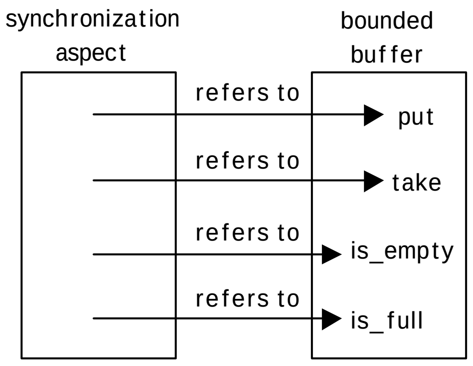


Figure 106 *Bounded buffer and its synchronization aspect*

Another important observation is that, whereas the buffer can be used "stand alone" (e.g. in a single threaded program), the synchronization aspect cannot. The synchronization aspect does not do anything useful on its own. In order to do something useful, it needs to be used with some concrete buffer. This example indicates the kind of asymmetry between components and their aspects that is common in practice (we often refer to the components as the *primary structure*). Nonetheless, a bounded buffer synchronization aspect is a reusable piece of code since you can reuse it on a queue, on a stack, or on some other kind of buffer.

Primary structure

Now, let us take a look at what the composition requirements discussed in the previous sections mean in our context:

- *Minimal coupling*: We want the aspect to mention as few details as possible about the component. In our case, the aspect refers to the methods put, take, is_empty, and is_full, which is necessary to formulate the synchronization constraints. Thus, we use operation join points to connect the synchronization aspect to the component.

- *Different binding times and modes*: In general, we want to be able to plug and unplug an aspect at any time. In the case of synchronization, this will be of particular interest if we have to coordinate a number of components (see e.g. Section 7.5.2.1).[98] For example, if the components are dynamically reconfigurable, we also should be able to plug and unplug synchronization aspects dynamically.

- *Noninvasive adaptability*: We should be able to add the synchronization aspect to a component without having to modify the component or the client code manually (see Figure 107). The composition is taken care of by weaving (static or dynamic). Weaving effectively "injects" aspect code into the component at appropriate places. We will discuss ways of implementing weaving later in Section 7.6.2.

In the following sections, we will gradually come to this ideal solution outlined above. In our discussion, we will use a simple stack component as an example of a bounded buffer. We first start with a "tangled" version of the stack, i.e. one that hardwires synchronization directly into the functional code. We then show how to separate the synchronization aspect using design patterns and the SOP composition mechanism. Finally, we give you the ideal solution in Java and AspectJ and in Smalltalk.

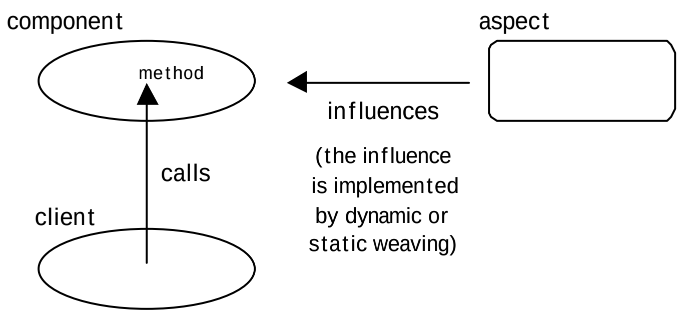


Figure 107 *Noninvasive addition of an aspect to a component*

7.4.3 "Tangled" Synchronized Stack

We start with a simple C++ implementation of a synchronized stack in which the synchronization code is not separated from the functionality code at all. The C++ code is shown in Figure 108 (the equivalent Java implementation is shown in Figure 121). The stack has a push() and a pop() method (line 14 and line 22) and the code responsible for synchronization has been highlighted. The synchronization code involves five synchronization variables: lock, push_wait, pop_wait (lines 32-34), monitor in push() (line 15) and monitor in pop() (line 24). lock is the most important synchronization variable on which all synchronization locking is performed. The remaining synchronization variables are just wrappers on lock allowing for different styles of locking. The implementation in Figure 108 uses the publicly available and portable ACE library [Sch94], which provides a number of synchronization classes, such as ACE_Thread_Mutex, ACE_Condition_Thread_Mutex, and ACE_Guard, which wrap the low-level synchronization variables provided by operating systems (see [Sch95] for a detailed discussion of synchronization wrappers).

Let us first state the synchronization constraints that this synchronization code implements:

1. push() is self exclusive, i.e. no two different threads can execute push() at the same time;

2. pop() is self exclusive;

3. push() and pop() are mutually exclusive, i.e. no two different threads can be executing push() and pop() at the same time;

4. push() can only proceed if the stack is not full;

5. pop() can only proceed if the stack is not empty.

Thus, push() and pop() are both *mutually exclusive* and *self exclusive*. This is achieved by locking the shared lock variable on entering push() and pop() and releasing it on leaving push() and pop(), respectively. This locking is automated using the class template ACE_Guard. We wrap lock into ACE_Guard by declaring the temporary variable monitor at the beginning of push() and pop() (lines 15 and 24). The effect of this wrapping is that lock is locked in the constructor of monitor at the beginning of the synchronized methods and released automatically in the destructor of monitor at the end of the methods (see the comment in lines 19-20). Thus, the programmer does not have to manually release lock at the end of the method, which has been a common source of errors. The declaration of monitor at the beginning of a method amounts to declaring the method as *synchronized* in Java (compare Figure 121).

In addition to making push() and pop() mutual and self exclusive, we also need to suspend any thread that tries to push an element on a full stack or pop an element from an empty stack (i.e. requirements 4 and 5). The implementation of these constraints involves the conditional synchronization variables push_wait and pop_wait (lines 33-34 and 12-13). The stack conditions of being full or empty are checked at the beginning of each method (lines 16 and 25) and, if necessary, the current thread is put to sleep by calling wait() on push_wait or on pop_wait. At the end of each method, signal() is called on pop_wait or on push_wait if the state of the stack transitions from empty to not empty or from full to not full. For example, when a thread waits to pop an element from an empty stack and some other thread pushes some element on the stack and executes signal() on pop_wait, the waiting thread wakes up, evaluates its condition (which is false), and pops the element. The synchronization of push() and pop() is an example of the classical *bounded buffer synchronization schema* which is used for synchronizing concurrent access to data buffers (see e.g. [Lea97]).

The problem with the implementation in Figure 108 is that it mixes the synchronization code with the functionality code. This has a number of disadvantages:

- *Hard-coded synchronization*: Since synchronization is not parameterized, it cannot be changed without manually modifying the stack class. A reusable component has to work in different contexts and different synchronization strategies might be required in different contexts. In the simplest case, we would also like to use the stack in a single-threaded environment, but the synchronization code introduces unnecessary overhead and, even worse, will cause a deadlock when trying to push an element on a full stack or pop an element from an empty stack.

- *Tangled synchronization and functionality aspects*: The functionality code is mixed with the synchronization code, which makes it more difficult to reason about each of these aspects in separation. This causes maintenance problems and impairs adaptability and reusability. More complex components might also require some extra state variables to be used in the synchronization conditions. Such variables are referred to as the *synchronization state*. Thus, our naive, tangled solution also mixes the synchronization state with the logical state of the component. In the stack example, we saw that the synchronization aspect crosscuts the methods of the stack. In general, the situation may be much worse since we often need to coordinate a number of objects and the synchronization code may crosscut all or some of them in different ways. We discuss different kinds of crosscutting found in OO programs in Section 7.4.8.

- *Non-intentional representation of the synchronization aspect*: The synchronization constraints are not represented explicitly in the code. For example, instead of the code involving lock and mutex, all we would have to say is to annotate the push and pop methods as mutually and self exclusive.[99] Also the implementation of the conditional synchronization is too implicit and exposes too much detail. We will see a better solution in Section 7.5.1.

```
#include "ace/Synch.h"                                                   //line 1
                                                                         //line 2
template<class Element, int S_SIZE>                                      //line 3
class Sync_Stack                                                         //line 4
{ public:                                                                //line 5
   enum {                                                                //line 6
      MAX_TOP = S_SIZE-1, // maximum top value                           //line 7
      UNDER_MAX_TOP = MAX_TOP-1 // just under the maximum top value      //line 8
  };                                                                     //line 9
   Sync_Stack()                                                          //line 10
      : top (-1),                                                        //line 11
        push_wait (lock),                                                //line 12
        pop_wait (lock) { };                                             //line 13
   void push(Element *element)                                           //line 14
   {   ACE_Guard<ACE_Thread_Mutex> monitor (lock);                       //line 15
       while (top == MAX_TOP) push_wait.wait();                          //line 16
       elements [++top] = element;                                       //line 17
       if (top == 0) pop_wait.signal(); // signal if was empty           //line 18
       // the lock is unlocked automatically                             //line 19
       // by the destructor of the monitor                               //line 20
   }                                                                     //line 21
   Element *pop()                                                        //line 22
   {   Element *return_val;                                              //line 23
       ACE_Guard<ACE_Thread_Mutex> monitor (lock);                       //line 24
       while (top == -1) pop_wait.wait();                                //line 25
       return_val = elements [top--];                                    //line 26
       if (top == UNDER_MAX_TOP) push_wait.signal();  // signal if was full //line 27
       return return_val;                                                //line 28
   }                                                                     //line 29
  private:                                                               //line 30
   // synchronization variables                                          //line 31
   ACE_Thread_Mutex lock;                                                //line 32
   ACE_Condition_Thread_Mutex push_wait;                                 //line 33
   ACE_Condition_Thread_Mutex pop_wait;                                  //line 34
                                                                         //line 35
   // stack variables                                                    //line 36
   int top;                                                              //line 37
   Element *elements [S_SIZE];                                           //line 38
};                                                                       //line 39
```

Figure 108 *"Tangled" implementation of a synchronized stack in C++*

7.4.4 Separating Synchronization Using Design Patterns

A standard way of separating synchronization code from the functionality code is to use inheritance (see [Lea97]). The usual implementation involves putting the synchronization into a subclass. The subclass wraps the methods of the superclass which have to be synchronized with appropriate synchronization code. In C++, we can additionally parameterize the superclass, so that the synchronization class can be reused on a number of related functionality classes. The implementation of the synchronization class uses the *inheritance-based static wrapper idiom* shown in Figure 75. The implementation of the stack and a stack synchronizer is shown in Figure 109.

Stack in Figure 109 does error checking which is needed if we also want to use it in a single-threaded environment. Trying to push an element on a full stack or pop an element from an empty stack in a single-threaded application is clearly an error. Stack checks for these conditions and throws an exception if necessary. For use in a multithreaded application, we can wrap Stack in the synchronization wrapper in order to make it thread safe as follows: Sync_Stack_Wrapper<Stack<...> >. One of the advantages of the wrapper solution is that we can reuse the synchronization wrapper for different stack implementations, e.g. stacks using different containers for storing their elements.

```
template<class Element, int S_SIZE>
class Stack
{  public:
      // export element type and maximum top value
      typedef Element Element;
      enum {MAX_TOP = S_SIZE-1};

      Stack() : top (-1) {}
      void push(Element *element)
      {   if (top < MAX_TOP) elements [++top] = element;
          else throw "attempt to push on a full stack";
      }
      Element *pop()
      {   if (top > -1) return elements [top--];
          else throw "attempt to pop from an empty stack";
          return NULL;
      }
   protected:
      int top;
   private:
      Element *elements [S_SIZE];
};
```

```
template<class UnsyncStack>
class Sync_Stack_Wrapper : public UnsyncStack
{  public:
     // import elemet type and maximum top value
     typedef UnsyncStack::Element Element;
     enum {MAX_TOP = UnsyncStack::MAX_TOP};
     // declare UNDER_MAX_TOP
     enum {UNDER_MAX_TOP = MAX_TOP-1};

     Sync_Stack_Wrapper()
           : UnsyncStack (),
             push_wait (lock),
             pop_wait (lock) { }
     void push(Element *element)
     {    ACE_Guard<ACE_Thread_Mutex> monitor(lock);
          while (top == MAX_TOP) push_wait.wait();
          UnsyncStack::push(element);
          if (top == 0) pop_wait.signal(); // signal if was empty
     }
     Element *pop()
     {   Element *return_val;
          ACE_Guard<ACE_Thread_Mutex> monitor (lock);
          while (top == -1) pop_wait.wait();
          return_val = UnsyncStack::pop();
          if (top == UNDER_MAX_TOP) push_wait.signal();  // signal if was full
          return return_val;
     }
   private:
     // synchronization variables
     ACE_Thread_Mutex lock;
     ACE_Condition_Thread_Mutex push_wait;
     ACE_Condition_Thread_Mutex pop_wait;
};
```

Figure 109 *Implementation of a stack and a synchronizing stack wrapper using parameterized inheritance*

The stack implementation in Figure 109 still has one major deficiency: If we wrap the stack into the synchronization wrapper to be used in a multithreaded environment, the base stack implementation still checks for errors, although the checking is not needed in the this case. One solution to this problem is to separate the error checking aspect from the base implementation of the stack using another inheritance-based static wrapper, just as we did it with the synchronization aspect. This is shown in Figure 110. Thus, we implemented the synchronization aspect as a “mixin class” with parameterized inheritance.

```
template<class Element, int S_SIZE>
class Stack
{  public:
     // export element type and maximum top value
     typedef Element Element;
     enum {MAX_TOP = S_SIZE-1};

     Stack() : top (-1) {}
     void push(Element *element)
     {   elements [++top] = element;
     }
     Element *pop()
     {   return elements [top--];
     }
   protected:
     int top;
   private:
     Element *elements [S_SIZE];
};

template<class UnsafeStack>
class Balking_Stack_Wrapper : public UnsafeStack
{  public:
     // import elemet type and stack size
     typedef UnsafeStack::Element Element;
     enum {MAX_TOP = UnsafeStack::MAX_TOP};

     Balking_Stack_Wrapper()
          : UnsafeStack () {}
     void push(Element *element)
     {    if (top < MAX_TOP) UnsafeStack::push(element);
          else throw "attempt to push on a full stack";
     }
     Element *pop()
     {    if (top > -1) return UnsafeStack::pop();
          else throw "attempt to pop from an empty stack";
          return NULL;
     }
};
```

Figure 110 *Implementation of a stack without error checking and a balking stack wrapper using parameterized inheritance*

Given the code in Figure 110, Balking_Stack_Wrapper<Stack<...> > has the same meaning as Stack<...> in Figure 109. Now, we can use Balking_Stack_Wrapper<Stack<...> > in a single-threaded application and Sync_Stack_Wrapper<Stack<...> > in a multi-threaded application (in both cases, we use Stack from Figure 110). Of course, we could further parameterize the synchronization and the error checking wrappers to provide different synchronization, error-checking, and error-response polices.

The stack example also illustrates an important point: there are interdependencies between aspects. Trying to push an element on a full stack or to pop an element from an empty stack concerns both error handling and synchronization. In fact, in a multithreaded environment, we treat the following different polices uniformly — simply as alternatives [Lea97]:

- *Inaction*: Ignoring an action if it cannot be performed.
- *Balking*: Throw an exception if an action cannot be performed (e.g. Balking_Stack_Wrapper).

- *Guarded suspension*: Suspending a thread until the precondition becomes true (e.g. Sync_Stack_Wrapper)

- *Provisional action*: Pretending to perform an action, but not committing to its effects until success assured.

- *Rollback/Recovery*: Trying to proceed, but upon failure, undoing any effects of partially completed actions.

- *Retry*: Repeatedly attempting failed actions after recovering from previous attempts.

The applicability of these policies is discussed in [Lea97] in detail.

Adding synchronization or error handling by method wrapping (as in Sync_Stack_Wrapper or Balking_Stack_Wrapper) is not always possible. Sometimes we need to make a call to a synchronizer or an error checker somewhere in the middle of a method. In this case, we can parameterize the method with the synchronizer or the error checker using the *strategy pattern* (see [GHJV95]]). If static binding is sufficient, we can use a static version of the strategy pattern based on a template parameter. The pattern is illustrated in see Figure 111. The listing shows three different strategies. Please note that the strategy method someAction() is declared as static in each strategy. If, in a certain context, no action is necessary, we can use EmptyStrategy which implements someAction() as an empty method. If we use inlining, someAction() will not incur any overhead. If necessary, we can also pass *this as a parameter to the strategy method. Defining the strategy method as static minimizes any overhead; however, if the strategy algorithm needs to retain some state between calls, we would define the strategy method as an *instance* method and add a strategy instance variable to the component. If we want to allow for both static and dynamic binding, we need to use the idiom presented in Section 7.10.

```
class StrategyA
{   public:
    static void someAction()
    {    // do some work
    }
};

class StrategyB
{   public:
    static void someAction()
    {    // do some other work
    }
};

class EmptyStrategy
{   public:
    static void someAction()
    {} // empty action
};

template <class Strategy>
class Component
{   public:
    //...
    void method()
    {   // some method-specific work
        Strategy::someAction();
        // some other method-specific work
    }
    //...
};
```

Figure 111 *Strategy pattern based on a template parameter*

Unfortunately, calling hook methods such as someAction() pollutes the code implementing the basic functionality. Sometimes we can solve this problem by splitting methods and classes in a way that the wrapper pattern can be used (e.g. the section of the code that needs to be synchronized could be moved into a separate method). In order to solve this problem, we need new composition mechanisms that address the different kinds of crosscutting discussed later in Section 7.4.8.

Once we decide to use the static wrapper solution presented in this section, we still need to somehow modify the client code in order to create object of Sync_Stack_Wrapper<Stack<...> > instead of Stack<...>.

Let us assume that Stack is declared in file stack.h and Sync_Stack_Wrapper is declared in sync_stack_wrapper.h. Furthermore, some client client.cpp includes stack.h and uses Stack. One way to make the client use the wrapped stack instead of Stack is to move stack.h into some new directory, e.g. original, and create a new stack.h at the same place where the original stack.h resided before moving it. The content of the new stack.h is shown in Figure 112.

```
//stack.h
namespace original {
    #include "original/stack.h"
}
namespace extension {
    #include "sync_stack_wrapper.h"
}
// wrap original::Stack into extension::Sync_Stack_Wrapper and publish it as Stack
template<class Element, int S_SIZE>
class Stack : public extension::Sync_Stack_Wrapper<original::Stack<Element, S_SIZE> >
{   public:
        Stack() : extension::Sync_Stack_Wrapper<original::Stack<Element, S_SIZE> >() {}
};
```

Figure 112 *Content of the new* stack.h

7.4.5 Separating Synchronization Using SOP

Using the SOP approach, we can nicely separate functionality code from the synchronization code by encapsulating them in two separate subjects. In the C++ version of SOP (see [SOP]) a subject is modeled simply as a C++ namespace.

Let us assume that Stack has been defined in the namespace original. The implementation is shown in Figure 113. Unfortunately, the current implementation of SOP/C++ does not support the composition of class templates. Therefore, we implement Stack as a C++ class. The constants Element and S_SIZE are then defined using the preprocessor directive #define in a separate include file (see Figure 114).

```
// stack.h
#include "constants.h"

namespace original {
class Stack
{  public:
      Stack() : top (-1) {}
      void push(Element *element)
      {   elements [++top] = element;
      }
      Element *pop()
      {   return elements [top--];
      }
   private:
      int top;
      Element *elements [S_SIZE];
};
} //namespace original
```

Figure 113 *Implementation of the class* Stack

```
//constants.h
#ifndef CONSTANTS_H
#define CONSTANTS_H

#define Element int
#define S_SIZE 10

#endif
```

Figure 114 *Content of* constants.h

Next, we define the new namespace sync which contains the stack synchronization class Stack. We will later compose original::Stack with sync::Stack in order to synchronize it. The code for sync::Stack is shown in Figure 115.

```
// sync_stack_extension.h                                                          // line  1
#include "ace/Synch.h"                                                             // line  2
#include "constants.h"                                                             // line  3
                                                                                   // line  4
namespace sync {                                                                   // line  5
class Stack                                                                        // line  6
{  public:                                                                         // line  7
    Stack ()                                                                       // line  8
        : push_wait (lock),                                                        // line  9
          pop_wait (lock) { }                                                      // line 10
    void push(Element *element)                                                    // line 11
    {    ACE_Guard<ACE_Thread_Mutex> monitor (lock);                               // line 12
        while (top == S_SIZE-1) push_wait.wait();                                  // line 13
        inner_push(element);                          //  <----- call inner_push   // line 14
        if (top == 0) pop_wait.signal();  // signal if was empty                   // line 15
    }                                                                              // line 16
    Element *pop()                                                                 // line 17
    {   Element *return_val;                                                       // line 18
        ACE_Guard<ACE_Thread_Mutex> monitor (lock);                                // line 19
        while (top == -1) pop_wait.wait();                                         // line 20
        return_val = inner_pop();                     //  <----- call inner_pop    // line 21
        if (top == S_SIZE-2) push_wait.signal();  // signal if was full            // line 22
        return return_val;                                                         // line 23
    }                                                                              // line 24
  private:                                                                         // line 25
    // inner methods declarations (implementations will be provided on composition)  // line 26
    void inner_push(Element *element);                                             // line 27
    Element *inner_pop();                                                          // line 28
                                                                                   // line 29
    int top;                                                                       // line 30
                                                                                   // line 31
    // synchronization variables                                                   // line 32
    ACE_Thread_Mutex lock;                                                         // line 33
    ACE_Condition_Thread_Mutex push_wait;                                          // line 34
    ACE_Condition_Thread_Mutex pop_wait;                                           // line 35
};                                                                                 // line 36
} // namespace sync                                                                // line 37
```

Figure 115 *Implementation of* sync::Stack

sync::Stack is a "vanilla" C++ class which is very similar to Sync_Stack_Wrapper in Figure 109. The differences between sync::Stack and Sync_Stack_Wrapper are the following:

- sync::Stack has no superclass;
- sync::Stack has the two additional method declarations inner_push() (line 27) and inner_pop() (line 28); the implementations for these methods will be provided upon composition;
- push() of sync::Stack calls inner_push() (line 14) and pop() of sync::Stack calls inner_pop() (line 21).

Composition in SOP is achieved by writing composition rules which specify the correspondence of the subjects (i.e. namespaces), classes, and members to be composed and how to combine them (see Section 7.2.1). Before we show the subject composition rules for composing the namespace original with the namespace sync, we will first describe the composition in plain English:

1. original and sync should be composed into the new namespace composed by merging their members, e.g. classes, operations, attributes, etc. and making sure that if two members have the same name, they correspond. In particular, original::Stack and sync::Stack correspond and should be merged into composed::Stack. Furthermore, when merging corresponding attributes, e.g. original::Stack::top and sync::Stack::top, only one copy should be included in the composite class, i.e. composed::Stack::top.

2. The "by name" correspondence implied by the previous requirement should be overridden for original::Stack::push() and sync::Stack::push() and for original::Stack::pop() and sync::Stack::pop(). Instead, we want sync::Stack::inner_push() and original::Stack::push() to correspond and be combined into combined::Stack::inner_push(). The combined operation should use the implementation of original::Stack::push(). We want the same to happen for sync::Stack::inner_pop() and original::Stack::pop(). Effectively, the push() and pop() of the synchronizer class will call push() and pop() of the original Stack class as their inner methods (see next requirement). This has the effect of wrapping the original push() and pop() into synchronization code.

3. If a method is called inside one of the classes to be composed on *self*, we want *self* in the resulting composite class to refer to the composite class. In particular, the call to inner_push in sync::Stack::push, once promoted to composed::Stack::push, should call the new inner method, i.e. composed::Stack::inner_push. The same applies to the call to inner_pop in sync::Stack::pop.

4. The constructors of both classes to be composed should also be composed. This requirement is taken care of automatically by the SOP system.

These composition requirements are satisfied by the composition rules shown in Figure 116.

```
// stack.rul

// requirements #1, #3, #4
ByNameMerge(composed, (original, sync))

// requirement #2
Equate(operation composed.inner_push, (sync.inner_push, original.push))
Equate(operation composed.inner_pop, (sync.inner_pop, original.pop))
```

Figure 116 *Rules for composing the* original *and* extension *namespaces*[100]

The *ByNameMerge rule* in Figure 116 is a correspondence-and-combination rule, which implements the requirements 1, 3, and 4. It implies *by name* correspondence and *merge* combination of the subjects original and sync. It also specifies that the name of the resulting subject is composed. Please note that the first parameter of this rule indicates the resulting name and the second parameter lists the names to be composed (lists are enclosed into parenthesis). The next two *equate rules* are correspondence rules. They implement requirement 2. Since the scope of these two rules is single methods and such scope is smaller than the scope of the merge rule (which was defined over subjects), the equate rules have precedence over the ByNameMerge rule.

It is important to note that once we compile some_client.cpp, stack.h, sync_stack_extension.h, stack.rul, the client code in some_client.cpp will automatically reference composed::Stack at any place it explicitly references original::Stack. Thus the SOP composition is noninvasive with respect to both Stack and the client code.

This example illustrated two kinds of composition rules available in SOP. There are more rules allowing a fine-grained control over the composition of subjects.

The SOP composition rules work on *labels* of subjects rather than their sources. A label contains all the symbolic information about a subject, such as the names of the classes, methods, and attributes contained in the subject. As a consequence, we can actually compose compiled subjects (in SOP, each subjects can be compiled separately). In other words, SOP supports the composition of binary components.

Variability-Oriented Programming (VOP) [Mez97a, Mez97b] is another composition approach, which we could have used to implement our synchronization example. The major difference between VOP and SOP is that VOP supports both static and dynamic composition of classes and class fragments.[101] Using dynamic fragment composition, we can simplify the implementation of many of the design patterns described in [GHJV95]. For example, the implementation of the visitor pattern given in [Mez97a] requires only a small fraction of the methods in the original implementation in [GHJV95]. We will discuss the advantages of fragment composition over the conventional design-pattern-based solutions in Section 7.4.6.

7.4.6 Some Problems of Implementing Design Patterns and Some Solutions

Design patterns described in [GHJV95], but also in [Cop92, Pre95, BMR+96], represent a tremendous advance towards more reusable software. When we discuss the design patterns collected in [GHJV95], we need at least to distinguish between two important contributions of design patterns:

- design patterns as a form of documenting recurring problems and solutions in OO designs and
- the concrete solutions proposed by specific design patterns.

In our discussion, we will focus on the second contribution and specifically on the concrete OO solutions proposed in [GHJV95].

Design Pattern	Aspect(s) That Can Vary
Adapter	interface to an object
Bridge	implementation of an object
Mediator	how and which objects interact with each other
State	states of an object
Strategy	an algorithm

Table 11 *Design aspects that design patterns let you vary (excerpt from Table 1.2 in [GHJV95])*

Design patterns collected in [GHJV95] help us to decouple and encapsulate various aspects of a software system using concrete implementation idioms based on the two object-oriented composition mechanisms: *inheritance* and *object composition* (by object composition we mean composition by referencing objects and by containing objects, which, of course, covers association and aggregation). Table 11 lists the aspects that some of the design patterns allow us to encapsulate and vary.

Unfortunately, implementation idioms based exclusively on inheritance and object composition also introduce more complexity than necessary. For example, in order to be able to vary some algorithm, we can apply the strategy pattern which involves factoring out the algorithm and encapsulating it in an extra strategy object. But often the algorithm is really part of the original object and by encapsulating it into the separate strategy object we add extra complexity of

- handling an additional strategy object and
- having to live with a level of indirection between the original object and the strategy object.

Similarly, when we decorate an object, we really want just to add or override some of its members. But when we apply the decorator pattern based on object composition, we also end up with two objects and an extra indirection level. A similar observation applies to adapter, state, bridge and other patterns.

Several researchers identified a number of problems with the implementation of design patterns in conventional OO languages (see e.g. [Sou95, SPL96, Mez97a, Bos98, SOPD]). We group these problems into three major problem categories: object schizophrenia, the preplanning problem, and the traceability problem.

7.4.6.1 Object Schizophrenia

The problem of the split of what intentionally supposed to be a single object is referred to as *object schizophrenia* [SOPD].

Broken delegation (i.e. the self problem)

One subproblem of object schizophrenia is *broken delegation* [SOPD] (also referred to as the *self problem* [Lie86a]; cf. Section 7.2.2). For example, if an algorithm is a part of an object and it needs to send a message to the object it is part of, it simply sends this message to *self*. However, if we factor out the algorithm as a strategy object, we need to send the message to the original object. We can no longer think of the algorithm simply as a part of the original object. Instead, we have the extra complexity of indirection, e.g. we need some extra arrangements to provide the algorithm with the data it needs from the original object.

Another problem results from the fact that we have to manage two object identities instead of one. This also causes extra complexity and increases the chance of programming errors, e.g. when the component “escapes” its wrapper (i.e. when we accidentally give away a reference to the wrapped object).

Fragment-based composition, as supported in SOP or VOP, allows us to avoid these problems by modeling extensions such as strategies and wrappers as true class fragments of the extended class.[102] In other words, we do not have to deal with two object identities or an indirection level and we can code the fragments as if they were simply parts of the resulting composite objects.

7.4.6.2 Preplanning Problem

Another problem of design patterns is that they allow for adaptability, but only if the need for certain adaptability was anticipated and the appropriate design patterns were applied in the design phase. This is referred to as the *preplanning problem* [SOPD].

As we have seen in previous sections, there are many situations where fragment-based composition of SOP allows us to noninvasively adapt a component, even if the adaptation has not been anticipated at the design time of the component. We will also show an implementation of a noninvasive composition mechanism in Smalltalk in Section 7.4.7.

7.4.6.3 Traceability Problem

Yet another problem with many design patterns and idioms[103] is their *indirect* representation in the source code. We refer to this problem as the *traceability problem* [Bos98] (also *indirection problem* in [SOPD]). As we already discussed in Section 6.4.3.1.1, given a concrete implementation, it is usually unclear which design patterns were actually applied. The code implementing a pattern is intertwined with other aspects and scattered over a number of components. As a consequence, it is difficult to impossible to distinguish which parts of the resulting program were contributed by which design pattern. Additionally, design patterns increase the fragmentation of the design by introducing many extra little methods and classes. This indirect representation of design patterns causes severe problems for maintenance, reuse, and evolution since the design information is lost.

New language features and composition mechanisms will allow a more direct representation of design patterns. For example, the Demeter approach replaces the conventional OO implementation idiom of the visitor pattern given in [GHJV95] with a more direct representation, which, as we discussed in Section 7.2.3, allows us to avoid some problems of the first one. Proposals of other language constructs for the direct representation of design patterns can be found in [SPL96, Mez98a, Bos98].

We can say that patterns and idioms represent just one phase in the evolution of abstractions, which, eventually, may become language features. Once some design patterns and idioms become features of programming languages, they loose much of their substance as patterns. We no longer think of them as design patterns or idioms in the conventional sense, just as we do not regard inheritance as a design pattern. On the other hand, the documentation format of design patterns can be still useful for documenting the applicability of language features for solving specific problems.[104]

7.4.7 Implementing Noninvasive, Dynamic Composition in Smalltalk

Noninvasive, dynamic composition based on intercepting messages can be easily implemented using the reflection mechanisms of a reflective language such as Smalltalk [GR83]. By reflective facilities we mean explicit representations of some language features, e.g. metaobjects representing classes, methods, and method execution contexts, which are part of the language itself and can be manipulated by the very same language features to affect their own semantics. In Smalltalk, these metaobjects exist at the runtime of an application. As an example of a noninvasive, dynamic composition mechanism, we will show a Smalltalk implementation of an extremely simplified version of the CF model (see Section 7.2.2). Later, in Section 7.5.2, we will use this mechanism to compose synchronization objects and functionality objects at runtime.

7.4.7.1 Model of the Composition

Listener objects

The model of the composition mechanism is shown in Figure 117. The idea is to be able to attach *listener objects* to a *base object* at runtime. Both listeners and base objects are simply instances of some arbitrary classes. We say that a listener is attached to a base object *at* a certain message *messageA* meaning that when we send *messageA* to the base object, the message is redirected to the listener.

By drawing the interface layer in Figure 117 around the base object, we indicate that the dispatch filters, which forward some messages to the listeners, and the references to the listeners are added after the base object has been created. However, please note that the interface layer should be transparent with respect to both the environment and the base object. In fact, we regard the interface layer as a true part of the base object. Thus, it is not appropriate to implement it as a wrapper based on object composition.

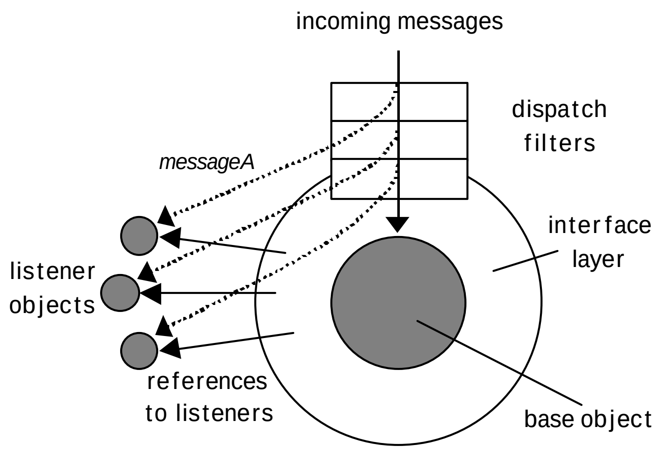


Figure 117 *Model of a simple composition mechanism based on message interception*

In order to implement this model, all we need is to be able to

- dynamically override methods on a per-instance basis, i.e. in a single instance rather than the whole class; given this ability, we can implement each dispatch filter as a method which forwards to the appropriate listener;

- dynamically add instance variables on a per-instance basis; this ability allows us to add references pointing at new listeners.

In summary, we can attach an object *B* to another object *A* at the message *messageA* by

- adding an instance variable to object *A* pointing at *B* and
- override the method *messageA* in object *A* so that *messageA* is forwarded to *B*.

7.4.7.2 Composition API[105]

Given two arbitrary objects referenced by the variables object2 and object1, we can attach object2 to object1 at message messageA as follows:

```
object1 attach: object2 at: #messageA.
```

Once we have attached an object, we can also detach it as follows:

```
object1 detachAt: #messageA.
```

In the attached object (e.g. object2), we can access the base object (i.e. object1) as follows:

```
self baseObject.
```

Once a message has been rerouted to the attached object, we can still call the original method in the base object from the new method in the attached object as follows (please note that this code remains the same even if the original message has some parameters):

```
self baseCall.
```

For example, we could wrap messageA in object1 in some actions by implementing messageA in object2 as follows:

```
messageA
   self someBeforeAction.
   self baseCall.
   self someAfterAction.
```

and attaching object2 to object1 at messageA:

```
object1 attach: object2 at: #messageA.
```

The complete code implementing the composition mechanism is shown in Sections 7.8 and 7.9.

Please note that we can compose any objects without having to modify the code of any of their classes or clients. Also, we did not have to provide any special "hooks" in any of the objects being composed. Thus, the composition is noninvasive and it also avoids the preplanning problem.

7.4.7.3 Instance-Specific Extension Protocol

As stated, the implementation of the composition mechanism utilizes reflection facilities of Smalltalk. These reflection facilities include metaobjects representing classes, methods, and method execution contexts and the Smalltalk compiler itself. We also use two important methods providing access to the implementation of Smalltalk, namely class: and become:. class: allows us to modify the class of an instance and become: replaces every reference to the receiver by the reference to the argument of this message.[106]

Metaobject protocols

Metaobjects provide representations for elements of a language. The interface defined by the metaobjects of a language is referred to as a *metaobject protocol* (*MOP*) [KRB91]. By allowing a write access to metaobjects, as in Smalltalk, we can modify the default semantics of the language. Indeed, in [KRB91], this has been recognized as one fundamental approach to language design. Instead of designing a fixed language, we can design a default language and a MOP which allows the user to customize the default language according to his needs. In other words, a reflective language such as Smalltalk, provides a substantial part of its own implementation as a library of metaobjects to be extended and/or modified by the programmer.

Our implementation of the composition mechanism utilizes a very simple protocol implemented on top of the Smalltalk MOP. Specifically, the protocol we need consists of two methods: one for adding/overriding methods to/in instances and another one for adding instance variables to instances. We refer to this simple protocol as the *instance-specific extension protocol*. We will discuss its implementation in the following two sections. The code of the Smalltalk\VisualWorks implementation of this protocol is given in Section 7.8.[107]

7.4.7.3.1 Defining Methods in Instances

Instance-specific behavior

We use the Smalltalk idiom *instance-specific behavior* in order to define a method in a particular instance without affecting the code of its class. This idiom was described by Beck in [Bec93a].[108] The following sample code shows how to define a method in an instance using Beck's implementation:

```
anObjectA
  specialize:
    'whoAreYou
      ^self printString'
```

The method specialize: inserts a new class between anObjectA and its class, i.e. ObjectA as shown in Figure 118. This is possible since we can use the method class: to modify the class of an instance. The new class is an instance of Behavior, which is a metaclass defining some properties for all classes. For example, Behavior defines the variable methodDict which stores a method dictionary (see Figure 119). A method dictionary is a table which stores the methods of a class. The methodDict variable is present in all classes in Smalltalk since classes are instances of metaclasses and the latter are subclasses of Behavior. We use an instance of Behavior instead of Class since Behavior incurs much less overhead. For example, aBehavior does not even have a name. Of course, we set the superclass of aBehavior to ObjectA.

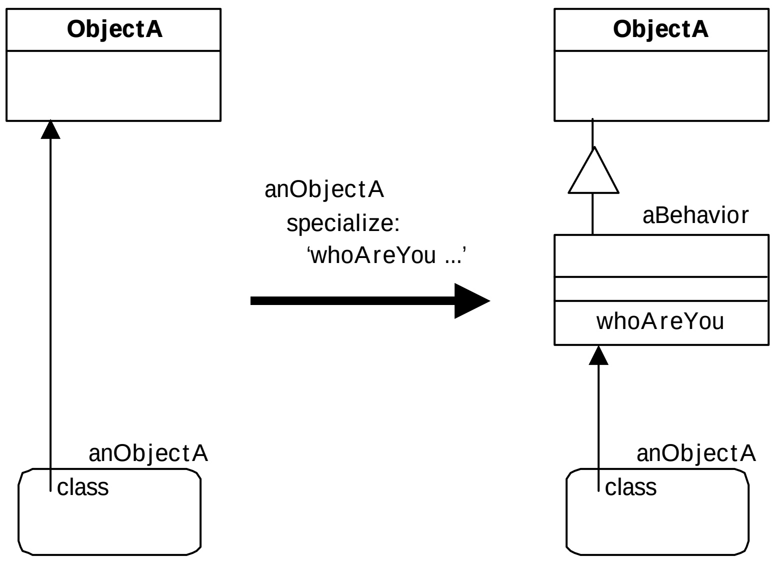


Figure 118 *Adding a method to an instance*

Once we made aBehavior the class of anObjectA, the string provided as a parameter to specialize: is treated as the source code of a new method. The string is compiled and the resulting method is inserted in the method dictionary of aBehavior.[109] In the case that the original class ObjectA already had an implementation of whoAreYou, this original implementation can still be accessed through super.

```
Object ()
  Behavior ('superclass' 'methodDict' 'format' 'subclasses')
    ClassDescription ('instanceVariables' 'organization')
      Class ('name' 'classPool' 'sharedPools')
        ... all the Metaclasses ...
      Metaclass ('thisClass')
```

Figure 119 *Metaclass hierarchy in Smalltalk/VisualWorks. Subclasses are indented. Instance variables defined by the corresponding class are shown in parenthesis.*

Finally, the standard method class is overridden in anObjectA to return its original class ObjectA. This way the new class aBehavior is not visible to the environment. We can say that the new class is a private class of anObjectA.

The implementation of instance-specific behavior makes use of the following reflective facilities in Smalltalk:

- Behavior and inserting a method into its method dictionary;
- changing the class of an instance using class:;
- compiling a method from a string using Compiler and the explicit handling of methods as instances of CompiledMethod.

7.4.7.3.2 Adding Instance Variables to Instances

The idiom *adding instance variables to instances* was originally described in [Cza96]. The following sample code demonstrates the use of this idiom:

```
“declare a temporary variable”
| anObjectA |
“initialize the variable with an instance of ObjectA”
anObjectA := ObjectA new.
“add the new instance variable someNewInstVar to anObject”
anObjectA addInstanceVariable: ‘someNewInstVar’.
“set the value of someNewInstVar’ to 1”
anObjectA someNewInstVar: 1.
“return the value of someNewInstVar’
^anObjectA someNewInstVar
```

The method addInstanceVariable: adds an instance variable to an instance in four steps:

1. Insert aBehavior between the receiver and its class (just as we did it for adding a method; see Figure 118).
2. Modify the object format contained in aBehavior (see the instance variable format in Figure 119), so that the number of instance variables encoded in the format is increased by one. The object format stored in a class encodes, among others, the number of instance variables of each of its instances.
3. Mutate the receiver, so that it contains the number of instance variables encoded in format. This is achieved by creating an new instance of aBehavior, copying the contents of the receiver into the new instance, and replacing the receiver with the new instance using become:.
4. Insert the accessing methods for the new instance in the method dictionary of aBehavior.

Here is the implementation of addInstanceVariable: (implemented in Object):

```
addInstanceVariable: aString
   "Add an instance variable and accessing
   methods based on the name aString"
   self specialize.
   self incrementNumberOfInstVarsInMyClassFormat.
   self mutateSelfToReflectNewClassFormat.
   self addAccessingMethodsForLastInstVarUsing: aString.
```

The full implementation code is included in Section 7.8.

In order to implement adding instance variables to instances, we have used the following reflective facilities of Smalltalk:

- modifying the object format in Behavior
- modifying the identity of an object using become:;
- all the reflective facilities we also used for instance-specific behavior.

7.4.8 Kinds of Crosscutting

The stack example in Figure 108 demonstrated the crosscutting of synchronization code and methods of one class. We also saw that method wrapping or before and after methods allow us to separate the synchronization aspect from the stack.

But what other kinds of crosscutting are common in object-oriented programs?

Class level vs. instance level crosscutting

First, we distinguish between crosscutting at the class level and crosscutting at the instance level. If some aspect code crosscuts classes, it means that crosscutting the instances of the class is controlled at the class level. For example, the synchronization code crosscuts the stack at the class level and, in our case, it crosscuts all stack instances in the same way. Class level crosscutting requires composition mechanisms allowing us to associate aspect code with classes. Furthermore, at the class level, we also distinguish between aspect state shared among all instances and per-instance aspect state.

Shared and per-instance aspect state

Instance-level crosscutting means that we have to be able to associate aspect code with individual instances. For example, different instances of a data structure could be associated with different aspect codes implementing different synchronization strategies. We saw an example of instance-level composition mechanism in the previous section.

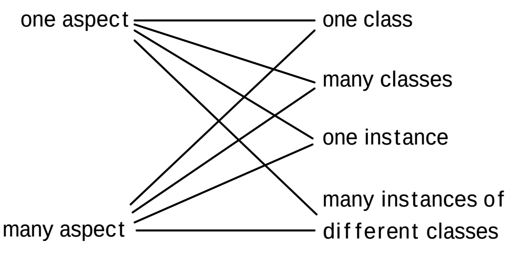


Figure 120 *Different kinds of crosscutting with respect to classes and instances*[110]

Kinds of message join points

Different kinds of crosscutting with respect to classes and instances are summarized in Figure 120. Each combination of one item on the left and one item on the right indicates a different kind of crosscutting.

Message join points are quite common in OO programming. Examples of aspects amenable to message join points are concurrency, real-time constraints, atomic transactions, precondition-like error handling, security, profiling, monitoring, testing, caching, and historization. These kinds of aspects can be usually implemented as objects and coupled to the primary structure objects using some kind of message coupling mechanism such as method wrapping or message interception. Indeed, we also have different possibilities at the method level: We can add a before action, an after action or both to a method implementation or to a method call. The latter actually allows us to add actions *inside* a method implementation by instrumenting calls to other methods inside it. In general, we can distinguish between instrumenting the definition and the use of some intention, e.g. instrumenting method definition vs. method calls. In addition to methods, we can also instrument attributes of objects. This is useful if we want to monitor read or write access to attributes.[111] Future versions of AspectJ, which we discuss in Section 7.5.1, will augment Java with composition mechanisms for addressing the different kinds of crosscutting common in OO programs.

Domain-specific optimizations

There are also cases of more complex crosscutting in practice. For example, domain-specific optimizations in vector and matrix code involve crosscutting at the statement level. The usual example for vector code is loop fusing and temporary variable elimination. In this case, the optimizations crosscut vector expression and statement structures. We will see examples of such optimizations in Section 10.2.6.1. An even more complex kind of crosscutting is common to cache-based optimizations in matrix code. This kind of optimizations requires very complex transformation. Indeed, due to the complexity of such transformations, the quality of the code generated by such optimizers is far beyond the quality of manually optimized code. In general, domain-specific optimizations in scientific computing require complex transformations on parse trees.

7.5 How to Express Aspects in Programming Languages

In the previous sections, we have seen that appropriate composition mechanisms (e.g. fragment-based composition in SOP or traversal strategies in Demeter) allow us to reduce the complexity of conventional OO programs. But reducing complexity by untangling aspects and relating them by an appropriate composition mechanisms is just the first step. We can further reduce complexity by representing the aspects themselves using appropriate linguistic means.

Whatever solution we find, our goal is enable a one-to-one encoding of requirements in the language we use. For example, the synchronization of a stack involves four synchronization constraints (see Section 7.4.3). Ideally, we want to be able to express these constraints in the programming language with four corresponding statements.

What different options do we have for capturing aspects? There are three possibilities:

- Encode the aspect support as a conventional library (e.g. class or procedure library). We will see an example of this strategy in Section 7.5.2. That section discusses the implementation of a very high level synchronization library that developers writing aspect program can use. The library uses our dynamic composition mechanism for attaching objects from Section 7.4.7 in order to address crosscutting.

- Design a separate language for the aspect. We will see an example of a separate synchronization language in Section 7.5.1.

- Design a language extension for the aspect. We will discuss this option in Section 7.6.1.

The implementation technologies and the pros and cons for each of these approaches are discussed in Section 7.6.

The following two section will demonstrate the separate language approach and the class library approach to aspects.

7.5.1 Separating Synchronization Using AspectJ Cool

Cool

As you remember, Section 7.4.5 showed how to separate the synchronization aspect of a stack using SOP. While the separation was quite satisfying (e.g. it allows an noninvasive replacement of the synchronization aspect with respect to both the stack implementation and stack client code), the representation of the synchronization aspect in Figure 115 was far from being intentional. In this section, we show a more intentional solution written in *Cool*, an aspect language for expressing synchronization in concurrent OO programs.

AspectJ

Cool was originally designed by Lopes in her dissertation [Lop97] and implemented as a part of the AspectJ[112] environment developed at Xerox Palo Alto Research Center [XER98a, XER98b, AJ].

AspectJ is an extension to Java that supports aspect-oriented programming. AspectJ version 0.2.0 includes general-purpose cross-cutting mechanisms that can be used to capture a range of cross-cutting modularities (see Section 7.4.8), including but not limited to synchronization and distribution control. The support for specific aspects is implemented in the form of class libraries, much in the style we will see in Section 7.5.2.

Ridl

At the time of preparing the examples for this chapter, the strategy of AspectJ (version 0.1.0) used to be to provide a separate aspect language for each aspect it addressed. The provided languages were Cool (i.e. the synchronization language) and Ridl. Ridl is an aspect language for expressing remote invocation and controlling the depth of parameter transfer (see [Lop97]). Both languages were implemented by a preprocessor which took Java source files and the aspect source files written in these languages and generated pure Java.

The example in this section is implemented using this older version 0.1.0 of AspectJ rather than the more general AspectJ 0.2.0. The older version allows us to demonstrate the use of specialized aspect languages for expressing aspects. The other approach involving class libraries and message-join-point composition mechanisms (as used by AspectJ 0.2.0) is demonstrated later in Section 7.5.2.

Before we show you the more intentional encoding of the stack synchronization aspect using AspectJ Cool, let us first re-state the synchronization constraints on the stack we came up with in Section 7.4.3:

1. `push` is self exclusive;

2. `pop` is self exclusive;

3. `push` and `pop` are mutually exclusive;

4. `push` can only proceed if the stack is not full;

5. `pop` can only proceed if the stack is not empty.

Coordinators

The stack is full if `top==stack_size-1` and is empty `if top==-1`. All these statements can be represented directly in Cool as a *coordinator* (i.e. a synchronizing agent). Figure 122 shows the Java implementation of the stack and the Cool implementation of the stack coordinator (for completeness, we also include the equivalent, “tangled” implementation of the synchronized stack in pure Java in Figure 121). We numbered the lines of the stack coordinator, so that we can exactly describe how the above-listed synchronization requirements are reflected in the coordinator code.

```
public class Stack
{    private int max_top;
     private int under_max_top;

     public Stack(int size)
     {    elements = new Object[size];
          top = -1;
          max_top = size-1;
          under_max_top = max_top-1;
     }
     public synchronized void push(Object element)
     {    while (top == max_top)
          {    try
               {    wait();
               }
               catch (InterruptedException e) {};
          }
          elements[++top] = element;
          if (top==0) notifyAll(); // signal if was empty
     }
     public synchronized Object pop()
     {    while (top==-1)
          {    try
               {    wait();
               }
               catch (InterruptedException e) {};
          }
          Object return_val = elements[top--];
          if (top==under_max_top) notifyAll(); // signal if was full
          return return_val;
     }

     private int top;
     private Object [] elements;
}
```

Figure 121 *"Tangled" implementation of a synchronized stack in Java*

On-exit and on-entry blocks

The name of the coordinator appearing on line 2 in Figure 122 is the same as the name of the Java class it synchronizes, i.e. Stack. Line 3 states that push and pop are self exclusive (i.e. requirement 1 and 2).[113] Line 4 states that push and pop are mutually exclusive (i.e. requirement 3). The states of the stack are modeled by the condition variables full and empty declared on line 5. Lines 7 and 8 say that push can only proceed if the stack is not full (i.e. requirement 4). Line 9 starts the definition of the so-called *on-exit block* of code that is to be executed just after the execution of push of the class Stack.[114] The on-exit block of push (lines 9-12) and the on-exit block of pop (lines 15-18) define the semantics of the states full and empty. Please note that these blocks access the stack variables s_size and top. In general, a read-only access of the variables of the coordinated object is possible, but we are not allowed to modify them. Finally, line 14 states that pop can only proceed if the stack is not empty (i.e. requirement 5).

This simple stack example demonstrates that the stack synchronization constraints can be quite directly translated into Cool. Thus, the Cool representation of the synchronization aspect is clearly more intentional than the subclass version in Figure 109 or the SOP version in Figure 115.

```
//in a separate JCore file Stack.jcore[115]
public class Stack
{    private int s_size;

     public Stack(int size)
     {    elements = new Object[size];
          top = -1;
          s_size = size;
     }
     public void push(Object element)
     {    elements[++top] = element;
     }
     public Object pop()
     {    return elements[top--];
     }

     private int top;
     private Object [] elements;
}
```

```
//in a separate Cool file Stacksync.cool                // line  1
coordinator Stack                                       // line  2
{    selfex push, pop;                                  // line  3
     mutex {push, pop};                                 // line  4
     condition full=false, empty=true;                  // line  5
                                                        // line  6
     guard push:                                        // line  7
         requires !full;                                // line  8
         onexit                                         // line  9
         {     if (empty) empty=false;                  // line 10
               if (top==s_size-1) full=true;            // line 11
         }                                              // line 12
     guard pop:                                         // line 13
         requires !empty;                               // line 14
         onexit                                         // line 15
         {     if (full) full=false;                    // line 16
               if (top==-1) empty=true;                 // line 17
         }                                              // line 18
}                                                       // line 19
```

Figure 122 *Java implementation of a stack and Cool implementation of a stack coordinator*[116]

The stack coordinator in Figure 122 is an example of a per-instance coordinator, i.e. each instance of Stack will have its own coordinator instance. In addition to the per-instance coordinators, Cool supports per-class coordinators, which can also be shared among a number of classes. The details of the Cool language are given in [XER98b, Lop97].

7.5.2 Implementing Dynamic Cool in Smalltalk

We can also implement Cool-like synchronization using a class library and the dynamic composition mechanism described in Section 7.4.7. Thus, composition based on reflective facilities represents an alternative to static composition using a preprocessor (or a compiler). In contrast to a preprocessor, dynamic composition allows us to connect and reconnect coordinators to individual objects at runtime. A prototype implementation of a dynamic version of Cool in Smalltalk is available at [DCOOL]. We will describe the overall architecture of this implementation in Section 7.5.2.2. But first, we demonstrate its application with an example.

7.5.2.1 Example: Synchronizing an Assembly System

The example involves the synchronization of an assembly system simulating the production of candies (the example was adapted from [XER97]). An assembly system consists of a number of workers, where each worker works independently in a separate thread and consumes and produces some materials (cf. [Lea97]).

Our sample assembly system is shown in Figure 123. There are two candy makers (instances of CandyMaker), which make candies and pass them to the packer (an instance of Packer) by sending it the message newCandy: with a new candy as a parameter. The packer collects a certain number of candies, packages them into a pack, and passes the new pack to a finalizer (an instance of Finalizer) by sending it the message newPack:. The finalizer also receives labels from a label maker (an instance of LabelMaker). Once the finalizer has a pack and a label, it glues the label onto the pack and ships the finished pack.

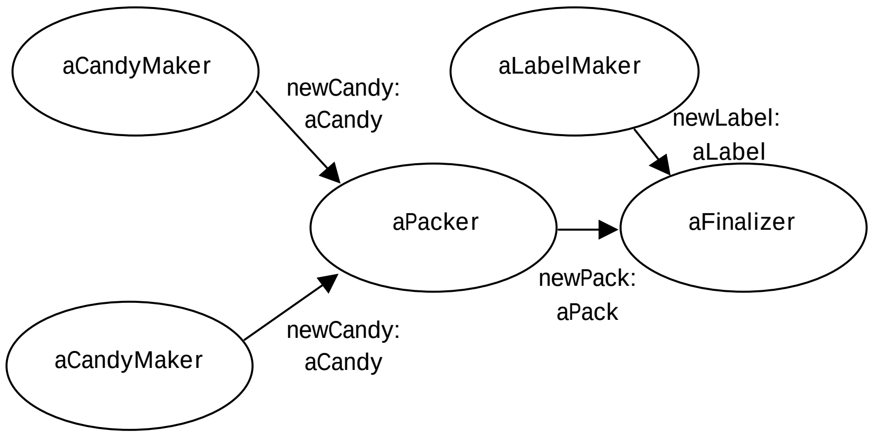


Figure 123 *Concurrent assembly system simulating the production of candy*

As stated before, each of the workers works independently in a separate thread. However, the workers have to synchronize their work at certain points. We have the following synchronization constraints:

1. The method newCandy: of Packer has to be self exclusive since we have more than one candy maker calling this method concurrently;

2. The method newCandy: can only be executed if the pack in the packer is not full, otherwise the thread sending newCandy: is suspended (the packer needs to start a new pack before the thread can be resumed).

3. The packer has to close a full pack by sending itself the message processPack. Thus, processPack requires that the current pack is full.

4. newPack: is sent to the finalizer after finishing the execution of processPack. However, newPack: can only be executed if the finalizer does not currently have a pack, otherwise the thread sending newPack: is suspended. When newPack: exits, the packer starts with a new, empty pack.

5. newLabel: can only be executed when the finalizer does not currently have a label, otherwise the thread sending newLabel: is suspended.

6. The finalizer glues a label to a pack by sending glueLabelToPack to itself. However, glueLabelToPack can only be executed if the finalizer has one pack and one label.

7. Finalizer ships a new candy pack by sending itself shipNewCandyPack. This message is sent right after finishing the execution of glueLabelToPack. When shipNewCandyPack exits, the finalizer has neither a pack nor a label.

We will implement these constraints in a coordinator (i.e. a coordination object), which will be connected to some of the workers. Indeed, as we will see later, we only need to attach the coordinator to the packer and the finalizer. In our implementation of *Dynamic Cool* (i.e. a version of Cool which supports dynamic reconfiguration), we connect an object to be synchronized to a *port* of a coordinator. A coordinator has one port for each object it synchronizes. This is illustrated in Figure 124.

Coordinator ports

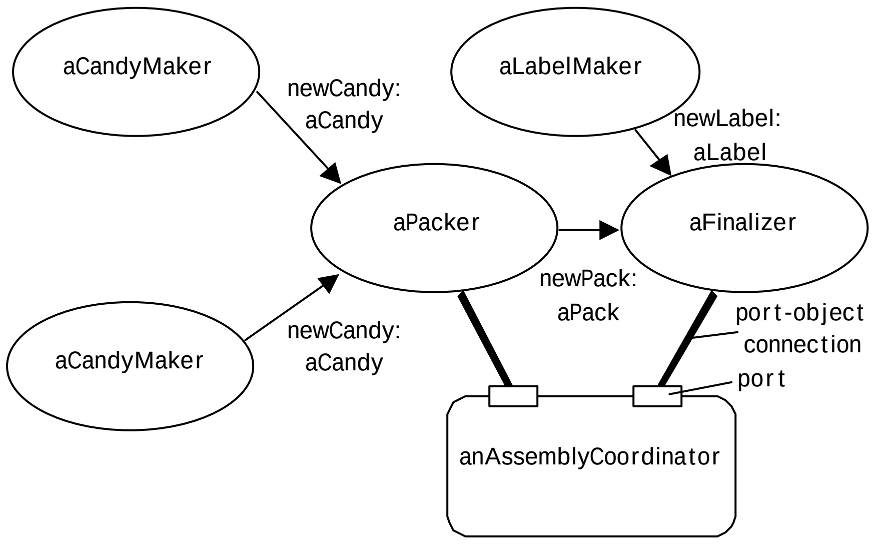


Figure 124 *Candy assembly system synchronized using a coordinator*

Once given the configuration in Figure 123, we achieve the configuration in Figure 124 as follows. First, we define the new class AssemblyCoordinator as a subclass of Coordinator, which is a class provided by the implementation of Dynamic Cool. Then we have to redefine the method defineCoordination in AssemblyCoordinator. This method defines the synchronization constraints. We will show the contents of this method in a moment. AssemblyCoordinator has two ports: the port #packer, for synchronizing the packer, and the port #finalizer, for synchronizing the finalizer. Given anAssemblyCoordinator (i.e. an instance of AssemblyCoordinator), the following code connects aPacker to the #packer port of anAssemblyCoordinator and aFinalizer to the #finalizer port of anAssemblyCoordinator:

```
anAssemblyCoordinator
   register: aPacker at: #packer;
   register: aFinalizer at: #finalizer.
```

The method defineCoordination in AssemblyCoordinator implements the synchronization constraints. The contents of this method is shown in Figure 125.

There is a close correspondence between the code in Figure 125 and the synchronization constraints listed earlier. We indicated which sections of the code implement which constraint in the comments (in Smalltalk, comments are enclosed in double quotation marks). Our goal was to make sure that the notation looks as close as possible to the Cool notation (cf. Figure 122), while its syntax is still valid Smalltalk syntax.

```
defineCoordination

  “initialize condition variables”                                              “line 1”
  packFull := false. “packer has no full pack”                                  “line 2”
  gotPack := false. “finalizer has no pack”                                     “line 3”
  gotLabel := false. “finalizer has no label”                                   “line 4”
                                                                                “line 5”
  self                                                                          “line 6”
                                                                                “line 7”
     “define the two ports: packer and finalizer”                               “line 8”
     addPorts: #( packer finalizer );                                           “line 9”
                                                                                “line 10”
     “requirement #1”                                                           “line 11”
     selfex: #( 'packer.newCandy:' );                                           “line 12”
                                                                                “line 13”
     “requirement #2”                                                           “line 14”
     guardOn: 'packer.newCandy:'                                                “line 15”
         requires: [ :packer | packFull not ]                                   “line 16”
         onExit: [ :packer |                                                    “line 17”
           packer candyPack candyCount = packer maxCandyCount ifTrue: [ packFull := true ]]; “line 18”
                                                                                “line 19”
     “requirement #3”                                                           “line 20”
     guardOn: 'packer.processPack'                                              “line 21”
         requires: [ :packer | packFull ];                                      “line 22”
                                                                                “line 23”
     “requirement #4”                                                           “line 24”
     guardOn: 'finalizer.newPack:'                                              “line 25”
         requires: [ :finalizer | gotPack not ]                                 “line 26”
         onEntry: [ :finalizer | gotPack := true ]                              “line 27”
         onExit: [ :finalizer | packFull := false ];                            “line 28”
                                                                                “line 29”
     “requirement #5”                                                           “line 30”
     guardOn: 'finalizer.newLabel:'                                             “line 31”
         requires: [ :finalizer | gotLabel not ]                                “line 32”
         onEntry: [ :finalizer | gotLabel := true ];                            “line 33”
                                                                                “line 34”
     “requirement #6”                                                           “line 35”
     guardOn: 'finalizer.glueLabelToPack'                                       “line 36”
         requires: [ :finalizer | gotPack & gotLabel ];                         “line 37”
                                                                                “line 38”
     “requirement #7”                                                           “line 39”
     guardOn: 'finalizer.shipNewCandyPack'                                      “line 40”
         onExit: [ :finalizer | gotPack := false. gotLabel := false ].          “line 41”
```

Figure 125 *Implementation of the synchronization constraints in* defineCoordination

The three condition variables initialized in lines 2-4 were declared as instance variables of AssemblyCoordinator. packFull indicates whether the current pack in the packer is full or not. gotPack is true if the finalizer has a pack and otherwise false. gotLabel indicates whether the finalizer currently has a label or not. The two ports #packer and #finalizer are defined on line 9 by sending addPorts: to self (i.e. anAssemblyCoordinator). On line 12, the set of self-exclusive methods is declared. In our case, we only declare newCandy: of the object connected to the port #packer as self exclusive. We refer to this method as follows: 'packer.newCandy:'. This *qualified method name* is an element of an array which is passed as an argument to the message selfex:, which is sent to self.[117] The remaining code defines six guards on six different methods. For example, the guard on 'packer.newCandy:' on line 15 is defined by sending guardOn:requires:onExit: to self. The first argument is the qualified method name, the second argument defines the *requires-condition block*, which is implemented as a Smalltalk block returning true or false, and the third argument defines the *onExit block*, which is also defined as a Smalltalk block. Both blocks receive the object connected to the port on which the guard is being defined, i.e., in our case, the object connected to #port.[118] The onExit block sets packFull to true if the current pack in the packer contains the maximum count of candy. The remaining guards are defined by sending either guardOn:requires: or guardOn:requires:onEntry: or guardOn:requires:onEntrys:onExit: to self.

Requires-condition, onEntry, and onExit blocks

The order of sending the messages, except for addPorts:, is not relevant. addPorts: has to be sent first. Of course, we can also send any of these messages later in order to modify the synchronization constraints at some point in time.

Some of the more interesting features of the Smalltalk implementation of Dynamic Cool include the following

- The synchronization constraints in a coordinator can be modified at any time.
- The coordinators can be connect and reconnect to objects at any time.
- One object can be connected to more than one port of different coordinators (this is shown in Figure 126) and/or of the same coordinator. Also, one coordinator may be used to coordinate a number of instance of different classes at the same time.
- The same method of one object can be synchronized by more than one coordinator (this is also shown in Figure 126);
- One coordinator can be reused for coordinating instances of different classes.
- A number of objects can be synchronized by a group of cooperating coordinators.

Applications of these features include the dynamic adaptation of the synchronization aspect (e.g. due to dynamic load balancing) and the dynamic configuration of components, where each component contributes its own coordinator which has to be composed with other coordinators.

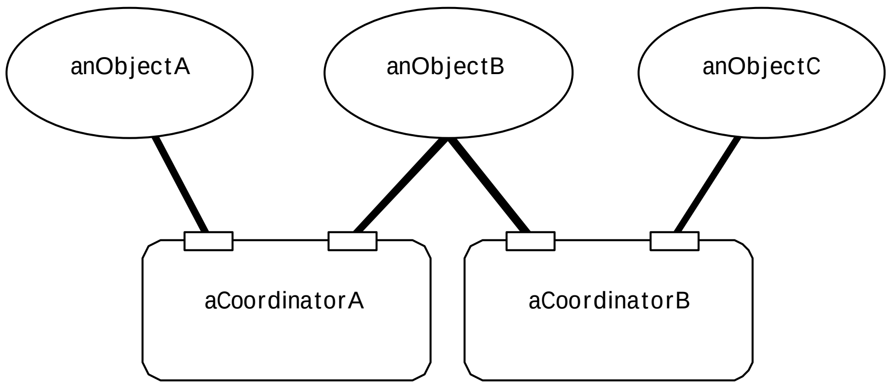


Figure 126 *Example of a configuration containing an object connected to two coordinators*

7.5.2.2 Architecture of the Smalltalk Implementation of Dynamic Cool

The main concepts of the Smalltalk implementation of Dynamic Cool are illustrated in Figure 127. This figure shows two coordinators and an object connected to them.

As stated before, a coordinator has one or more named ports.[119] Each port contains a number of *message coordinators*. There is one message coordinator for each method coordinated at a port. A message coordinator contains all the information required to coordinate one method. In particular, it contains the waiting-condition blocks (including the negated requires-condition block and the condition blocks generated according to the selfex and mutex sets) and the onEntry and onExit blocks.

Message coordinators

One message coordinator is connected to one *synchronization listener*. A synchronization listener is a special kind of a listener in the sense of Section 7.4.7. It is connected to the synchronized object at the message it synchronizes using the mechanism also described in Section 7.4.7. It wraps the corresponding method of the synchronized object into appropriate synchronization code. Specifically, in the before action of this synchronization code, the synchronization listener checks all the waiting-condition blocks of all the message coordinators it is connected to. If any of these blocks returns true, the current thread is suspended. Otherwise, the onEntry blocks of the connected message coordinators are executed. Next, the original method of the synchronized object is executed. Finally, in the after action, the onExit blocks of the connected message coordinators are executed and the waiting threads are signaled to reevaluate their waiting-condition blocks.

Synchronization listeners

As shown in Figure 127, one synchronization listener can be connected to more than one message coordinators (however, the message coordinators have to belong to different ports).

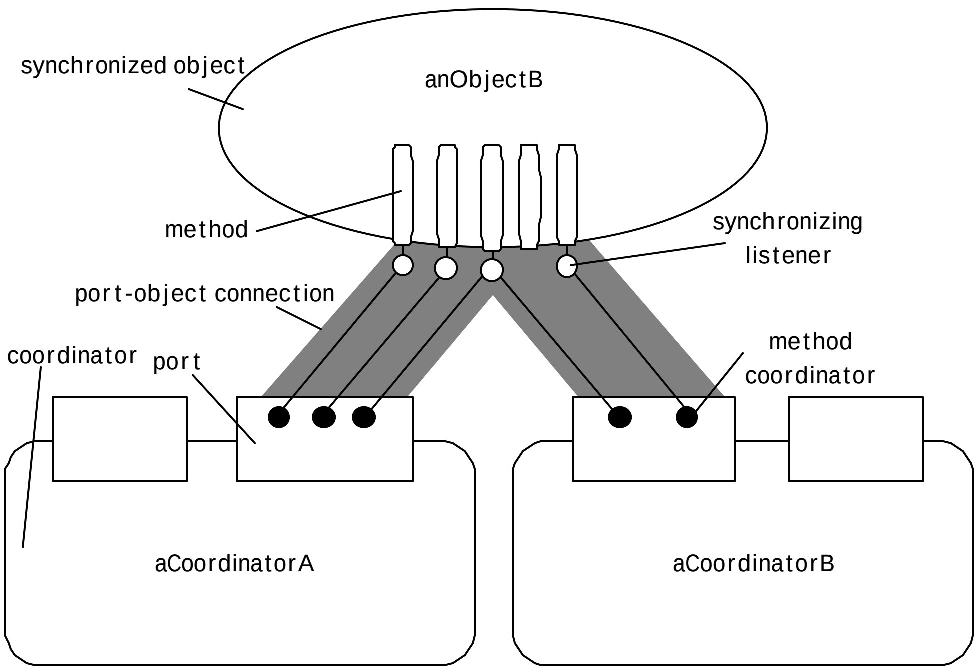


Figure 127 *Main concepts of the Dynamic Cool implementation*

7.6 Implementation Technologies for Aspect-Oriented Programming

Providing support for an aspect involves two things:

- implementing abstractions for expressing the aspect, and
- implementing weaving for composing the aspect code with the primary code and/or the code for other aspects.

We address each of these points in the rest of this chapter.

7.6.1 Technologies for Implementing Aspect-Specific Abstractions

As stated, we have three main approaches to implementing the abstractions for expressing an aspect:

- Encode the aspect support as a conventional library (e.g. class or procedure library). We saw an example of this approach in Section 7.5.2. If we use this approach, we still need to provide an appropriate composition mechanism which addresses the kind of crosscutting the aspect involves.
- Design a separate language for the aspect. This was demonstrated in Section 7.5.1. The language can be implemented by a preprocessor, compiler, or an interpreter. Preprocessors, while often being a simple and economical solution, have a number of problems caused by the lack of communication between different language levels. We discuss these problems in Section 9.4.1.
- Design a language extension for the aspect. By a language extension we mean a modular language extension that we can plug into whatever language (or, more appropriately, configuration of language extensions) we currently use. This solution differs from the previous one in technology rather than at the language level. We can actually use a “separate language” as a language extension. We will just use it in a separate module or in some other scoping mechanism. The main question here is whether the programming platform we use supports pluggable, modular language extensions or it confines us to

using one fixed language or a fixed set of separate languages. An example of an extendible platform is the Intentional Programming system discussed in Section 6.4.3.

Using a separate language or a language extension specialized to the problem at hand has a number of advantages over using the conventional library approach. They are summarized in Table 12. These advantages are usually cited in the context of domain-specific languages (i.e. specialized languages; see e.g. [DK98]).

The conventional library approach does not only have disadvantages. It also has one important advantage: Given the currently available technologies, it is often the only choice. This should change when tools such as the IP system become widely available.

Advantages of modular language extensions

Modular language extensions have a number of advantages over a fixed set of separate languages:

- Language extensions are more scalable. We can plug and unplug extensions as we go. This is particularly useful, when, during the development of a system, we have to address more and more aspects. We can start with a smaller set of aspects and add new ones as we go (just as we add libraries).

- Language extensions allow the reuse of compiler infrastructure and language implementation. We do not have to write a new compiler for each new aspect language. Also, one extension can work with many others. Except for some glue code, most of the implementation of a larger language extension can be reused for different configurations.

Specialized languages or language extensions	**Conventional libraries**
declarative representation Requirements can be directly translated into the language, e.g. synchronization constraints in Cool. Specialized languages can use whatever domain-specific notation is appropriate, e.g. two-dimensional mathematical symbols for matrix code.	less direct representation Implementing requirements in a general programming language often involves obscure idioms and low-level language details. For example, the Smalltalk implementation of the Dynamic Cool uses the Smalltalk block syntax with the the special characters absent in the solution using the Cool language.
simpler analysis and reasoning The language constructs capture the intentions of the programmer directly in the most appropriate form. No complex programming pattern or cliche analysis is needed to recognize domain abstractions since each of them has a separate language construct.	analysis often not practicable General purpose constructs can be combined in a myriad of ways. The intention of why they were combined in this and not the other way is usually lost (or, at best, represented as a comment). We also discussed the related traceability problem of design patterns in Section 7.4.6.3. The analysis problem is being addressed in the automatic program analysis community with relatively small or no success. The whole area of program reengineering is also wrestling with this problem.
force you to capture all of the important design information What otherwise makes a useful comment in a program is actually a part of the language.	design information gets lost See comments above.
allow domain-level error checking Since you provide all the domain knowledge in a form that the compiler can make use of, the compiler can do more powerful error checking based on this knowledge.	domain-level errors cannot be found statically
allow domain-level optimizations Domain-knowledge allows us to provide more powerful optimizations than what is possible at the level of a general purpose language. We demonstrated this fact in Section 6.4.1.	domain-level optimizations hard to achieved (only limited to impossible) Most of the languages currently used in the industry do not provide any support for static metaprogramming. Template metaprogramming in C++ allows us to implement only relatively simple domain-specific optimizations.

Table 12 *Advantages of specialized languages over conventional libraries*

Specialization and abstraction level

We often say that a language is higher level if it allows us to solve problems with less effort. But it is not widely recognized that the “level” of a language has at least two dimensions: *specialization level* and *abstraction level* (see Figure 128).

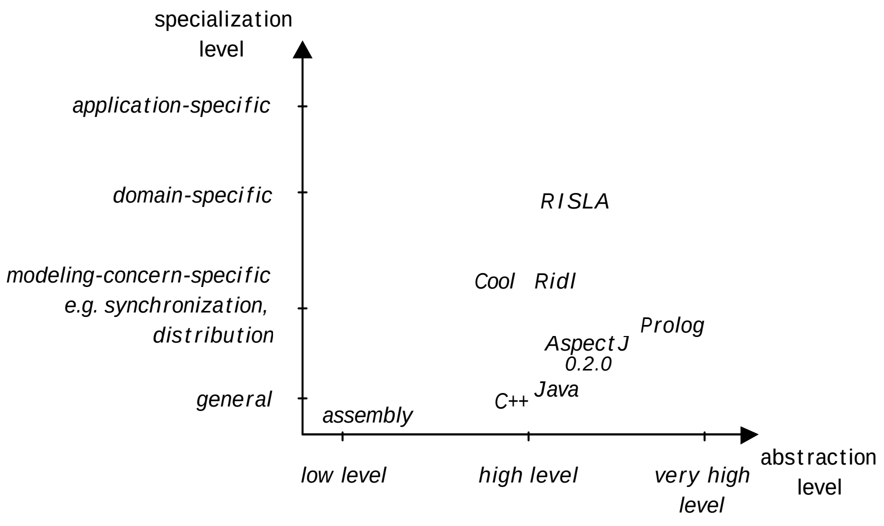


Figure 128 *Classifying languages according to their specialization and abstraction levels*[120]

The horizontal axis indicates the abstraction level of a language. If a language has a higher abstraction level, it means that it does more work for you. In other words, you write less program code, but the code does more.[121] On the language implementation side, higher level implies that the implementation does much more code expansion, the runtime libraries are larger, etc. An assembly language is an example of a very low-level language.

The vertical axis in Figure 128 represents the specialization level of a language. More specialized languages are applicable to smaller classes of problems. We listed the advantages of applying specialized languages in Table 12. Examples of modeling-concern-specific languages are Cool and Ridl (Section 7.5.1). An example of a domain-specific language is RISLA [DK98], which is a language for defining financial products.

7.6.2 Technologies for Implementing Weaving

In addition to the specialization and the abstraction level, aspect languages address a third dimension which is the *level of crosscutting* (see Figure 129). As we will see below, weaving separated, crosscutting code modules involves merging them in a coordinated way. If you use a compiler for the weaving, the crosscutting level tells you how much transformation work the compiler does for you.

Crosscutting level

Conceptually, composing separated aspects involves code transformation. This is illustrated in Figure 130. Instead of writing the low-level tangled code (on the right), we write code with well separated aspects (on the left) and use a transformation in order to obtain the lower-level tangled code (also referred to as the “woven” code). We can use at least two different technologies for implementing such transformations:

- source transformation and
- dynamic reflection.

Both technologies are examples of *metaprogramming*. Metaprogramming involves a domain shift: The metacode is *about* the base code (just as an aspect is about some component(s); see Figure 106). A more implementation-oriented definition of metaprogramming often found in literature characterizes it as “manipulating programs as data”.

Metaprogramming

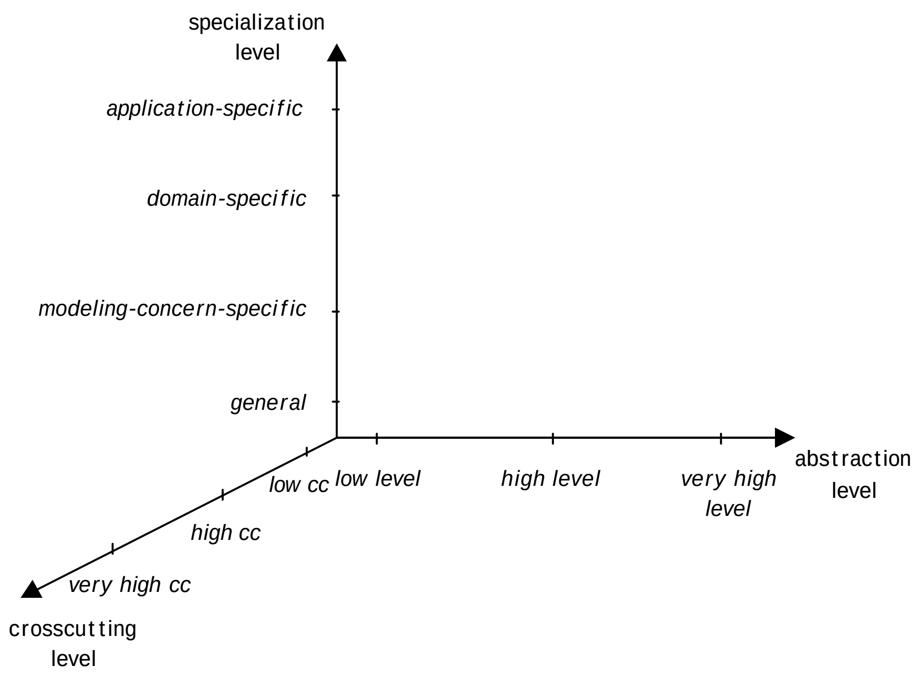


Figure 129 *Crosscutting level as the third dimension of languages*[122]

Source transformation can be implemented in a compiler, a preprocessor, or a transformation system. Compilers and preprocessors usually provide only a low level interface for writing transformations consisting of parse tree node creation and editing operations (we assume that you have access to the implementation of the compiler or the preprocessor). Transformation systems, on the other hand, give you a more convenient interface including pattern matching facilities, Lisp-like quoting facilities, program analysis facilities (e.g. data and control flow analysis), etc. We discussed transformation systems in Chapter 6.

If the woven code need not have to be rearranged at runtime, we can do the program transformation before runtime and generate efficient, statically-bound code.[123] For example, the AspectJ version described in Section 7.5.1 weaves the code before runtime using a transforming preprocessor. Alternatively, we could also provide some generating capability at runtime (e.g. by including some transforming component and/or the compiler in the runtime version), so that we can optimize the code if certain aspect constellations are repeatedly needed.

Dynamic reflection

The other technique for weaving aspects is to use dynamic reflection. Dynamic reflection involves having explicit representations of some elements of the programming language being used (i.e. the *metalevel*) at runtime (see [KRB91] for a stimulating discussion of this topic). As stated before, examples of such representations are metaobjects representing classes, methods, message sends, method-execution contexts, etc. Using such metarepresentations, we can modify the meaning of some language elements at runtime and, this way, arrange for a transparent transfer of control between aspects.[124] We have seen a concrete example of applying these techniques in Sections 7.4.7 and 7.5.2.

Interpreting aspects

Now the question is: how do we achieve the transformation of the separated code into the tangled code shown in Figure 130 using dynamic reflection? Instead of explicitly transforming the high-level code with separated aspects into the tangled code, the high-level code is interpreted at runtime and the control between the aspects is transferred so often that this execution is effectively equivalent to executing the tangled code. Thus, we achieve the desired effect through *interpretation* of the aspect rather than explicitly generating the tangled code. However, this approach has both its advantages and disadvantages.

The positive side to dynamic reflection is the ability to dynamically reconfigure the code. For example, the Smalltalk implementation of Dynamic Cool allows us to reattach coordinators at runtime. The running system could even produce new configuration and components which were not preplanned before runtime. On the other hand, the dynamic transfer of control between aspects incurs certain runtime overhead, e.g. dynamic method calls, dynamic reification (e.g. requesting the object representing the execution context of a method in Smalltalk[125]), etc. If the performance is critical and static configuration is sufficient, static code transformation is certainly the better choice.

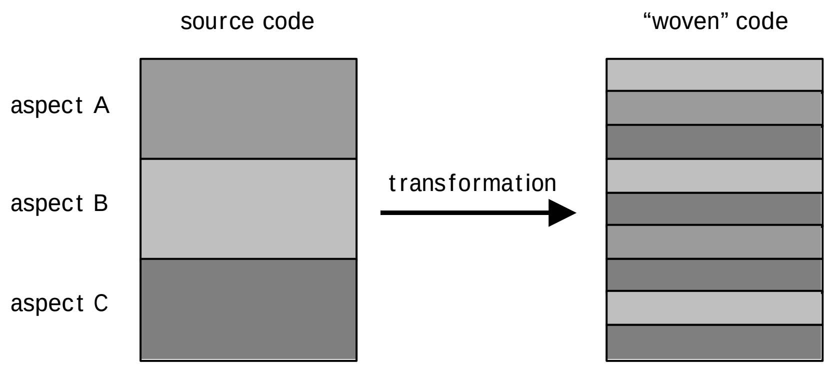


Figure 130 *Composing separated aspects involves transformation*

Separating relevant aspects may often require quite complicated transformations to get back to the lower-level, tangled version. Vandevoorde [Van98] proposed an illuminating model for visually representing some of this complexity. His idea is to show the traceability relationships between the locations in the high-level source and the locations in the low-level code in a two-dimensional Cartesian coordinate system (see Figure 131). The horizontal axis of this coordinate system represents the code locations (e.g. line numbers) in the high-level source and the vertical axis represents the code locations in the lower-level, tangled code. Next, we plot the points which represent the correspondence between the locations in the source and the tangled code. For example, the left diagram in Figure 131 shows a linear transformation between the location in the source and the transformed code. The semantics of this diagram are as follows: Given the location x in the source, the corresponding location y indicates the portion of the transformed code that was derived from the source code at x. Next, we indicate the portions of the source code implementing some aspects, e.g. A, B, or C. Given the transformation of locations, we can tell which part of the transformed code implements which aspect. Of course, the whole idea of AOP is that the location transformation is nonlinear. For example, the transformation might look like the one in Figure 131 on the right.

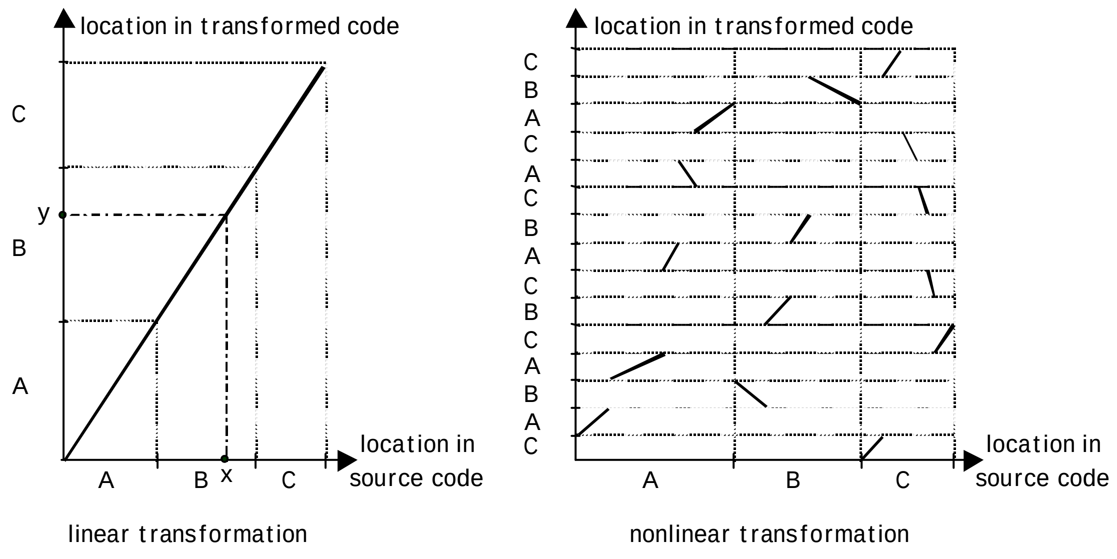


Figure 131 *Diagrams illustrating traceability relationships between locations in source and transformed code*

The traceability relationship between the code locations in the source and the tangled code is often a many-to-many one (see Figure 132). Of course, the diagram shows the relationship just for one particular source. A slightly different source might result in a radically different transformation.

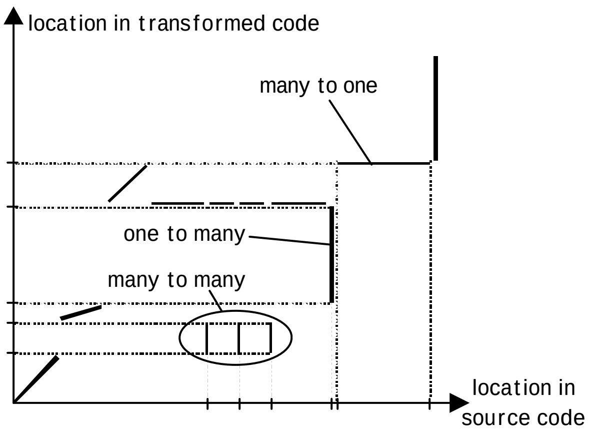


Figure 132 *Traceability relationship with one-to-many, many-to-one, and many-to-many regions*[126]

The amount of transformation work we have to do depends on the level of crosscutting we are addressing. In Section 7.4.8, we gave a whole set of important aspects of OO programs that can be addressed by the relatively simple message-join-point composition mechanisms, which do not involve complex transformation work. Thus, by adding a few reflective constructs such as the different kinds of before and after methods to a language, we can already cover a large number of aspects. However, there are also other aspects, such as domain-specific optimizations in numerical computing, which may involve extremely complex, multi-level transformations. But even in this case, at the different individual levels, we often deal with crosscutting that follows some well structured patterns such as data or control flow, geometry of the problem, etc. The kind of programming abstractions need for capturing the different kinds of crosscutting represents an exciting area for future research.

The code location traceability model we discussed above allows us to make a number of important points [Van98]:

- *Verification problem*: The more non-linear the transformation is the more difficult it is to compute it and thus more difficult to implement it. In particular, given an implementation, e.g. using a transformation system, verifying the correctness of the generated code is extremely difficult. This problem is basically the compiler verification problem. The verification is the more difficult the more higher abstraction and crosscutting level the source code is. Thus, adequate debugging support for the transformation code is crucial.

- *Debugging problem*: In order to be able to find errors in the transforms, we have to be able, as in IP (see Section 6.4.3.3), to debug the generated code at various levels (i.e. source level, intermediate transformation levels, and the lower level). The debugging support is further complicated by the fact that the transformation may implement a many-to-many relationship between the higher-level concepts and the lower-level concepts. As we discussed in Section 6.4.3.3, we also need domain-specific debugging code that allows us to compute the values of some higher-level variables, which are not directly represented in the executable. Finally, domain-specific debugging models are required for the application programmers using the high-level languages since the traditional stepping through the source code is not adequate for all high-level aspects. For example, the mutual exclusion aspect in AspectJ coordinators requires special debugging support, such as tools for monitoring waiting threads, deadlock detection, etc. Another useful feature for the domain-specific level is to be able to navigate from any statement to the aspect modules that affect its semantics. For example, you could click on a method and get a menu of the coordinators that mention this method.[127]

- *Transformations allowing separation of some aspects only*: Supposed we had a set of exemplar programs and, after the programs were written, we came up with a number of aspects which are tangled in these programs and we would like to separate them. Thus, we are looking for a transformation which can transform the equivalent versions of the programs with separated aspects into their tangled versions and, of course, one that we can also use for weaving new aspect programs. In other words, we succeed in separating these aspects in all these programs, only if we find the transformation for weaving them. Unfortunately, the higher abstraction and crosscutting level the aspects are, the higher is the chance that the transformation we are looking for is extremely difficult to compute. Thus, in some cases, it may turn out that a less complicated transformation is more economical, even if it just allows us to write programs where only some of the aspects are separated.

The challenge of high-level programming is obvious: as we increase the abstraction and the crosscutting levels of a set of aspect languages (while increasing their generality level or keeping it constant), the complexity of the required weaving transformations usually grows rapidly. In this context, it is interesting to realize the impact of this level increase on our industry: The software industry employs hordes of programmers, who work as aspect weavers by transforming high level requirements into low level code. By increasing the abstraction and crosscutting levels of programming languages, we eliminate some of the manual, algorithmic work performed by the programmers. One interpretation of this observation is that the advances in programming language design and implementation compete with masses of programmers possessing a huge spectrum of highly specialized skills and thus the automatic transformations required for any progress are inevitably getting more and more complex. The other interpretation is that by moving some of the manual work into the aspect weavers, we allow the programmers to concentrate on the more creative parts of software development and enable the construction of even more complex systems.

7.6.3 AOP and Specialized Language Extensions

Each aspect needs some appropriate linguistic support, i.e. concern-specific abstractions and composition mechanisms addressing crosscutting. In Section 7.6.1, we discussed the advantages of specialized languages over conventional libraries and the conclusion of Section 7.6.2 was that a fixed set of composition mechanisms cannot support all kinds of crosscutting.

Thus, the model of modular language extensions, as exemplified by the IP system, seems to be very appropriate for AOP. It allows a fast dissemination of new composition mechanisms and domain-specific language features or feature frameworks supporting various aspects. Most importantly, these extensions can be used in different constellations, each one tuned towards some domain or domain category. Given this language extension model, the discussion whether to support a number of aspects with one separate language per aspect or with one comprehensive language or with some other number of languages becomes pointless.

7.6.4 AOP and Active Libraries

If we subscribe to the idea that the language extension model is well suited for AOP, IP-like extension libraries (see Section 6.4.3.6.2) become an attractive vehicle for packaging and distributing such extensions. Indeed, we already observe a trend towards shifting some of the responsibilities of compilers to libraries. An example of this shift is the Blitz++ library developed by Veldhuizen [Vel96, Vel98b, Bli] (also see Section 8.11 for other examples).

Blitz++ is a C++ array library for numeric computing. The two most important design goals of Blitz++ are to support a high-level mathematical notation for array expressions and, at the same time, to achieve high efficiency in terms of execution speed and memory consumption. The library is quite successful in satisfying these design goals. For example, instead of writing

```
Array<float,1> A(12);
for (int i=0; i<12; ++i)
   A(i) = sin(2*M_PI / 12*i);
```

we can simply write

```
A = sin(2*M_PI / 12*i);
```

where i is a special placeholder variable. Blitz++ achieves an excellent performance (cf. benchmark results in [VJ97, Vel97, Vel98a]) by applying optimizations such as loop fusing and elimination of temporary variables. An important property of these optimizations is that they need some knowledge of the abstractions they are applied to, e.g. the array components. Thus, they are most appropriately packaged in the library which contains these components rather being implemented in the compiler.

According to Veldhuizen [Vel98a], there are also strong economical reasons for putting the domain-specific optimizations into the library rather than the compiler. First, specialized domains such as scientific computing represent still a small segment market and major compiler vendors focus on providing best support for the average case rather than specialized niches. Supporting a niche is also quite expensive since the cost of compiler development is very high (according to [Vel98a] $80 and more per line of code). Finally, compiler releases represent only an extremely slow channel for distributing new domain-specific features. Thus, putting domain-specific optimizations into libraries is certainly the better solution. However, it is important to note that this solution requires some compile-time metaprogramming capabilities of the language used to program the library. In the case of C++, this only became possible after all the C++ language features enabling template metaprogramming became part of the language in the final standard (see Chapter 8).

Generative libraries

Both Blitz++ and the generative matrix computation library described in Chapter 10 are examples of libraries that contain code which extends the compiler. We refer to such libraries as *generative libraries*. The benefit of generative libraries is that we can write high-level code as opposed to tangled code and still achieve an excellent performance. Unfortunately, there are severe limitations to template metaprogramming, such as the lack of user-defined error reporting and inadequate debugging support. The situation does not improve substantially if we use preprocessors. Although they allow error reporting at the level of the preprocessed code, they require more development effort for implementing the processing infrastructure and also introduce a split between the preprocessor level and the compiler level. This split makes debugging and reporting of compiler-level errors even more difficult.

As we saw in Section 6.4.3, in addition to domain-specific optimizations, we would also like to package other responsibilities in libraries, such as domain-specific editing and displaying capabilities (including aspect-specific views on the code), domain-specific debugging support, domain-specific type systems, code analysis capabilities, etc. Given an extendible programming environment such as IP, the packaging of all these capabilities becomes more economical since we do not have to reinvent the compiler and programming infrastructure every time we need some new language extension. Also, the problems of template metaprogramming and preprocessors disappear. We refer to libraries extending the whole programming environment as *programming environment extension libraries*.

Programming environment extension libraries

In general, we use the term *active libraries*[128] to refer to libraries which, in addition to the base code implementing domain concepts to be linked to the executable of an application, also contain metacode which can be executed at different times and in different contexts in order to compile, optimize, adapt, debug, analyze, visualize, and edit the base concepts. Furthermore, they can describe themselves to tools (such as compilers, profilers, code analyzers, debuggers, etc.) in an intelligible way.

Active libraries

Active libraries may contain more than one generation metalevel, e.g. there could be code that generates compilation code based on the deployment context of the active library (e.g. they could query the hardware and the operating system about their architecture). Furthermore, the same metarepresentations may be used at different times and in different contexts, e.g. based on the compile-time knowledge of some context properties which remain stable during runtime, some metarepresentations may be used to perform optimizations at compile time and other metarepresentations may be injected into the application in order to allow optimization and reconfiguration at runtime.[129] Which metarepresentations are evaluated at which time will depend on the target application.

This perspective outlined above forces us to redefine the conventional interaction between compilers, libraries, and applications. In a sense, active libraries can be viewed as a kind of knowledgeable agents, which interact with each other in order to produce more specialized agents, i.e. applications, which allow the user to solve specific problems. All the agents need some infrastructure supporting communication between agents, generation, transformation, interaction with the programmers, versioning, etc. This infrastructure could be referred to as *compiler middleware* or as more general *programming environment middleware*.

Compiler middleware, programming environment middleware

7.7 Final Remarks

The initial contribution of AOP is to focus our attention on the important deficiency of current component technologies: the inability to adequately capture many important aspects of software systems using generalized procedures only.

However, as we have seen in this chapter, AOP has also a profound impact on analysis and design by introducing a new style of decomposition (we will characterize this style as a multiparadigm decomposition style later in this section). It also motivates the development of new kinds of composition mechanisms and new concern- and domain-specific languages. As we go along, we will be able to componentize more kinds of aspects.

Our feeling is that, as we come up with more and more systems of aspects, they will become qualities on their own. From the modeling perspective, we can view systems of aspects as reusable frameworks of system decomposition. Thus, the impact of AOP on modeling methods will be fundamental. In order to understand this impact, we have to realize that most of the current analysis, design, and implementation methods are centered around single paradigms, e.g. OO methods around objects, structured methods around procedures and data structures. The insight of AOP is that we need different paradigms for different aspects of a system we want to build. This kind of thinking is referred to as the *multiparadigm view* (see e.g. [Bud94]).

Multiparadigm view

We certainly do not have start from scratch. There are whole research communities who have been working on paradigms for different aspects for decades. We refer here simply to communities working on various aspects, e.g. synchronization, distribution, error handling, etc.

By committing to the multiparadigm view in language design and in the development of modeling methods, we will be able to transfer more of the research results of these communities into practice. Thus, while the recognition of the limitations of generalized procedures will be a true "eye opener" for our industry, this is just the beginning.

7.8 Appendix: Instance-Specific Extension Protocol

This appendix contains the implementation of the instance-specific extension protocol described in Section 7.4.7.3 for Smalltalk\VisualWorks. [130] The protocol allows us to define methods in instances and add instance variables to instances. This code has been tested with VisualWorks versions 2.0, 2.5, and 3.0. [131] It is also available at [DCOOL].[132]

Object methods in protocol 'instance specialization':

specialize: aString
```
  "Compile aString as a method for this instance only."

  self specialize.
  self basicClass compile: aString notifying: nil.
```

addInstanceVariable: aString
```
  "Add an instance variable and accessing
  methods based on the name aString"

  self specialize.
  self incrementNumberOfInstVarsInMyClassFormat.
  self mutateSelfToReflectNewClassFormat.
  self addAccessingMethodsForLastInstVarUsing: aString.
```

unspecialize
```
  "Get rid of my private aBehavior, if any."

  self isSpecialized ifFalse: [^self].
  self changeClassToThatOf: self class basicNew.
```

isSpecialized
```
  "Check if I am specialized by checking if my class is aBehavior."

  ^self basicClass shouldBeRegistered not
```

basicClass
```
  "Answer the object which is the receiver's class."

  <primitive: 111>
  self primitiveFailed
```

Object methods in protocol 'private - instance specialization':

specialize
```
  "Insert a private instance of Behavior between me and my class."

  | class |
  self isSpecialized ifTrue: [^self].
  class := Behavior new
    superclass: self class;
    setInstanceFormat: self class format;
    methodDictionary: MethodDictionary new.
  self changeClassToThatOf: class basicNew.
  self basicClass compile:
    'class
      ^super class superclass'
    notifying: nil.
```

incrementNumberOfInstVarsInMyClassFormat

```
  "Set the format of my class so that it defines the number
  of instance variables to be higher by one than previously"

  | format  newInstVarListSize |
  newInstVarListSize := self basicClass instSize - self class instSize + 1.
  format := ClassBuilder new
    genFormat: newInstVarListSize
    under: self class
    format: self basicClass format.
  self basicClass setInstanceFormat: format.
```

mutateSelfToReflectNewClassFormat

```
  "Mutate self to get in sync with the new format of my class"

  | newIVSize mappingArray newInst |
  newIVSize := self basicClass instSize.
  mappingArray := Array new: newIVSize.
  1 to: newIVSize - 1 do: [ :i | mappingArray at: i put: i].
  mappingArray at: newIVSize put: 0.
  newInst := ClassBuilder new
    createCopy: self
    under: self basicClass
    using: mappingArray.
  self become: newInst.
```

addAccessingMethodsForLastInstVarUsing: aString

```
  "Add getters and setters for the last instance
  variable using aString for naming"

  | lastIVIndex getMeth putMeth |
  lastIVIndex := self basicClass instSize.
  getMeth := aString, ' ^self instVarAt: ',
    lastIVIndex printString.
  self basicClass compile: getMeth notifying: nil.
  putMeth := aString, ': aValue self instVarAt: ',
    lastIVIndex printString, ' put: aValue'.
  self basicClass compile: putMeth notifying: nil.
```

7.9 Appendix: Attaching Objects

This appendix contains the implementation of the non-invasive, dynamic composition mechanism described in Section 7.4.7 for Smalltalk\VisualWorks.[133] This code has been tested with VisualWorks versions 2.0, 2.5, and 3.0.[134] It is also available at [DCOOL].

Object methods in protocol 'attaching objects':

attach: anObject at: selector

```
  "Attach anObject at selector. All messages selector
  sent to self are redirected to anObject."

  self makeObjectAttachable.
  self redirectMessage: selector.

  self listenersDictionary at: selector put: anObject.
```

attach: anObject at: selector1 and: selector2

```
  "Attach anObject at selector1 and selector2. All messages selector1
  and selector2 sent to self are redirected to anObject."

  self makeObjectAttachable.
  self redirectMessage: selector1.
  self redirectMessage: selector2.

  self listenersDictionary at: selector1 put: anObject.
  self listenersDictionary at: selector2 put: anObject.
```

attach: anObject atSelectors: aCollection

```
  "Attach anObject at selectors contained in  aCollection."

  self makeObjectAttachable.
  aCollection do: [ :selector |
    self redirectMessage: selector.
    self listenersDictionary at: selector put: anObject ].
```

attach: anObject at: selector1 sending: selector2

```
  "Attach anObject at selector1. All messages selector1
  sent to self are converted to selector2 and redirected to anObject.
  selector2 has to be an unary message selector, i.e. expects no arguments."

  self makeObjectAttachable.
  self redirectMessage: selector1 as: selector2.

  self listenersDictionary at: selector1 put: anObject.
```

detachAt: selector

```
  "Stop redirecting messages of the form ‘selector’."

  self basicClass removeSelector: selector.
  self listenersDictionary removeKey: selector.
```

baseCall

```
  "This method is intended to be invoked directly or indirectly by a redirected message.
  baseCall calls the base method, i.e. the method which would have been called
  if it the message was not redirected."

  | baseContext selector baseMethod na args |
  baseContext := self baseContext.

  "find out the selector of the originally dispatched method"
  selector := baseContext receiver basicClass
    selectorAtMethod: baseContext
    method ifAbsent: [ self error: 'method not found - should not happen' ].

  "retrieve the arguments of the originally dispatched method"
  na := selector numArgs.
  args := Array new: na.
  1 to: na do: [ :i | args at: i put: (baseContext localAt: i) ].

  "look up the base method; if not found, report an error"
  baseMethod := baseContext receiver class findSelector: selector.
  baseMethod isNil ifTrue: [
    Object messageNotUnderstoodSignal
      raiseRequestWith: (Message selector: selector arguments: args)
      errorString: 'superCall: Message not understood by the base object: ' , selector. ].
  baseMethod := baseMethod at: 2.

  "execute the base method"
  ^baseContext receiver performMethod: baseMethod arguments: args
```

baseContext

```
  "Return the initial method context of the redirected message.
  Override this method if you have a direct reference to baseObject in a sublass.
            In that case, test if receiver == baseObject instead of isObjectAttachable.”

  | ctx |
  ctx := thisContext.
  [ ctx == nil or: [
    ctx receiver isObjectAttachable and: [ ctx receiver ~~ self]]] whileFalse: [ ctx := ctx sender ].
  ctx == nil ifTrue: [ self error: 'base context not found' ].
  ^ctx
```

baseObject

```
  "Return the original receiver of the redirected message."

  ^self baseContext receiver
```

Object methods in protocol 'private - attaching objects':

```
makeObjectAttachable
  "Prepare this object for redirecting messages."

  self isObjectAttachable ifTrue: [ ^self ].

  self
    addInstanceVariable: 'listenersDictionary';
    listenersDictionary: IdentityDictionary new.

isObjectAttachable
  "Is this object prepared to redirect messages?"

  ^self basicClass includesSelector: #listenersDictionary

redirectMessage: selector
  "Redirect messages of the form ‘selector’ to self to the appropriate listener object."

  self redirectMessage: selector as: nil

redirectMessage: selector1 as: selector2OrNil
  "Redirect messages of the form ‘selector1’ to self to the appropriate listener object.
  If selector2OrNil is not nil, convert the message to selector2OrNil."

  | dispatchMeth aStream selectorWithArguments newSelector |

  (self messageIsRedirected: selector1) ifTrue: [ ^self ].
  selectorWithArguments := self insertArgumentsInto: selector1.
  newSelector := selector2OrNil isNil
    ifTrue: [ selectorWithArguments ]
    ifFalse: [ selector2OrNil ].

  aStream := WriteStream on: (String new: 100).
  aStream
    nextPutAll: selectorWithArguments;
    nextPutAll: ' ^(self listenersDictionary at: #';
    nextPutAll: selector1;
    nextPutAll: ') ';
    nextPutAll: newSelector.

  dispatchMeth := aStream contents.
  self basicClass compile: dispatchMeth notifying: nil.

messageIsRedirected: selector

  ^self basicClass includesSelector: selector

insertArgumentsInto: selector
  "self insertArgumentsInto: 'messageArg1:with:' returns 'messageArg1: t1 with: t2' "

  | aStream numArgs |
  aStream := WriteStream on: (String new: 60).
  (numArgs := selector numArgs) = 0
    ifTrue: [ aStream nextPutAll: selector ]
    ifFalse: [
      selector keywords with: (1 to: numArgs) do: [ :word :i |
        aStream nextPutAll: word; nextPutAll: ' t'; print: i; space ]].
  ^aStream contents
```

7.10 Appendix: Strategy Pattern With Parameterized Binding Mode

```
#include <iostream.h>

template<class StrategyType>
class Context
{
  public:
    Context(StrategyType *s)
      : strategy(s)
    {}
    void doWork()
    {
      strategy->doAlgorithm();
    }
    void setStrategy(StrategyType *newStrategy)
    {
      strategy=newStrategy;
    }
  private:
    StrategyType *strategy;
};

class DynamicStrategy
{
  public:
    virtual void doAlgorithm() = 0;
};

class StaticStrategy
{};

template<class ParentStrategy>
class StrategyA : public ParentStrategy
{
  public:
    void doAlgorithm()
    { cout << "Strategy A: doAlgorithm()\n";
    }
};

template<class ParentStrategy>
class StrategyB : public ParentStrategy
{
  public:
    void doAlgorithm()
    { cout << "Strategy B: doAlgorithm()\n";
    }
};


void main()
{
  cout << "test context with a static strategy:\n";
  StrategyA<StaticStrategy> statStrategyA;
  Context<StrategyA<StaticStrategy> > context_with_static_strategy(&statStrategyA);
  context_with_static_strategy.doWork();

  cout << "\n";
  cout << "test context with a dynamic strategy:\n";
  StrategyA<DynamicStrategy> dynStrategyA;
  StrategyB<DynamicStrategy> dynStrategyB;

  Context<DynamicStrategy> context_with_dynamic_strategy(&dynStrategyB);
  context_with_dynamic_strategy.doWork();
  context_with_dynamic_strategy.setStrategy(&dynStrategyA);
  context_with_dynamic_strategy.doWork();
}
```

7.11 References

[Aks89] M. Aksit. On the Design of the Object-Oriented Language Sina. Ph.D. Thesis, Department of Computer Science, University of Twente, The Netherlands, 1989

[ABSB94] M. Aksit, J. Bosch, W. v.d. Sterren, and L. Bergmans. Real-Time Specification Inheritance Anomalies and Real-Time Filters. In *Proceedings of the ECOOP'94 Conference*, LNCS 821, Springer-Verlag, July 1994, pp. 386-407, see [CF]

[ABV92] M. Aksit, L. Bergmans, and S. Vural. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In *Proceedings of the ECOOP'92 Conference*, LNCS 615, Springer-Verlag, 1992

[AJ] Homepage of AspectJ^{TM}, XEROX Palo Alto Research Center (XEROX PARC), Palo Alto, California, http://www.parc.xerox.com/aop/aspectj

[AOP] Homepage of the Aspect-Oriented Programming Project, XEROX Palo Alto Research Center (XEROX PARC), Palo Alto, California, http://www.parc.xerox.com/aop/

[AOP97] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales, (Organizers). *Proceedings of the ECOOP'98 Workshop on Aspect-Oriented Programming*, Jyväskylä, Finland, June, 1997, see http://wwwtrese.cs.utwente.nl/aop-ecoop97/

[AOP98] C. Lopes, G. Kiczales, G. Murphy, and A. Lee (Organizers). *Proceedings of the ICSE'98 Workshop on Aspect-Oriented Programming*, Kyoto, Japan, April 20, 1998, see [AOP]

[AT88] M. Aksit and A. Tripathi, Data Abstraction Mechanisms in Sina/ST. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'88)*, ACM SIGPLAN Notices, vol. 23, no. 11, 1988, pp. 265-275

[AT96] G. Attardi and T. Flagella, Memory Management in the PoSSo Solver. In *Journal of Symbolic Computing*, 21, 1996, pp. 1-20, see http://www.di.unipi.it/~attardi/papers.html

[ATB96] M. Aksit, B. Tekinerdogan, and L. Bergmans. Achieving adaptability through separation and composition of concerns. In Proceedings of the ECOOP Workshop on Adaptability in Object-Oriented Software Development, Linz, Austria, July 1996, published in [Müh97], pp. 12-23

[AWB+93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, LNCS 791, Springr-Verlag, 1993, pp. 152-184

[BC90] G. Bracha and W. Cook. Mixin-Based Inheritance. In *Proceedings of the 8th Conference on Object-Oriented Programming, Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP), ACM SIGPLAN Notices*, vol. 25, no. 10, 1990, pp. 303-311

[BDF98] L. Berger, A.M. Dery, M. Fornarino. Interactions between objects: an aspect of object-oriented languages. In [AOP98], pp. 13-18

[Bec93] K. Beck. Smalltalk Idioms: Instance specific behavior, Part 1. In *Smalltalk Report*, no. 2, vol. 6, March/April 1993, pp. 13-15

[Bec93] K. Beck. Smalltalk Idioms: Instance specific behavior, Part 2. In *Smalltalk Report*, no. 2, vol. 7, May/June 1993.

[Bec95] K. Beck. Smalltalk Idioms: What? What happened to garbage collection? In *Smalltalk Report*, no. 4, vol. 4, March/April 1995, pp. 27-32

[Ber94] L. M. J. Bergmans. Composing Concurrent Objects — Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs. Ph.D. Thesis, Department of Computer Science, University of Twente, The Netherlands, 1994, see ftp://ftp.cs.utwente.nl/pub/doc/TRESE/bergmans.phd.tar

[BFJR98] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to Rescue. In Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98), E. Jul (Ed.), LNCS 1445, Springer-Verlag, 1998, pp. 396-417

[BG98] C. Becker and K. Gheis. Quality of Service — Aspect of Distributed Programs. In [AOP98], pp. 7-12

[Bli] Homepage of the Blitz++ Project, http://monet.uwaterloo.ca/blitz/

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. Wiley, Chichester, UK, 1996

[Bos98] J. Bosch. Design Patterns as Language Constructs. In *Journal of Object-Oriented Programming (JOOP)*, May 1998, pp. 18-32

[Bud94] T. Budd. *Multiparadigm Programming in Leda*. Addison-Wesley, 1994

[CF] Homepage of the TRESE Project, University of Twente, The Netherlands, http://wwwtrese.cs.utwente.nl/; also see the online tutorial on Composition Filters at http://wwwtrese.cs.utwente.nl/sina/cfom/index.html

[CIOO96] L. Bergans and P. Cointe (Organizers). Proceedings of the ECOOP'96 Workshop on Composability Issues in Object Orientation (CIOO'96), Linz, Austria, July 8-9, 1996. In [Müh97], pp. 53-123

[Cop92] J. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1992

[Cza96] K. Czarnecki. Metaprogrammierung für jedermann: Ein Mustersystem für leichtgewichtige Framework-Erweiterungen in Smalltalk, Teil II. In *OBJEKTspektrum*, Juli/August 1996, pp. 96-99, http://nero.prakinf.tu-ilmenau.de/~czarn/

[DCOOL] K. Czarnecki, Dynamic Cool, a prototype implementation of a dynamic version of Cool in Smalltalk, available at http://nero.prakinf.tu-ilmenau.de/~czarn/aop

[Dem] Homepage of the Demeter Project, Northeastern University, Boston, Massachusetts, http://www.ccs.neu.edu/research/demeter/

[DG87] L. Demighiel and R. Gabriel. The Common Lisp Object System: An overview. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP87)*, 1987, pp. 151-170

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976

[DK98] A. van Deursen and P. Klint. Little Languages: Little Maintenance? In *Journal of Software Maintenance*, no. 10,1998, pp. 75-92, see http://www.cwi.nl/~arie

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[Gla95] M. Glandrup. Extending C++ Using the Concepts of Composition Filters. Master's Thesis, Department of Computer Science, University of Twente, The Netherlands, November 1995, see [CF]

[GR83] A. Goldberg and D. Robson. *Smalltalk 80: The Language and its Implementation*. Addison-Wesley, 1983

[HJ96] B. Hinkle and R. E. Johnson. Deep in the Heart of Smalltalk: The active life is the life for me! In *Smalltalk Report*, May, 1996, see http://www.sigs.com/publications/docs/srpt/9605/srpt9605.f.hinkle.html

[HO93] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings of the 8th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '93), ACM SIGPLAN Notices*, vol. 28, no. 10, 1993, pp. 411-428

[ILG+97] J. Irwin, J.-M. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-Oriented Programming of Sparse Matrix Code. XEROX PARC Technical Report SPL97-007 P9710045, February 1997, see [AOP]

[Kee89] S. Keene. *Object-oriented programming in Common Lisp: a Programmer's Guide to CLOS*. Addison-Wesley, 1989

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings ECOOP'97 — Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 1997*, Mehmet Aksit and Satoshi Matsuoka (Eds.), LNCS 1241, Springer-Verlag, 1997, also see [AOP]

[KRB91] G. Kiczales, J. des Rivières, and D.-G. Bobrow. *The art of the metaobject protocol*. The MIT Press, 1991

[Lea97] D. Lea. *Concurrent Programming in Java^{TM}: Design Principles and Patterns*. Addison-Wesley Longman, 1997

[LHR88] K. J. Lieberherr, I. Holland, and A. J. Riel. Object-oriented programming: an objective sense of style. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'88), ACM SIGPLAN Notices*, vol. 23, no.11, 1988, pp. 323-334

[Lie86a] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, ACM SIGPLAN Notices, vol. 21, no. 11, 1986, pp. 214-223

[Lie86b] H. Lieberman. Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object Oriented Systems. In 3eme Journees d'Etudes Langages Orientes Objets, J. Bezivin, P. Cointe, (Eds.), AFCET, Paris, France, 1986

[Lie92] K. Lieberherr. Component Enhancement: An Adaptive Reusability Mechanism for Groups of Collaborating Classes. In *Information Processing'92, 12th World Computer Congress, Madrid, Spain*, J. van Leeuwen, (Ed.), Elsevier, 1992, pp. 179-185

[Lie96] K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996

[Lie97] K. Lieberherr. Demeter and Aspect-Oriented Programming. In *Proceedings of the STJA'97 Conference*, Erfurt, Germany, September 10-11, 1997, pp. 40-43, see [DM]

[Lie98] K. Lieberherr. Connections between Demeter/Adaptive Programming and Aspect-Oriented programming. Informal note at http://www.ccs.neu.edu/home/lieber/connection-to-aop.html

[LK97] C. V. Lopes and G. Kiczales. D: A Language Framework for Distributed Programming. XEROX PARC Technical Report SPL97-010 P9710047, February 1997, see [AOP]

[LL94] C. Lopes and K. Lieberherr. *Abstracting Process-to-Function Relations in Concurrent Object-Oriented Applications.* In Proceedings of the *European Conference on Object-Oriented Programming (ECOOP'94)*, Bologna, Italy, 1994, pp. 81-99

[LO97] K. Lieberherr and D. Orleans. Preventive Program Maintenance in Demeter/Java. In *Proceedings of International Conference on Software Engineering (ICSE 1997)*, Boston, MA, pp. 604-605, see [Dem]

[Lop95] C. Lopes. Graph-based optimizations for parameter passing in remote invocations. In *Proceeding of the 4th International Workshop on Object Orientation in Operating Systems (I-WOOOS'95)*, Lund, Sweden, IEEE Computer Society Press, August 1995

[Lop97] C. V. Lopes. D: A Language Framework for Distributed Programming. Ph.D. Thesis, Graduate School of College of Computer Science, Northeastern University, Boston, Massachusetts, 1997, see [Dem]

[LP97] K. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, Northeastern University, Boston, Massachusetts, September 1997, see [Dem]

[MB85] S. L. Messick and K. L. Beck. Active Variables in Smalltalk-80. Technical Report CR-85-09, Computer Research Lab, Tektronix, Inc., 1985

[MD95] J. Mordhorst and W. van Dijk. Composition Filters in Smalltalk. Master's Thesis, Department of Computer Science, University of Twente, The Netherlands, July 1995, see [CF]

[Mez97a] M. Mezini. Variation-Oriented Programming: Beyond Classes and Inheritance. Ph.D. Thesis, Fachbereich Elektrotechnik und Informatik, Universität-Gesamthochschule Siegen, Germany, 1997, see http://www.informatik.uni-siegen.de/~mira/thesis.html

[Mez97b] M. Mezini. Dynamic Object Evolution Without Name Collisions. In *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP '97)*, M. Aksit and S. Matsuoka, (Eds.), LNCS 1241, Springer-Verlag 1997, pp. 190-219, http://www.informatik.uni-siegen.de/~mira/public.html

[MKL97] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A Case-Study for Aspect-Oriented Programming. XEROX PARC Technical Report SPL97-009 P9710044, February 1997, see [AOP]

[ML98] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In Proceedings of the *Conference on Object-Oriented Programming Languages and Applications (OOPSLA '98)*, 1998 and Technical Report NU-CCS-98-3, Northeastern University, Boston, Massachusetts, 1998, see [Dem]

[Moo86] D. A. Moon. Object-Oriented Programming with Flavors. In *Proceedings of the 1st ACM Conference on Object-Oriented Programming Languages and Applications (OOPSLA '86), ACM SIGPLAN Notices*, vol. 21, no. 11, 1986, pp. 1-8

[Müh97] Mühlhäuser (Ed.). *Special Issues in Object-Oriented Programming: Workshop Reader of ECOOP'96*, dpunkt.verlag, 1997

[MWY90] S. Matsuoka, K. Wakita, and A. Yonezawa. Synchronization constraints with inheritance: What is not possible — so what is? Technical Report 10, Department of Information Science, the University of Tokyo, 1990

[MY93] S. Matsuoka and A. Yonezawa. Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner, and A. Yonezawa, (Eds.), MIT Press, April 1993, pp. 107-150

[NT95] O. Nierstrasz and D. Tsichritzis. *Object-Oriented Software Composition*. Prentice Hall, 1995

[OKH+95] H. Ossher, M. Kaplan, W. Harrison, A. Katz and V. Kruskal. Subject-Oriented Composition Rules. In *Proceedings of the 10th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '95), ACM SIGPLAN Notices*, vol. 30, no. 10, 1995, pp. 235-250

[OKK+96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying Subject-Oriented Composition. In *Theory and Practice of Object Systems*, vol. 2, no. 3, 1996

[OT98] H. Ossher and P. Tarr. Operation-Level Composition: A Case in (Join) Point. In [AOP98], pp. 28-31

[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995

[RMI97] Java Remote Method Invocation. Specification, Sun Microsystems, December 1997 http://java.sun.com/products/jdk/rmi/index.html

[Sch94] D.C. Schmidt. The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications. In *Proceedings of the 12th Annual Sun Users Group Conference (SUG)*, San Francisco, California, June 1994, pp. 214-225; the ACE toolkit is available at http://www.cs.wustl.edu/~schmidt/ACE-obtain.html

[Sch95] D.C. Schmidt. An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit. Technical Report WUCS-95-31, Washington University, 1995, see http://www.cs.wustl.edu/~schmidt/ACE-concurrency.ps.gz

[SHS94] I. Silva-Lepe, W. Hürsch, and G. Sullivan. A Report on Demeter C++. In *C++ Report*, vol. 6, no. 2, November 1994, pp. 24-30, see [Dem]

[SOP] Homepage of the Subject-Oriented Programming Project, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, http://www.research.ibm.com/sop/

[SOPD] Subject-oriented programming and design patterns. Draft, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, see http://www.research.ibm.com/sop/sopcpats.htm

[Sou95] J. Soukup. Implementing Patterns. In *Pattern Languages of Programm Design*, J. O. Coplien and D. C. Schmidt, (Eds.), Addison-Wesley, 1995, pp. 395-412

[SPL 96] L. M. Seiter, J. Palsberg, K. Lieberherr. Evolution of Object Behavior Using Context Relations. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Garlan D., (Ed.), Software Engineering Notes, vol. 21, no. 6, ACM Press, 1996, pp. 46-56, see [Dem]

[Ste87] L. A. Stein. Delegation Is Inheritance. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '87), ACM SIGPLAN Notices*, vol. 22, no. 12, 1987, pp. 138-146

[Str94] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994

[Van98] D. Vandevoorde, Hewlett-Packard, California Language Lab, Cupertino, California. Personal communication, 1998

[Vel98a] T. Veldhuizen. Scientific computing using the Blitz++ class library. Talk presented at the Dagstuhl Seminar 9817 "Generic Programming", Schloß Dagstuhl, April 27-30, 1998, see [Bli]

[Vel98b] T. Veldhuizen. The Blitz++ Array Model. Submitted to the Second International Symposium on Computing in Object-oriented Parallel Environments (ISCOPE'98), Santa Fe, New Mexico, December 8-11, 1998, see [Bli]

[Vel97] T. Veldhuizen. Scientific Computing: C++ vs. Fortran. In *Dr. Dobb's Journal*, November 1997, see [Bli]

[Vel96] T. Veldhuizen. Rapid Linear Algebra in C++. In *Dr. Dobb's Journal*, August 1996, see [Bli]

[VJ97] T. Veldhuizen and M. Ed Jernigan. Will C++ be faster than Fortran? In *Proceedings of the Second International Symposium on Computing in Object-oriented Parallel Environments (ISCOPE'97)*, 1997, see [Bli]

[XER97] AspectJ^{TM}: Primer. XEROX Palo Alto Research Center (XEROX PARC), Palo Alto, California, October 1997

[XER98a] AspectJ^{TM}: User's Guide and Primer. XEROX Palo Alto Research Center (XEROX PARC), Palo Alto, California, March 1998, see [AJ]

[XER98b] AspectJ^{TM} Specification. XEROX Palo Alto Research Center (XEROX PARC), Palo Alto, California, March 1998, see [AJ]

Chapter 8 Static Metaprogramming in C++

8.1 What Is Static Metaprogramming?

Metalanguage

In linguistics, a *metalanguage* is defined as follows [Dict]:

> *"any language or symbolic system used to discuss, describe, or analyze another language or symbolic system"*

Metaprogramming

Generative metaprograms

This definition characterizes the main idea of *metaprogramming*, which involves writing programs related by the meta-relationship, i.e. the relationship of "being about". A program that manipulates another program is clearly an instance of metaprogramming. In Section 7.4.7, we saw other instances of metaprogramming which do not necessarily involve program manipulation (e.g. before and after methods[135]); however, in this chapter we focus on *generative metaprograms*, i.e. programs manipulating and generating other programs.

generative metaprogram = algorithm + program representation

Static metaprograms

Metaprograms can run in different contexts and at different times. In this chapter, however, we only focus on *static metaprograms*. These metaprograms run before the load time of the code they manipulate.[136]

The most common examples of systems involving metaprogramming are compilers and preprocessors. These systems manipulate representations of input programs (e.g. abstract syntax trees) in order to transform them into other languages (or the same language as the input language but with a modified program structure). In the usual case, the only people writing metaprograms for these systems are compiler developers who have the full access to their sources.

Open compilers

The idea of an *open compiler* is to make metaprogramming more accessible to a broader audience by providing well-defined, high-level interfaces for manipulating various internal program representations (see e.g. [LKRR92]). A transformation system is an example of an open compiler which provides an interface for writing transformations on abstract syntax trees (ASTs). We discussed them in Section 6.3. An open compiler may also provide access to the parser, runtime libraries, or any other of its parts. The Intentional Programming System is a good example of a widely-open programming environment. Unfortunately, currently no industrial-strength open compilers are commercially available (at least not for the main-stream object-oriented languages).[137]

8.2 Template Metaprogramming

A practicable approach to metaprogramming in C++ is *template metaprogramming*. Template metaprograms consist of class templates operating on numbers and/or types as data.

Algorithms are expressed using template recursion as a looping construct and class template specialization as a conditional construct. Template recursion involves the direct or indirect use of a class template in the construction of its own member type or member constant.

Here is an example of a class template[138] which computes the factorial of a natural number:

```
template<int n>
struct Factorial
{
  enum { RET = Factorial<n-1>::RET * n };
};

//the following template specialization terminates the recursion
template<>
struct Factorial<0>
{
  enum { RET = 1 };
};
```

We can use this class template as follows:

```
void main()
{
  cout << Factorial<7>::RET << endl; //prints 5040
}
```

The important point about this program is that Factorial<7> is instantiated at compile time. During the instantiation, the compiler also determines the value of Factorial<7>::RET. Thus, the code generated for this main() program by the C++ compiler is the same as the code generated for the following main():

```
void main()
{
  cout << 5040 << endl; //prints 5040
}
```

We can regard Factorial<> as a *function* which is evaluated at compile time. This particular function takes one number as its parameter and returns another in its RET member (RET is an abbreviation for RETURN; we use this name to mimic the return statement in a programming language). It is important to note that we are dealing with a shift of intentionality here: The job of the compiler is to do type inference, template instantiation, and type construction, which involve computation. We use the fact that the compiler does computation and, by encoding data as numbers and types, we can actually use (or abuse) the compiler as a processor for *interpreting* metaprograms.

Metafunctions

Thus, we refer to functions such as Factorial<> as *metafunctions*. Factorial<> is a metafunction since, at compilation time, it computes constant data of a program which has not been generated yet.

A more obvious example of a metafunction would be a function which returns a type, especially a class type. Since a class type can represents computation (remember: it may contain methods), such a metafunction actually manipulates representations of computation.

The following metafunction takes a Boolean and two types as its parameters and returns a type:

```
template<bool cond, class ThenType, class ElseType>
struct IF
{    typedef ThenType RET;
};

template<class ThenType, class ElseType>
struct IF<false, ThenType, ElseType>
{ typedef ElseType RET;
};
```

As you have probably recognized, this function corresponds to an if statement: it has a condition parameter, a “then” parameter, and an “else” parameter. If the condition is true, it returns ThenType in RET. This is encoded in the base definition of the template. If the condition is false, it returns ElseType in RET. Thus, this metafunction can be viewed as a meta-control statement.

We can use IF<> to implement other useful metafunctions. For example, the following metafunction determines the larger of two numeric types:

```
#include <limits>
using namespace std;

template<class A, class B>
struct PROMOTE_NUMERIC_TYPE
{
   typedef IF<
      numeric_limits<A>::max_exponent10 < numeric_limits<B>::max_exponent10
      ||
      (numeric_limits<A>::max_exponent10==numeric_limits<B>::max_exponent10
         &&
       numeric_limits<A>::digits < numeric_limits<B>::digits),

      B,
      A>::RET RET;
};
```

This metafunction determines the larger numeric type as follows:

1. it returns the second type if its exponent is larger than the exponent of the first type or, in case the exponents are equal, it returns the second type if it has a larger number of significant digits;

2. otherwise, it returns the first type.

For example, the following expression evaluates to float:

```
PROMOTE_NUMERIC_TYPE<float, int>::RET
```

Metainformation

Traits templates

PROMOTE_NUMERIC_TYPE<> retrieves information about number types (i.e. *metainformation*) such as maximum exponent and number of significant digits from the numeric_limits<> templates provided by the standard C++ include file limits. Templates providing information about other types are referred to as *traits templates* [Mye95] (cf. traits classes in Section 6.4.2.4).

The standard traits template numeric_limits<> encodes many important properties of numeric types. The following is the base template (the base template is specialized for different numeric types):

```
template<class T>
   class numeric_limits
{
  public:
   static const bool has_denorm = false;
   static const bool has_denorm_loss = false;
   static const bool has_infinity = false;
   static const bool has_quiet_NaN = false;
   static const bool has_signaling_NaN = false;
   static const bool is_bounded = false;
   static const bool is_exact = false;
   static const bool is_iec559 = false;
   static const bool is_integer = false;
   static const bool is_modulo = false;
   static const bool is_signed = false;
   static const bool is_specialized = false;
   static const bool tinyness_before = false;
   static const bool traps = false;
   static const float_round_style round_style = round_toward_zero;
   static const int digits = 0;
   static const int digits10 = 0;
  static const int max_exponent = 0;
  static const int max_exponent10 = 0;
  static const int min_exponent = 0;
  static const int min_exponent10 = 0;
  static const int radix = 0;
  static T denorm_min() throw();
  static T epsilon() throw();
  static T infinity() throw();
  static T max() throw();
  static T min() throw();
  static T quiet_NaN() throw();
  static T round_error() throw();
  static T signaling_NaN() throw();
  };
```

As stated, this base template is specialized for different numeric types. Each specialization provides concrete values for the members, e.g.:

```
template<>
  class numeric_limits <float>
{
 public:
   //...
   static const bool is_specialized = true; // yes, we have metainformation information about float
   //...
  static const int digits = 24;
  //...
  static const int max_exponent10 = 38;
  //...
};
```

Given the above definition, the following expression evaluates to 24:

```
numeric_limits<float>::digits
```

In general, we have three ways to provide information about a type:

- define a traits template for the type,
- provide the information directly as members of the type, or
- define a configuration repository (we discuss this approach in Section 8.7).

The first approach is the only possible one if the type is a basic type or if the type is user defined and you do not have access to its source or cannot modify the source for some other reason. Moreover, the traits solution is often the preferred one since it avoids putting too much information into the type.

There are also situations however, where you find it more convenient to put the information directly into the user-defined type. For example, later in Section 10.3.1.3, we will use templates to describe the structure of a matrix type. Each of these templates represents some matrix parameter or parameter value. In order to be able to manipulate matrix structure descriptions, we need some means for testing whether two types were instantiated from the same template or not. We can accomplish this by marking the templates with IDs. Let us take a look at an example. Assume that we want to be able to specify the parameter temperature of some metafunction in Fahrenheit and in Celsius (i.e. centigrade). We could do so by wrapping the specified number in the type indicating the unit of temperature used, e.g. fahrenheit<212> or celsius<100>. Since we want to be able to distinguish between Fahrenheit and Celsius, each of these two templates will have an ID. We provide the IDs through a base class:

```
struct base_class_for_temperature_values
{
  enum {
    // IDs of values
    celsius_id,
    fahrenheit_id };
};
```

And here are the two templates:

```
//celsius
template<int Value>
struct celsius : public base_class_for_temperature_values
{ enum {
    id= celsius_id,
    value= Value };
};

//fahrenheit
template<int Value>
struct fahrenheit: public base_class_for_temperature_values
{ enum {
    id= fahrenheit_id,
    value= Value };
};
```

Please note that each template "publishes" both its ID and its parameter value as enum constants.[139] Now, let us assume that some metafunction SomeMetafunction<> expects a temperature specification in Fahrenheit or in Celsius, but internally it does its computations in Celsius. Thus, we need a conversion metafunction, which takes a temperature specification in Fahrenheit or in Celsius and returns a temperature value in Celsius:

```
template<class temperature>
struct ConvertToCelsius
{ enum {
    RET=
      temperature::id==temperature::fahrenheit_id ?
        (temperature::value-32)*5/9                   :
         temperature::value };
};
```

Here is an example demonstrating how this metafunction works:

```
cout << ConvertToCelsius<fahrenheit<212> >::RET << endl;   // prints "100"
cout << ConvertToCelsius<celsius<100> >::RET << endl;      // prints "100"
```

Now, you can use the conversion function in SomeFunction<> as follows:

```
template<class temperature>
class SomeFunction
{
  enum {
    celsius = ConvertToCelsius<temperature>::RET,
    // do some other computations...
    //... to compute result
    };
  public:
    enum {
      RET = result };
};
```

The latter metafunction illustrates a number of points:

- You do some computation in the private section and return the result in the public section.
- The enum constant initialization is used as a kind of "assignment statement".
- You can split computation over a number of "assignment statements" or move them into separate metafunctions.

There are two more observations we can make based on the examples discussed so far:

- Metafunctions can take numbers and/or types as their arguments and return numbers or types.

- The equivalent of an assignment statement for types is a member typedef declaration (see the code for PROMOTE_NUMERIC_TYPE<> above).

8.3 Metaprograms and Code Generation

The previous examples of metafunctions demonstrated how to perform computations at compile time. But we can also arrange metafunctions into metaprograms that generate code.

We start with a simple example. Assume that we have the following two types:

```
struct Type1
{
  static void print()
  {
    cout << "Type1" << endl;
  }
};

struct Type2
{
  static void print()
  {
    cout << "Type2" << endl;
  }
};
```

Each of them defines the static inline method print(). Now, consider the following statement:

```
IF< (1<2), Type1, Type2>::RET::print();
```

Since the condition 1<2 is true, Type1 ends up in RET. Thus, the above statement compiles into machine code which is equivalent to the machine code obtained by compiling the following statements:

```
cout << "Type1" << endl;
```

The reason is that print() is declared as a static inline method and the compiler can optimize away any overhead associated with the method call. This was just a very simple example, but in general you can imagine a metafunction that takes some parameters and does arbitrarily complex computation in order to select some type providing the right method:

```
SomeMetafunction</* takes some parameters here*/>::RET::executeSomeCode();
```

The code in executeSomeCode() can also use different metafunctions in order to select other methods it calls. In effect, we are able to combine code fragments at compile time based on the algorithms embodied in the metafunctions.

As a further example of static code generation using metafunctions, we show you how to do loop unrolling in C++. In order to do this, we will use a meta while loop available at [TM]. In order to use the loop we need a statement and a condition. Both will be modeled as structs. The statement has to provide the method execute() and the next statement in next. The code in execute() is actually the code to be executed by the loop:

```
template <int i>
struct aStatement
{ enum { n = i };
  static void execute()
  { cout << i << endl;
  }
  typedef aStatement<n+1> next;
};
```

The condition provides the constant evaluate, whose value is used to determine when to terminate the loop.

```
struct aCondition
{ template <class Statement>
  struct Test
  { enum {  evaluate = (Statement::n <= 10) };
  };
};
```

Now, you can write the following code:

```
WHILE<Condition,aStatement<1> >::EXEC();
```

The compiler expands this code into:

```
cout << 1 << endl;
cout << 2 << endl;
cout << 3 << endl;
cout << 4 << endl;
cout << 5 << endl;
cout << 6 << endl;
cout << 7 << endl;
cout << 8 << endl;
cout << 9 << endl;
cout << 10 << endl;
```

You can use this technique to implement optimizations by unrolling small loops, e.g. loops for vector assignment.

Another approach to code generation is to compose class templates. For example, assume that we have the following two class templates:

- `List<>`, which implements the abstract data type list and
- `ListWithLengthCounter<>`, which represents a wrapper on `List<>` implementing a counter for keeping track of the length of the list.

Now, based on a flag, we can decide whether to wrap `List<>` into `ListWithLengthCounter<>` or not:

```
typedef IF<flag==listWithCounter,
              ListWithLengthCounter<List<ElementType> >
              List<ElementType> >::RET list;
```

We will use this technique later in Section 8.7 in order to implement a simple list generator.

8.4 List Processing Using Template Metaprograms

Nested templates can be used to represent lists. We can represent many useful things using a list “metadata structure”, e.g. a meta-case statement (which is shown later) or other control structures (see e.g. Section 10.3.1.6). In other words, template metaprogramming uses types as compile-time data structures. Lists are useful for doing all sorts of computations in general. Indeed, lists are the only data structure available in Lisp and this does not impose any limitation on its expressiveness.

In Lisp, you can write a list like this:

```
(1 2 3 9)
```

This list can be constructed by calling the list constructor `cons` four times:

```
(cons 1 (cons 2 (cons 3 (cons 9 nil))))
```

where `nil` is an empty list. Here is the list in the prefix notation:

```
cons(1, cons(2, cons(3, cons(9, nil))))
```

We can do this with nested templates, too:

```
Cons<1, Cons<2, Cons<3, Cons<9, End> > > >
```

Here is the code for `End` and `Cons<>`:

```
//tag marking the end of a list
const int endValue = ~(~0u >> 1); //initialize with the smallest int

struct End
```

```
{
   enum    { head = endValue };
   typedef End Tail;
};

template<int head_, class Tail_ = End>
struct Cons
{
   enum    { head = head_ };
   typedef Tail_ Tail;
};
```

Please note that End and Cons<> publish head and Tail as their members.

Now, assume that we need a metafunction for determining the length of a list. Here is the code:

```
template<class List>
struct Length
{
  // make a recursive call to Length and pass Tail of the list as the argument
  enum { RET= Length<List::Tail>::RET+1 };
};

// stop the recursion if we've got to End
template<>
struct Length<End>
{
  enum { RET= 0 };
};
```

We can use this function as follows:

```
typedef Cons<1,Cons<2,Cons<3> > > list1;
cout << Length<list1>::RET << endl; // prints "3"
```

The following function tests if a list is empty:

```
template<class List>
struct IsEmpty
{
  enum { RET= 0 };
};

template<>
struct IsEmpty<End>
{
  enum { RET= 1 };
};
```

Last<> returns the last element of a list:

```
template<class List>
struct Last
{
  enum {
    RET=IsEmpty<List::Tail>::RET ?
      List::head                              :
      Last<List::Tail>::RET * 1 //multiply by 1 to make VC++ 5.0 happy
  };
};

template<>
struct Last<End>
{
  enum { RET= endValue };
};
```

Append1<> takes the list List and the number n and returns a new list which is computed by appending n at the end of List:

```
template<class List, int n>
struct Append1
{
  typedef Cons<List::head, Append1<List::Tail, n>::RET> RET;
};

template<int n>
struct Append1<End, n>
{
  typedef Cons<n> RET;
};
```

Please note that the recursion in Append1<> is terminated using partial specialization, where List is fixed to be End whereas n remains variable.

A general Append<> for appending two lists is similar:

```
template<class List1, class List2>
struct Append
{
  typedef Cons<List1::head, Append<List1::Tail, List2>::RET > RET;
};

template<class List2>
struct Append<End, List2>
{
  typedef List2 RET;
};
```

The following test program tests our list metafunctions:

```
void main()
{
  typedef Cons<1,Cons<2,Cons<3> > > list1;

  cout << Length<list1>::RET << endl;       // prints 3
  cout << Last<list1>::RET << endl;        // prints 3

  typedef Append1<list1, 9>::RET list2;

  cout << Length<list2>::RET << endl;    // prints 4
  cout << Last<list2>::RET << endl;          // prints 9

  typedef Append<list1, list2>::RET list3;

  cout << Length<list3>::RET << endl;    // prints 7
  cout << Last<list3>::RET << endl;          // prints 9

}
```

You can easily imagine how to write other functions such as Reverse(List), CopyUpTo(X,List), RestPast(X,List), Replace(X,Y,List), etc. We can make our list functions more general by declaring the first parameter of Cons<> to be a type rather than an int and providing a type wrapper for numbers:

```
template<class head_, class Tail_ = End>
struct Cons
{
   typedef head_ head;
   typedef Tail_ Tail;
};

template <int n>
struct Int
{
  enum { value=n };
};
```

Given the above templates, we can build lists consisting of types and numbers, e.g.

```
Cons<Int<1>, Cons<SomeType, Cons<Int<3> > > >
```

Indeed, this technique allows us to implement a simple Lisp as a template metaprogram (see Section 8.12).

8.5 Workarounds for Doing Without Partial Specialization of Class Templates

As you saw in the previous section, partial specialization allows you to terminate the recursion of a template whose some parameters remain fixed during all iterations and other parameters vary from iteration to iteration. Unfortunately, currently only very few compilers support partial specialization. But here is the good news: We have always been able to find a workaround. This section gives you the necessary techniques for doing without partial specialization.

We will consider two cases: a recursive metafunction with numeric arguments and a recursive metafunction with type parameters.

We start with a recursive metafunction with numeric arguments. A classic example of a numeric function requiring partial specialization is raising a number to the m-th power. Here is the implementation of the power metafunction with partial specialization:

```
//power
template<int n, int m>
struct Power
{
  enum { RET = m>0 ? Power< n, (m>0) ? m-1:0 >::RET * n
                        : 1 };
};

// terminate recursion
template<int n>
struct Power<n, 0>
{
  enum { RET = 1 };
};
```

This function works as follows: It takes `n` and `m` as arguments and recursively computes the result by calling `Power<n, m-1>`. Thus, the first argument remains constant, while the second argument is “consumed”. We know that the final call looks like this: `Power<n, 0>`. This is the classic case for partial specialization: One argument could be anything and the other one is fixed.

If our C++ compiler does not support partial specialization, we need another solution. The required technique is to map the final call to one that has all of its arguments fixed, e.g. `Power<1, 0>`. This can be easily done by using an extra conditional operator in the first argument to the call to `Power<n, m-1>`. Here is the code:

```
//power
template<int n, int m>
struct Power
{
  enum { RET = m>0 ? Power<(m>0) ? n:1, (m>0) ? m-1:0>::RET * n
                        : 1 };
};

template<>
struct Power<1,0>
{
  enum { RET = 1 };
};

cout << "Power<2,3>::RET = " << Power<2,3>::RET << endl; // prints “8”
```

We can use the same trick in order to reimplement the metafunction Append1<> (see Section 8.4) without partial specialization. All we have to do is to map the recursion termination to a case where both arguments are fixed:

```
template<class List, int n>
struct Append1
{
  enum {
    new_head = IsEmpty<List>::RET  ?
      endValue                     :
      n
  };

  typedef
    IF<IsEmpty<List>::RET,
      Cons<n>,
      Cons<List::head, Append1<List::Tail, new_head>::RET >
    >::RET RET;
};

template<>
struct Append1<End, endValue>
{
  typedef End RET;
};
```

The same technique also works with type arguments. Here is the metafunction Append<> without partial specialization (remember that the second argument was constant during recursion):

```
template<class List1, class List2>
struct Append
{
  typedef
    IF<IsEmpty<List1>::RET,
      End,
      List2
    >::RET NewList2;

  typedef
    IF<IsEmpty<List1>::RET,
      List2,
      Cons<List1::head, Append<List1::Tail, NewList2>::RET >
    >::RET RET;
};

template<>
struct Append<End, End>
{
  typedef End RET;
};
```

You may ask why we worry about terminating recursion in the else branch of the second IF<> if in the terminal case we return List2 and not the value of the recursive call. The reason why we have to worry is that the types in both branches of a meta-if (i.e. IF<>) are actually built by the compiler. Thus, meta-if has a slightly different behavior than an if statement in a conventional programming language: we have always to keep in mind that both branches of a meta-if are "evaluated" (at least on VC++ 5.0).

We have used meta-if in order to eliminate partial specialization in Append<>. But if you take a look at the implementation of meta-if in Section 8.2, you will realize that meta-if uses partial specialization itself. Thus, we also need to provide an implementation of meta-if without partial specialization.

A meta-if involves returning the first of its two argument types if the condition is true and the second one if the condition is false. Let us first provide two metafunctions Select, where each of them takes two parameters and the first one returns the first parameter and the second one returns the second parameter. Additionally, we wrap the metafunctions into classes, so that we can return them as a result of some other metafunction. Here is the code for the two metafunctions:

```
struct SelectFirstType
{
  template<class A, class B>
  struct Select
  {
    typedef A RET;
  };
};

struct SelectSecondType
{
  template<class A, class B>
  struct Select
  {
    typedef B RET;
  };
};
```

Now, we implement another metafunction which takes a condition number as its parameter and returns SelectFirstType if the condition is other than 0 and SelectSecondType if the condition is 0:

```
template<int condition>
struct SelectSelector
{
  typedef SelectFirstType RET;
};

template<>
struct SelectSelector<0>
{
  typedef SelectSecondType RET;
};
```

Finally, we can implement our meta-if:

```
template<int condition, class A, class B>
class IF
{
    typedef SelectSelector<condition>::RET Selector;

  public:
    typedef Selector::Select<A, B>::RET RET;
};
```

Please note that this implementation of meta-if does not use partial specialization. Another interesting thing to point out is that we returned metafunctions (i.e. SelectFirstType and SelectSecondType) as a result of some other metafunction (we could use the same technique to pass metafunctions as arguments to some other metafunctions). Thus, template metaprogramming supports *higher-order metafunctions*.

Higher-order metafunctions

8.6 Control Structures

The structure of metaprograms can be improved by using basic metafunction which provide the functionality equivalent to the control structures used in programming languages such as if, switch, while, for, etc.

You have already seen the meta-if in Section 8.2. A *meta-switch* is used as follows:

```
SWITCH < 5
  , CASE < 1, TypeA
  , CASE < 2, TypeB
  , DEFAULT < TypeC > > >
  >::RET                          //returns TypeC
```

As you probably recognized, this switch uses the list processing techniques we discussed in Section 8.4. The implementation is given in Section 8.13.

You also saw the *meta-while loop* in Section 8.3. Other loops such as *meta-for* and *meta-do* also exist (see [TM]).

8.7 Configuration Repositories and Code Generation

In Section 6.4.2.4, we showed how to use a configuration repository class in order to encapsulate the horizontal parameters of a configuration and to propagate types up and down aggregation and inheritance hierarchies. Such configuration repositories can be combined with template metaprogramming into a very powerful static generation technique.

We will present this generation technique step by step, starting with a simple example of configurable list components based on a configuration repository.

We start with the implementation of the base component representing a list. The component implements methods such as head() for reading the head of the list, setHead() for setting the head, and setTail() for setting the tail of the list. The component takes a configuration repository class as its parameter and retrieves a number of types from it:

```
template<class Config_>
class List
{
  public:
    // publish Config so that others can access it
    typedef Config_ Config;

  private:
    // retrieve the element type
    typedef Config::ElementType ElementType;

    // retrieve my type (i.e. the final type of the list); this is necessary since
    // we will derive from List<> later and
    // we want to use the most derived type as the type of tail_;
    // thus, we actually pass a type from the subclass
    // to its superclass
    typedef Config::ReturnType ReturnType;

  public:
    List(const ElementType& h, ReturnType *t = 0) :
      head_(h), tail_(t)
    {}

    void setHead(const ElementType& h)
    { head_ = h; }

    const ElementType& head() const
    { return head_; }

    void setTail(ReturnType *t)
    { tail_ = t; }

    ReturnType *tail() const
    { return tail_; }

  private:
    ElementType head_;
    ReturnType *tail_;
};
```

Next, imagine that you need to keep track of the number of elements in the list. We can accomplish this using an inheritance-based wrapper (see Section 6.4.2.4):

```
template<class BaseList>
```

```
class ListWithLengthCounter : public BaseList
{
  public:
    typedef BaseList::Config Config;

  private:
    typedef Config::ElementType ElementType;
    typedef Config::ReturnType ReturnType;

    //get the type for the length counter
    typedef Config::LengthType LengthType;

  public:
    ListWithLengthCounter(const ElementType& h, ReturnType *t = 0) :
     BaseList(h,t), length_(computedLength())
     { }

    //redefine setTail() to keep track of the length
    void setTail(ReturnType *t)
    {
      BaseList::setTail(t);
      length_ = computedLength();
    }

    //an here is the length() method
    const LengthType& length() const
    { return length_; }

  private:
    LengthType computedLength()
    {
      return tail()   ? tail()->length()+1
                      : 1; }

    LengthType length_;
};
```

Furthermore, we might be also interested in logging all calls to head(), setHead(), and setTail(). For this purpose, we provide yet another wrapper, which is implemented similarly to ListWithLengthCounter<>:

```
template<class BaseList>
class TracedList : public BaseList
{
  public:
    typedef BaseList::Config Config;

  private:
     typedef Config::ElementType ElementType;
    typedef Config::ReturnType ReturnType;

  public:
    TracedList(const ElementType& h, ReturnType *t = 0) :
     BaseList(h,t)
     { }

    void setHead(const ElementType& h)
    {  cout << "setHead(" << h << ")"<< endl;
       BaseList::setHead(h);
    }
    const ElementType& head() const
    {  cout << "head()"<< endl;
       return BaseList::head();
    }

    void setTail(ReturnType *t)
    {  cout << "setTail(t)"<< endl;
```

```
      BaseList::setTail(t);
    }
};
```

Given these three components, we can construct four different configurations:

- SimpleList: a simple list;
- ListWithCounter: a list with a counter;
- TracedList: a list with tracing;
- TracedListWithCounter: a list with a counter and with tracing.

Here is the code:

```
//SimpleList
struct ListConfig
{
  typedef int ElementType;
  typedef List<ListConfig> ReturnType;
};
typedef ListConfig::ReturnType SimpleList;

// ListWithCounter
struct CounterListConfig
{
  typedef int ElementType;
  typedef int LengthType;
  typedef ListWithLengthCounter<List<CounterListConfig> > ReturnType;
};
typedef CounterListConfig::ReturnType ListWithCounter;

// TracedList
struct TracedListConfig
{
  typedef int ElementType;
  typedef TracedList<List<TracedListConfig> > ReturnType;
};
typedef TracedListConfig::ReturnType TracedList;

// TracedListWithCounter
struct TracedCounterListConfig
{
  typedef int ElementType;
  typedef int LengthType;
  typedef TracedList<ListWithLengthCounter<List<TracedCounterListConfig> > > ReturnType;
};
typedef TracedCounterListConfig::ReturnType TracedListWithCounter;
```

The problem with this solution is that we have to provide an explicit configuration repository for each of these four combinations. This is not a big problem in our small example. However, if we have thousands of different combinations (e.g. the matrix components described in Chapter 10 can be configured in over 140 000 different ways), writing all these configuration repository classes becomes unpractical (remember that for larger configurations each of the configuration repository classes is much longer than in this example).

A better solution is to generate the required configuration repository using template metaprogramming. For this purpose, we define the metafunction LIST_GENERATOR<> which takes a number of parameters describing the list type we want to generate and returns the generated list type.

Here is the declaration of this metafunction:

```
template<class ElementType, int counterFlag, int tracingFlag,  class LengthType>
class LIST_GENERATOR;
```

The semantics of the parameters is as follows:

- ElementType: This is the element type of the list.
- counterFlag: This flag specifies whether the generated list should have a counter or not. Possible values are with_counter or no_counter.
- tracingFlag: This flag specifies whether the generated list should have a counter or not. Possible values are with_tracing or no_tracing.
- LengthType: This is the type of the length counter (if any).

We first need to provide the values for the flags:

```
enum {with_counter, no_counter};
enum {with_tracing, no_tracing};
```

And here is the code of the list generator metafunction:

```
template<
  class ElementType_ = int,
  int counterFlag = no_counter,
  int tracingFlag = no_tracing,
  class LengthType_ = int
>
class LIST_GENERATOR
{
  public:
    // provide the type of the generator as Generator
    typedef LIST_GENERATOR<ElementType_, counterFlag, tracingFlag, LengthType_> Generator;

  private:
    //define a simple list; please note that we pass Generator as a parameter
    typedef List<Generator> list;

    // wrap into ListWithLengthCounter if specified
    typedef
      IF<counterFlag==with_counter,
        ListWithLengthCounter<list>,
        list
      >::RET list_with_counter_or_not;

    // wrap into TracedList if specified
    typedef
      IF<tracingFlag==with_tracing,
        TracedList<list_with_counter_or_not>,
        list_with_counter_or_not
      >::RET list_with_tracing_or_not;

  public:
    // return finished type
    typedef list_with_tracing_or_not RET;

    // provide Config; Config is retrieved by List<> from Generator and passed on to its wrappers
    struct Config
    {
      typedef ElementType_ ElementType;
      typedef LengthType_ LengthType;
      typedef RET ReturnType;
    };
};
```

Please note that we passed Generator to List<> rather than Config. This is not a problem since List<> can retrieve Config from Generator. Thus, we need to change two lines in List<>:

```
template<class Generator>
class List
{
  public:
    //publish Config so that others can access it
    typedef Generator::Config Config;
    // the rest of the code remains the same as above
    //...
};
```

Using the list generator, we can produce each of the four list types as follows:

```
typedef LIST_GENERATOR<>::RET SimpleList;
typedef LIST_GENERATOR<int, with_counter>::RET ListWithCounter;
typedef LIST_GENERATOR<int, no_counter, with_tracing>::RET TracedList;
typedef LIST_GENERATOR<int, with_counter, with_tracing>::RET TracedListWithCounter;
```

The list generator is an example of a very simple generator. In Section 10.3.1.5, we show you an advanced matrix generator, which performs parsing, computing default values for parameters, and assembling components.

8.8 Template Metaprograms and Expression Templates

The main idea behind the technique of expression templates [Vel95b] is the following: if you implement binary operators to be used in an expression, e.g. A+B+C, where A, B, and C are vectors, each operator does not return the resulting vector (or whatever the result is), but an object representing the expression itself. Thus, the operators are actually type constructors and if you implement them as overloaded template operators, they will derive the exact type of the expression at compile time. This type encodes the structure of the expression and you can pass it to a template metafunction which analyzes it and generates efficient code for it using the techniques presented in Section 8.3.

The arguments to the expressions could be abstract data types generated using a generator such as the one described in the previous section (i.e. LIST_GENERATOR<>). In this case, the configuration repositories contain all the metainformation about the ADTs they belong to. Please note that you can access the configuration repository at compile time, e.g.:

```
TracedListWithCounter::Config
```

Furthermore, you can provide traits about the types of the expression objects. These traits could define the algebraic properties of the involved operators. The ADT metainformation and the operator metainformation can then be used by the code generating metafunction to perform a complex analysis of the whole expression type and to generate highly-optimized code.

Section 10.3.1.7 will demonstrate some of these techniques.

8.9 Relationship Between Template Metaprogramming and Other Metaprogramming Approaches

As we stated at the beginning of this chapter, generative metaprograms manipulate program representations. Compilers and transformation systems usually manipulate abstract syntax trees (ASTs). Lisp allows us to write metaprograms which manipulate language constructs, more precisely, lists representing functions. Smalltalk and CLOS allow us to (dynamically) manipulate classes. Template metaprograms can manipulate (e.g. compose) program representations such as template classes and methods.

An important question is whether metaprograms have to be written in a different language than the programs being manipulated. This is often the case with transformation systems which usually provide a specialized transformation API which is independent of the structures being transformed. In the case of Lisp, CLOS, and Smalltalk, both the metaprograms and the programs being manipulated are part of the same language in a reflective way, i.e. base programs can call metaprograms and metaprograms can call base programs. In the case of template metaprogramming, we use a “metaprogramming sublanguage” of C++, which can manipulate types (e.g. compose class templates, select functions for inlining), but it has no access to their metainformation other than what was explicitly encoded using techniques such as traits. Also, template metaprograms run before the programs they generate and the base code cannot call the metacode.

In this context, we can view C++ as a two-level language. Two-level languages contain static code (which is evaluated at compile-time) and dynamic code (which is compiled, and later executed at run-time). Multi-level languages [GJ97] can provide a simpler approach to writing program generators (see e.g., the Catacomb system [SG97]).

8.10 Limitations of Template Metaprogramming

Template metaprogramming has a number of limitations which are discussed in Section 10.3.1.8.

Table 116 (in Section 10.3.2) compares the applicability of template metaprogramming and of the Intentional Programming System for implementing algorithmic libraries.

8.11 Historical Notes

The first documented template metaprogram was written by Unruh [Unr94]. The program generated prime numbers as compiler warnings. To our knowledge, the first publication on template metaprogramming was the article by Veldhuizen [Vel95a], who pioneered many of the template metaprogramming techniques by applying them to numerical computing. In particular, he’s been developing the numerical array package Blitz++ [Vel97], which we already mentioned in Section 7.6.4.

Currently, there is a number of other libraries using template metaprogramming, e.g. Pooma [POOMA], A++/P++ [BDQ97], MTL [SL98], and the generative matrix package described in Chapter 10.

The contribution of Ulrich Eisenecker and the author was to provide explicit meta-control structures. A further contribution is the combination of configuration repositories and metafunctions into generators. The concept of a meta-if in C++ was first published in [Cza97], but only after developing the workarounds for partial specialization (see Section 8.5), we were able to fully develop our ideas (since we did not have a compiler supporting partial specialization).

8.12 Appendix: A Very Simple Lisp Implementation Using Template Metaprogramming

```
#include <iostream.h>
#include "IF.h"

/*
The siplest Lisp implementation requires the following
primitive expressions:

 *data types: numbers, symbols, and lists
 *primitive functions: list constructor (cons),
  list accessors (first, rest), type-testing predicates
  (numberp, symbolp, listp), equivalence test (eq),
  algebraic operations on numbers (plus, minus, times,
  divide, rem), numeric comparison (eqp, lessp, greaterp)
*/

//**************************
// Symbolic data of Lisp:
//     *Symbolic atoms: numbers and symbols
//     *Symbolic expressions (S-expressions): lists
```

```
//

// In Lisp, we have numbers, symbols, and lists.
// We also need primitive methods for checking
// the type of a Lisp type.

//LispObject defines the interface
//for all lisp objects.
//Some of the members do not make
//sence for some subclasses.
//Yet they are required
//in order for numbers, symbols,
//and lists to be treated
//polymorphicaly by the compiler.

struct LispObject
{
  enum {
    N_VAL=0,
    ID=-1,
    IS_NUMBER=0,
    IS_SYMBOL=0,
    IS_LIST=0
  };
  typedef LispObject S_VAL;
  typedef LispObject FIRST;
  typedef LispObject REST;
};


struct NumberType : public LispObject
{
  //the following enums should accessed by the type testing predicates
  //numberp, symbolp, and listp, only.
  enum {
    IS_NUMBER=1,
    IS_SYMBOL=0,
    IS_LIST=0
  };
};

struct SymbolType : public LispObject
{
  enum {
    IS_NUMBER=0,
    IS_SYMBOL=1,
    IS_LIST=0
  };
};

struct ListType : public LispObject
{
  enum {
    IS_NUMBER=0,
    IS_SYMBOL=0,
    IS_LIST=1
  };
};

//numbers
//Numbers are defined as "wrapped" classes.
//We need a wrapper for numbers in
//order to be able to pass number to class templates
//expecting classes. The problem is that C++
//does not allow us provide two versions of a class template
//one with class and other with number parameters.
//Unfortunately, we can only handle integral types at the moment.
template <int n>
```

```
struct Int : NumberType
{
  enum { N_VAL=n };
};

//symbols
//symbols are defined as "wrapped" classes
//We need a wrapper for classes in
//order to be able provide them with IDs.
//IDs allow us to test for equivalence.
template <class Value, int SymbolID>
struct Symbol : public SymbolType
{
  typedef Value S_VAL;
  enum { ID=SymbolID };
};

//lists
//Lists are represented by nesting the list contructor
//cons defined later

//but first we need an empty list
struct EmptyList : public ListType {};


//******************************************
//Primitive function
//******************************************

//List constructor cons(X,L)
//
//an example of a list:
//Cons<Int<1>,Cons<Int<2>,Cons<Symbol<SomeClass,1>,EmptyList>>>
//In Lisp syntax, this list would look like this:
//cons(1,cons(2,cons("SomeClass",()))) = (1,2,"SomeClass")
//
//In this implementation of cons, we do not define RET
//since this would unnecessary complicate our C++ Lisp syntax.
//The list type is the type of an intantiated cons.

template <class X, class List>
struct Cons : public ListType
{
  typedef X FIRST;
  typedef List REST;
};


//list accessors first and rest

//first(List)
// returns the first element of List
//
//first is achieved through:
//Cons<Int<1>,Cons<Int<2>,Cons<Symbol<SomeClass,1>,
//  EmptyList>>>::FIRST = Int<1>

template <class List>
struct First
{
  typedef List::FIRST RET;
};

//rest(List)
// retuns the List without the first element

//rest is achieved through:
//Cons<Int<1>,Cons<Int<2>,Cons<Symbol<SomeClass,1>,
//  EmptyList>>>::REST
//    = Cons<Int<2>,Cons<Symbol<SomeClass,1>,EmptyList>>

template <class List>
struct Rest
{
  typedef List::REST RET;
};


//type testing predicates numberp, symbolp, listp

//numberp(X)
//this predicate returns true (i.e. 1) if X is a number
//otherwise false (i.e. 0)

template <class X>
struct Numberp
{
  enum { RET=X::IS_NUMBER};
};

//symbolp(X)
//this predicate returns true (i.e. 1) if X is a symbol
//otherwise false (i.e. 0)

template <class X>
struct Symbolp
{
  enum { RET=X::IS_SYMBOL};
};

//listp(X)
//this predicate returns true (i.e. 1) if X is a list
//otherwise false (i.e. 0)

template <class X>
struct Listp
{
  enum { RET=X::IS_LIST};
};

//empty(List)
//return true List=EmptyList
//otherwise false
template <class List>
struct IsEmpty
{
  enum { RET=0 };
};

struct IsEmpty<EmptyList>
{
  enum { RET=1 };
};


//equivalence test

//eq(X1,X2)
//return true if X1 is equivalent to X2
//otherwise false

template <class X1, class X2> struct Eq;
template <class N1, class N2> struct EqNumbers;
template <class S1, class S2> struct EqSymbols;
template <class L1, class L2> struct EqLists;
template <class X1, class X2>
struct Eq
{
  enum {
    RET=
      Numberp<X1>::RET && Numberp<X2>::RET ?
        EqNumbers<X1,X2>::RET :
        Symbolp<X1>::RET && Symbolp<X2>::RET ?
          EqSymbols<X1,X2>::RET :
          Listp<X1>::RET && Listp<X2>::RET ?
            EqLists<X1,X2>::RET :
            //should be 0, but VC++ 4.0 does not take it
            1 ?
              0:
              0
  };
};

struct Eq<EmptyList, EmptyList>
{
  enum { RET=1 };
};

struct Eq<LispObject,LispObject>
{
  enum { RET=1 };
};

template <class N1, class N2> //private
struct EqNumbers
{
  enum { RET=N1::N_VAL==N2::N_VAL };
};

template <class S1, class S2> //private
struct EqSymbols
{
  enum { RET=S1::ID==S2::ID };
};

template <class L1, class L2> //private
struct EqLists
{
  enum {
    RET=
      IsEmpty<L1>::RET ?
        0 :
        IsEmpty<L2>::RET ?
          0 :
          Eq<First<L1>::RET,First<L2>::RET>::RET ?
            EqLists<Rest<L1>::RET,Rest<L2>::RET>::RET :
            0
  };
};

struct EqLists<EmptyList, EmptyList>
{
  enum { RET=1 };
};
//to avoid a compile error
struct EqLists<LispObject, LispObject>
{
  enum { RET=1 };
};


//arithmetic functions
```

```
//plus
template <class N1, class N2>
struct Plus
{
  enum { RET=N1::VAL+N2::VAL };
};

//minus, times, and divide are defined similarly

//lessp, greaterp
//...


//let expression corresponds to typedef or enum

//define expression corresponds to a class template
//note: a template can be passed to a template by defining it
//as member of a concrete class (corresponds to passing functions)

//****************** end of primitives ***************

//********************************************
//convenience functions

//length(List)
template <class List>
struct Length
{
  enum { RET=Length<Rest<List>::RET>::RET+1 };
};

struct Length<EmptyList>
{
  enum { RET=0 };
};

//second(List)
template <class List>
struct Second
{
  typedef First<Rest<List>::RET>::RET RET;
};

//third(List)
template <class List>
struct Third
{
  typedef Second<Rest<List>::RET>::RET RET;
};

//n-th(n,List)
struct nil {};

template <int n, class List>
struct N_th
{
  typedef
    IF<n==1,
      EmptyList,
      Rest<List>::RET
    >::RET tail;

  typedef
    IF<n==1,
      First<List>::RET,
      N_th<n-1, tail>::RET
    >::RET RET;
};
```

```
template<>
struct N_th < 0, EmptyList >
{
  typedef nil RET;
};

//last(List)
template <class List>
struct Last
{
  typedef N_th<Length<List>::RET,List>::RET RET;
};


//append1(List,LispObject)
template<class List, class LispObject>
struct Append1
{
  typedef
    IF<IsEmpty<List>::RET,
      nil,
      LispObject
    >::RET new_object;

  typedef
    IF<IsEmpty<List>::RET,
      nil,
      Rest<List>::RET
    >::RET new_list;

  typedef
    IF<IsEmpty<List>::RET,
      Cons<LispObject, EmptyList>,
      Cons<First<List>::RET, Append1<new_list, new_object>::RET >
    >::RET RET;
};

template<>
struct Append1<nil, nil>
{
  typedef EmptyList RET;
};

//here should come some other functions:
//append(L1, L2), reverse(List), copy_up_to(X,L), rest_past(X,L), replace(X,Y,L), etc.



// just for testing:
class SomeClass1 {};
class SomeClass2 {};

typedef Symbol<SomeClass1,1> symb1;
typedef Symbol<SomeClass2,2> symb2;

void main()
{
  cout << "typedef Cons<Int<1>,Cons<Int<2>,Cons<Int<3>,EmptyList> > > l1;" << endl;
  typedef Cons<Int<1>,Cons<Int<2>,Cons<Int<3>,EmptyList> > > l1;
  cout << "typedef Cons<Int<1>,Cons<Int<2>,Cons<Int<3>,EmptyList> > > l2;" << endl;
  typedef Cons<Int<1>,Cons<Int<2>,Cons<Int<3>,EmptyList> > > l2;
  cout << "typedef Cons<Int<1>,Cons<Int<2>,Cons<Int<3>,Cons<Int<5>,EmptyList> > > > l3;" << endl;
  typedef Cons<Int<1>,Cons<Int<2>,Cons<Int<3>,Cons<Int<5>,EmptyList> > > > l3;
  cout << "typedef Cons<symb1,Cons<Int<2>,Cons<symb2,EmptyList> > > l4;" << endl;
  typedef Cons<symb1,Cons<Int<2>,Cons<symb2,EmptyList> > > l4;

  cout << "EqNumbers<Int<1>,Int<1> >::RET =" << endl;
  cout << EqNumbers<Int<1>,Int<1> >::RET << endl;

  cout << "EqSymbols<symb1,symb2 >::RET =" << endl;
  cout << EqSymbols<symb1,symb2 >::RET << endl;

  cout << "Eq<Int<1>,Int<1> >::RET =" << endl;
  cout << Eq<Int<1>,Int<1> >::RET << endl;

  cout << "Eq<symb1,symb2>::RET =" << endl;
  cout << Eq<symb1,symb2>::RET << endl;

  cout << "Third<l3>::RET::N_VAL =" << endl;
  cout << Third<l3>::RET::N_VAL << endl;

  cout << "N_th<4, l3>::RET::N_VAL =" << endl;
  cout << N_th<4, l3>::RET::N_VAL << endl;

  cout << "Last<l2>::RET::N_VAL =" << endl;
  cout << Last<l2>::RET::N_VAL << endl;

  cout << "Eq<l1,l3>::RET =" << endl;
  cout << Eq<l1,l3>::RET << endl;

  cout << "Length<l1>::RET =" << endl;
  cout << Length<l1>::RET << endl;

  cout << "typedef Append1<l1, Int<9> >::RET l5;" << endl;
  typedef Append1<l1, Int<9> >::RET l5;

  cout << "N_th<4, l5>::RET::N_VAL =" << endl;
  cout << N_th<4, l5>::RET::N_VAL << endl;

  cout << "Length<l5>::RET =" << endl;
  cout << Length<l5>::RET << endl;
}
```

8.13 Appendix: Meta-Switch Statement

```
//****************************************************************
//Authors: Ulrich Eisenecker and Johannes Knaupp
//****************************************************************

#include "../if/IF.H"

const int NilValue       = ~(~0u >> 1); //initialize with the smallest int
const int DefaultValue  = NilValue+1;

struct NilCase
{
  enum   { tag              = NilValue
         , foundCount       = 0
         , defaultCount     = 0
       };
  typedef NilCase RET;
  typedef NilCase DEFAULT_RET;
};

class MultipleCaseHits {};   // for error detection
class MultipleDefaults {};   // for error detection

template< int Tag, class Statement, class Next = NilCase >
struct CASE
{
  enum   { tag = Tag };
  typedef Statement statement;
  typedef Next next;
};
```

```
template< class Statement, class Next = NilCase >
struct DEFAULT
{
  enum   { tag = DefaultValue };
  typedef Statement statement;
  typedef Next next;
};

template <int Tag, class aCase, bool acceptMultipleCaseHits = false >
struct SWITCH
{
  typedef aCase::next   NextCase;

  enum  { tag          = aCase::tag
        , nextTag      = NextCase::tag
        , found          = (tag == Tag)
        , isDefault    = (tag == DefaultValue)
        };

  typedef IF< (nextTag != NilValue)
           , SWITCH< Tag, NextCase >
           , NilCase
           >::RET NextSwitch;

  enum  { foundCount   = found     + NextSwitch::foundCount
        , defaultCount = isDefault + NextSwitch::defaultCount
        };

  typedef IF< isDefault
           , aCase::statement
           , NextSwitch::DEFAULT_RET
           >::RET DEFAULT_RET;

  typedef IF< found
           , IF< ((foundCount == 1) || (acceptMultipleCaseHits == true))
              , aCase::statement
              , MultipleCaseHits
              >::RET
           , IF< (foundCount != 0)
              , NextSwitch::RET
              , DEFAULT_RET
              >::RET
           >::RET ProvisionalRET;

  typedef IF< (defaultCount <= 1)
           , ProvisionalRET
           , MultipleDefaults
           >::RET RET;
};
```

8.14 References

[BDQ97] F. Bassetti, K. Davis, and D. Quinlan. A Comparison of Performance-enhancing Strategies for Parallel Numerical Object-Oriented Frameworks. In *Proceedings of the first International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conference*, Marina del Rey, California, Dec 1997, http://www.c3.lanl.gov/~dquinlan/

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'95), ACM SIGPLAN Notices*, vol. 30, no. 10, 1995, pp. 285-299, http://www.softlab.is.tsukuba.ac.jp/~chiba/openc++.html

[Cza97] K. Czarnecki. Statische Konfiguration in C++. In *OBJEKTspektrum* 4/1997, pp. 86-91, see http://nero.prakinf.tu-ilmenau.de/~czarn

[Dict] *Webster's Encyclopedic Unabridged Dictionary of the English Language*. Portland House, New York, 1989

[GJ97] R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. In *Lisp and Symbolic Computation*, vol. 10, no. 2, 1997, pp. 113-158

[LKRR92] J. Lamping, G. Kiczales, L. H. Rodriguez Jr., and E. Ruf. An Architecture for an Open Compiler. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992, see http://www.parc.xerox.com/spl/groups/eca/pubs/complete.html

[Mye95] N. C. Myers. Traits: a new and useful template technique. In *C++ Report*, June 1995, see http://www.cantrip.org/traits.html

[POOMA] POOMA: Parallel Object-Oriented Methods and Applications. A framework for scientific computing applications on parallel computers. Available at http://www.acl.lanl.gov/pooma

[SG97] J. Stichnoth and T. Gross. Code composition as an implementation language for compilers. In *Proceedings USENIX Conference on Domain-Specific Languages (DSL'97)*, 1997

[SL98] J. G. Siek and A. Lumsdaine. A Rational Approach to Portable High Performance: The Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (FAST) Library. In *Proceedings of the ECOOP'98 Workshop on Parallel Object-Oriented Computing (POOSC'98)*, 1998, see http://www.lsc.nd.edu/

[TM] Meta-control structures for template metaprogramming available at http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm

[Unr94] E. Unruh. Prime number computation. ANSI X3J16-94-0075/SO WG21-462

[Vel95a] T. Veldhuizen. Using C++ template metaprograms. In *C++ Report*, vol. 7, no. 4, May 1995, pp. 36-43, see http://monet.uwaterloo.ca/blitz/

[Vel95b] T. Veldhuizen. Expression Templates. In *C++ Report*, vol. 7 no. 5, June 1995, pp. 26-31, see http://monet.uwaterloo.ca/blitz/

[Vel97] T. Veldhuizen. Scientific Computing: C++ versus Fortran. In *Dr. Dobb's Journal*, November 1997, pp. 34-41, see http://monet.uwaterloo.ca/blitz/

Part IV

PUTTING IT ALL TOGETHER: DEMRAL AND THE MATRIX CASE STUDY

Chapter 9 Domain Engineering Method for Reusable Algorithmic Libraries (DEMRAL)

9.1 What Is DEMRAL?

DEMRAL is a Domain Engineering method for developing algorithmic libraries, e.g. numerical libraries, container libraries, image processing libraries, image recognition libraries, speech recognition libraries, graph computation libraries, etc. The characteristics of algorithmic libraries include the following:

- main concepts are adequately captured as abstract data types (ADTs) and algorithms that operate on the ADTs;
- the ADTs often have container-like properties, e.g. matrices, images, graphs, etc.;
- there is usually a well developed underlying mathematical theory, e.g. linear algebra [GL96], image algebra [RW96], graph theory [BM76];
- the ADTs and the algorithms usually come in large varieties. For example, there are many different kinds of matrices (sparse or dense, diagonal, square, symmetric, and band matrices, etc.) and many specialized versions of different matrix factorization algorithms (e.g. the different specializations of the general LU include LDL^T and Cholesky factorizations with specialized versions for different matrix shapes and with and without pivoting);

DEMRAL supports the following library design goals:

- providing the client with a high-level, intentional library interface;
    - the client code specifies problems in terms of high-level domain concepts;
    - the interface supports large numbers of concept variants in an effective way;
    - the client code is able to specify problems at the most appropriate level of detail (of course, which level is most appropriate depends on the problem, the client, and other contextual issues);
- achieving high efficiency in terms of execution time and memory consumption;
    - the large number of variants should not have any negative effect on the efficiency;
    - possibilities of optimizations should be analyzed and useful optimizations should be implemented;

  - unused functionality should be removed and, whenever possible, static binding should be used;

- achieving high quality of library code;

  - high adaptability and extendibility;

  - minimal code duplication;

  - minimal code tangling;

DEMRAL has been developed as a specialization of the ODM method (see Section 3.7.2), while applying ODM in the development of the matrix computation library described in Chapter 10. DEMRAL combines ideas from several areas including

- Domain Engineering (Chapter 3),

- Generators and Metaprogramming (Chapter 6),

- Aspect-Oriented Programming (Chapter 7), and

- Object-Oriented Software Development.

DEMRAL is an ongoing effort. We are currently refining the method based on our experience with analyzing and implementing matrix algorithms. We still need to test DEMRAL on other domains.

This chapter gives a high-level overview of the method and explains its basic concepts. Chapter 10 contains a comprehensive case study of applying DEMRAL in the development of a matrix computation library.

9.2 Outline of DEMRAL

The major activities of DEMRAL are shown in Figure 133. Of course, the development process is an iterative and incremental one.[140] During development, the various activities may be scheduled and rescheduled in arbitrary order. For example, identification of key concepts (1.2.1.) and feature modeling (1.2.2.) usually require many iterations. Also, the domain definition often needs to be revised based on insights from Domain Modeling, Domain Design, and Domain Implementation. Similarly, feature models are usually refined and revised in Domain Design and Implementation. Furthermore, activities may be revisited because of external changes, e.g. changes of the stakeholder goals, environmental changes, etc. We encourage prototyping at any time since it represents an excellent tool for gaining a better understanding of new ideas and for evaluating alternative solutions.

Thus, the process outlined in Figure 133 should be viewed as a default process and a starting point for the user, who should customize it to suite his needs. However, we use this default outline to structure the documentation of the matrix computations case study in Chapter 10.

Reenskaug et al. give an excellent characterization of the role of process outlines [Ree96]:

> *"Documentation is by its nature linear, and must be strictly structured. Software development processes are by their nature creative and exploratory, and cannot be forced into the straightjacket of a fixed sequence of steps. In an insightful article, Parnas et al. state that many have sought a software process that allows a program to be derived systematically from a precise statement of requirements [PC86]. Their paper proposes that although we will not succeed in designing a real product that way, we can produce documentation that makes it appear as if the software was designed by such a process.*
>
> *The sequences of steps we describe in the following sections and in the rest of the book are therefore to be construed as default work processes and suggested documentation*

structures. We also believe that you will have to develop your own preferred sequence of steps, but you may want to take the steps proposed here as a starting point."

1. Domain Analysis
    - 1.1. Domain Definition
        - 1.1.1. Goal and Stakeholder Analysis
        - 1.1.2. Domain Scoping and Context Analysis
            - 1.1.2.1. Analysis of application areas and existing systems (i.e. exemplars)
            - 1.1.2.2. Identification of domain features
            - 1.1.2.3. Identification of relationships to other domains
    - 1.2 Domain Modeling
        - 1.2.1. Identification of key concepts
        - 1.2.2. Feature modeling of the key concepts (i.e. identification of commonalities, variabilities, and feature dependencies/interactions)
2. Domain Design
    - 2.2. Identification of the overall implementation architecture
    - 2.1. Identification and specification of domain-specific languages
3. Domain Implementation (implementation of the domain-specific languages, language translators, and implementation components)

Figure 133 *Outline of DEMRAL*

The DEMRAL activities have been derived from the ODM phases and tasks (see Figure 15 and Table 5 in Section 3.7.2.1). However, there are several differences:

- *Different divisions into top-level activities*: DEMRAL subscribes to the more widely accepted division of Domain Engineering into Domain Analysis, Domain Design, and Domain Implementation. This division does not represent any substantial difference to ODM. The ODM phases (see Figure 15) are easily mapped to these top-level activities: Plan Domain and Model Domain correspond to Domain Analysis. Scope Asset Base and Architect Asset Base correspond to Domain Design. Finally, Implement Asset Base corresponds to Domain Implementation.

- *Stronger focus on technical issues*: A unique feature of ODM is its focus on organizational issues. The description of DEMRAL, on the other hand, is more focused on technical issues. Of course, whenever appropriate, DEMRAL can be extended with the organizational tasks and workproducts of ODM.

- *Only a subset of ODM tasks and workproducts*: As a specialization of a very general Domain Engineering method, DEMRAL covers only a subset of the ODM tasks and workproducts. Additionally, the DEMRAL process outline is less detailed than the ODM process outline. In a sense, DEMRAL can be seen as a "lightweight" specialization of ODM.

The special features of DEMRAL as a specialization of ODM include

- focus on the two concept categories: ADTs and algorithms;

- specialized feature starter sets for ADTs and algorithms;

- application of feature diagrams for feature modeling (ODM does not prescribe any particular feature modeling notation);

- focus on the development of DSLs.

In the remaining sections of this chapter, we describe each of the activities of DEMRAL. A comprehensive case study of applying DEMRAL to the domain of matrix computations is presented in Chapter 10.

9.3 Domain Analysis

Domain Analysis involves two main activities: Domain Definition and Domain Modeling. They are described in the following two sections.

9.3.1 Domain Definition

The first activity of Domain Definition is the identification of goals and stakeholders. The complexity of this activity depends on the size and the context of the project. We will not further elaborate on this activity. We assume that the result of this activity are cross-checked, prioritized lists of goals and stakeholders.

The next activity is to determine the scope and characterize the contents of the domain by defining its domain features. Two important sources of domain features are

- the analysis of the application areas of the systems in the domain and
- the analysis of the existing exemplar systems.

For example, if our goal is to define the domain of matrix computation libraries, we need to analyze the application areas of matrix computations and the features of existing matrix computation libraries. The results of these analyses are shown in Table 13 through Table 16. Please note that the format of the application areas tables (i.e. Table 13 and Table 14) and the existing libraries tables (i.e. Table 15 and Table 16) is similar. The conclusion of both analyses is that the main features of matrix computation libraries are different types of matrices and different types of computations they implement.

Domain feature diagram

The results of the analysis of the application areas and the exemplar systems are summarized in a *domain feature diagram*. The domain feature diagram for the domain of matrix computation libraries is shown in Figure 137. This diagram describes which features *are* part and which *can* be part of a matrix computation library. For example, band matrices are optional, but at least dense or sparse matrices have to be implemented. A matrix computation library can also implement both dense and sparse matrices. This is indicated in the diagram by the fact that dense and sparse matrices have been shown as or-features. Please note that alternative *concept features*, e.g. dense or sparse of the concept *matrix*, emerge in the *domain feature* diagram as or-features.

We found it useful to annotate the domain features with priorities. There are at least three important factors influencing the priority of a domain feature:

- typicality rate of the domain feature in the analyzed application areas,
- typicality rate of the domain feature in the analyzed exemplar systems, and
- importance of the domain feature according to the stakeholder goals.

The priorities are assigned on a rather informal basis. However, they are still very useful. They indicate the importance of the various parts of a domain and will help to decide which parts of the domain will be implemented first. Of course, the priorities may have to be adjusted as the goals evolve and more knowledge about the domain becomes available over time. The domain feature diagram represents a definition of a domain from the probabilistic viewpoint (see Section 2.2.3). It is a concise and convenient style of defining a domain.

Domain dictionary and domain knowledge

Right from the beginning of Domain Analysis, it is essential to establish a *domain dictionary* (see e.g. Section 10.4) and a register of sources of *domain knowledge* (see e.g. Section 10.1.1.2.2). As the analysis progresses, both the dictionary and the register need to be updated.

Analogy or support domains

Finally, we analyze related domains, such as *analogy* or *support domains*. As you remember from Section 3.6.3, an analogy domain has significant similarities to the domain being analyzed and thus may provide some useful insights about the latter domain. A support domain, on the other hand, may be used to express some aspects of the domain being analyzed. Examples of both types of related domains can be found in Section 10.1.1.2.6.

9.3.2 Domain Modeling

Domain Modeling involves two main activities: *identification of key concepts* and *feature modeling of the key concepts*. We describe these activities in the following two sections.

9.3.2.1 Identification of Key Concepts

By definition, DEMRAL focuses on domains whose main categories of concepts are ADTs and algorithms. Identifying the key ADTs is usually quite simple, e.g. the key ADTs in matrix computation libraries are matrices and vectors and the key ADTs in image processing libraries are various kinds of images.

An ADT defines a whole family of data types. Thus, a matrix ADT defines a family of matrices (e.g. sparse, dense, diagonal, square, symmetric, etc.). Similarly, we usually have whole families of algorithms operating on the ADTs. For example, matrix computation libraries may include factorization algorithms and algorithms for solving eigenvalue problems. Furthermore, each general version of a factorization algorithm may be specialized for matrices of different properties, e.g. there is over a dozen of important specializations of the general LU factorization (see Section 10.1.2.2.2).

Basic ADT operations and the algorithm families

We make a distinction between *basic ADT operations* and the *algorithm families* which access the ADTs through accessing operations and the basic operations. Examples of basic operations in matrix computations are matrix addition, subtraction, and multiplication. We analyze basic operations together with the ADTs since they all define a cohesive *kernel algebra*. We usually implement the kernel algebra in one *component* which is separate from the more complex algorithm families. For example, a matrix component would include a matrix type and a vector type and the basic operations on matrices and vectors.

It is important to note that if we model types using OO classes, we usually do not define the basic operations directly in the class interfaces, but rather as free-standing operators or operator templates.[141] This, of course, is only possible if the programming language we use supports free-standing operators (e.g. C++). The class interfaces should include a minimal set of necessary methods, e.g. accessing methods for accessing directly stored state or abstract (i.e. computed) state. This way, we avoid “fat class interfaces” and improve modularity since we can define families of operators in separate modules. Furthermore, it is easier to add new operators. Another reason for defining operators outside the class definitions is that they often cannot be thought of as a conceptual part of just one class. For example, in the expression M*V, where M is a matrix and V is a vector, the multiplication operation * is equally part of the matrix interface and the vector interface. If dynamic binding is required, * is best implemented as a dynamic multi-method[142], otherwise we can implement it as a free-standing operator or as a (possibly specialized) operator template. Furthermore, if expression optimizations are required, the operators are best implemented as *expression templates* (see Section 8.8). Indeed, for our matrix computation library, we will implement matrix operations using expression templates (Section 10.3.1.7).

The more complex algorithms (e.g. solving systems of linear equations) are often more appropriately implemented as classes rather than procedures or functions (see e.g. [Wal97]). This allows us to organize them into family hierarchies. The algorithms usually call both basic operations and ADT accessing methods.

9.3.2.2 Feature Modeling

The purpose of feature modeling is to develop feature models of the concepts in the domain. Feature models define the common and variable features of concept instances and the dependencies between the features.

We already discussed how to perform feature modeling in Chapter 5. The only addition of DEMRAL to what we said there are the feature starter sets for ADTs and algorithms. These are listed in Sections 9.3.2.2.1 and 9.3.2.2.2.

9.3.2.2.1 Feature Starter Set for ADTs

The following is the feature starter set for ADTs:

- *Attributes*: Attributes are named properties of ADT instances, such as the number of rows in a matrix or the length of a vector. Important features of an attribute are its type and whether it is mutable or not (i.e. if its possible to change its value). Other features concern the implementation of an attribute, e.g. whether the attribute is stored directly or computed, whether its value is cached or not, whether it is *owned* by the ADT or not (see [Eis96]). If an attribute is owned by the ADT, the ADT is responsible for creating and destroying the values of the attribute. Also, accessing a value owned by an ADT usually involves copying the value. Each of the features of an attribute could be parameterized.

- *Data structures*: The ADT may be implemented on top of some complex data structures. This is especially the case for container-like ADTs such as matrices or images. We will discuss this aspect later in more detail.

- *Operations*: Examples of operations are accessing operations and kernel algebra operations (i.e. basic operations used by more complex algorithms). Other operations may be added during the analysis of algorithms. In addition to operation signatures (i.e. operation name, operands and operand types), we need to analyze various possible implementations of operations. In particular, possible optimizations need to be documented. For example, matrix operations may be optimized with respect to the shape of the operands. Also the optimization of matrix expressions based on the expression structure is possible (e.g. loop fusing). Binding mode (e.g. static or dynamic) and binding time (e.g. compile time or runtime) are other features of an operation (see Section 5.4.4.3). Binding mode and binding time of an operation can be parameterized (see e.g. Section 7.10).

- *Error detection, response, and handling*: Error detection is often performed in the form of pre-, intra-, and post-condition[143] and invariant checking. What is an error and what is not may depend on the usage context of an ADT. Thus, we may want to parameterize error detection (also referred to as *error checking*). In certain contexts, it may be also appropriate to switch off checking for certain error condition. For example, if the client of a container is guaranteed to access elements using only valid indices, no bounds checking is necessary. Once an error condition has been detected, various responses to the error condition are possible: We can throw an exception, abort the program, issue an error report [Eis95]. Finally, on the client side, we need to handle exceptions by performing appropriate actions. An important aspect of error response and handling is *exception safety*. We say that an ADT is exception safe if exceptions thrown by the ADT code or the client code do not leave the ADT in an undefined, broken state. Interestingly, this important aspect has been addressed in the Standard Template Library (STL) only in its final standardization phase.

- *Memory management*: By memory management we mean approaches to allocating and relinquishing memory and various approaches to managing virtual memory. We can allocate memory on the stack or on the heap using standard mechanisms available in most languages. We can also manage memory ourselves by allocating large chunks of memory (so-called *memory pool*) at once and allocating objects within this customarily managed memory. We can also use automatic memory management approaches, e.g. reference counting or some more sophisticated garbage collection approach (see e.g. [Wil92]). In a multithreading environment, it may be useful to manage thread-specific memory, i.e. per-thread memory where a thread can allocate its own objects not shared with other threads. Other kinds of memory are memory shared among a number of processes and persistent store. Memory management interacts with other features, e.g. exception safety: One aspect of exception safety is making sure that exceptions do not cause memory leaks. The memory allocation aspect of container elements in STL is parameterized in the form of *memory allocators* (see [KL98]), which can be passed to a container as a template parameter. Paging and cache behaviors often need to be tuned to the requirements of an application (e.g. blocking in matrix algorithms; see Section 10.1.2.2.1.3.1). An important aspect of memory management in databases is location control and clustering, i.e. where and how the data is stored.

- *Synchronization*: If we want to use an ADT in a multithreaded environment, we have to synchronize the access to shared data. This is usually done by providing appropriate synchronization code (see Chapter 7). Synchronization variability may include not only different synchronization constraints but also different implementation strategies. For example, synchronization can be implemented at different levels: the interface level of an ADT or the internal data level. The interface level locking is, as a rule, less complex than the data level locking. On the other hand, data level locking allows more concurrency. For this reason, data level locking is usually used in large collections, e.g. in databases. If we also want to use an ADT in a sequential environment, it should be possible to leave out its synchronization code entirely, or, in some cases, replace it by error checking code. Thus, there is also an interaction between synchronization and error checking. We already saw an example of this interaction in Section 7.4.4.

- *Persistency*: Some application may require the ability to store an ADT on disk (e.g. in a file or in a database). In such cases, we need to provide mechanisms for storing appropriate parts of the state of an ADT as well as for restoring them. This aspect is closely related to memory management.

- *Perspectives and subjectivity*: Different stakeholders and different client programs usually have different requirements on an ADT. Thus, we may consider organizing ADTs into subjects based on the different perspectives of the stakeholders or client programs on the ADTs. Some of the perspectives may delineate different parts of an ADT according to their service (e.g. printing, storing, etc.) and other perspectives may require different realizations of the same service, e.g. different attributes and operations, attribute and operation implementations, etc. In this case, we might want to develop a model of the relevant subjective perspectives and define ADT features that correspond to these perspectives.

If the ADT has a container-like character (e.g. matrix or image), we also should consider the following aspects:

- *Element type*: What is the type of the elements managed by the ADT?

- *Indexing*: Are the elements to be accessed through an index (e.g. integral index, symbolic key, or some other, user-defined key)?

- *Structure*: How are the elements stored? What data structures are used? For example, the structure of a matrix has a number of subfeatures such as entry type, shape, format, and representation (see Section 10.1.2.2.1.3).

An ADT may have a number of different interfaces. For example, a matrix will have a base interface including the basic matrix operations as well as a configuration interface, which allows us to select different storage formats, error checking strategies, etc. The configuration interface may be needed at different times, e.g. compile time or runtime. Also, the base interface may be organized into subinterfaces and be configurable (e.g. in order to model subjectivity).

Some features in the feature starter set are aspects in the AOP sense, e.g. synchronization or memory management.

Of course, the starter set should be extended as soon as new relevant features are identified.

9.3.2.2.2 Feature Starter Set for Algorithms

The feature starter set for algorithms includes the following feature categories:

- *Computational aspect*: This is the main aspect of an algorithm: the abstract, text-book formulation of the algorithm without the more involved implementation issues. We may specify this aspect using pseudocode. Furthermore, we have to investigate the relationships between algorithms, e.g. specialization and use, and organize them into families. It may also be useful to classify them according to the general algorithm categories such as search algorithms, greedy algorithms, divide-and-conquer algorithms, etc. (see Figure 100, Section 6.4.4). For example, iterative methods in matrix computations are search algorithms.

- *Data access*: Algorithms access ADTs through accessing operations and basic operations. Careful design of the basic operations is crucial in order to achieve both flexibility and good performance. For example, algorithms benefit from optimizations of basic operations. We can use different techniques in order to minimize the coupling between algorithms and data structures, e.g. iterators [GHJV95, MS96] and data access templates (or data accessors) [KW97].

- *Optimizations*: There are various opportunities for domain-specific optimizations, e.g. optimization based on the known structure of the data, caching, in-place computation (i.e. storing result data in the argument data to avoid copying), loop restructuring, algebraic optimizations, etc.

- *Error detection, response, and handling*: We discussed this aspect in the previous section.

- *Memory management*: Algorithms may also make direct calls to memory management services for tuning purposes.

There are also more specialized domain-specific features, e.g. pivoting strategies in matrix computations (see Section 10.1.2.2.2).

An aspect we did not discuss is parallelization, which is relevant in high-performance areas such as scientific computing. Parallelization is a very advanced topic and we refer the interested reader to [GO93].

9.4 Domain Design

The purpose of Domain Design in DEMRAL is to develop a library architecture consisting of a decomposition into packages and the specifications of the user DSLs. The DSL specifications include both abstract syntax and implementation specifications in a form which can serve as a basis for their implementation using some appropriate language extension technology. Domain Design builds on the results of Domain Modeling, i.e. feature models of different aspects of ADTs and algorithm families.

Since DSLs are central to Domain Design in DEMRAL, we first review the advantages of DSLs, the sources of DSLs, and implementation technologies for DSLs in Section 9.4.1. Then we describe the Domain Design activities in DEMRAL in Section 9.4.2.

9.4.1 Domain-Specific Languages Revisited

A domain-specific language (DSL) is a specialized, problem-oriented language. Similarly as in the case of domains, some DSLs are more general modeling DSLs (e.g. a DSL for expressing synchronization)[144] and others are more specialized, application-oriented DSLs (e.g. DSL for defining financial products).

Compared to conventional libraries, DSLs have a number of advantages. We already summarized them in Section 7.6.1, but let us restate some main points (see Table 12 for other advantages):

- *Intentional representation*: DSLs are designed to allow a direct, well-localized expression of requirements without obscure language details. Also, they enforce the inclusion of all relevant design information which would otherwise be lost if we used a general-purpose language. Thus, programs written in DSLs are easier to analyze, understand, modify, extend, reuse, and maintain.

- *Error checking*: DSLs enable error checking based on domain knowledge.

- *Optimizations*: DSLs allow optimizations based on domain knowledge. Such optimizations are usually much more effective than low-level optimization at the level of a general-purpose language (see Section 6.4.1). They are the key to being able to write cleanly designed, abstract code and still have it compiled into a high-performance executable. For example, if we implement matrix addition naively in an OO language using overloaded binary operators, the performance of such implementation will be unacceptable. Given the expression M1+M2+M3, the sum of the matrices M2 and M3 will be computed first and the intermediate result will be passed as the second argument to the first plus operation in the expression. Unfortunately, intermediate results can cause a significant overhead, especially if the matrices are large. Therefore, such an expression is often manually implemented as a pair of nested loops iterating through rows and columns and adding the corresponding elements of all matrices at once, which does not require any intermediate results and an extra pair of loops. The manual implementation, despite its better performance, is much more difficult to maintain than the original expression M1+M2+M3. Domain specific optimizations offer a solution to this dilemma. In our case, we have to extend the compilation process with the matrix expression optimization computing the efficient implementation from the abstract expression automatically. Thus, we can write abstract expressions and get maximum performance at the same time.

The conclusion of Chapter 7 was that aspectual decomposition requires specialized language constructs for dealing with crosscutting. Also, we concluded that language extensions are well suited for capturing aspects in an intentional way.

Furthermore, we saw in Section 7.6.3 that the distinction between languages and language extensions disappears if we subscribe to the idea of modular language extensions. In this case, instead of using some (possibly extended) fixed language, we rather use configurations of language extensions. DSLs as language extensions have three important advantages (also see Section 7.6.1):

- *Reusability*: Breaking monolithic languages into modular language extensions allows us to use them in different configurations.

- *Scalability*: We can develop a small system using a small configuration of necessary language extensions. When the system grows to encompass new aspects, we can add new language extensions addressing the new aspects to the initial configuration. Modular language extensions avoid the well-known problems of large and monolithic DSLs that are hard to evolve (see e.g. [DK98]).

- *Fast feature turnover*: Modular language extensions are a faster vehicle for distributing new language features than closed compilers. Also, modular language feature extensions have to survive based on their merits since they can be loaded and unloaded at any time. This is not the case for language features of fixed languages, in which case it is usually not possible to get rid of questionable features once they are part of a language (since there may be users depending on them).

An important issue is how to compose the language extensions. We could classify the composition mechanisms into three categories:

- *Embedding*: Small sections of code written in one language extension are "embedded" in the code written in other language extensions, e.g. embedded SQL statements in a general-purpose programming language. The boundaries between the embedded code and the surrounding code can be usually easily identified since both codes use a different style, paradigm, etc.

- *"Seamless" integration*: One language extension "naturally" extends another one. An example of such integration is the extension of Java 1.1 with *inner classes* in Java 1.2.

- *Referential integration*: Different modules are written in different language extensions. The modules reference each other at appropriate join points. A good example of referential integration is the composition of Jcore and Cool modules in AspectJ (see Section 7.5.1).

It is important to note that, from the linguistic viewpoint, conventional libraries of procedures or classes extend a programming language within its syntax since they introduce new vocabulary (i.e. new abstractions) for describing problems at a higher level. Such extensions are adequate as long as we do not require

- syntax extensions,

- semantic extensions or modifications of language constructs,

- domain-specific optimizations,

- domain-specific error checking, and

- domain-specific type systems.

Some languages allow implementing domain-specific optimizations, error checking, and type systems without leaving the syntax of the language, e.g. C++ thanks to static metaprogramming. Other languages allow us to extend their syntax and semantics by extending the libraries defining them, e.g. Smalltalk and CLOS (see Section 7.4.7).

In general, technologies for implementing language extensions covering domain-specific optimizations, error checking, type systems, syntactic extensions, and modifications of language constructs include the following:

- *Preprocessors*: Preprocessors are popular for extending existing programming languages. They usually expand some embedded macros into the target programming language, which is referred to as the host language. The advantage of preprocessors is that they do not have to understand the host language entirely and thus their development cost can be much smaller than the cost of developing a compiler. Moreover, they can be simply deployed in front of any favorite compiler for the host language. Unfortunately, this advantage is also their disadvantage. Errors in the target source are reported by the compiler in terms of the target source and not in terms of the source given to the preprocessor. Also, debuggers for the host language do not understand the extended language. Thus, preprocessors usually do not adequately support the programmer. Furthermore, if a preprocessor does not completely understand the host language, macros cannot utilize other information contained in the source (e.g. host language constants, constant expressions, etc.) and thus many important kinds of domain-specific optimizations cannot be implemented. This problem could be solved if the preprocessor had access to the internal representation of the compiler. In an extreme case, we could imagine a pipeline of preprocessors, all storing their metainformation in one repository. Indeed, such an architecture can be viewed as a modularly extendible compiler, which we mention in the last point.

- *Languages with metaprogramming support*: Some languages have built-in language extension capabilities. Templates in C++ allow us to implement domain-specific optimizations while staying within the C++ syntax and semantics. Reflective languages such as Smalltalk or CLOS allow us to implement any kinds of extensions since their definition is accessible to the programmer as a modifiable and extendable library. However, the languages do not provide the kind of architecture for modular extensions as modularly extendible compilers or programming environments do.

- *Modularly extendible compilers and modularly extendible programming environments*: An example of a system in this category is the Intentional Programming (IP) environment described in Section 6.4.3. Language extensions in IP are implemented as extension libraries which extend editors, compilers, debuggers, etc.

The latter two technologies allow us to develop active libraries, i.e. libraries which, in addition to the runtime code, also include domain-specific optimizations, or any other kind of compiler or programming environment extensions (see Section 7.6.4).

In the following text, whenever we use the term DSL, we actually mean a domain-specific language extension.

9.4.2 Domain Design Activities

Domain Design in DEMRAL involves the following activities:

- identify packages,
- identify user DSLs,
- identify interactions between DSLs,
- scope DSLs, and
- specify DSLs.

These activities are described in following five sections.

9.4.2.1 Identify packages

We divide the library to be developed in a number of packages. Packages serve several of purposes:

- defining the high-level modules of the library which helps to minimize dependencies and improve understandability;
- assigning different packages to different developers for concurrent development;
- selective import of required packages by different applications.

The usual strategy in DEMRAL is to put each ADT and each algorithm family into a separate package. We can use the UML package diagrams in order to represent the package view on the library under development.

9.4.2.2 Identify User DSLs

User DSLs are the DSLs provided by the library to their users as Application Programmer's Interfaces (APIs). There are two important kinds of user DSLs in DEMRAL:

Configuration DSLs

- *Configuration DSLs*: Configuration DSLs are used to configure ADTs and algorithms. We discuss configuration DSLs in Section 9.4.3.

Expression DSLs

- *Expression DSLs*: Expression DSLs are DSLs for writing expressions involving ADTs and operations on them, e.g. matrix expressions. We discuss expression DSLs in Section 9.4.4

Other, more problem-specific DSLs are also possible, e.g. a language extension for expressing pivoting in matrix computation algorithms (see [ILG+97]).

9.4.2.3 Identify Interactions Between DSLs

Next, we need to address the following question: What kind of information has to be exchanged between the implementations of the DSLs? For example, the implementation of a matrix expression DSL will need access to different properties of matrices (e.g. element type, shape, format, etc.), which are described by the matrix configuration DSL. This information is necessary in order to implement the operations contained in an expression by selecting optimal algorithms for the given matrix arguments. Furthermore, it is used to compute the matrix types for the intermediate results.

9.4.2.4 Scope DSLs

The features to be covered by an implementation of a DSL have to be selected from the feature models based on the priorities recorded in the feature models and the current project goals and resources.

9.4.2.5 Specify DSLs

Abstract and concrete syntax

We specify a language by specifying its syntax and semantics. At this point, we will only specify the *abstract syntax* of each DSL and leave the *concrete syntax* (also referred to as the *surface syntax*) to Domain Implementation. The difference between abstract and concrete syntax is shown in Figure 134. The abstract syntax of a language describes the structure of the abstract syntax trees used to represent programs in the compiler (or another language processor), whereas the concrete syntax describes the structure of programs displayed on the screen. Technologies such as Intentional Programming allow us to easily implement many alternative concrete syntaxes for one abstract syntax. We specify abstract syntax in the form of an abstract grammar (see e.g. Figure 134) and we use the *Backus-Naur Form* (BNF; see e.g. [Mey90]) for the concrete syntax.

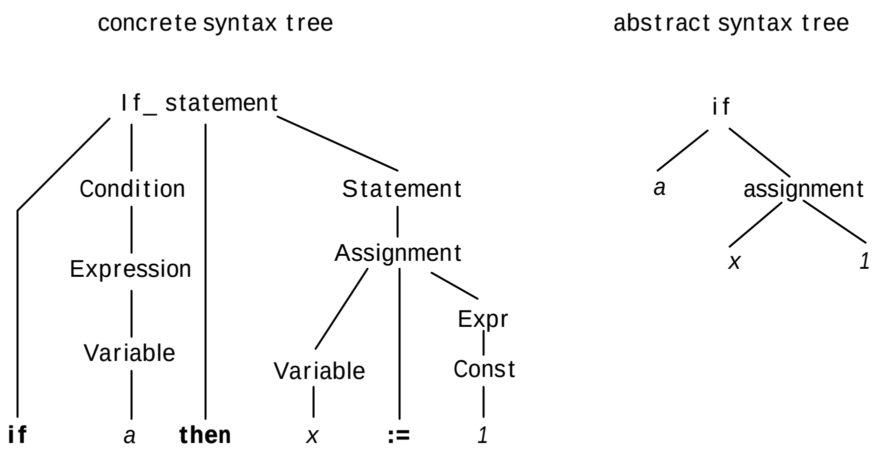


Figure 134 *Concrete and abstract syntax tree for the statement if a then x:= 1 (adapted from [WM95])*

The specification of the semantics of a language is a more complex task. Meyer describes five fundamental approaches to specifying semantics [Mey90]:

- *Attribute grammars*, which extends the grammar by a set of rules for computing properties of language constructs.
- *Translational semantics*, where the semantics is expressed by a translation scheme to a simpler language.
- *Operational semantics*, which specifies the semantics by providing an abstract interpreter.

- *Denotational semantics*, which associates with every programming construct a set of mathematical functions defining its meaning.
- *Axiomatic semantics*, which for a programming language defines a mathematical theory for proving properties of programs written in this language.

Each of these approaches has its advantages and disadvantages. In any case, however, formal specification of the semantics of a language is an extremely laborious enterprise.

A simple and practicable approach is to specify how to translate DSL programs into pieces and patterns of code in some appropriate programming language or pseudo code.

Finally, we have to specify how the different DSLs should be integrated (e.g. embedding, seamless integration, or referential integration).

Since the two most important categories of DSLs in DEMRAL are configuration DSLs and expression DSLs, we discuss them in Sections 9.4.3 and 9.4.4, respectively.

9.4.3 Configuration DSLs

A configuration DSL allows us to specify a concrete instance of a concept, e.g. data structure, algorithm, object, etc. Thus, it defines a family of artifacts, just as a feature model does. Indeed, we derive a configuration DSL of a concept from a feature model by tuning it to the needs of the reuser.

Implementation components configuration language (ICCL)

We usually implement a configuration using a generative component (see Section 6.2) or a configurable runtime component. The model of a generative component is shown in Figure 135. An instance specification in a configuration DSL (e.g. specification of a matrix: complex elements, lower-diagonal shape, stored in an array, and with bounds checking) is given to the generative component which assembles the concrete component instance (e.g. the concrete matrix class) from a number of *implementation components* (e.g. parameterized classes implementing element containers, bounds checkers, adapters, etc.) according to the specification. The implementation components can be connected only in certain ways. This is specified by an *implementation components configuration language* (*ICCL*).

A configurable runtime component has the same elements as the generative component. The only difference is that the finished configuration, the translator, and the implementation components are all contained in one configurable runtime component. Of course, a generative component may also generate a configurable runtime component, which implements a subset of its original configuration DSL and thus includes a subset of its implementation components and a smaller translator.

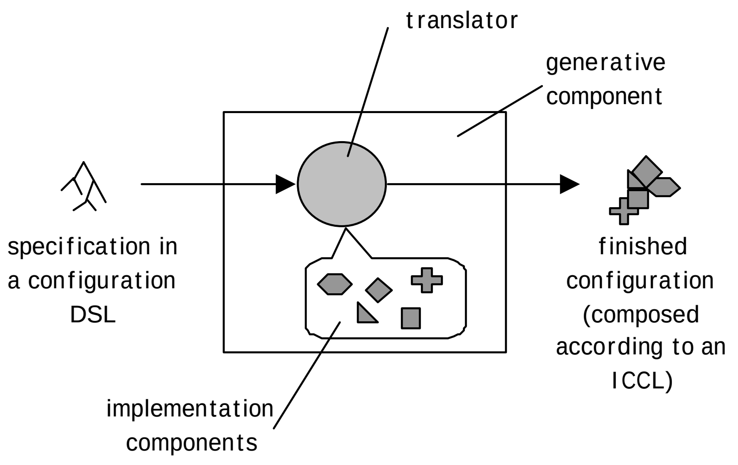

translator
generative component
specification in a configuration DSL
finished configuration (composed according to an ICCL)
implementation components

Figure 135 *Generative component implementing a configuration DSL*

Thus, the complete specification of a configuration DSL consists of a DSL grammar (see Section 10.2.3 and Figure 150), the specification of the ICCL (see Section 10.2.4), and the specification of how to translate statements in the configuration DSL into ICCL statements (see Section 10.2.5).

Why do we need both configuration DSLs and ICCLs? The reason is that both kinds of languages have different foci. The focus of a configuration DSL is to allow the user (or the client program) of a component to specify his needs at a level of detail that suites him best. The focus of an ICCL is to allow maximum flexibility and reusability of the implementation components. Thus, the configuration DSL describes the problem space, whereas the ICCL describes the solution space.

Direct or computed defaults

As stated, the configuration DSL should allow the user of a component to specify his needs at a level of detail that suites him best. He should not be forced to specify any implementation-dependent details if he needs not to. This way, we make sure that a client program does not introduce any unnecessary dependencies on the implementation details of the server component. For example, a client might just request a matrix from a generative matrix component. The matrix component should produce a matrix with some reasonable defaults, e.g. rectangular shape, real element values, dynamic row and column numbers, etc. In general, the client should be able to leave out a feature in a specification, in which case the feature should be determined by the generative component as a *direct default* or a *computed default* (i.e. a default determined based on some other specified features and other feature defaults). The client could be more specific and specify features stating some usage profile, e.g. need dense or sparse matrix or need space- or speed-optimized matrix. The next possibility would be to provide more precise specifications, e.g. the matrix density is 5% nonzero elements. Furthermore, the client should be able to specify some implementation features directly, e.g. what available storage format the matrix should use. Finally, it should be possible for the client to contribute its own implementation of some features, e.g. its own storage format. The different levels of detail for specifying a configuration are summarized in Figure 136.

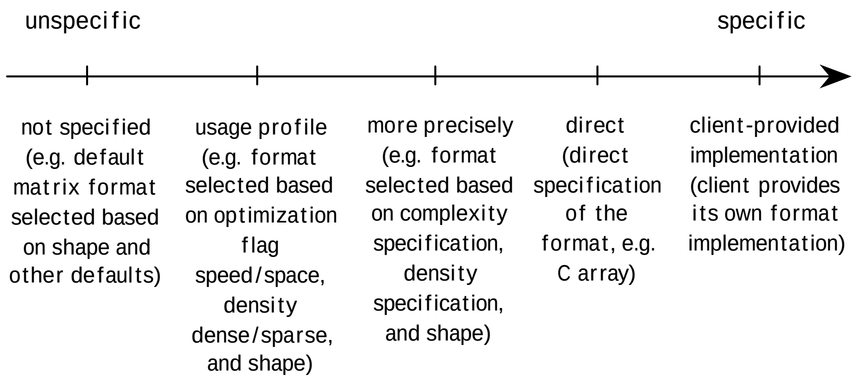

unspecific
specific
not specified (e.g. default matrix format selected based on shape and other defaults)
usage profile (e.g. format selected based on optimization flag speed/space, density dense/sparse, and shape)
more precisely (e.g. format selected based on complexity specification, density specification, and shape)
direct (direct specification of the format, e.g. C array)
client-provided implementation (client provides its own format implementation)

Figure 136 *Different levels of detail for specifying variable features*

Metainterface

The idea of the different levels of detail was inspired by the Open Implementation approach to design [KLL+97, MLMK97]. In this approach, the configuration interface of a component is referred to as its *metainterface*.

The focus of an ICCL is on the reusability and flexibility of the implementation components. This requirement may sometimes conflict with the requirements on a configuration DSL: We strive for small, atomic components that can be combined in as many ways as possible. We want to avoid any code duplication by factoring out similar code sections into small, (parameterized) components. This code duplication avoidance and the opportunities for reuse of some implementation components (e.g. containers, bounds checkers, etc.) in other servers may lead to implementation components that do not align well with the feature boundaries that the user wants to use in his or her specifications. An example of a library which only provides an ICCL is the C++ Standard Template Library (STL). The user of the STL has to configure the STL implementation components manually (i.e. he has to hardcode ICCL statements in the client code).

The use of a configuration DSL and an ICCL separates the problem space and the solution space. This separation allows a more independent evolution of the client code, which uses a configuration DSL as the configuration interface to a component, and the component implementation code, which implements the ICCL. Indeed, during the development of the matrix computation library described in Chapter 10, there were cases in which we had to modify the ICCL and these modifications had no effect on the DSL whatsoever.

The configuration DSL of a concept can be directly derived from the feature model of the concept. The approach for this derivation involves answering the following questions:

- *Which features are relevant for the DSL?* Obviously, a configuration DSL will consist of variation points and variable features only. Other features of a feature model are not relevant for the configuration DSL.

- *Are any additional, more abstract features needed?* For example, during Domain Design of the matrix computation library described in Section 10.2.3, we added the *optimization flag* with the alternative subfeatures *speed* and *space* to the original matrix feature model created during feature modeling. The optimization flag allows us to decide which matrix storage format to use based on the matrix shape and density features (e.g. a dense triangular matrix is faster in access when we store its elements in an array than in a vector since we do not have to convert the subscripts on each element access; however, using a vector requires half the storage of an array since only the nonzero half of the matrix needs to be stored in the vector).

- *Is the nesting of features optimal for the user?* It may be useful to rearrange the feature diagrams of a feature model according to the needs of the users (which may be different). Also, the target implementation technology may impose some constraints on the DSL. For example, the matrix configuration DSL described in Section 10.2.3 uses dimensions as the only kind of variation points. This way we can easily implement it using C++ class templates.

- *Which features should have direct defaults and what are these defaults?* Some features should be selected by default if not specified. For example, the *shape* of a matrix is a dimension and should have the default value *rectangular.* Other dimensions of a matrix which should have direct defaults include element type, index type, optimization flag, and error checking flag (see Table 42).

- *For which features can defaults be computed and how to compute them?* Features, for which no direct defaults were defined should be computable from the specified features and the direct defaults. For example, the storage format of a matrix can be determined based on its shape, density, and optimization flag. The computed defaults can be specified using dependency tables (see Table 24 in Section 10.2.3 and, e.g., Table 48 in Section 10.2.3.4.3).[145]

The implementation components may be based on different component architectures. For the matrix package, we have used the GenVoca architecture described in Section 6.4.2. A GenVoca component represents a parameterized layer containing a number of mixin classes (thus, the layers are also referred to as *mixin layers* [SB98]). These layers can be configured according to a predefined grammar. Thus, a GenVoca grammar and a set of GenVoca layers may describe a whole family of OO frameworks since a configuration of a number of layers may represent a framework. In a degenerate case, a GenVoca layer may contain one class only. Configuring such layers corresponds to configuring a number of parameterized classes, some of which are parameterized by their superclasses.

If we use the GenVoca architecture as our implementation components architecture, our ICCL is obviously a GenVoca grammar (see e.g. Figure 173). In this case, we can think of our generative component to produce whole customized class hierarchies based on specifications in a configuration DSL.

The generated component or the configured runtime component should keep the metainformation describing its current configuration in an accessible form since this information may be interesting to other components. For example, the metainformation of a matrix is needed by the generative component implementing matrix expressions, so that it can select optimal algorithm implementations for the given argument matrices.

Configuration repository

We prefer to keep this metainformation in a *configuration repository* in the form of name-value pairs for all features (i.e. both the explicitly specified features and the computed defaults) rather than the original DSL description since we do not have to parse the description on each feature access. Each component has such a repository as its "birth certificate" (for the statically generated component) or current "data sheet" (for the dynamically reconfigurable component).

9.4.4 Expression DSLs

An important class of DSLs in algorithmic domains are expression DSLs. An expression DSL allows us to write algebraic expressions involving ADTs and operations on ADTs. Examples of expressions are matrix expressions, e.g. M1+M2*M3+M4–M5, where +, *, and – are matrix addition, multiplication, and matrix subtraction and M1 through M5 are matrices. Another example are image algebra expressions, e.g.

$$\left((a \oplus s)^2 + (a \oplus s)^2\right)^{1/2}$$

where a is an image, s is a template (i.e. an image whose values are images), and $\oplus$ is the right linear product operator. This expression is used in the image algebraic formulation of the Roberts edge detector (see [RW96]).

First, we need to list the operations and the ADTs they operate on. For our matrix case study in Section 10.2.6, for example, we only consider matrix addition, subtraction, and multiplication. The BLAS vector and matrix operations (see Section 10.1.1.2.4) provide a more comprehensive set of basic operations for matrix algorithms. Image Algebra [RW96] defines several dozens of basic operations.

The operations usually have some systematic relationships with the properties of their arguments. For example, adding two lower triangular matrices results in another lower triangular matrix, adding a lower triangular matrix and an upper triangular matrix results in a general square matrix, adding two dense matrices results in a dense matrix, etc.

Dependency tables

If we know that a certain property of a matrix does not change during the entire runtime, we can encode this property in its static type. We can then use the static properties of the arguments to derive the static properties of operation results. This corresponds to type inference. In general, we use the configuration repositories of the argument ADTs (a configuration repository contains all the static features of an ADT) to compute the configuration repository of the resulting ADT. Thus, we need to specify the mapping function for computing the features of operation results from the features of the arguments. For this purpose, we use *dependency tables*, which we define in Section 10.2.3. The functions for computing result types for matrix operations are given in Section 10.2.7.

Finally, we need to investigate opportunities for optimizations. We distinguish between two types of optimizations:

- *Optimizations of single operations*: This kind of optimization is performed by selecting specialized algorithms based on the information in the configuration repository of the participating ADTs. For example, we use different addition and multiplication algorithms depending on the shape of the argument matrices. The specification of such optimizations involves specifying the different algorithms and the criteria for selecting them (see e.g. Section 10.2.6.3).

- *Optimizations of whole expressions*: This kind of optimization involves the structural analysis of the entire expression and generating customized code based on this analysis.

Examples of such optimizations are elimination of temporaries and loop fusing. The optimizations may be performed at several levels of refinement. Sophisticated optimization techniques for expression DSLs are described in [Big98a, Big98b].

9.5 Domain Implementation

Different parts of an algorithmic library require different implementation techniques:

- ADTs, operations, and algorithms can be adequately implemented using parameterized functions, parameterized classes, and mixin layers (i.e. the GenVoca model). All of these abstraction mechanisms are available in C++.

- The implementation of configuration generators and expression optimizations require static metaprogramming capabilities. Here we can use built-in language capabilities such as template metaprogramming in C++, custom-developed preprocessors or compilers, or specialized metaprogramming environments and tools such as IP (Section 6.4.3) or Open C++ [Chi95].

- Domain-specific syntax extensions require preprocessors, custom compilers, or extendible compilers (e.g. IP). The C++ vector library Blitz++ (see Section 7.6.4), however, demonstrates that a rich language such as C++ allows us to simulate a great deal of mathematical notations without the need of syntax extensions.

Chapter 10 will demonstrate two implementation approaches:

- implementation in C++ in Section 10.3.1 (including implementation of a GenVoca architecture using C++ class templates, implementation of a configuration generator using template metaprogramming, implementation of the expression DSL using expression templates) and

- implementation in the IP System in Section 10.3.2.

Both approaches are evaluated and compared in Sections 10.3.1.8 and 10.3.2.

9.6 References

[Big98a] T. Biggerstaff. Anticipatory Optimization in Domain Specific Translation. In *Proceedings of the Fifth International Conference on Software Reuse (ICSR'98)*, P. Devanbu and J. Paulin, (Eds.), IEEE Computer Society Press, 1998, pp. 124-133

[Big98b] T. Biggerstaff. Composite Folding and Optimization in Domain Specific Translation. Technical Report, MSR-TR-98-22, Microsoft Research, June 1998

[BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. North-Holland, New York, 1976

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'95), ACM SIGPLAN Notices*, vol. 30, no. 10, 1995, pp. 285-299, http://www.softlab.is.tsukuba.ac.jp/~chiba/openc++.html

[Cop92] J. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1992

[DK98] A. van Deursen and P. Klint. Little Languages: Little Maintenance? In *Journal of Software Maintenance*, no. 10,1998, pp. 75-92, see http://www.cwi.nl/~arie

[Eis95] U. W. Eisenecker. Recht auf Fehler. In *iX*, no. 6, 1996, pp. 184-189 (in German)

[Eis96] Ulrich W. Eisenecker: Attribute im Zugriff. In *OBJEKTspektrum*, no.5, September/October 1996, pp. 98-101 (in German)

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[GL96] G. Golub and C. van Loan. *Matrix Computations*. Third edition. The John Hopkins University Press, Baltimore and London, 1996

[GO93] G.H. Golub and J.M. Ortega. *Scientific Computing: An Introduction with Parallel Computing*. Academic Press, Boston, 1993

[ILG+97] J. Irwin, J.-M. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-Oriented Programming of Sparse Matrix Code. XEROX PARC Technical Report SPL97-007 P9710045, February 1997, see http://www.parc.xerox.com/aop

[Kee89] S. Keene. *Object-oriented programming in Common Lisp: a Programmer's Guide to CLOS*. Addison-Wesley, 1989

[KL98] K. Kreft and A. Langer. Allocator Types. In *C++ Report*, vol. 10, no. 6, 1998, pp. 54-61 and p. 68

[KLL+97] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Open Implementation Design Guidelines. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, 1997, pp. 481-490

[KW97] D. Kühn and K. Weihe. Data access templates. In *C++ Report*, July/August, 1997

[Mey90] B. Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, 1990

[MLMK97] C. Maeda, A. Lee, G. Murphy, and G. Kiczales. Open Implementation Analysis and Design. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, M. Harandi, (Ed.), ACM Software Engineering Notes, vol 22, no. 3, May 1997, pp. 44-52

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide. Addison-Wesley*, Reading, Massachusetts, 1996

[PC86] D. L. Parnas and P. C. Clemens. A Rational Design Process: How and Why to Fake It. In *IEEE Transactions to Software Engineering*, vol. SE-12, no. 2, February 1986, pp. 251-257

[Pes98] C. Pescio. Multiple Dispatch: A New Approach Using Templates and RTTI. In *C++ Report*, vol. 10, no. 6, June 1998

[Ree96] T. Reenskaug with P. Wold and O.A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning, 1996

[RW96] G. X. Ritter and Joseph N. Wilson. *Handbook of Computer Vision Algorithms in Image Algebra*. CRC Press, 1996

[SB98] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceeding of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, E. Jul, (Ed.), 1998, pp. 550-570

[Wal97] R. van der Wal. Algorithmic Object. In *C++ Report*, vol. 9, no. 6, June 1997, pp. 23-27

[Wil92] P.R. Wilson. Uniprocessor garbage collection techniques. In *Memory Management*, Y. Bekkers and J. Cohen, (Eds.), Lecture Notes in Computer Science, no. 637, Springer-Verlag, 1992, pp. 1-42

[WM95] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995

Chapter 10 Case Study: Generative Matrix Computation Library (GMCL)

10.1 Domain Analysis

10.1.1 Domain Definition

10.1.1.1 Goals and Stakeholders

Our goal is to develop a *matrix computation library*. Thus, our domain is the *domain of matrix computation libraries*. The most important group of stakeholders are users solving linear algebra problems. We want the library to be highly reusable, adaptable, and very efficient (in terms of execution speed and memory consumption) and provide a highly intentional interface to application programmers. For our case study, we will always prefer technologically better solutions and ignore organizational issues. In a real world setting of a software development organization, the analysis of stakeholders and their goals, strategic project goals, and other organizational issues may involve a significant effort.

10.1.1.2 Domain Scoping and Context Analysis

10.1.1.2.1 Characterization of the Domain of Matrix Computation Libraries

Our general domain of interest is referred to as *matrix computations*, which is a synonym for applied, algorithmic linear algebra. Matrix computations is a mature domain with a history of more than 30 years (e.g. [Wil61]). The domain includes both the well-defined mathematical theory of linear algebra as well as the knowledge about efficient implementations of algorithms and data structures for solving linear algebra problems on existing computer architectures. This implementation knowledge is well documented in the literature, e.g. [GL96, JK93].

In particular, we are interested in the *domain of matrix computation libraries*. A matrix computation library contains ADTs and algorithm families for matrix computations and is intended to be used as a part of a larger application. Thus, it is an example of a *horizontal* domain. Examples of *vertical* domains involving matrix computations would be matrix computation environments (e.g. Matlab [Pra95]) or specialized scientific workbenches (e.g. for electromagnetics or quantum chemistry). They are vertical domains since they contain entire applications including GUIs, graphical visualization, persistent storage for matrices, etc.

The main concepts in matrix computations are vectors, matrices, and computational methods, e.g. methods for solving a system of linear equations or computing the eigenvalues. A glossary of some of the terms used in matrix computations is given in 10.4.

10.1.1.2.2 Sources of Domain Knowledge

The following sources of domain knowledge were used in the analysis of the matrix computation libraries domain:

- literature on matrix computations: [GL96, JK93];
- documentation, source code, and articles describing the design of existing matrix computation libraries: [LHKK79, DDHH88, DDDH90, CHL+96] and those listed in Table 15 and Table 16;
- online repository of matrices: [MM].

10.1.1.2.3 Application Areas of Matrix Computation Libraries

In this section, we will identify features characterizing matrix computation libraries by analyzing different application areas of matrix computations.

Table 13 and Table 14 list some typical application areas of matrix computations and the types of matrices and computations which are required for solving the problems in the listed areas. The application areas were grouped into two categories: one requiring dense matrices (Table 13) and the other one requiring sparse matrices (Table 14). In general, large matrix problems usually involve sparse matrices and large dense matrix problems are much less common.

Application area	**Dense matrix types**	**Computational problems**
electromagnetics (Helmholtz equation), e.g. radar technology, stealth (i.e. “radar-invisible”) airplane technology	complex, Hermitian (rarely also non-Hermitian), e.g. 55296 by 55296	boundary integral solution (specifically the method of moments)
flow analysis (Laplace or Poisson equation), e.g. airflow past an airplane wing, flow around ships	symmetric, e.g. 12088 by 12088	boundary integral solution (specifically the panel method)
diffusion of solid bodies in liquids	block Toeplitz	i. n. a.[146]
diffusion of light through small particles	block Toeplitz	i. n. a.
noise reduction	block Toeplitz	i. n. a.
quantum mechanical scattering (computing the scattering of elementary particles from other particles and atoms; involves Schrödinger wave function)	i. n. a.	dense linear systems
quantum chemistry (Schrödinger wave function)	real symmetric, occasionally Hermitian, small and dense (large systems are usually sparse)	symmetric eigenvalue problems
material science	i. n. a.	unsymmetric eigenvalue problems
real-time signal processing	i. n. a.	rank-revealing factorizations

applications		and the updating of factorizations after low rank changes

Table 13 *Examples of application areas for dense matrix computations (based on examples found in [Ede91, Ede93, Ede94,Hig96])*

Application area	**Sparse matrix types**	**Computational problems**
static analyses in structural engineering[147], e.g. static analysis of buildings, roofs, bridges, airplanes, etc.	real symmetric positive definite, pattern symmetric indefinite, e.g. 3948 by 3948 with 60882 entries	generalized symmetric eigenvalue problem, finite-element modeling, linear systems
dynamic analysis in structural engineering, e.g. dynamic analysis of fluids, suspension bridges, transmission towers, robotic control	real symmetric and positive definite or positive semi-definite or indefinite	symmetric eigenvalue problems, linear systems
hydrodynamics	real unsymmetric, e.g. 100 by 100 with 396 entries	eigenvalues of the Jacobi matrix
oceanic modeling, e.g. models of the shallow waves for the Atlantic and Indian Oceans	real symmetric indefinite, real skew symmetric, e.g. 1919 by 1919 with 4831 entries	finite-difference model
acoustic scattering	complex symmetric	i. n. a.
fluid flow modeling, fluid dynamics, flow in networks	real unsymmetric, symmetric structure, e.g. 511 by 511, 2796 entries and 23560 by 23560 with 484256 entries	iterative and direct methods, eigenvalue and eigenvector problems (in perturbation analysis), Lanczos method
petroleum engineering, e.g. oil recovery, oil reservoir simulation	real unsymmetric, symmetric structure, e.g. 2205 by 2205 with 14133 entries	i. n. a.
electromagnetic field modeling, e.g. integrated circuit applications, power lines	real pattern symmetric indefinite, real pattern symmetric positive definite, real unsymmetric, e.g. 1074 by 1074 with 5760 entries	finite-element modeling, symmetric and unsymmetric eigenvalue problem
power systems simulations, power system networks	real unsymmetric, real symmetric indefinite, real symmetric positive definite, e.g. 4929 by 10595 with 47369 entries	symmetric and unsymmetric eigenvalue problems
circuit simulation	real unsymmetric, 58 by 59 with 340 entries	i. n. a.
astrophysics, e.g. nonlinear radiative transfer and statistical equilibrium in astrophysics	real unsymmetric, e.g. 765 by 765 with 24382 entries	i. n. a.
nuclear physics, plasma physics	real unsymmetric, e.g. 1700 by 1700 with 21313 entries	Large unsymmetric generalized eigenvalue

		problems
quantum chemistry	complex symmetric indefinite, e.g. 2534 by 2534 with 463360 entries	symmetric eigenvalue problems
chemical engineering, e.g. simple chemical plant model, hydrocarbon separation problem	real unsymmetric, e.g. 225 by 225 with 1308 entries	conjugate gradient eigenvalue computation, initial Jacobian approximation for sparse nonlinear equations
probability theory and its applications, e.g. simulation studies in computer systems involving Markov modeling techniques	real unsymmetric, e.g. 163 by 163 with 935 entries	unsymmetric eigenvalues and eigenvectors
economic modeling e.g. economic models of countries, models of economic transactions	real unsymmetric, e.g. 2529 by 2529 with 90158 entries	i. n. a.
demography, e.g. model of inter-country migration	real unsymmetric, often relatively large fill-in with no pattern, e.g. 3140 by 3140 with 543162 entries	i. n. a.
surveying	real unsymmetric, e.g. 480 by 480 with 17088 entries	least squares problem
air traffic control	sparse real symmetric indefinite, e.g. 2873 by 2873 with 15032 entries	conjugate gradient algorithms
ordinary and partial differential equations	real symmetric positive definite, real symmetric indefinite, real unsymmetric, e.g. 900 by 900 with 4322 entries	symmetric and unsymmetric eigenvalue problems

Table 14 *Examples of application areas for sparse matrix computations (based on examples found in [MM])*

10.1.1.2.4 Existing Matrix Computation Libraries

As of writing, the most comprehensive matrix computation libraries available are written in Fortran. However, several object-oriented matrix computation libraries (for performance reasons, they are written mostly in C++) are currently under development. Table 15 and Table 16 list some of the publicly and commercially available matrix computation libraries in Fortran and in C++ (also see [OONP]).

Matrix computations library	**Features**
LINPACK a matrix computation library for solving dense linear systems; superseded by LAPACK see [DBMS79] and http://www.netlib.org/linpack	*language*: Fortran *matrix types*: dense, real, complex, rectangular, band, symmetric, triangular, and tridiagonal *computations*: factorizations (Cholesky, QR), systems of linear equations (Gaussian elimination, various factorizations), linear least

	squares problems, and singular value problems
EISPACK a matrix computation library for solving dense eigenvalue problems; superseded by LAPACK see [SBD+76] and http://www.netlib.org/eispack	*language*: Fortran *matrix types*: dense, real, complex, rectangular, symmetric, band, and tridiagonal *computations*: eigenvalues and eigenvectors, linear least squares problems
LAPACK a matrix computation library for dense linear problems; supersedes both LINPACK and EISPACK see [ABB+94] and http://www.netlib.org/lapack	*language*: Fortran *matrix types*: dense, real, complex, rectangular, band, symmetric, triangular, and tridiagonal *computations*: systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems
ARPACK a comprehensive library for solving real or complex and symmetric or unsymmetric eigenvalue problems; uses LAPACK and BLAS (see text below Table 16) see [LSY98] and http://www.caam.rice.edu/software/ARPACK	*language*: Fortran *matrix types:* provided by BLAS and LAPACK *computations*: Implicitly Restarted Arnoldi Method (IRAM), Implicitly Restarted Lanczos Method (IRLM), and supporting methods for solving real or complex and symmetric or unsymmetric eigenvalue problems
LAPACK++ a matrix computation library for general dense linear problems; provides a subset of LAPACK functionality in C++ see [DPW93] and http://math.nist.gov/lapack++	*language*: C++ *matrix types*: dense, real, complex, rectangular, symmetric, symmetric positive definite, band, triangular, and tridiagonal *computations*: factorizations (LU, Cholesky, QR), systems of linear equations and eigenvalue problems, and singular value problems
ARPACK++ subset of ARPACK functionality in C++ (using templates) see [FS97] and http://www.caam.rice.edu/software/ARPACK/arpack++.html	*language*: C++ *matrix types*: dense, sparse (CSC), real, complex, rectangular, symmetric, band *computations*: Implicitly Restarted Arnoldi Method (IRAM)
SparseLib++ library with sparse matrices; intended to be used with IML++ see [DLPRJ94] and http://math.nist.gov/sparselib++	*language*: C++ *matrix types*: sparse
IML++ (Iterative Methods Library) library with iterative methods; requires a library implementing matrices, e.g. SparseLib++ see [DLPR96] and http://math.nist.gov/iml++	*language*: C++ *matrix types*: library implementing matrices *computations*: iterative methods for solving both symmetric and unsymmetric linear systems of equations (Richardson Iteration,

	Chebyshev Iteration, Conjugate Gradient, Conjugate Gradient Squared, BiConjugate Gradient, BiConjugate Gradient Stabilized, Generalized Minimum Residual, Quasi-Minimal Residual Without Lookahead)
Newmat, version 9 a matrix computation library for dense linear problems; does not use C++ templates see http://nz.com/webnz/robert/nzc_nm09.html	*language*: C++ *matrix types*: dense, real, rectangular, diagonal, symmetric, triangular, band *computations*: factorizations (Cholesky, QR, singular value decomposition), eigenvalues of a symmetric matrix, Fast Fourier
TNT (Template Numerical Toolkit) a C++ matrix computation library for linear problems; it has a template-based design; eventually to supersede LAPACK++, SparseLib++, and IMC++; as of writing, with rudimentary functionality see [Poz96] and http://math.nist.gov/tnt	*language*: C++, extensive use of templates *matrix types*: dense, sparse, real, complex, rectangular, symmetric, triangular *computations*: factorizations (LU, Cholesky, QR), systems of linear equations contains an interface to LAPACK
MTL (Matrix Template Library) a C++ matrix library; it has an STL-like template-based design; its goal is to provide only one version of any algorithm and adapt it for various matrices using parameterization and iterators; it implements register blocking using template metaprogramming; it exhibits an excellent performance comparable to tuned Fortran90 code see [SL98a, SL98b] and http://www.lsc.nd.edu/	*language*: C++, extensive use of templates *matrix types*: dense, sparse, real, complex, rectangular, symmetric, band

Table 15 *Some of the publicly available matrix computation libraries and their features*

Matrix computations library	**Features**
Math.h++ C++ vector, matrix, and an array library in one Rogue Wave Software, Inc., see http://www.roguewave.com/products/math	*language*: C++ *matrix types*: dense, real, complex, rectangular *computations*: LU factorization, FFT
LAPACK.h++ works on top of Math.h++; offers functionality of the Fortran LAPACK library in C++ Rogue Wave Software, Inc., see http://www.roguewave.com/products/lapack	*language*: C++ *matrix types*: dense (some from Math.h++), sparse, real, complex, symmetric, hermitian, skew-symmetric, band, symmetric band, hermitian band, lower triangular, and upper triangular matrices *computations*: factorizations (LU, QR, SVD, Cholesky, Schur, Hessenberg, complete orthogonal, tridiagonal), real/complex and

	symmetric/unsymmetric eigenvalue problems
Matrix<LIB> LINPACK and EISPACK functionality in C++ Matlab-like syntax MathTools Ltd, see http://www.mathtools.com	*language*: C++ *matrix types*: dense, real, complex *computations*: factorizations (Cholesky, Hessenberg, LU, QR, QZ, Schur and SVD), solving linear systems, linear least squares problems, eigenvalue/eigenvector problems
ObjectSuite™ C++: IMSL Math Module for C++ a matrix computation library for dense linear problems Visual Numerics, Inc., see http://www.vni.com/products/osuite	*language*: C++ *matrix types*: dense, real, complex, rectangular, symmetric/Hermitian and symmetric/Hermitian positive definite *computations*: factorizations (LU, Cholesky, QR, and Singular Value Decomposition), linear systems, linear least squares problems, eigenvalue and eigenvector problems, two-dimensional FFTs

Table 16 *Some of the commercially available matrix computation libraries and their features*

A set of basic matrix operations and formats for high-performance architectures has been standardized in the form of the *Basic Linear Algebra Subprograms (BLAS)*. The operations are organized according to their complexity into three levels: Level-1 BLAS contain operations requiring O(n) of storage for input data and O(n) time of work, e.g. vector/vector operations (see [LHKK79]), Level-2 BLAS contain operations requiring O(n^2) of input and O(n^2) of work, e.g. matrix-vector multiplication (see [DDHH88]), Level-3 BLAS contain operations requiring O(n^2) of input and O(n^3) of work, e.g. matrix-matrix multiplication (see [DDDH90, BLAS97]). There are also Sparse BLAS [CHL+96], which are special BLAS for sparse matrices. The Sparse BLAS standard also defines various sparse storage formats. Different implementations of BLAS are available from http://www.netlib.org/blas/.

10.1.1.2.5 Features of the Domain of Matrix Computation Libraries

From the analysis of application areas and existing matrix computation libraries (Table 13, Table 14, Table 15, and Table 16), we can derive a number of major matrix types and computational method types which are common in the matrix computations practice. They are listed in Table 17 and Table 18, respectively. The types of matrices and computations represent the main features of the domain of matrix computation libraries and can be used to describe the scope of a matrix computation library. The rationale for including each of these features in a concrete matrix computation library implementation are also given in Table 17 and Table 18. Some features such as dense matrices and factorizations are basic features required by many other features and they should be included in any matrix computation library implementation. Some other features such as complex matrices and methods for computing eigenvalues are — unless directly required by some stakeholders — optional and their implementation may be deferred.

Matrix type	**Rationale for inclusion**
dense matrices	Dense matrices are ubiquitous in linear algebra computations and are mandatory for any matrix computation library.
sparse matrices	In practice, large linear systems are usually sparse.
real matrices	Real matrices are very common in linear algebra problems.
complex matrices	Complex matrices are less common than real matrices but still very important for a large class of problems.
rectangular, symmetric, diagonal, and triangular matrices	Rectangular, symmetric, diagonal, and triangular matrices are very common in linear algebra problems and are mandatory for any matrix computation library.
band matrices	Band matrices are common in many practical problems, e.g. a large percentage of the matrices found in [MM] are band matrices.
other matrix shapes (e.g. Toeplitz, tridiagonal, symmetric band)	There is a large number of other matrix shapes which are specialized for various problems. In general, providing all possible shapes in a general purpose matrix computation library is not possible since new applications may require new specialized shapes.

Table 17 *Major matrix types and the rationale for their inclusion in the implemented feature set*

Computational methods	**Rationale for inclusion**
factorizations (decompositions)	Factorizations are needed for direct methods and matrix analysis and are mandatory for any matrix computation library.
direct methods for solving linear systems	Direct methods (e.g. using the LU factorization) are standard methods for solving linear systems.
least squares methods	The least squares approach is concerned with the solution of overdetermined systems of equations. It represents the standard scientific method to reduce the influence of errors when fitting models to given observations.
symmetric and unsymmetric eigenvalue and eigenvector methods	Eigenvalue methods have numerous applications in science and engineering.
iterative methods for linear systems	Iterative methods for linear systems are the methods of choice for some large sparse systems. There are iterative methods for solving linear systems and for computing eigenvalues.

Table 18 *Major matrix computational methods types and the rationale for their inclusion in the implemented feature set*

Figure 137 summarizes the features of a matrix computation library in a feature diagram. The priorities express the typicality rate of the variable features. These typicality rates are informal and are intuitively based on the analysis of application areas and existing matrix computation libraries.

Another important feature of a matrix computation library, which will not be considered here, is its target computer architecture, e.g. hierarchical memory, multiple processors with distributed memory or shared memory, etc.

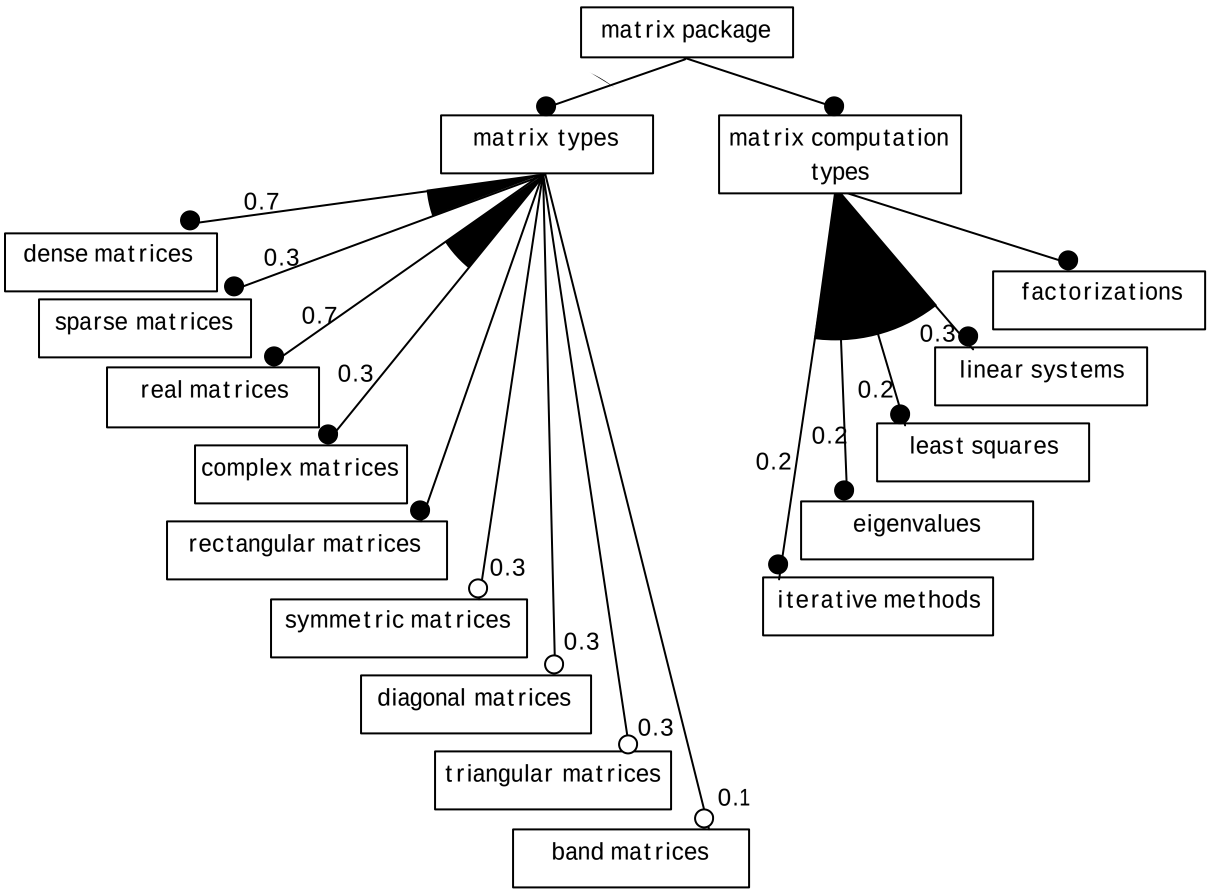


Figure 137 *Feature diagram of a matrix computation library*

10.1.1.2.6 Relationship to Other Domains

The *domain of array libraries* is an example of an *analogy domain* (see Section 3.6.3) of the domain of matrix computation libraries. Array libraries implement arrays (including two-dimensional arrays) and numerical computations on arrays. Thus, there are significant similarities between array libraries and matrix computation libraries. But there are also several differences:

- Array libraries, in contrast to matrix computation libraries, also cover arrays with more than two dimensions.

- Array operations are primarily elementwise operations. For example, in an array library * means elementwise multiply, whereas in a matrix computation library * designates matrix multiplication.

- Arrays usually support a wide range of element types, e.g. int, float, char, bool, and user defined types, whereas the type of matrix elements is either real or complex numbers.

- Array libraries usually do not provide a comprehensive set of algorithms for solving complicated linear problems. They rather focus on other areas, e.g. signal processing.

- Array libraries usually do not provide any special support for different shapes and densities.

Blitz++ [Vel97] is an example of an array library. Math.h++ (see Table 16) combines aspects of both an array and a matrix computation library in one.

An example of another analogy domain is the domain of *image processing libraries*. Images are somewhat similar to matrices. However, there are also several differences:

- The elements of an image are binary, gray scale, or color pixel values. Binary and color pixel values require a different representation than matrix elements. For example, color pixels may be represented using three values and a collection of binary pixels is usually represented by one number.

- The operations and algorithms required in image processing are different than those used for solving linear problems.

An example of a *support domain* is the *domain of container libraries*. A container library, e.g. the Standard Template Library (STL; [MS96, Bre98]) could be used to implement storage for matrix elements in a matrix computation library.

10.1.2 Domain Modeling

10.1.2.1 Key Concepts of the Domain of Matrix Computation Libraries

The key concepts of the domain of matrix computation libraries are

- abstract data types: *vectors* and *matrices*;

- algorithm families: *factorizations*, *solving systems of linear equations*, *solving least squares problems*, *solving eigenvalue and eigenvector problems*, and *iterative methods*.

10.1.2.2 Feature Modeling of the Key Concepts

10.1.2.2.1 Features of Vectors and Matrices

This section describes the features of vectors and matrices. Since the vector features represent a subset of the matrix features, we only list the matrix features and indicate if a feature does not apply to vectors. Please note that vectors can be adequately represented as matrices with number of rows equal one or number of columns equal one.

We have the following matrix features:

- *element type*: type of the matrix elements;

- *subscripts*: subscripts of the matrix elements;

- *structure*: the arrangement and the storage of matrix elements:

- *entry type*: whether an entry is a scalar or a matrix;

- *density*: whether the matrix is sparse or dense;

- *shape*: the arrangement pattern of the nonzero matrix elements (this feature does not apply to vectors);

- *representation*: the data structures used to store the elements;

- *format*: the layout of the elements in the data structures;

- *memory management*: allocating and relinquishing memory;

- *operations*: operations on matrices (including their implementations);

- *attributes*: matrix attributes, e.g. number of rows and columns;
- *concurrency and synchronization*: concurrent execution of algorithms and operations and synchronization of memory access;
- *persistency*: persistent storage of a matrix instance;
- *error handling*: error detection and notification, e.g. bounds checking, compatibility checking for vector-vector, matrix-vector, and matrix-matrix operations.

10.1.2.2.1.1 Element Type

The only element types occurring in linear algebra (and also in the application areas listed in Table 13 and Table 14) are real and complex numbers. Existing libraries (Table 15 and Table 16) typically support single and double precision real and complex element types. Other element types (e.g. bool, user defined types, etc.) are covered by array libraries (see Section 10.1.1.2.6).

10.1.2.2.1.2 Subscripts (Indices)

The following are the subfeatures concerning subscripts:

- *index type*: The type of subscripts is an integral type, e.g. char, short, int, long, unsigned short, unsigned int, or unsigned long.
- *maximum index value*: The choice of index type, e.g. char or unsigned long, determines the maximum size of a matrix or vector.
- *index base*: There are two relevant choices for the start value of indices: *C-style indexing* (or *0-base indexing*), which starts at 0, and the *Fortran-style indexing* (or *1-base indexing*), which starts at 1. Some libraries, e.g. TNT (see Table 15), provide both styles at the same time (TNT provides the operator "[]" for 0-base indexing and the operator "()" for 1-base indexing).
- *subscript ranges*: Additionally, we could also have a subscript type representing subscript ranges. An example of range indexing is the Matlab indexing style [Pra95], e.g. 1:4 denotes a range from 1 to 4, 0:9:3 denotes a range from 0 to 9 with stride 3. An example of a library supporting subscript ranges is Matrix<LIB> (see Table 16).

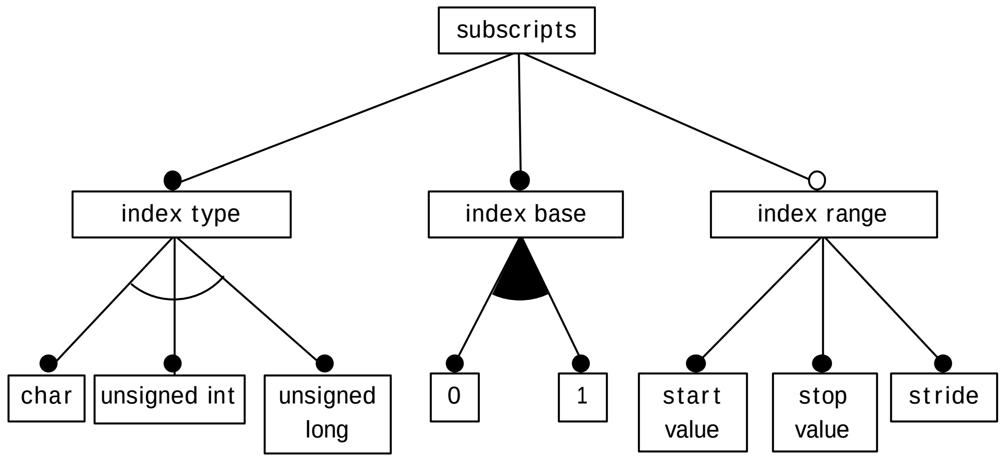


Figure 138 *Subscripts*

10.1.2.2.1.3 Structure

Structure is concerned with the arrangement and the storage of matrix or vector elements. We can exploit the arrangement of the elements in order to reduce storage requirements and to provide specialized and faster variants of basic operations and more complex algorithms. The subfeatures of the *structure* of matrices are shown in Figure 139.

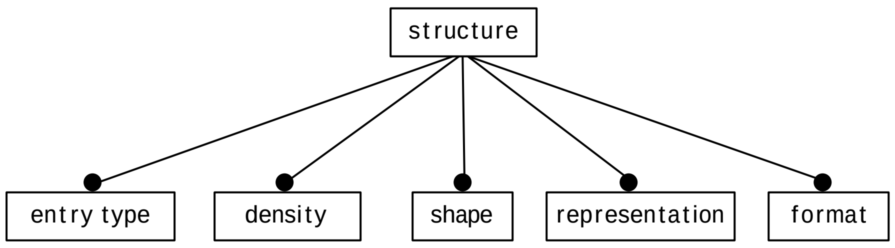


Figure 139 *Structure*

10.1.2.2.1.3.1 Entry Type

The entries in a matrix are usually scalars (e.g. real or complex). These types of matrices are referred to as *point-entry matrices* [CHL+96]. There are also matrices whose entries are matrices and they are referred to as *block matrices* [CHL+96, GL96]. Block matrices are common in high-performance computing since they allow us to express operations on large matrices in terms of operations on small matrices. This formulation enables us to take advantage of the hierarchical memory organization on modern computer architectures.

The memory of modern computer architectures is usually organized into a hierarchy: The higher levels in the hierarchy feature memory fast in access but of limited capacity (e.g. processor cache). As we move down the hierarchy, the memory speed decreases but its capacity increases (e.g. main memory, disk).

When performing a matrix operation, it is advantageous to keep all the operands in cache in order to eliminate excessive data movements between the cache and the main memory during the operation. If the operands are matrices which entirely fit into the cache, we can use the point-entry format. But if a matrix size exceeds the cache size, the block format should be preferred. Operations on block matrices are performed in terms of operations on their blocks, e.g. matrix multiplication is performed in terms of multiplications of the block submatrices. By properly adjusting the block size, we are able to fit the arguments of the submatrix operation into the cache.

Furthermore, we distinguish between *constant blocks* (i.e. blocks have equal sizes) and *variable blocks* (i.e. blocks have variable sizes). The subfeatures of *entry type* are summarized in Figure 140.

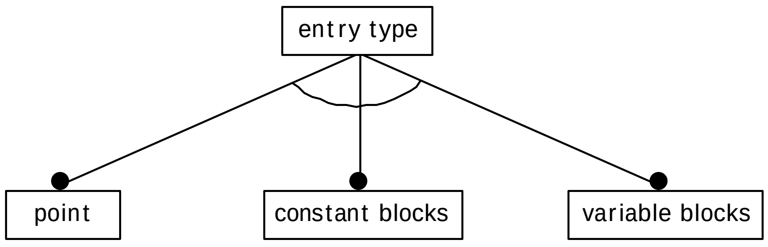


Figure 140 *Entry Type*

Blocking is used in high performance linear algebra libraries such as LAPACK (see Table 15).

10.1.2.2.1.3.2 Density

One of the major distinctions between matrices is whether a matrix is *dense* or *sparse*. A dense matrix is a matrix with a large percentage of *nonzero elements* (i.e. elements not equal zero). A sparse matrix, on the other hand, contains a large percentage of *zero elements* (usually more than 90%). In [Sch89], Schendel gives an example of a sparse matrix in the context of the frequency analysis of linear networks which involves solving a system of linear equations of the form $A(w_i)x = b$. In this example, A is a 3304-by-3304 Hermitian matrix with 60685 nonzero elements. Thus, the nonzero elements make up only 0.6% of all elements in A (i.e. the fill-in is 0.6%). Furthermore, the LU-factorization of A yields a new matrix with an even smaller fill-in of 0.4% (see Table 14 for more examples of sparse matrices). The representation of A as a dense matrix would require several megabytes of memory. However, it is necessary to store only the nonzero elements, which dramatically reduces the storage requirements for sparse matrices. The knowledge of the density of a matrix allows us not only to optimize the storage consumption, but also the processing speed since we can provide specialized variants of operations which take advantage of sparseness.

Most matrix computation libraries provide dense matrices and some matrix computation libraries also implement sparse matrices (e.g. LAPACK.h++; see Table 16). Since most of the large matrix problems are sparse (see Table 14), a general-purpose matrix computation library is much more attractive if it implements both dense and sparse matrices.

10.1.2.2.1.3.3 Shape

Matrix computations involve matrices with different arrangement patterns of the nonzero elements. Such arrangement patterns are referred to as *shapes*. Some of the more common shapes include the following (see Figure 141):

- *Rectangular* and *square matrices*: A rectangular matrix has a different number of rows than the number of columns. The number of rows and the number of columns in a square matrix are equal.

- *Null matrix*: A null matrix consists of only zero elements. No elements have to be stored for a null matrix but only the number of rows and columns.

- *Diagonal matrix*: A diagonal matrix is a square matrix with all zero elements except the diagonal elements (i.e. elements whose row index and column index are equal). Only the (main) diagonal elements have to be stored. If they are all equal, the matrix is referred to as a *scalar matrix* and only the scalar has to be stored.

- *Identity matrix*: An identity matrix is a diagonal matrix whose diagonal entries are all equal 1. No elements have to be stored for an identity matrix but only the number of rows and columns.

- *Symmetric*, *skew-symmetric* or *anti-symmetric*, *Hermitian*, and *skew-Hermitian matrices*: For all elements of a symmetric matrix the following equation holds $a_{ij} = a_{ji}$. For a skew-symmetric matrix, we have a slightly different equation: $a_{ij} = - a_{ji}$. A complex-valued matrix with symmetric real part and skew-symmetric imaginary part is referred to as a Hermitian. If, on the other hand, the real part is skew-symmetric and the imaginary part is symmetric, we have a skew-Hermitian matrix. For all these four matrix types we only need to store one half of the matrix. One possible storage format is to consecutively store all the rows (or columns or diagonals) of one half of the matrix in a vector and use an indexing formula to access the matrix elements.

- *Upper* or *lower triangular* or *unit triangular* or *Hessenberg matrices*: An upper triangular matrix is a square matrix which has nonzero elements only on and above the main diagonal. If the diagonal elements are only ones, the matrix is referred to as *unit upper triangular.* If the diagonal elements are only zeros, the matrix is referred to as *strictly upper triangular.* If, on the other hand, the main diagonal and also the diagonal below contains nonzeros, the matrix is referred to as an upper Hessenberg. The lower triangular, lower unit triangular, and lower Hessenberg matrices are defined analogously. Similarly as in the case of symmetric matrices, only one half of the elements of a triangular matrix has to be stored.

- *Upper* or *lower bidiagonal*, and *tridiagonal matrices*: These matrices are diagonal matrices with an extra nonzero diagonal above, or below, or both above and below the main diagonal.

- *Band matrices*: Band matrices have nonzero fill-in in one or more adjacent diagonals (see Figure 141. Diagonal and triangular matrices can be regarded as a special case of band matrices. An example of a general storage schema for band matrices would be storing the nonzero diagonals in a smaller matrix, with one diagonal per row and accessing the elements using an indexing formula. Special types of band matrices are *upper* and *lower band triangular matrices*, *band diagonal matrices*, and *symmetric band matrices*.

- *Toeplitz matrices*: A Toeplitz matrix is a square matrix, where all elements within each of its diagonals are equal. Thus, a Toeplitz matrix requires the same amount of storage as a diagonal matrix.

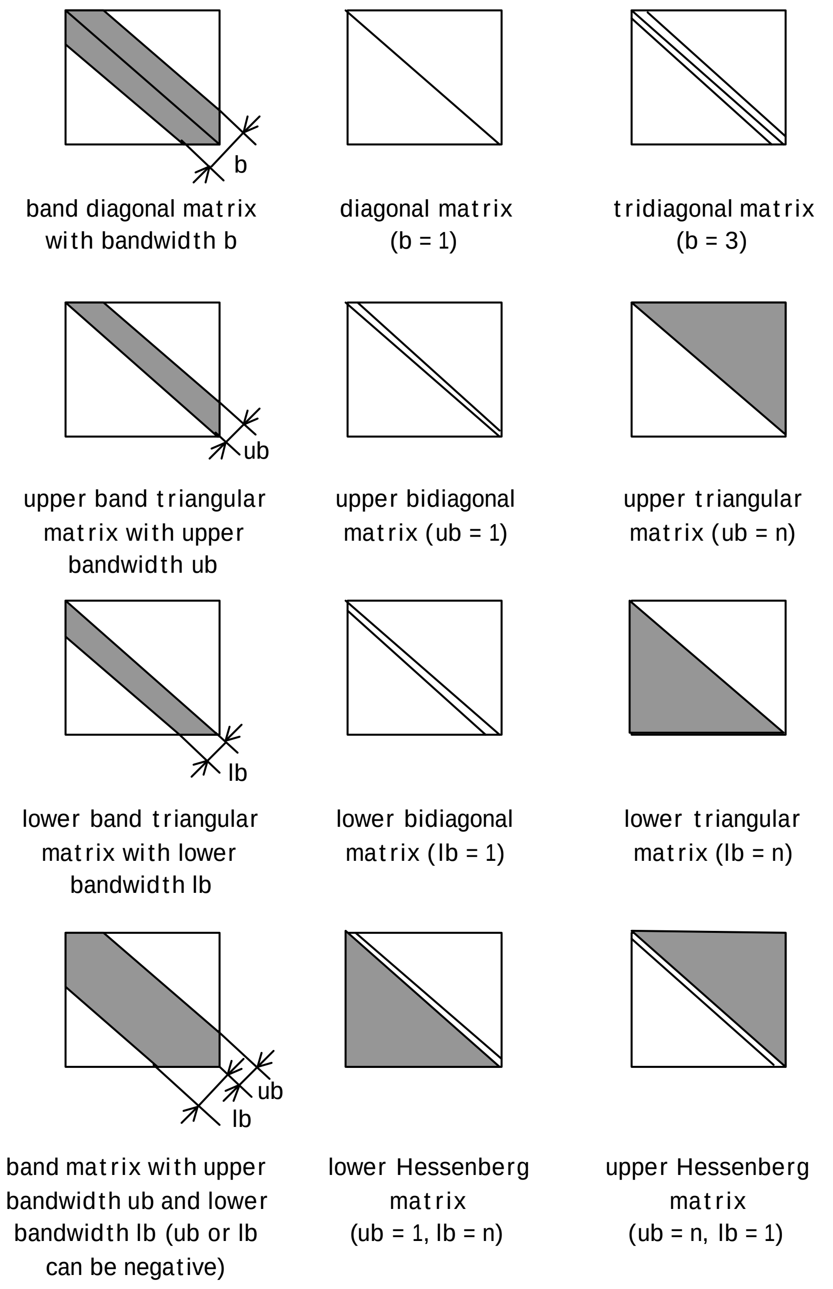

b
band diagonal matrix with bandwidth b
diagonal matrix (b = 1)
tridiagonal matrix (b = 3)
ub
upper band triangular matrix with upper bandwidth ub
upper bidiagonal matrix (ub = 1)
upper triangular matrix (ub = n)
lb
lower band triangular matrix with lower bandwidth lb
lower bidiagonal matrix (lb = 1)
lower triangular matrix (lb = n)
ub
lb
band matrix with upper bandwidth ub and lower bandwidth lb (ub or lb can be negative)
lower Hessenberg matrix (ub = 1, lb = n)
upper Hessenberg matrix (ub = n, lb = 1)

Figure 141 *Examples of n´n band matrices (only the gray region and the shown diagonals may contain nonzeros)*

Some of the above-listed shapes also apply to block matrices, e.g. a block matrix with null matrix entries except for the diagonal entries is referred to as a *block diagonal matrix*. There are also numerous examples of “more exotic”, usually sparse matrix types in the literature, e.g. in [Sch89]:

strip matrix, *band matrix with margin (also referred to as a bordered matrix)*, *block diagonal matrix with margin*, *band matrix with step* (see Figure 142).

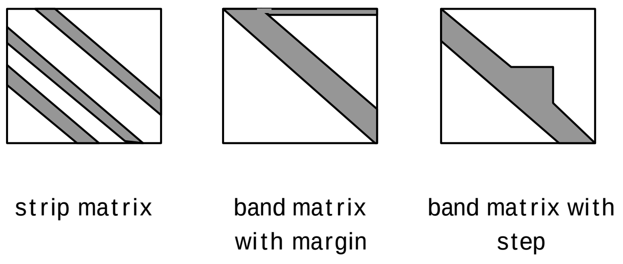


Figure 142 *Some more exotic matrix shapes*

Most matrix computation libraries provide rectangular, symmetric, diagonal, and triangular matrices. Some libraries also provide band matrices (e.g. LAPACK, LAPACK++, ARPACK++, Newmat, LAPACK.h++; see Table 15 and Table 16). Other shapes are less commonly supported.

10.1.2.2.1.3.4 Representation

The elements of a matrix or a vector can be stored in a variety of data structures, e.g. arrays, lists, binary trees, dictionaries (i.e. maps). Each data structure exhibits different performance regarding adding, removing, enumerating, and randomly accessing the elements.

10.1.2.2.1.3.5 Memory Management in Data Structures

The data structures for storing matrix elements may use different memory allocation strategies. We discussed different strategies in Section 9.3.2.2.1. Here, we require at least static and dynamic memory allocation. We extend the representation feature with the *memory allocation* subfeature. The resulting diagram is shown in Figure 143.

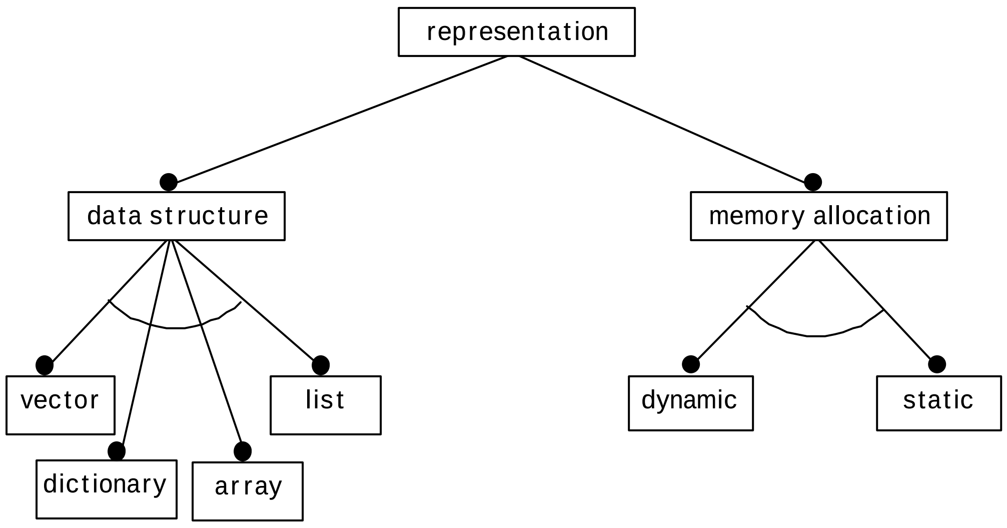


Figure 143 *Representation*

10.1.2.2.1.3.6 Format

Format describes how the elements of a matrix of certain entry type, shape, and density are stored in concrete data structures. For the sake of simplicity, we will further investigate only dense or sparse, point-entry matrices with the most common shapes: rectangular, symmetric, triangular, diagonal, and band. We first describe the common formats for rectangular dense matrices and general sparse matrices and then discuss the special storage and access requirements of other shapes.

10.1.2.2.1.3.6.1 Rectangular Dense Matrices

There are two major formats for storing the elements of a rectangular dense matrix:

1. *row-major* or *C-style format*: In the row-major format, the matrix elements are stored row-wise, i.e. the neighboring elements of one row are also adjacent in memory. This format corresponds to the way arrays are stored in C.

2. *column-major* or *Fortran-style format*: In the column-major format, the matrix elements are stored column-wise, i.e. the neighboring elements of one column are also adjacent in memory. This format corresponds to the array storage convention of Fortran.

Newer matrix computation libraries usually provide both formats (e.g. LINPACK++, TNT). The column-major format is especially useful for interfacing to Fortran libraries.

10.1.2.2.1.3.6.2 General Sparse Matrices

There are several common storage formats for general sparse matrices, i.e. formats that do not assume any specific shape. However, they are also used to represent shaped sparse matrices. The general sparse storage formats include the following:

- *coordinate format* (COO): Only the nonzero matrix elements along with their coordinates are stored. This format is usually implemented using three vectors, one containing the nonzeros and the other two containing their row and the column indices, respectively. Another possibility is to use one array or list with objects, where each of the objects encapsulates a matrix element and its coordinates. Yet another possibility is to use a hash dictionary data structure, where the keys are the coordinates and the values are the nonzeros.

- *compressed sparse column format* (CSC): The nonzeros are stored column-wise, i.e. the nonzeros of a column are stored in the order of their occurrence within the columns. One possibility is to store the columns containing nonzeros in sparse vectors.

- *compressed sparse row format* (CSR): The nonzeros are stored row-wise, i.e. the nonzeros of a row are stored in the order of their occurrence within the rows. One possibility is to store the rows containing nonzeros in sparse vectors.

There are also several other sparse formats including *sparse diagonal* (DIA), *ellpack/itpack* (ELL), *jagged diagonal* (JAD), and *skyline formats* (SKY) and several *block matrix formats* (see [CHL+96]). Table 19 summarizes when to use which sparse format.

Sparse Format	When to Use?
coordinate (COO)	Most flexible data structure when constructing or modifying a sparse matrix.
compressed sparse column (CSC)	Natural data structure for many common matrix operations including matrix multiplication and constructing or solving sparse triangular factors.
compressed sparse row (CSR)	Natural data structure for many common matrix operations including matrix multiplication and constructing or solving sparse triangular factors.
sparse diagonal (DIA)	Particularly useful for matrices coming from finite difference approximations to partial differential equations on uniform grids.
ellpack/itpack (ELL)	Appropriate for finite element or finite volume approximations to partial differential equations where elements are of the same type, but the gridding is irregular.
jagged diagonal (JAD)	Appropriate for matrices which are highly irregular or for a general-purpose matrix multiplication where the properties of the matrix are not known a priori.
skyline (SKY)	Appropriate for band triangular matrices. Particularly well suited for Cholesky or LU decomposition when no pivoting is required. In this case, all fill will occur within the existing nonzero structure.

Table 19 *Choice of sparse format (adapted from [CHL+96])*

10.1.2.2.1.3.6.3 Dependency Between Density, Shape, and Format

The storage and access requirements of dense or sparse, point-entry matrices with the shapes rectangular, symmetric, triangular, diagonal, and band are set out in Table 20.

Structure Type	Storage and access requirements
dense rectangular	We store the full matrix in the row- or the column-major dense format.
dense symmetric	We store one half of the matrix, e.g. row-, column-, or diagonal-wise in a dense vector, and use an indexing formula to access the elements. Alternatively, we can store the elements in a full-size two-dimensional array using only half of it. The latter approach needs double as much memory as the first one, but it is faster in access since we do not have to transform the indices. Assigning a value to the element a_{ij} with $i \neq j$, automatically assigns the same value to a_{ji}.
dense triangular	We store the nonzero half of the matrix, e.g. row-, column-, or diagonal-wise in a dense vector. Alternatively, we can store the elements in a two-dimensional array, which requires more space but is faster in access. Reading an element from the other half returns 0 and setting such an element to a value other than 0 results in an error.
dense diagonal	We store only the diagonal in a dense vector. Reading an element off the diagonal returns 0 and setting such an element to a value other than 0 results in an error.
dense band	We store the band only, e.g. diagonal-wise in a dense vector or a smaller two-dimensional array. Alternatively, we can store the elements in a full-size two-dimensional array, which requires more space but is faster in access. Reading an element off the band returns 0 and setting such an element to a value other than 0 results in an error.
sparse rectangular	We store only the nonzero elements in one of the sparse formats, e.g. CSR, CSC, COO, ELL, JAD;
sparse symmetric	We store only one half of the matrix and only the nonzero elements using one of the sparse formats (esp. SKY or DIA). Assigning a value to the element a_{ij} with $i \neq j$, automatically assigns the same value to a_{ji}.
sparse triangular	We use one of the sparse formats (esp. SKY or DIA) to store the nonzero elements. Setting an element in the zero-element half of the matrix to a value other than 0 results in an error.
sparse diagonal	We use one of the sparse matrix formats (esp. DIA) to store the nonzero elements or we store them in a sparse vector. Reading elements off the diagonal returns 0 and assigning a value other than 0 to them causes an error.
sparse band	We use one of the sparse formats (esp. DIA for band diagonal and DIA or SKY for band triangular) to store the nonzero elements. Setting an element off band to a value other than 0 results in an error.

Table 20 *Storage and access requirements of matrices of different structures*

The particular shape of a matrix — especially of a sparse matrix — is application dependent and not all of the possible shapes and formats can be provided by a matrix computation library. Thus, it is important to allow a client program to supply specialized formats.

10.1.2.2.1.3.7 Error Checking: Checking Assignment Validity

Checking the validity of an assignment, e.g. checking whether the value assigned to an element within the zero-element half of a triangular matrix is actually a zero, should be parameterized.

10.1.2.2.1.4 Matrix and Vector Operations

We will consider the following operations on matrices and vectors as parts of a matrix component:

- access operations, i.e. set element and get element and

- basic mathematical operations directly based on access operations, e.g. matrix addition and multiplication.

More complex operations, such as computing the inverse of a matrix or solving triangular systems, will be analyzed together with the algorithm families (e.g. solving linear systems).

The basic operations can be clustered according to their arity and argument types. The unary operations are listed in Table 21. The operation type indicates the input argument type and result type. They are separated by an arrow.

Operation type	**Operations**
vector → scalar	Vector norms, e.g. p-norms (1-norm, 2-norm, etc.)
vector → vector	transposition
matrix → scalar	matrix norms, e.g. Frobenius norm, p-norms determinant
matrix → matrix	transposition

Table 21 *Unary operations*

The binary operations are set out in Table 22. An *update* operation stores the result in one of its input arguments. The definitions of the operations listed in Table 21 and Table 22 can be found in [GL96].

Operation type	Operations
(scalar, vector) $\rightarrow$ vector	scalar-vector multiplication
(scalar, matrix) $\rightarrow$ matrix	scalar-matrix multiplication
(scalar, vector) $\rightarrow$ update vector	saxpy, which is defined as follows y := ax + y. where x, y $\in R^n$ and a $\in R$
(vector, vector) $\rightarrow$ vector	vector addition, vector difference, vector multiply (or the Hadamard product)
(vector, vector) $\rightarrow$ scalar	dot product
(vector, vector) $\rightarrow$ matrix	outer product
(vector, vector) $\rightarrow$ update matrix	outer product update, which is defined as follows A := A + xy^T, where x $\in R^m$, y $\in R^n$ A $\in R^{m \times n}$
(matrix, vector) $\rightarrow$ vector	matrix-vector multiplication
(matrix, vector) $\rightarrow$ update vector	gaxpy (i.e. generalized saxpy), which is defined as follows y := Ax + y, where x, y $\in R^n$ and A $\in R^{m \times n}$
(matrix, matrix) $\rightarrow$ matrix	matrix addition, matrix difference, matrix multiplication

Table 22 *Binary operations*

A standard set of vector and matrix operations is defined in the form of the BLAS (see Section 10.1.1.2.4). The BLAS are a superset of the operations listed in Table 21 and Table 22. Matrix algorithms can be expressed at different levels, e.g. at the level of operations on matrix elements or at the Level-2 or Level-3 BLAS. Level-3 BLAS formulation of an algorithm contains matrix-matrix operations as their smallest operations. This formulation is especially suited for block matrices.

10.1.2.2.1.4.1 Error Checking: Bounds Checking

Invoking the get or the set operation on a matrix or vector with subscripts which are outside the matrix dimensions or vector dimension is an error. Checking for this condition should be parameterized.

10.1.2.2.1.4.2 Error Checking: Checking Argument Compatibility

The vectors and matrices supplied as input arguments to one of the binary operations must have compatible dimensions. For addition and subtraction of vectors and matrices and dot and outer product, the corresponding dimensions of both arguments must be equal. For the matrix-matrix multiplication, the number of columns of the first matrix must be equal to the number of rows in the second matrix. Similarly, the dimension of the vector in a matrix-vector product must be equal to the number of columns of the matrix. Moreover, a determinant can be computed only for square matrices. Checking argument compatibility should be parameterized. If the numbers of rows and columns are available at compile time, the checking should be performed at compile time.

10.1.2.2.1.5 Interaction Between Operations and Structure

The operations on matrices and vectors interact with their structures in various ways:

- There are dependencies between the shape of the arguments and the shape the result of operations.

- There are dependencies between the density of the arguments and the density of the result of operations.
- The implementation algorithms of the matrix operations can be specialized based on the shape to save floating point operations.
- The implementation of an operation's algorithm depends on the underlying representation and format of the arguments, e.g. dense storage provides fast random access. This is not the case with most sparse storage formats.

The following is true about the shape of the result of an operation:

- the result of multiplying a matrix by a scalar is a matrix of the same shape;
- adding, subtracting, or multiplying two lower triangular matrices results in a lower triangular matrix;
- adding, subtracting, or multiplying two upper triangular matrices results in an upper triangular matrix;
- adding or subtracting two symmetric matrices results in a symmetric matrix;
- adding, subtracting, or multiplying two diagonal matrices results in a diagonal matrix.

When we consider rectangular, triangular, and diagonal matrices, the addition, subtraction, or multiplication of two such matrices can potentially produce a matrix whose shape is equal to the shape resulting from superimposing the shapes of the arguments, e.g. rectangular and diagonal matrices yield rectangular matrices and lower diagonal and upper diagonal matrices also yield rectangular matrices, but diagonal and lower triangular matrices yield lower triangular matrices.

Adding, subtracting, or multiplying two dense matrices results — in most cases — in a dense matrix. Adding or subtracting two sparse matrices results in a sparse matrix. Multiplying two sparse matrices can result in a sparse or a dense matrix.

The algorithms of the matrix operations can be specialized based on the shape of the arguments. For example, the multiplication of two lower triangular matrices requires about half the floating point operations needed to multiply two rectangular matrices. Some of the special cases are adding, subtracting, and multiplying two diagonal matrices, two lower or upper matrices, or a diagonal and a triangular matrix, or multiplying a matrix by a null or identity matrix.

10.1.2.2.1.6 Optimizations

In addition to specializing algorithms for different shapes of the argument matrices, we can also optimize whole expressions. For example, more than one adjacent matrix addition operations in an expression should be all performed using one pair of nested loops adding the matrices elementwise without any intermediate results. Thus, this optimization involves the elimination of temporaries and loop fusing. We already described it in Section 9.4.1.

10.1.2.2.1.7 Attributes

An important attribute of a vector is its *dimension* (or *length*), which is the number of elements the vector contains. Since matrices are two dimensional, they have two attributes describing their size: *number of rows* and *number of columns*. For a square matrix, the number of rows and the number of columns are equal. Thus, we need to specify only one number, which is referred to as the *order*. For band matrices, we have to specify the *bandwidth* (i.e. the number of nonzero diagonals; see Figure 141). It should be possible to specify all these attributes statically or dynamically.

10.1.2.2.1.8 Concurrency and Synchronization

Matrix operations are well suited for parallelization (see [GL96,p. 256]). However, parallelization of matrix algorithms constitutes a complex area on its own and we will not further investigate this topic. A simple form of concurrent execution, however, can be achieved using threads, i.e. lightweight processes provided by the operating system. In this case, we have to synchronize the concurrent access to shared data structures, e.g. matrix element containers, matrix attributes. This can be achieved through various locking mechanisms, e.g. semaphores or monitors. We use the locking mechanisms to make all operations of a data structure mutually exclusive and to make the writing operations self exclusive (see Section 7.4.3). In the simplest case, we could provide a matrix synchronization wrapper, which makes get and set methods mutually exclusive and the set method self exclusive.

10.1.2.2.1.9 Persistency

We need to provide methods for storing a matrix instance on a disk in some appropriate format and for restoring it back to main memory.

10.1.2.2.2 Matrix Computation Algorithm Families

During Domain Definition in Section 10.1.1.2.5, we identified the main areas of matrix computations:

- factorizations,
- solving linear systems,
- computing least squares solutions,
- eigenvalue computations, and
- iterative methods.

Each of these areas contain large families of matrix computation algorithms.

As an example, we will discuss the family of factorization algorithms. The discussion focuses on the structure of this family rather than on explaining all the mathematical concepts behind the algorithms. The interested reader will find detailed explanations of these concepts in [GL96].

In general, factorizations decompose matrices into factor matrices with some desired properties by applying a number of transformations. Factorizations are used in nearly all the major areas of matrix computations: solving linear systems of equations, computing least squares solutions, and eigenvalue computations. For example, the LU factorization of a matrix A computes the lower triangular matrix L and the upper triangular matrix U, such that A = L*U. The LU factorization can be used to solve a linear system of the form A*x=b, where A is the coefficient matrix, b is the right-hand side vector, and x is the sought-after solution vector. After factoring A into L and U, solving the system involves solving two triangular systems: L*y=b and U*x=y, which is very simple to do using *forward* or *back substitution*.

In general, we can solve a linear system using either factorizations such as the LU, which are also referred to as *direct methods*, or we can use so-called *iterative methods*. Iterative methods generate series of approximate solutions, which hopefully converge on a single solution. Examples of iterative methods for solving linear systems are Jacobi iterations, Gauss-Seidel iterations, SOR iterations, and the Chebyshev semi-iterative method (see [GL96]).

There are two important categories of factorizations: the *LU family* and *orthogonal factorizations*. Examples of the latter are Singular Value Decomposition (SVD) and the QR factorizations (e.g. Hausholder QR, Givens QR, Fast Givens QR, Gram-Schmidt QR).

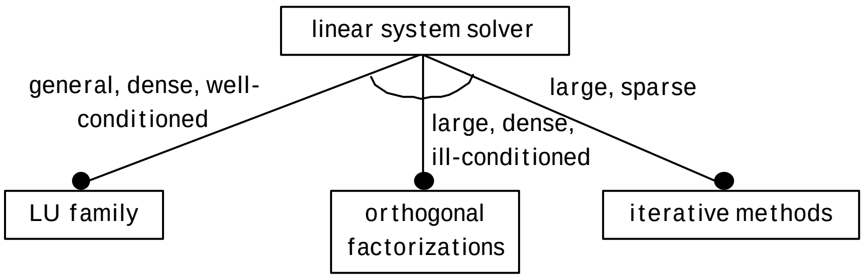

linear system solver
general, dense, well-conditioned
large, dense, ill-conditioned
large, sparse
LU family
orthogonal factorizations
iterative methods

Figure 144 *Approaches to solving linear systems*

Figure 144 explains when to use LU factorizations and when orthogonal factorizations or iterative methods to solve a linear system. Each alternative method is annotated with the properties of the coefficient matrix A as preconditions. These preconditions indicate when a given method is most appropriate. For example, if A is ill-conditioned, LU factorizations should not be used. A is ill-conditioned if it is nearly singular. Whether A is ill-conditioned or not is determined using *condition estimators* (see [GL96] for details).

In the rest of this section, we will concentrate on LU factorizations. The LU factorization in its general form, i.e. *general LU*, corresponds to the Gaussian elimination method. The general LU can be specialized in order to handle systems with special properties more efficiently. For example, if A is square and positive definite, we use the Cholesky factorization, which is a specialization of the general LU.

There are specialized versions of LU factorizations for different matrix shapes, e.g. band matrices or Hessenberg matrices, and for different entry types, i.e. point-entry and block-entry variants (see [GL96]).

An important issue in factorization algorithms is *pivoting*. Conceptually, pivoting involves data movements such as the interchange of two matrix rows (and columns, in some approaches). Gaussian elimination without pivoting fails for a certain class of well-conditioned systems. In this case, we have to use pivoting. However, if pivoting is not necessary, it should be avoided since it degrades performance. We have various pivoting strategies, e.g. no pivoting, partial pivoting, or complete pivoting. Some factorization algorithms have special kinds of pivoting, e.g. symmetric pivoting or diagonal pivoting. In certain cases, e.g. when using the band version of LU factorizations, pivoting destroys the shape of the matrix. This is problematic if we want to factor dense matrices *in place*, i.e. by storing the resulting matrices in the argument matrix. In-place computation is an important optimization technique in matrix computations allowing us to avoid the movement of large amounts of data.

The pivoting code is usually scattered over the base algorithm causing the code tangling problem we discussed in Chapter 7. Thus, pivoting is an example of an aspect in the AOP sense and we need to develop mechanisms for separating the pivoting code from the base algorithm (see e.g. [ILG+97]).

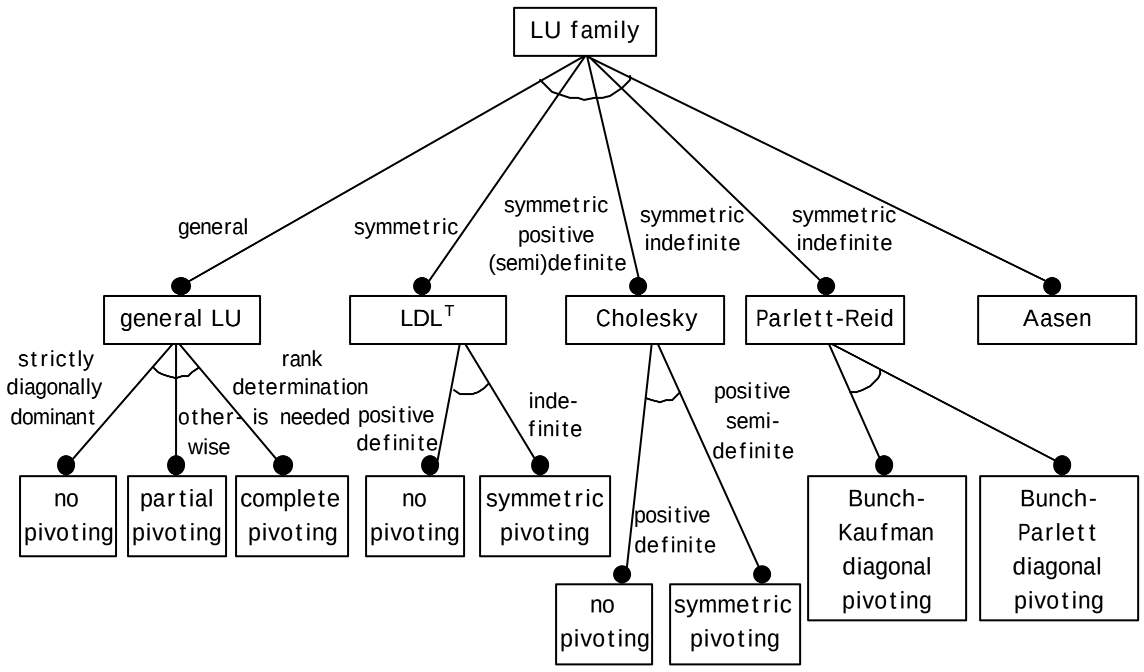


Figure 145 *Feature diagram of the LU family*

A selection of important LU factorizations is shown in Figure 145. The algorithm variants are annotated with matrix properties. An algorithm variant is well-suited for solving a given linear system if the properties of the coefficient matrix of this system match the variant's annotation.

There are also special variants of the LU factorizations for different matrix shapes (not shown in Figure 145). For example, Golub and van Loan describe specializations of general LU, LDL^T, and Cholesky for different band matrices in [GL96]. These specializations work fine without pivoting. Unfortunately, pivoting, when used, destroys the band shape of the factored matrix. As stated, this is problematic if we want to factor dense matrices in place.

In addition to shape, the algorithm selection conditions also include other mathematical properties which are not as easy to determine as shape, e.g.:

- *Positive definite*: Given $A \in R^{n \times n}$, A is positive definite if $x^T * A * x > 0$, for all nonzero $x \in R^n$.
- *Positive semidefinite*: Given $A \in R^{n \times n}$, A is positive definite if $x^T * A * x \geq 0$, for all $x \in R^n$.
- *Indefinite*: Given $A \in R^{n \times n}$, A is indefinite if $A = A^T$ and $x^T * A * x$ takes on both positive and negative values for different $x \in R^n$.

Since all the important properties of a matrix should be encoded in its type, we need to extend the matrix feature model from Section 10.1.2.2.1 with these new mathematical properties. The encoding of these properties in the matrix type allows us to arrange for the automatic selection of the most efficient algorithm variant.

One problem that we will have to address in the implementation is the type mutation in the case of in-place computation, i.e. we want to store the results in the arguments, but the results have different properties than the arguments. One possibility to address this problem is to divide a matrix into an element container and a matrix wrapper which encodes the matrix properties (see [BN94, pp. 459-464]). Given this arrangement, we can do the storing at the element container level, then mark the old matrix wrapper as invalid, and use a new one, which encodes the new matrix properties.

10.2 Domain Design

Each ADT and each algorithm family should be implemented in a separate component. In the rest of this chapter, we only present the detailed design and implementation of the matrix component.

10.2.1 Architecture of the Matrix Component

Before we discuss the architecture of the matrix component, we first take a look at a concrete example demonstrating how to use it. For our example, we have chosen the C++ implementation although later in Section 10.3.2 you will see an alternative implementation of the matrix component in the IP System. Here is the example:

```
//define a general rectangular matrix with element type double.
typedef MATRIX_GENERATOR<
        matrix< double,
                structure< rect<>
                         >
              >
>::RET RectMatrixType;

//define a scalar matrix with 3 rows and 3 columns
//scalar value is 3.4
typedef MATRIX_GENERATOR<
        matrix< double,
                structure< scalar< stat_val<int_number<int, 3> >,
                                   stat_val<float_number<double, 3400> >
                                 >
                         >
              >
>::RET ScalarMatrixType;

//declare some matrices
RectMatrixType RectMatrix1(3, 3), RectMatrix2(3, 3);
ScalarMatrixType ScalarMatrix;

//initialization of a dense matrix
RectMatrix1=
          1, 2, 3,
          4, 5, 6,
          7, 8, 9;

//multiplication of two matrices
RectMatrix2= ScalarMatrix * (RectMatrix1+ ScalarMatrix);
```

The first two gray regions indicate *two matrix configuration expressions* and the last one a *matrix expression*. The two kinds of expressions represent two important interfaces to the matrix component:

- *Matrix Configuration DSL Interface*: Configuration expressions are used to define concrete matrix types. The structure of a configuration expressions is described by the Matrix Configuration DSL (MCDSL).

- *Matrix Expression DSL Interface*: Matrix expressions are expressions involving matrices and operations on them. The structure of a matrix expressions is described by the Matrix Expression DSL (MEDSL).

Figure 146 shows the high-level architecture of the matrix component. The matrix configuration expressions are compiled by the *MCDSL generator* and the matrix expressions are compiled by the *MEDSL generator*. The MCDSL generator translates a matrix configuration expression into a matrix type by composing a number of *implementation components* (*ICs*). The MEDSL generator translates a matrix expression into an efficient implementation, e.g. by composing code fragments.

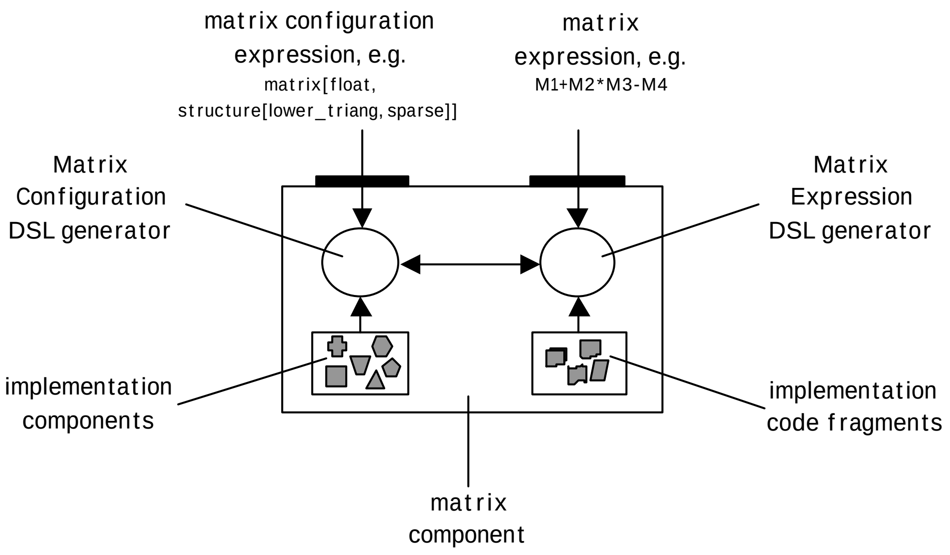

matrix configuration expression, e.g.
matrix[float, structure[lower_triang, sparse]]
matrix expression, e.g.
M1+M2*M3-M4
Matrix Configuration DSL generator
Matrix Expression DSL generator
implementation components
implementation code fragments
matrix component

Figure 146 *High-level Architecture of the Generative Matrix Component*

A more detailed view of the matrix component architecture is given in Figure 147. The pipeline on the left corresponds to the MCDSL generator and compiles matrix configuration expressions. The pipeline on the right corresponds to the MEDSL generator and compiles matrix expressions. A matrix configuration expression is compiled by parsing it (which retrieves the values of the features explicitly specified in the configuration expression), assigning default values to the unspecified features (some of which are computed), and assembling the implementation components according to the values of the features into a concrete matrix type. The matrix type also includes a *configuration repository* containing the value of all its configuration DSL features and some other types. The implementation components can be composed only into some valid configurations. These are specified by the *Implementation Components Configuration Language* (*ICCL*). In effect, the MCDSL generator has to translate matrix configuration expressions into corresponding ICCL expressions.

A matrix expression is parsed and then typed by computing the type records of all subexpressions (which requires accessing the configuration repositories of the argument matrices), and finally the efficient implementation code is generated.

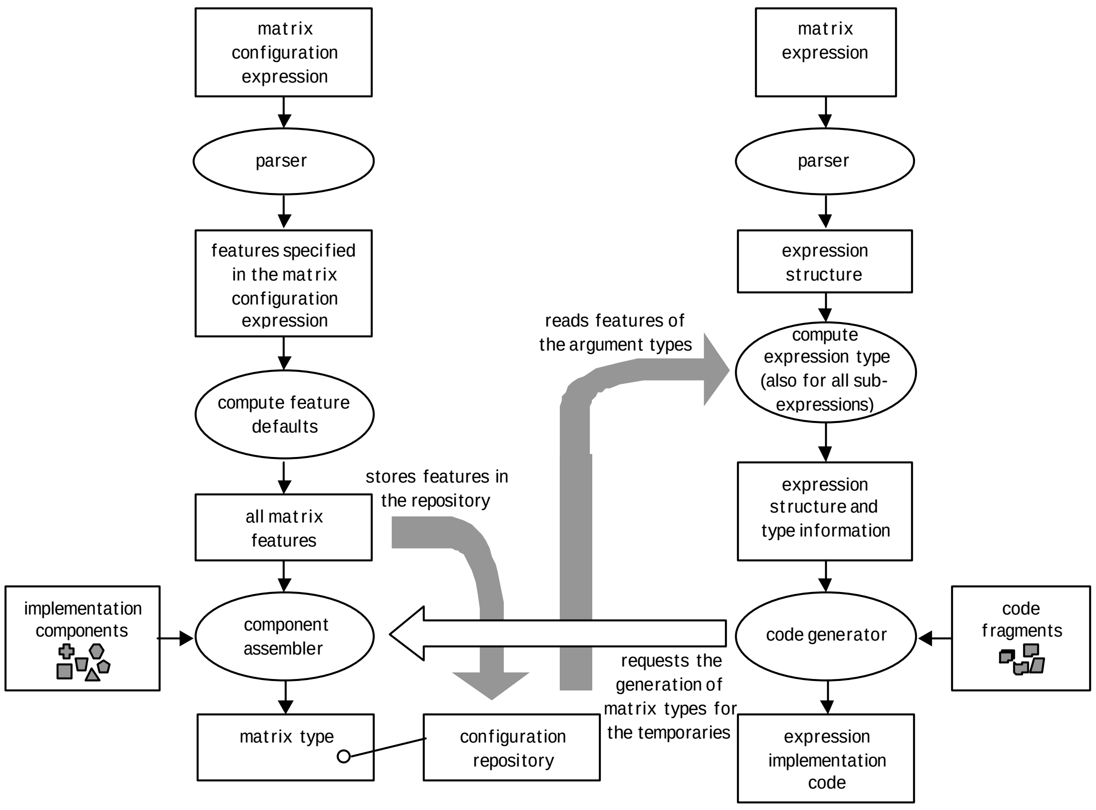

matrix configuration expression
parser
features specified in the matrix configuration expression
compute feature defaults
all matrix features
implementation components
component assembler
matrix type
configuration repository
stores features in the repository
reads features of the argument types
requests the generation of matrix types for the temporaries
matrix expression
parser
expression structure
compute expression type (also for all sub-expressions)
expression structure and type information
code generator
code fragments
expression implementation code

Figure 147 *More detailed architecture view of the matrix component*

The following sections contain the design specifications for

1. Matrix Configuration DSL (Section 10.2.3);
2. Matrix Implementation Components and the Matrix ICCL (Section 10.2.4);
3. Compiling Matrix Configuration DSL into ICCL (Section 10.2.5);
4. Matrix Expression DSL and Optimizing Matrix Expressions (Section 10.2.6);
5. Computing Result Type of the Expressions (Section 10.2.7).

Before we start with the Matrix Configuration DSL, we first need to define the scope of the matrix feature model that we are going to implement.

10.2.2 Matrix Component Scoping

The matrix component will cover a subset of the feature model we came up with in Section 10.1.2.2.1. In particular, we made the following decisions:

- *Element type*: We only support real numbers. Complex numbers are not supported, but we can add them later.
- *Subscripts*: Currently, we only support C-style indexing. Any integral type can be used as index type.
- *Entry type*: We only support point-entry matrices. Blocking can be added for dense matrices using blocking iterators, i.e. iterators that return matrices on dereferencing. Blocking iterators are described in [SL98a]. Sparse matrices would require special blocked formats (see [CHL+96]).

- *Density*: We support both dense and sparse matrices.

- *Shapes*: Currently we support rectangular, scalar, diagonal, triangular, symmetric, band diagonal, and band triangular matrices. Other shapes can be easily added.

- *Formats*: Dense matrices are stored in an array (row- or column-wise) or alternatively in a vector (diagonal-wise). The supported sparse formats include CSR, CSC, COO, DIA, and SKY.

- *Representation*: The elements are stored in dynamic or static, one or two dimensional containers. Other containers could also be easily integrated using adapters.

- *Error checking*: We provide optional bounds checking, compatibility checking, and memory allocation error checking.

- *Concurrency and synchronization*: Concurrency and synchronization are currently not supported.

- *Persistency*: We provide a way to write a matrix to a file in an uncompressed ASCII format and to read it back in.

- *Operations*: We only support matrix-matrix addition, subtraction, and multiplication.

10.2.3 Matrix Configuration DSL

The Matrix Configuration DSL is used to specify the features of a matrix configuration. We designed it according to the strategies described in Section 9.4.3. In particular, the DSL should allow the programmer to formulate matrix specifications at a level of detail which is most suitable for the particular client code. We address this goal by providing direct and computed feature defaults. If a configuration specification does not specify a feature for which we have a direct default, the direct default is assumed. The remaining features are computed from the specified features and other defaults. Furthermore, we introduced two new abstract features: *optimization flag* and *error checking flag*. The possible values of the optimization flag are *speed* and *space* and they allow us to specify whether the matrix should be optimized for speed or space. The error checking flag is used to specify the default for other error checking features. Thus, if we set the error checking flag to *check for errors*, the assumed default for bounds checking, compatibility checking, and memory allocation error checking will be to check for errors. Of course, each of the individual error checking features may be explicitly specified to have another value.

The Matrix Configuration DSL uses parameterization as its only variability mechanism, i.e. it only contains mandatory and alternative features. The reason for this is that we want to be able to represent it using C++ templates. We can always convert an arbitrary feature diagram into one that uses dimensions as its only kind of variability, although the resulting diagram will usually be more complex. If the implementation technology does not limit us to parameterization, we can represent a configuration DSL using the full feature diagram notation. For example, we could implement a GUI to specify configurations using feature diagrams directly or by menus or design some new configuration language.

The Matrix Configuration DSL specification consists of four parts:

- grammar specification,

- description of the features,

- specification of the direct feature defaults, and

- specification of the computation procedures for the computed feature defaults.

The sample grammar in Figure 148 demonstrates the notation we will use later to specify the Matrix Configuration DSL grammar. It describes a very simple matrix concept and is equivalent to the feature diagram in Figure 149. We use brackets to enclose *parameters* (i.e. dimensions) and a vertical bar to separate *alternative parameter values*. The symbols on the right are the *nonterminal symbols*. The first nonterminal (i.e. Matrix) represents the concept. The remaining nonterminals are the parameters (i.e. ElementType, Shape, Format, ArrayOrder, OptimizationFlag). As you may remember from Section 6.4.2, we used this kind of grammars to specify GenVoca architectures.

```
Matrix:            matrix[ElementType, Shape, Format, OptimizationFlag]
ElementType:       real | complex
Shape:             rectangular | lowerTriangular | upperTriangular
Format:            array[ArrayOrder] | vector
ArrayOrder:        cLike | fortranLike
OptimizationFlag:  speed | space
```

Figure 148 *Sample configuration DSL grammar*

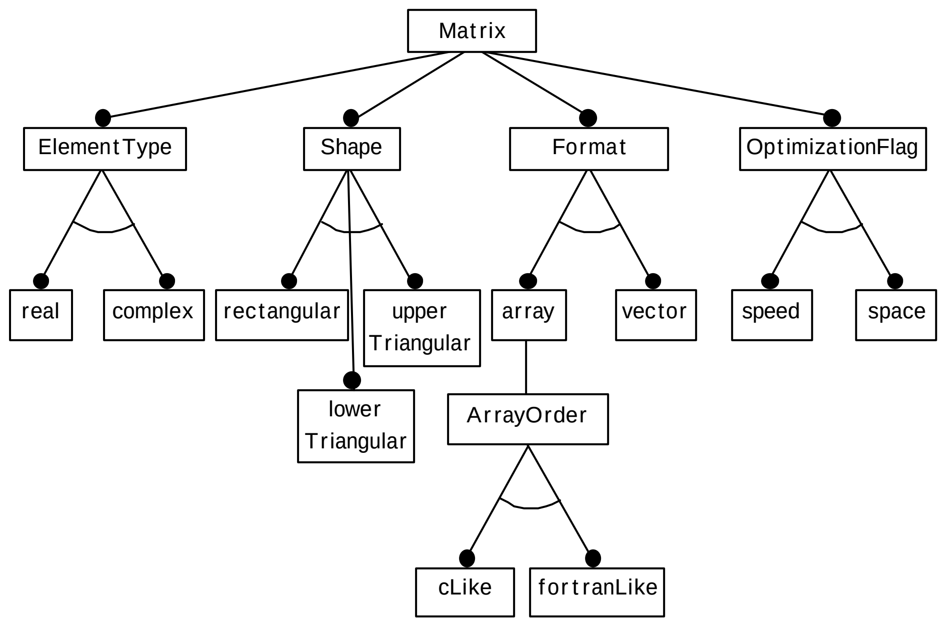


Figure 149 *Feature diagram equivalent to the sample grammar in Figure 148*

Our sample grammar defines 2*3*(2+1)*2 = 36 valid configuration expressions:

```
matrix[real,rectangular,array[cLike],speed]
matrix[real,rectangular,array[cLike],space]
matrix[real,rectangular,array[fortranLike],speed]
matrix[real,rectangular,array[fortranLike],space]
matrix[real,rectangular,vector,speed]
matrix[real,rectangular,vector,space]
matrix[real,lowerTriangular,array[cLike],speed]
matrix[real,lowerTriangular,array[cLike],space]
matrix[real,lowerTriangular,array[fortranLike],speed]
matrix[real,lowerTriangular,array[fortranLike],space]
matrix[real,lowerTriangular,vector,speed]
matrix[real,lowerTriangular,vector,space]
matrix[real,upperTriangular,array[cLike],speed]
matrix[real,upperTriangular,array[cLike],space]
matrix[real,upperTriangular,array[fortranLike],speed]
matrix[real,upperTriangular,array[fortranLike],space]
matrix[real,upperTriangular,vector,speed]
matrix[real,upperTriangular,vector,space]
```

```
matrix[complex,rectangular,array[cLike],speed]
matrix[complex,rectangular,array[cLike],space]
matrix[complex,rectangular,array[fortranLike],speed]
matrix[complex,rectangular,array[fortranLike],space]
matrix[complex,rectangular,vector,speed]
matrix[complex,rectangular,vector,space]
matrix[complex,lowerTriangular,array[cLike],speed]
matrix[complex,lowerTriangular,array[cLike],space]
matrix[complex,lowerTriangular,array[fortranLike],speed]
matrix[complex,lowerTriangular,array[fortranLike],space]
matrix[complex,lowerTriangular,vector,speed]
matrix[complex,lowerTriangular,vector,space]
matrix[complex,upperTriangular,array[cLike],speed]
matrix[complex,upperTriangular,array[cLike],space]
matrix[complex,upperTriangular,array[fortranLike],speed]
matrix[complex,upperTriangular,array[fortranLike],space]
matrix[complex,upperTriangular,vector,speed]
matrix[complex,upperTriangular,vector,space]
```

Next, we assume that the parameters of our simple grammar have the direct defaults specified in Table 23. In general, there is usually no such thing as the "absolute" choice for a default value. We often have a strong feeling about some defaults and some others are quite arbitrary. For example, we choose rectangular for Shape since a rectangular matrix can also hold a triangular matrix. This reflects a common principle: We usually choose the most general value to be the default value. This principle, however, does not have to be followed in all cases. Even if complex is more general than real, we still select real as the default value for ElementType. This is so since real reflects the more common case. Of course, what is the more common case and what not depends on the context, our knowledge, etc. Finally, we could have features for which no reasonable defaults can be assumed at all, e.g. the static number of rows and columns of a matrix. These features must be specified explicitly, otherwise we have an error.

ElementType:	real
Shape:	rectangular
ArrayOrder:	cLike
OptimizationFlag:	space

Table 23 *Direct feature defaults for the sample grammar in Figure 148*

Please note that we did not specify a direct default for Format. The reason is that we can compute it based on Shape and OptimizationFlag. The default will be computed as follows. If the value of Shape is rectangular and Format is not specified, we assume Format to be array since (dense) rectangular matrices are optimally stored in an array. If the value of Shape is triangular and Format is not specified, the value of Format depends on OptimizationFlag. If OptimizationFlag is space, we assume Format to be vector since the vector format stores only the nonzero half of the matrix (e.g. diagonal-wise). In this case, the element access functions have to convert the two-dimensional subscripts of a matrix into the one-dimensional subscripts of a vector. If, on the other hand, OptimizationFlag is speed, we assume Format to be array. Storing a triangular matrix in an array wastes space but it allows a faster element access since we do not have to convert indices. This dependency between Shape, OptimizationFlag, and Format is specified in Table 24.

Shape	OptimizationFlag	Format
rectangular	*	array
lowerTriangular upperTriangular	speed	array
	space	vector

Table 24 *Computing the DSL feature Format for the sample grammar in Figure 148*

We will use tables similar to Table 24 for specifying any dependencies between features. We refer to them as *feature dependency tables*.

Each dependency table represents a function. The columns to the left of the double vertical divider specify the arguments and the columns to the right specify the corresponding result values. For example, Table 25 specifies how to compute the product of two numbers. A dependency table is evaluated row-wise from top to bottom. Given some concrete Factor1 and Factor2, you try to match them to the values specified in the first row. If they match, you terminate the search and take the result from the last corresponding Product cell. If they do not match, you proceed with the next row.

The argument cells of a dependency table may contain one of the following:

- one or more concrete values for the corresponding variable; multiple values are interpreted as alternatives;
- "*", which matches any value;
- a local variable, e.g. (factor); local variables are enclosed in parentheses and denote the current value of the argument;
- "---", which indicates that the corresponding argument does not apply to this row; in terms of matching, it is equivalent to "*";

The result cells may contain one of the following:

- a concrete value,
- an expression; an expression starts with "=" and can refer to the table arguments and local variables;

Factor1	**Factor2**	**Product**
0	*	0
*	0	0
*	1	= Factor1
1	*	= Factor2
(factor)	(factor)	=(factor)^2
25	75	1875
*	*	= Factor1*Factor2

Table 25 *Sample specification of the product of two factors [Neu98]*

Now there is the question how the feature defaults are used. Feature defaults allow us to leave out some features in a configuration expression. For example, we could specify a matrix as simply as

```
matrix[]
```

Given the defaults specified above, this expression is equivalent to

```
matrix[real,rectangular,array[cLike],space]
```

How did we come up with this expression? This is simple. We took the default values for ElementType, Shape, and OptimizationFlag directly from Table 23 and then we determined Format based on Table 24. Finally, we took the value for ArrayOrder from Table 23. Other examples are shown in Table 26. Please note that we can leave out one or more parameters of a parameter list only if they constitute the last *n* parameters in the list (this corresponds to the way parameter defaults are used in C++ class templates).

Abbreviated expression	Equivalent, fully expanded expression
matrix[complex]	matrix[complex,rectangular,array[cLike],space]
matrix[real,lowerTriangular]	matrix[real,lowerTriangular, vector,space]
matrix[real,lowerTriangular, array[]]	matrix[real,lowerTriangular,array[cLike],space]
matrix[real,rectangular, vector]	matrix[real,rectangular, vector,space]

Table 26 *Examples of expressions abbreviated by leaving out trailing parameters*

There is also a method for not specifying a parameter in the middle of a parameter list: we use the value unspecified. Some examples are shown in Table 27.

Expression with unspecified	Equivalent, fully expanded expression
matrix[complex,upperTriangular,unspecified,speed]	matrix[complex,upperTriangular,array[cLike],speed]
matrix[complex,unspecified,unspecified,speed]	matrix[complex,rectangular,array[cLike],speed]

Table 27 *Examples of expressions with unspecified values*

10.2.3.1 Grammar of the Matrix Configuration DSL

The grammar of the Matrix Specification DSL is shown in Figure 150. All the features are explained in the following section.

Matrix:	matrix[ElementType, Structure, OptFlag, ErrFlag, BoundsChecking, CompatChecking, IndexType]
ElementType:	float | double | long double | short | int | long | unsigned short | unsigned int | unsigned long
Structure:	structure[Shape, Density, Malloc]
Shape:	rect[Rows, Cols, RectFormat] | diag[Order] | scalar[Order, ScalarValue] | ident[Order] | zero[Order] | lowerTriang[Order, LowerTriangFormat] | upperTriang[Order, UpperTriangFormat] | symm[Order, SymmFormat] | bandDiag[Order, Diags, BandDiagFormat] | lowerBandTriang[Order, Diags, LowerBandTriangFormat] | upperBandTriang[Order, Diags, UpperBandTriangFormat]
RectFormat:	array[ArrOrder] | CSR | CSC | COO[DictFormat]
LowerTriangFormat:	vector | array[ArrOrder] | DIA | SKY
UpperTriangFormat:	vector | array[ArrOrder] | DIA | SKY
SymmFormat:	vector | array[ArrOrder] | DIA | SKY
BandDiagFormat:	vector | array[ArrOrder] | DIA
LowerBandTriangFormat:	vector | DIA | SKY
UpperBandTriangFormat:	vector | DIA | SKY
ArrOrder:	cLike, fortranLike
DictFormat:	hashDictionary[HashWidth] | listDictionary
Density:	dense | sparse[Ratio, Growing]
Malloc:	fix[Size] | dyn[MallocErrChecking]
MallocErrChecking:	checkMallocErr | noMallocErrChecking
OptFlag:	speed | space
ErrFlag:	checkAsDefault | noChecking
BoundsChecking:	checkBounds | noBoundsChecking
CompatChecking:	checkCompat | noCompatChecking
IndexType:	char | short | int | long | unsigned char | unsigned short | unsigned int | unsigned long | signed char
Rows:	statVal[RowsNumber] | dynVal
Cols:	statVal[ColsNumber] | dynVal
Order:	statVal[OrderNumber] | dynVal
Diags:	statVal[DiagsNumber] | dynVal
ScalarValue:	statVal[ScalarValueNumber] | dynVal
Ratio, Growing:	float_number[Type, Value]
RowsNumber, ColsNumber, OrderNumber, DiagsNumber, Size, HashWidth:	int_number[Type, Value]
ScalarValueNumber:	float_number[Type, Value] | int_number[Type, Value]

Figure 150 *Grammar of the Matrix Configuration DSL*

10.2.3.2 Description of the Features of the Matrix Configuration DSL

Figure 151 through Figure 166 show the entire feature diagram for the Matrix Configuration DSL. Each single diagram covers some part of the complete feature diagram. The partial diagrams are followed by tables explaining the features they contain. Each feature has a traceability link back to the section describing its purpose.

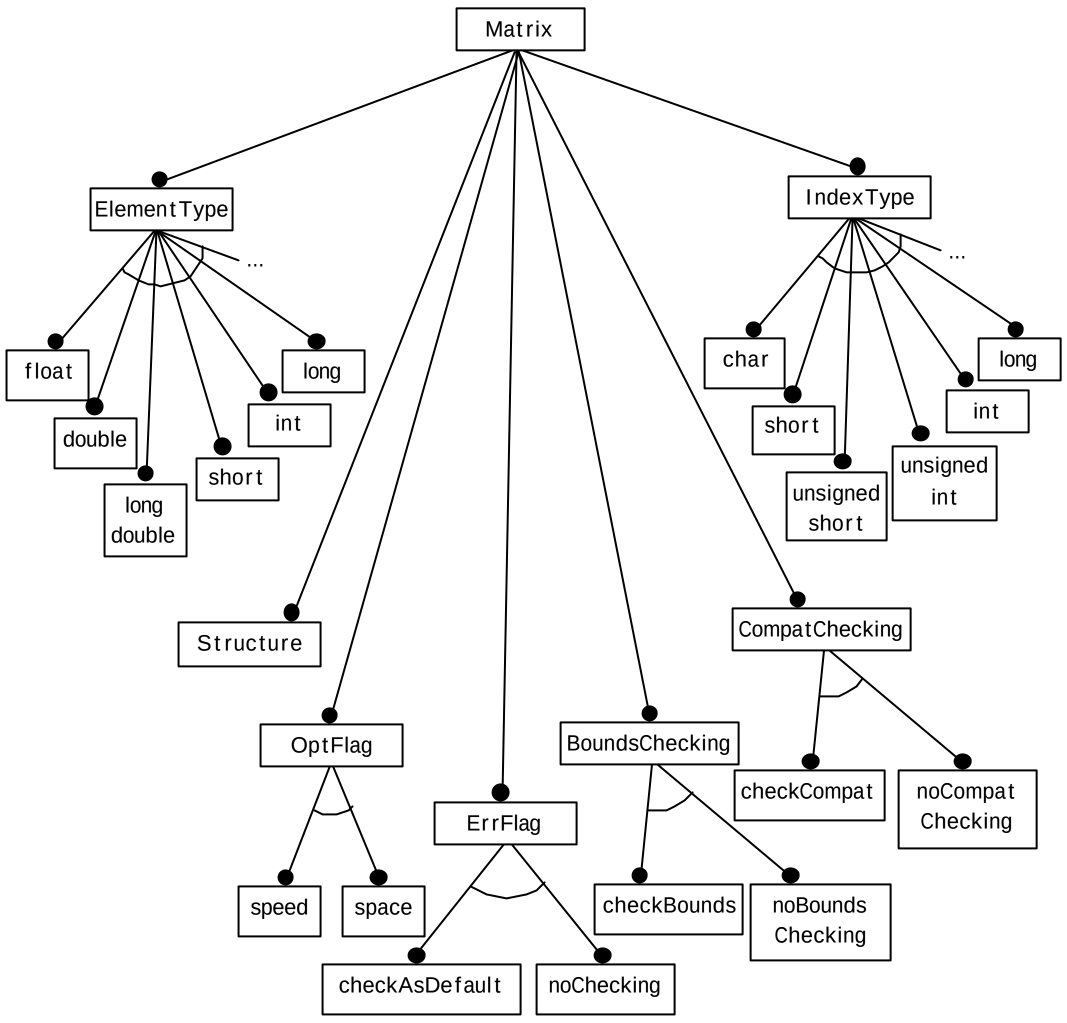


Figure 151 *Matrix features (see Table 28 for explanations)*

Feature name	Description	
ElementType	Type of the matrix elements (see Section 10.1.2.2.1.1).	
	Subfeatures	Possible values of *ElementType* include *float*, *double*, *long double*, *short*, *int*, *long*, *unsigned short*, *unsigned int*, and *unsigned long*. Other number types, if supported on the target platform, are also possible. Note: *complex* currently not supported
Structure	Structure describes shape, density, and memory allocation strategy (but also, indirectly, format and representation) of a matrix (see Section 10.1.2.2.1.3).	
	Subfeatures	see Figure 152
OptFlag	Optimization flag indicating whether the matrix should be optimized for speed or space (see 10.2.3).	
	Subfeatures	Possible values of *OptFlag*: *speed*, *space*
ErrFlag	Error checking flag determines whether all error checking should be done or no checking by default (see 10.2.3). The default is used for a specific error checking feature (e.g. *BoundsChecking* or *CompatChecking*) only if the feature is not specified by the user. For example, if *BoundsChecking* is not specified and *ErrFlag* is *checkAsDefault*, then bounds checking is *checkBounds*.	
	Subfeatures	Possible values of *ErrFlag*: *checkAsDefault*, *noChecking*
BoundsChecking	Bounds checking flag determines whether the validity of the indices used in each access operation to the matrix elements is checked or not (see Section 10.1.2.2.1.4.1).	
	Subfeatures	Possible values of *BoundsChecking*: *checkBounds*, *noBoundsChecking*
CompatChecking	Compatibility checking flag determines whether the compatibility of sizes of the arguments to an operation is checked or not (see Section 10.1.2.2.1.4.2). (For example, two matrices are compatible for multiplication if the number of rows of the first matrix is equal to the number of columns of the second matrix.)	
	Subfeatures	Possible values of *CompatChecking*: *checkCompat*, *noCompatChecking*
IndexType	Type of the index used to address matrix elements (see Section 10.1.2.2.1.2).	
	Subfeatures	Possible values of *IndexType* include *char*, *short*, *int*, *long*, *unsigned char*, *unsigned short*, *unsigned int*, *unsigned long*, and *signed char*. Other number types, if supported on the target platform, are also possible.

Table 28 *Description of matrix features*

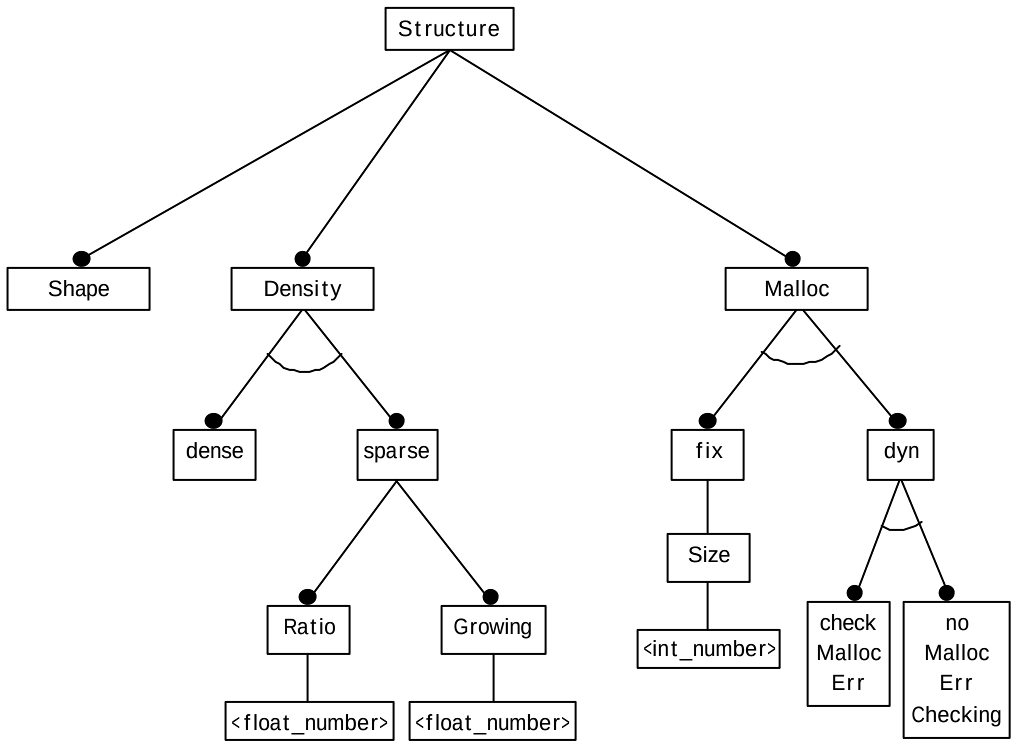


Figure 152 *Subfeatures of Structure (see Table 29 for explanations)*

Feature name	**Description**	
Shape	Shape describes matrix shape, but also, indirectly, format and representation (see Section 10.1.2.2.1.3.3).	
	Subfeatures	see Figure 153
Density	Density specifies whether a matrix is sparse or dense (see Section 10.1.2.2.1.3.2).	
	Subfeatures	Possible values of *Density*: *sparse, dense* *sparse* has two additional subfeatures: *Ratio* and *Growing*. *Ratio* specifies the estimated number of nonzero elements divided by the total number of elements of a matrix, i.e. it is a number between 0 and 1. *Growing* ratio specifies the relative density growth. The density of a matrix grows when nonzero numbers are assigned to zero elements or it decreases when zero is assigned to nonzero elements. *Growing* specifies the relative density change per time unit and it is a float number between 0 and +∞. 0 means no change. 1 means the doubling of the number of the nonzero elements. For example, *Growing* 1 means that the number of nonzero elements grows by factor 4 over two time units. In the matrix implementation, a time unit is the time between points at which the element container allocates extra memory.
Malloc	Memory allocation strategy for the matrix element container (see Section 10.1.2.2.1.3.5).	
	Subfeatures	Possible values of *Malloc*: *fix, dyn* *fix* implies static allocation. *dyn* implies dynamic allocation. *fix* has *Size* as its subfeature. *Size* specifies the size of the memory block which is statically allocated for the matrix elements. *Size* specifies only one dimension, i.e. the actual size of the allocated memory block is *Size* *size_of(*ElementType*) for 1D containers and *Size* * *Size* * size_of(*ElementType*) for 2D containers. If the number of rows and the number of columns are specified statically, *Size* is ignored. *dyn* has two alternative subfeatures: *checkMallocErr*, *noMallocChecking*. *checkMallocErr* implies checking for memory allocation errors. *noMallocChecking* implies no checking.

Table 29 *Description of subfeatures of Structure*

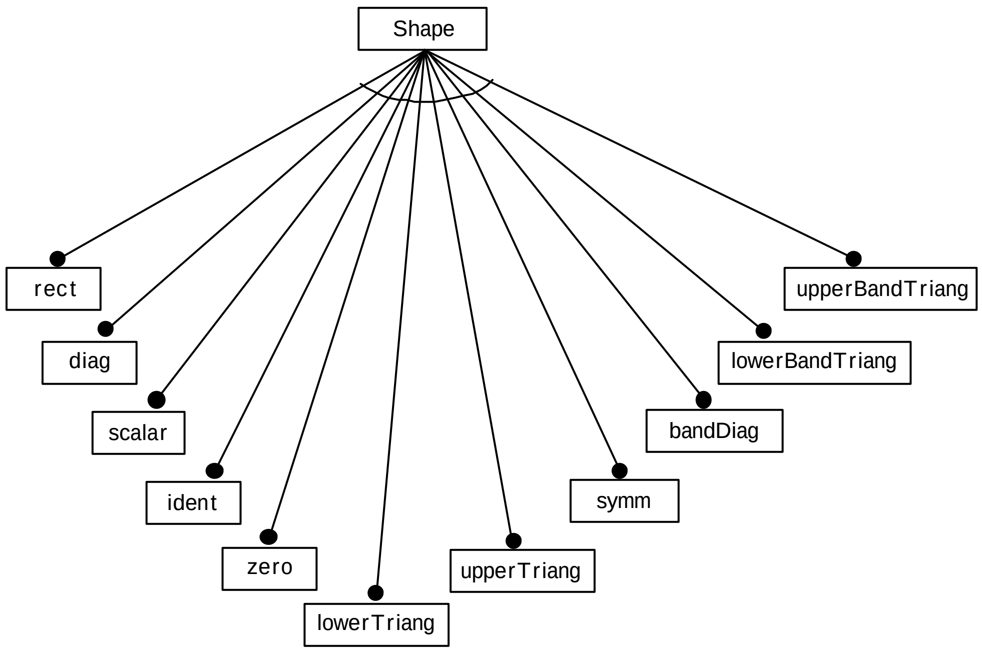


Figure 153 *Subfeatures of Shape (see Table 30 for explanations)*

Feature name	Description
rect	Rectangular matrix (see Figure 154 for subfeatures).
diag	Diagonal matrix (see Figure 155 for subfeatures).
scalar	Scalar matrix (see Figure 156 for subfeatures).
ident	Identity matrix (see Figure 157 for subfeatures).
zero	Zero matrix (see Figure 158 for subfeatures).
lowerTriang	Lower triangular matrix (see Figure 159 for subfeatures).
upperTriang	Upper triangular matrix (see Figure 160 for subfeatures).
symm	Symmetric matrix (see Figure 161 for subfeatures).
bandDiag	Band diagonal matrix (see Figure 162 for subfeatures).
lowerBandTriang	Lower band triangular matrix (see Figure 163 for subfeatures).
upperBandTriang	Upper band triangular matrix (see Figure 164 for subfeatures).

Table 30 *Description of subfeatures of Shape (see Section 10.1.2.2.1.3.3 for explanations)*

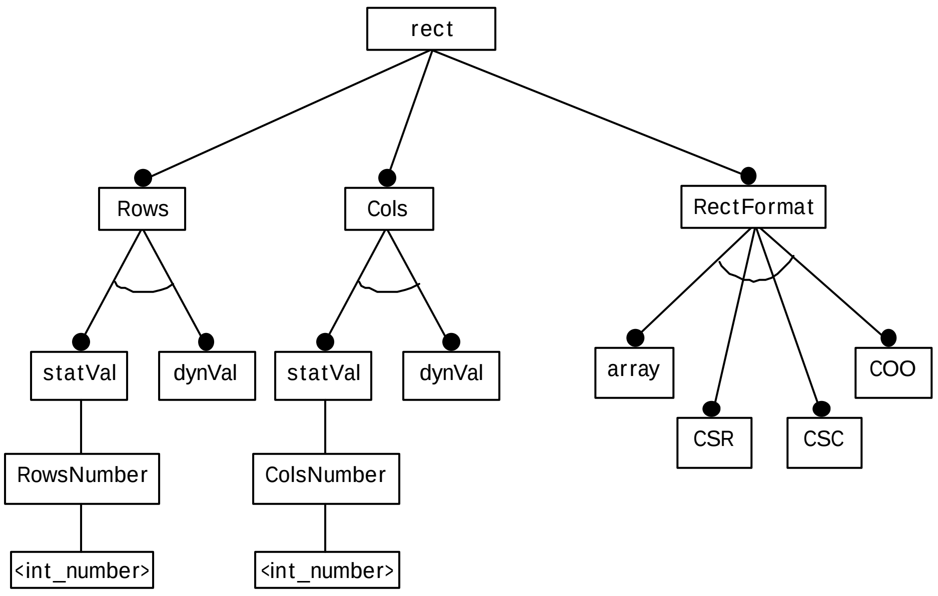


Figure 154 *Subfeatures of rect (see Table 31 for explanations)*

Feature name	**Description**	
Rows	*Rows* allows us to specify the number of rows of a matrix statically or to indicate that the number can be set at runtime (see Section 10.1.2.2.1.7).	
	Subfeatures	Possible values of *Rows*: *statVal*, *dynVal* *statVal* indicates that the number of rows is specified statically. The subfeature *RowsNumber* specifies the number of rows. *dynVal* indicates that the number of rows is specified dynamically.
Cols	*Cols* allows us to specify the number of columns of a matrix statically or to indicate that the number can be set at runtime (see Section 10.1.2.2.1.7).	
	Subfeatures	Possible values of *Rows*: *statVal*, *dynVal* *statVal* indicates that the number of columns is specified statically. The subfeature *ColsNumber* specifies the number of columns. *dynVal* indicates that the number of columns is specified dynamically.
RectFormat	*RectFormat* specifies the format of a rectangular matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *RectFormat*: *array*, *CSC*, *CSR*, *COO* *array* implies column- or row-wise storage in a two-dimensional vector (see Section 10.1.2.2.1.3.6.1). *CSC* implies compressed sparse column format (see Section 10.1.2.2.1.3.6.2). *CSR* implies compressed sparse row format (see Section 10.1.2.2.1.3.6.2). COO coordinate format (see Section 10.1.2.2.1.3.6.2 and Table 41).

Table 31 *Description of subfeatures of rect*

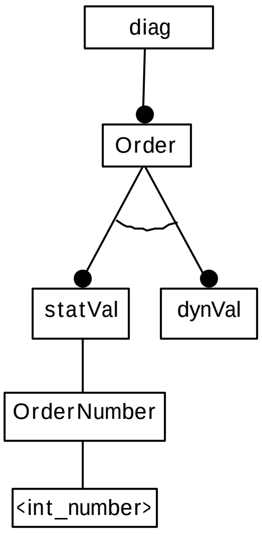


Figure 155 *Subfeatures of diag (see Table 32 for explanations)*

Feature name	**Description**	
Order	*Order* describes the number of columns and rows of a square matrix (see Section 10.1.2.2.1.7). We can specify order statically or indicate that it can be set at runtime.	
	Subfeatures	Possible values of *Order*: *statVal*, *dynVal* *statVal* indicates that order is specified statically. The subfeature *OrderNumber* specifies the order value. *dynVal* indicates that order is specified dynamically.

Table 32 *Description of Order, subfeature of diag, scalar, ident, zero, lowerTriang, upperTriang, symm, bandDiag, lowerBandDiag, and upperBandDiag*

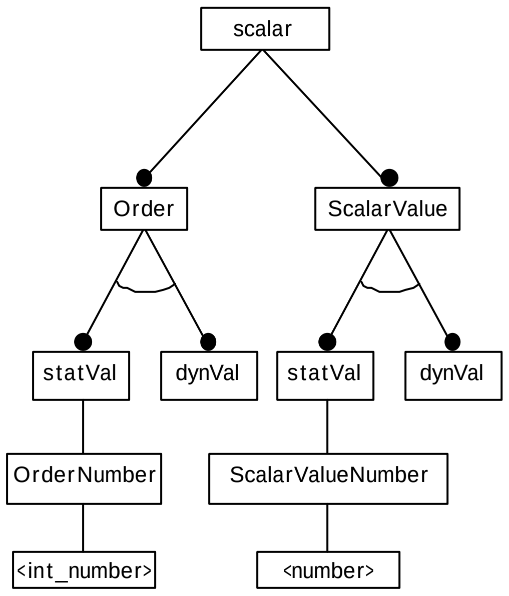


Figure 156 *Subfeatures of scalar (see Table 33 for explanations)*

Feature name	**Description**	
Order	see Table 32	
ScalarValue	*ScalarValue* allows us to specify the scalar value of a scalar matrix (i.e. the value of the diagonal elements) statically or to indicate that the value can be set at runtime (see Section 10.1.2.2.1.3.3).	
	Subfeatures	Possible values of *ScalarValue*: *statVal*, *dynVal* *statVal* indicates that the scalar value is specified statically. The subfeature *ScalarValueNumber* specifies the scalar value. *dynVal* indicates that the scalar value is specified dynamically.

Table 33 *Description of subfeatures of scalar*

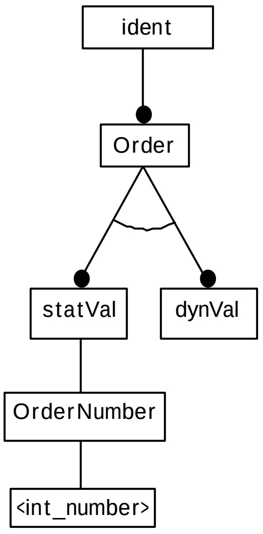


Figure 157 *Subfeatures of ident (see Table 32 for explanations)*

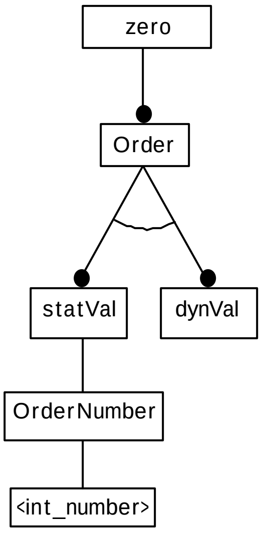


Figure 158 *Subfeatures of zero (see Table 32 for explanations)*

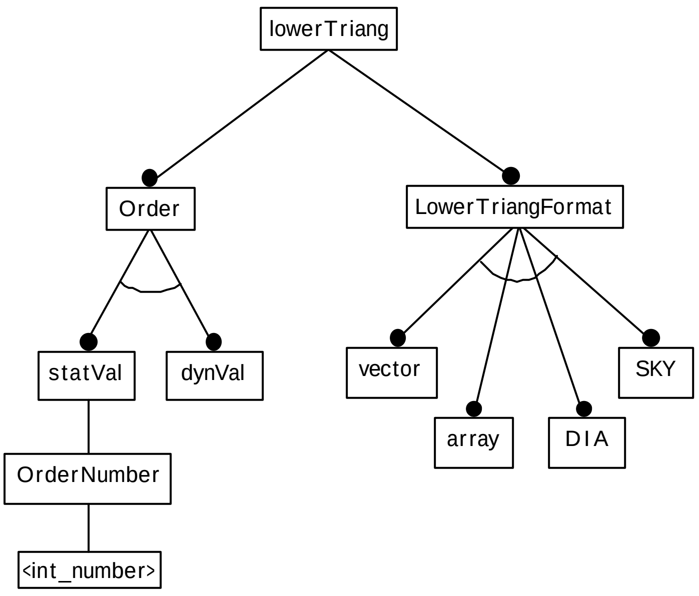


Figure 159 *Subfeatures of lowerTriang (see Table 34 for explanations)*

Feature name	Description	
Order	see Table 32	
LowerTriangFormat	*LowerTriangFormat* specifies the format of a lower triangular matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *LowerTriangFormat*: *vector, array, DIA, SKY* *vector* implies diagonal-wise storage in a vector. *array* implies column- or row-wise storage in a two-dimensional vector (see Section 10.1.2.2.1.3.6.1). *DIA* implies diagonal sparse format (see Section 10.1.2.2.1.3.6.2). *SKY* implies skyline format (see Section 10.1.2.2.1.3.6.2).

Table 34 *Description of subfeatures of lowerTriang*

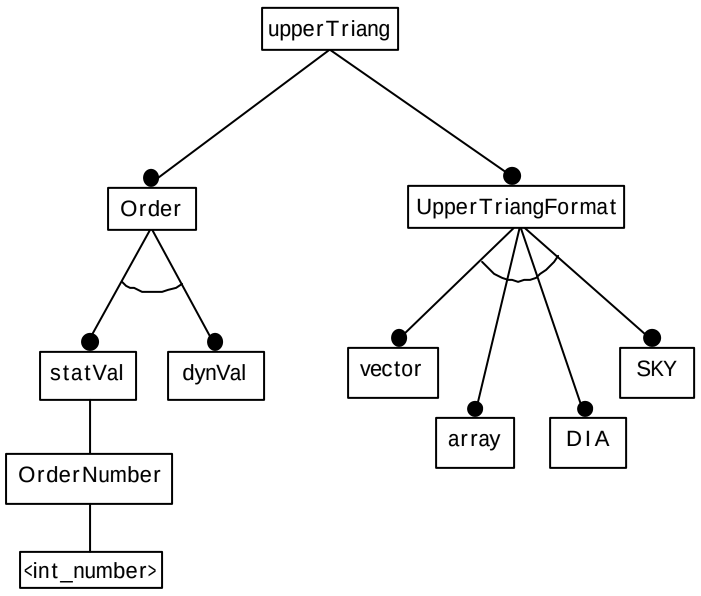


Figure 160 *Subfeatures of upperTriang (see Table 35 for explanations)*

Feature name	Description	
Order	see Table 32	
UpperTriangFormat	*UpperTriangFormat* specifies the format of an upper triangular matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *UpperTriangFormat*: *vector, array, DIA, SKY* *vector* implies diagonal-wise storage in a vector. *array* implies column- or row-wise in a two-dimensional vector (see Section 10.1.2.2.1.3.6.1). *DIA* implies diagonal sparse format (see Section 10.1.2.2.1.3.6.2). *SKY* implies skyline format (see Section 10.1.2.2.1.3.6.2).

Table 35 *Description of subfeatures of upperTriang*

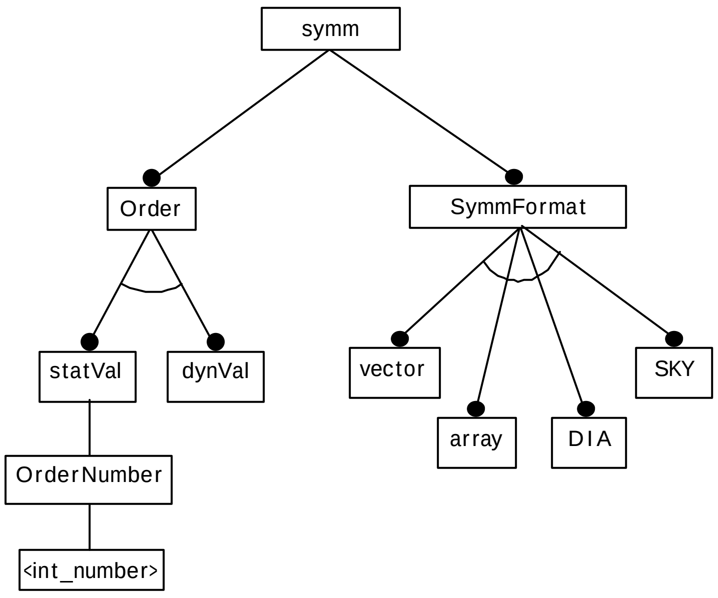


Figure 161 *Subfeatures of symm (see Table 36 for explanations)*

Feature name	**Description**	
Order	see Table 32	
SymmFormat	*SymmFormat* specifies the format of a symmetric matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *SymmFormat*: *vector*, *array*, *DIA*, *SKY* *vector* implies diagonal-wise storage in a vector. *array* implies column- or row-wise storage in a two-dimensional vector (see Section 10.1.2.2.1.3.6.1). *DIA* implies diagonal sparse format (see Section 10.1.2.2.1.3.6.2). *SKY* implies skyline format (see Section 10.1.2.2.1.3.6.2).

Table 36 *Description of subfeatures of symm*

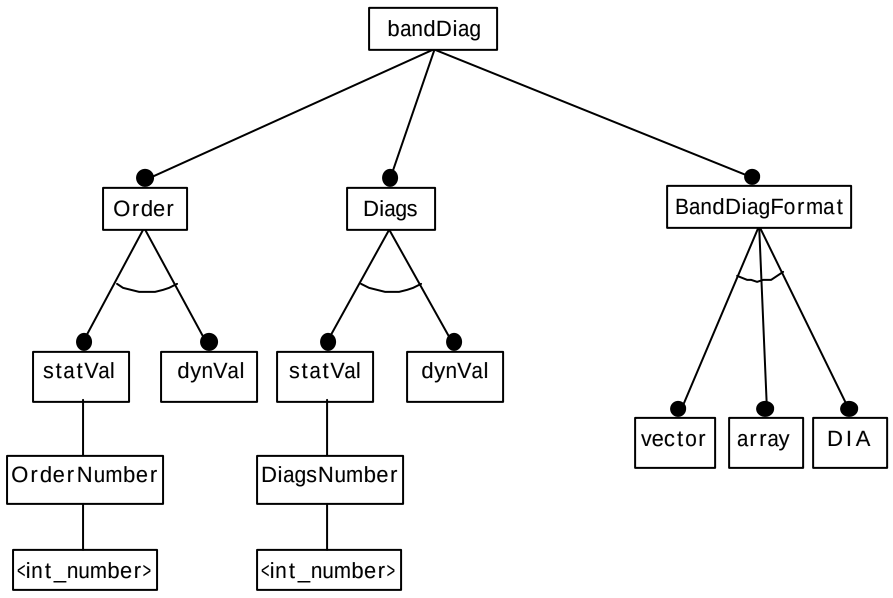


Figure 162 *Subfeatures of bandDiag (see Table 37 for explanations)*

Feature name	**Description**	
Order	see Table 32	
Diags	*Diags* allows us to statically specify the bandwidth (i.e. number of nonzero diagonals) of a band matrix or to indicate that the bandwidth can be set at runtime (see Section 10.1.2.2.1.7).	
	Subfeatures	Possible values of *Rows*: *statVal*, *dynVal* *statVal* indicates that the bandwidth is specified statically. The subfeature *DiagsNumber* specifies the number of nonzero diagonals. *dynVal* indicates that the bandwidth is specified dynamically.
BandDiagFormat	*BandDiagFormat* specifies the format of a band diagonal matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *BandDiagFormat*: *vector*, *array*, *DIA* *vector* implies diagonal-wise storage in a vector. *array* implies column- or row-wise storage in a two-dimensional vector (see Section 10.1.2.2.1.3.6.1). *DIA* implies diagonal sparse format (see Section 10.1.2.2.1.3.6.2).

Table 37 *Description of subfeatures of bandDiag*

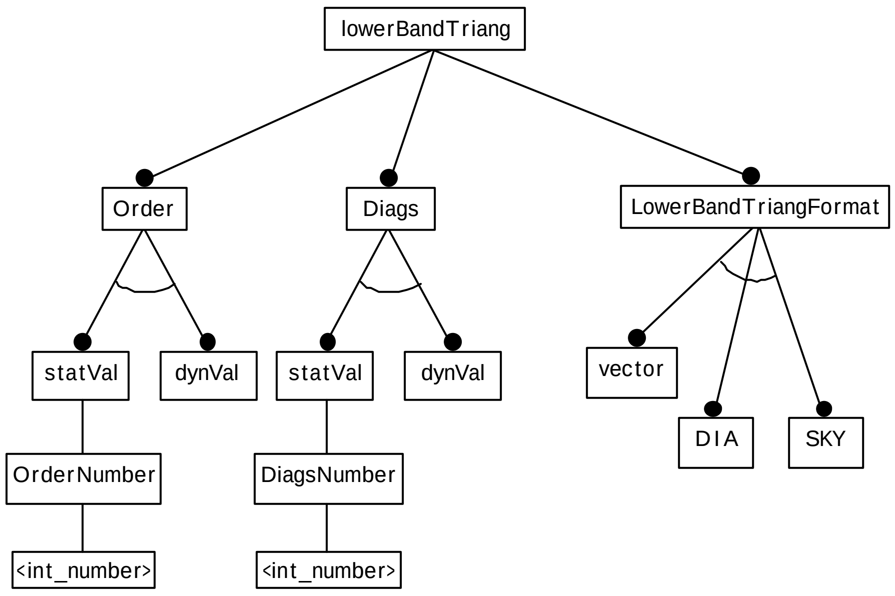


Figure 163 *Subfeatures of lowerBandTriang (see Table 38 for explanations)*

Feature name	**Description**	
Order	see Table 32	
Diags	see Diags in Table 37	
LowerBandTriangFormat	*LowerBandTriangFormat* specifies the format of a lower triangular matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *LowerBandTriangFormat*: *vector, DIA, SKY* *vector* implies diagonal-wise storage in a vector. *DIA* implies diagonal sparse format (see Section 10.1.2.2.1.3.6.2). *SKY* implies skyline format (see Section 10.1.2.2.1.3.6.2).

Table 38 *Description of subfeatures of lowerBandTriang*

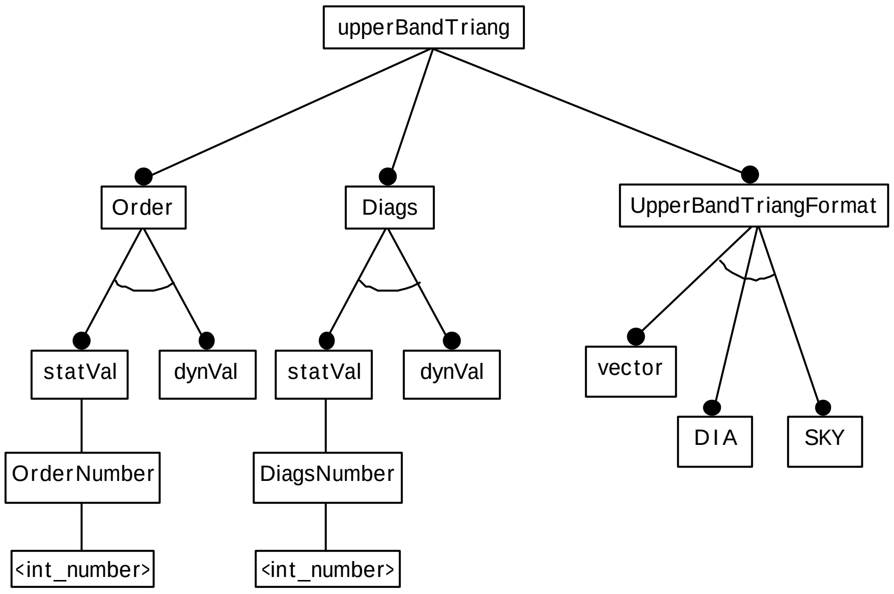


Figure 164 *Subfeatures of upperBandTriang (see Table 39 for explanations)*

Feature name	**Description**	
Order	see Table 32	
Diags	see Diags in Table 37	
UpperBandTriangFormat	*UpperBandTriangFormat* specifies the format of an upper band triangular matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *UpperBandTriangFormat*: *vector, DIA, SKY* *vector* implies diagonal-wise storage in a vector. *DIA* implies diagonal sparse format (see Section 10.1.2.2.1.3.6.2). *SKY* implies skyline format (see Section 10.1.2.2.1.3.6.2).

Table 39 *Description of subfeatures upperBandTriang*

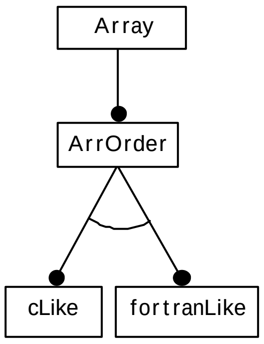


Figure 165 *Subfeatures of Array (see Table 40 for explanations)*

Feature name	Description	
ArrOrder	*ArrOrder* specifies whether to store elements row- or column-wise (see Section 10.1.2.2.1.3.6.1).	
	Subfeatures	Possible values of *Array*: *cLike, fortranLike* *CLike* implies row-wise storage. *fortranLike* implies column-wise storage.

Table 40 *Description of subfeatures of Array*

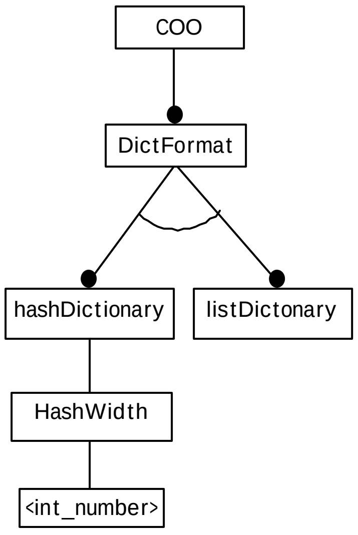


Figure 166 *Subfeatures of COO (see Table 41 for explanations)*

Feature name	Description	
DictFormat	*DictFormat* specifies the dictionary to be used for the COO matrix format (see Section 10.1.2.2.1.3.6.2).	
	Subfeatures	Possible values of *DictFormat*: *hashDict*, *listDict* *hashDict* implies a dictionary with a hashed key. The subfeature *HashWidth* is used in the hash function.[148] *listDict* implies a dictionary implemented using three vectors: two index vectors (for rows and columns indices) and one value vector.

Table 41 *Description of subfeatures of COO*

10.2.3.3 Direct Feature Defaults

A subset of the Matrix Configuration DSL parameters have direct defaults. In Table 42, we propose some, to our taste, reasonable default values. If necessary, they can be easily modified. Please note that some values are specified as ---. This symbol indicates that no reasonable default value exists for the corresponding parameter and, if the parameter is relevant in a given configuration, it has to be specified explicitly.

ElementType:	double
Structure:	structure
Shape:	rect
Malloc:	dyn
ArrOrder:	cLike
DictFormat:	hashDictionary
OptFlag:	space
ErrFlag:	checkAsDefault
IndexType:	unsigned int
Rows:	dynVal
Cols:	dynVal
Order:	dynVal
Diags:	dynVal
ScalarValue:	dynVal
RowsNumber:	---
ColsNumber:	---
OrderNumber:	---
DiagsNumber:	---
ScalarValueNumber:	---
Size:	int_number[int, 100]
Ratio:	float_number[double, 0.1]
Growing:	float_number[double, 0.25]
HashWidth:	int_number[IndexType, 1013]

Table 42 *Direct feature defaults for the Matrix Configuration DSL [Neu98]*

Density represents a special case. If Density is specified and Format[149] is not, Density is used to compute Format. On the other hand, if Format is specified and Density is not, Format is used to compute Density. Finally, if neither Format nor Density is specified, we use the following default for Density (and then compute Format):

Density:	dense

10.2.3.4 Computed Feature Defaults

The default values of the following parameters are computed:

Density
MallocChecking

BoundsChecking
CompatChecking
RectFormat
LowerTriangFormat
UpperTriangFormat
SymmFormat
BandDiagFormat
LowerBandTriangFormat
UpperBandTriang

Table 43 through Table 51 specify how to compute them. The semantics of the feature dependencies tables are explained in Section 10.2.3.

10.2.3.4.1 Density

Format	**Density**
array	dense
vector	dense
CSR	sparse
CSC	sparse
COO	sparse
DIA	sparse
SKY	sparse
*	dense (from Section 10.2.3.3)

Table 43 *Computing default value for* Density *[Neu98]*

10.2.3.4.2 Error Checking Features

ErrFlag is used as the primary default for all error checking features (see Section 10.2.3).

ErrFlag	**MallocChecking**
checkAsDefault	checkMallocErr
noChecking	noMallocErrChecking

Table 44 *Computing default value for MallocChecking [Neu98]*

ErrFlag	**BoundsChecking**
checkAsDefault	checkBounds
noChecking	noBoundsChecking

Table 45 *Computing default value for BoundsChecking [Neu98]*

ErrFlag	**CompatChecking**
checkAsDefault	checkCompat
noChecking	noCompatChecking

Table 46 *Computing default value for CompatChecking [Neu98]*

10.2.3.4.3 Format Features

The default format for a certain matrix shape is determined based on Density and OptFlag. We already explained the relationship between shape, format and OptFlag in Section 10.2.3

OptFlag	Density	RectFormat
*	dense	array
*	sparse	COO

Table 47 *Computing default value for RectFormat [Neu98]*

OptFlag	Density	LowerTriangFormat or UpperTriangFormat
speed	dense	array
speed	sparse	DIA
space	dense	vector
space	sparse	SKY

Table 48 *Computing default value for LowerTriangFormat and UpperTriangFormat [Neu98]*

OptFlag	Density	SymmFormat
speed	dense	array
space	dense	vector
*	sparse	SKY

Table 49 *Computing default value for SymmFormat [Neu98]*

OptFlag	Density	BandDiagFormat
*	dense	array
*	sparse	DIA

Table 50 *Computing default value for BandDiagFormat [Neu98]*

OptFlag	Density	LowerBandTriangFormat or UpperBandTriangFormat
*	dense	vector
speed	sparse	DIA
space	sparse	SKY

Table 51 *Computing default value for LowerBandTriangFormat and UpperBandTriang [Neu98]*

10.2.3.5 Flat Configuration Description

A concrete matrix type is completely described by specifying the values of the DSL parameters for which two or more alternative values are possible. We refer to the set of these DSL parameters as the *"flat" configuration description* (or simply *"flat" configuration*) or the *type record* of a matrix. The flat configuration for the matrix component is the following set of DSL parameters:

ElementType
Shape
Format

ArrOrder
DictFormat
Density
Malloc
MallocErrChecking
OptFlag
ErrFlag
BoundsChecking
CompatChecking
IndexType
Rows
Cols
Order
Diags
ScalarValue
Ratio
Growing
RowsNumber
ColsNumber
OrderNumber
DiagsNumber
Size
HashWidth
ScalarValueNumber

Please note that we combined all format parameters into Format since only one of them is used at a time. Furthermore, not all parameters are relevant in all cases. For example, if Format is vector, DictFormat is irrelevant.

The flat configuration is computed by analyzing the given matrix configuration expression and assigning defaults to the unspecified features.

10.2.4 Matrix ICCL

We construct a concrete matrix type by composing a number of *matrix implementation components*. The matrix package contains a set of parameterized implementation components, some of which are alternative and some optional. The parameterized components can be configured in a number of different ways. The exact specification of valid configurations is given by a grammar. This grammar specifies the *Matrix Implementations Components Configuration Language*, i.e. Matrix ICCL. In our case, the ICCL is specified by a GenVoca-style grammar, which we will show in Section 10.2.4.8. But before that, we first give an overview of the available matrix implementation components.

The matrix implementation components are organized into a number of generic layers (see Figure 167; we discussed this kind of diagrams in Section 6.4.2.2). The layers provide a high-level overview of the relationships between the components and how they are used to construct concrete matrix types.

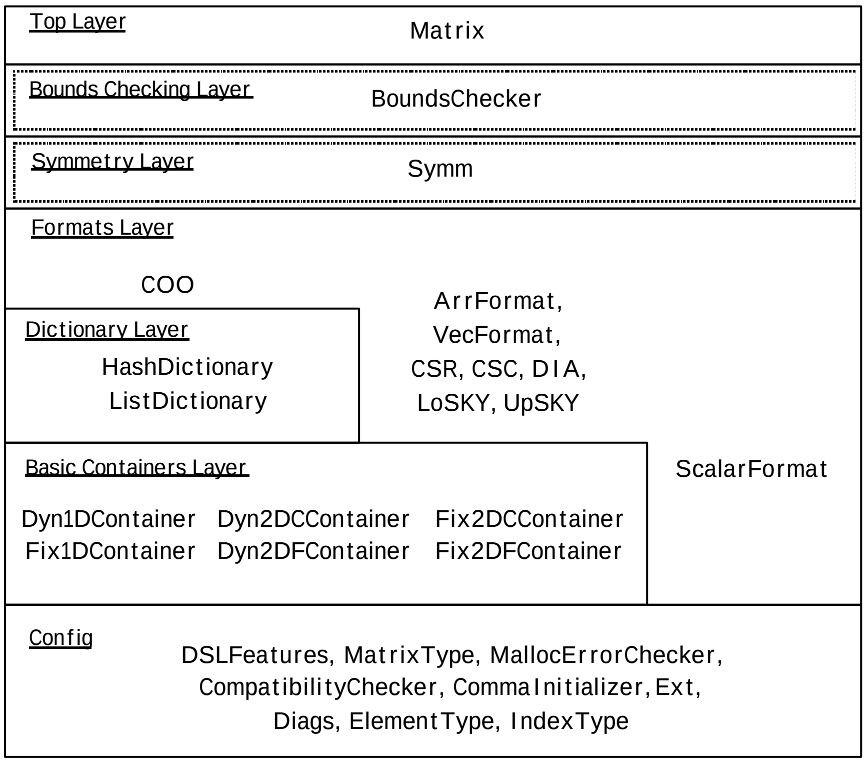

Top Layer
Matrix
Bounds Checking Layer
BoundsChecker
Symmetry Layer
Symm
Formats Layer
COO
ArrFormat, VecFormat, CSR, CSC, DIA, LoSKY, UpSKY
Dictionary Layer
HashDictionary
ListDictionary
Basic Containers Layer
Dyn1DContainer Dyn2DCContainer Fix2DCContainer
Fix1DContainer Dyn2DFContainer Fix2DFContainer
ScalarFormat
Config
DSLFeatures, MatrixType, MallocErrorChecker, CompatibilityChecker, CommaInitializer, Ext, Diags, ElementType, IndexType

Figure 167 *Layered matrix implementation component*

We have the following layers (the names of the layers are underlined in Figure 167):

- *Top layer*: The top layer consists of only one component, which is Matrix. Matrix represents the outer wrapper of every matrix configuration. Its purpose is to express their commonality as matrices. This is useful for writing generic operations on matrices.

- *Bounds checking layer*: This layer contains the optional wrapper BoundsChecker, which provides bounds checking functionality.

- *Symmetry layer*: This layer contains the optional wrapper Symm, which implements symmetry. Symm turns a lower triangular format into a symmetric format.

- *Formats layer*: Formats layer provides components implementing various dense and sparse storage formats.

- *Dictionary layer*: Dictionaries are required by the COO format.

- *Basic containers layer*: Any format that physically stores matrix elements uses basic containers.

The box at the bottom of Figure 167 is the *configuration repository*, which we also refer to as Config. Any component from any layer can reach down to the repository and get the information it needs (we already explained this idea in Sections 6.4.2.4 and 8.7). In particular, the repository is used to store some global components, the type of the matrix being constructed, some horizontal parameters (see Section 6.4.2.2), all the abstract matrix features which were computed from the matrix configuration description (i.e. the flat configuration, which is contained in DSLFeatures) and the description itself.

Each of the matrix components is parameterized. The basic containers are parameterized with Config. The dictionaries are parameterized with the basic containers. The formats are parameterized either with Config, or with the basic containers, or with the dictionaries. More precisely, ScalarFormat is parameterized with Config, whereas COO is parameterized with the dictionaries and the remaining components in this layer are parameterized with the basic containers. Symm is parameterized with the formats. Since the symmetry layer has a dashed inner box, Symm is an optional component. Thus, BoundsChecker can be parameterized with Symm or with the formats directly. Similarly, Matrix can be parameterized with BoundsChecker, or with Symm, or with the formats.

The names appearing in the Config box are variables rather than components. They are assigned concrete components when a configuration is built, so that other components can access them as needed.

An example of a concrete matrix configuration is shown in Figure 168. As we will see later, some formats and the dictionaries use more than one different basic containers at once. Thus, in general, a matrix configuration can look like a tree rather than a stack of components.

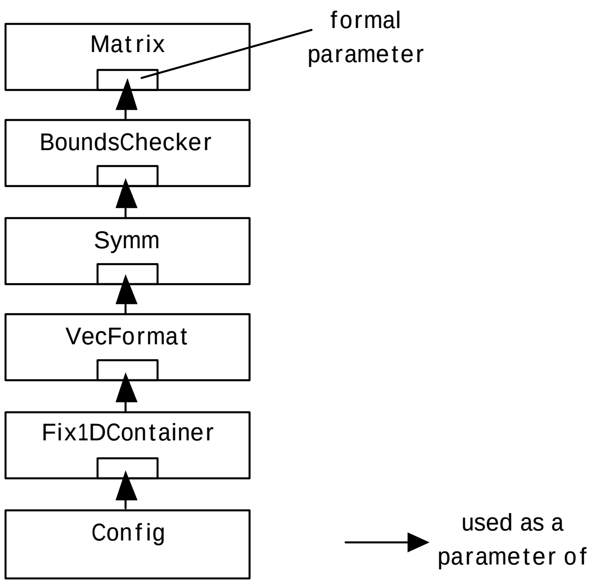


Figure 168 *Example of a matrix configuration*

Now, let us take a closer look at each group of matrix implementation components starting with the bottom layer.

10.2.4.1 Basic Containers

Basic containers are the basic components for storing objects. They are used by the matrix format components to store matrix elements and, in some cases, also element indices and pointers to containers. They are also used to implement dictionaries.

We have six basic containers (see Table 52). They can all be thought of as one or two-dimensional arrays. The containers whose names start with “Dyn” allocate their memory dynamically and the ones starting with “Fix” use static memory allocation (i.e. the size of the allocated memory is specified at compile time). The next two characters in the name of a container indicate whether the container is one or two dimensional. Finally, the two-dimensional containers store their elements either row-wise (this is indicated with an extra C, which stands for C-like storage) or column-wise (which is indicated by an extra F, which stands for Fortran-like storage).

Basic Container	Memory Allocation	Number of Dimensions	Storage Format
Dyn1Dcontainer	dynamic	1	---
Fix1Dcontainer	static	1	---
Dyn2DCContainer	dynamic	2	row-wise
Dyn2DFContainer	dynamic	2	column-wise
Fix2DCContainer	static	2	row-wise
Fix2DFContainer	static	2	column-wise

Table 52 *Basic containers (adapted from [Neu98])*

The basic containers have the following parameters:

Dyn1DContainer[ElementType, Ratio, Growing, Config]
Fix1DContainer[ElementType, Size, Config]
Dyn2DCContainer[Config]
Dyn2DFContainer[Config]
Fix2DCContainer[Size, Config]
Fix2DFContainer[Size, Config]

ElementType is the type of the elements stored in the container. Only the one-dimensional containers have ElementType as their explicit parameter since they are sometimes used more than once in a configuration, e.g. to store matrix elements and matrix element indices. The two-dimensional containers, on the other hand, are only used to store matrix elements. Thus, they can get their element type from Config (which is the element type of the whole matrix). We already explained Size, Ratio, Growing in Table 29. All containers get their index types from Config.

In general, we do not really have to specify ElementType as an explicit parameter of the one-dimensional containers. This is so since the component which gets a container as its parameter can internally request the vector storage type from the container and specify the element type at this time. For example, we could pass just one container to the CSR-format component, and the component would internally request two different types from this container: one vector of index type and one vector of element type. We illustrated this idea with some C++ code in Figure 169.

The solution in Figure 169 has the advantage that we do not have to specify ElementType as an explicit parameter of Dyn1Dcontainer. This leads to a much simpler description of the possible component combinations: We would be able to replace the three productions VerticalContainer, IndexVec, and ElemVec later in Figure 173 with just one production:

```
Vec:            Dyn1DContainer[Ratio, Growing, Config] | Fix1DContainer[Size, Config]
```

Unfortunately, we were not able to use this variant since VC++5.0 reported an internal error when compiling this valid C++ code.

```
//int_number and Configuration are not shown here
...

template<class Size, class Config>
class Dyn1DContainer
{
  public:
    //export Config
    typedef Config Config;

    //components using me can request a Vector as many times as they want;
    //they can specify a different ElementType each time
    template<class ElementType>
    class Vector
    {
      ...
    };
};

template<class Container1D>
class CSR
{
  public:
    //retrieve matrix index type and matrix element type from Config
    typedef Container1D::Config Config;
    typedef Config::IndexType IndexType;
    typedef Config::ElementType ElementType;

    //the CSR component requests the index vector and the element vector
    Container1D::Vector< IndexType> indexVec;
    Container1D::Vector< ElementType> elementVec;
    ...
};

main()
{
  CSR<Dyn1DContainer<int_number<long, 100>, Configuration> > myFormat;
}
```

Figure 169 *C++ Code demonstrating how CSR could request two vectors with different element types from a 1D container (adapted from [Neu98])*

10.2.4.2 Dictionaries

Dictionaries are used by the coordinate matrix format, i.e. COO. Their elements are addressed by a pair of integral numbers. Many variant implementations of a dictionary are possible. However, we consider here only two variants:

```
HashDictionary[VerticalContainer, HorizontalContainer, HashFunction]
ListDictionary[IndexVector, ElementVector]
```

HashDictionary uses a hashing function to index VerticalContainer, which contains references to element buckets (see Figure 170). We substitute one of the one-dimensional basic containers with a constant size for VerticalContainer. The buckets themselves are instantiated from HorizontalContainer. HorizontalContainer is also a dictionary and we use ListDictionary here. HashFunction is a component providing the hash function.

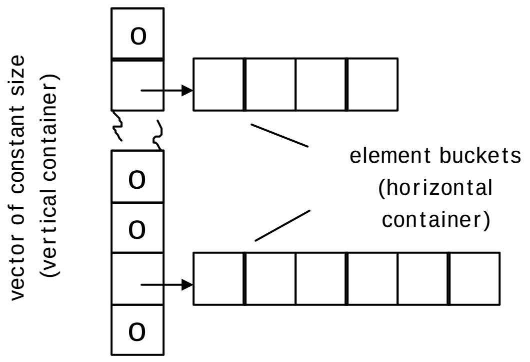


Figure 170 *Structure of a hash dictionary (from [Neu98])*

ListDictionary is a dictionary implementation storing element indices in two instances of IndexVector and elements in ElementVector. Both vector parameters are substituted by one of the one-dimensional basic containers.

10.2.4.3 Formats

We have nine matrix format components. They are summarized in Table 53.

Format Component	Purpose
ArrFormat	ArrFormat stores elements in a two-dimensional basic container. If the indices of a matrix element are i and j, the same indices are used to store the element in the container. ArrFormat is used to store dense matrices of different band shapes (see Figure 141). It optionally checks to make sure that nonzero elements are assigned to locations within the specified band only.
VecFormat	VecFormat stores elements in a one-dimensional basic container (thus, it uses a formula to convert matrix element indices into the container element indices). The elements are stored diagonal-wise. It is used to store dense matrices of different band shapes. Only the nonzero diagonals are stored. It optionally checks to make sure that nonzero elements are assigned to locations within the specified band only.
ScalarFormat	ScalarFormat is used to represent scalar matrices (including zero and identity matrix). The scalar value can be specified statically or dynamically. This format does not require any storage container.
CSR	CSR implements the compressed sparse row format. CSR is used for general rectangular sparse matrices.
CSC	CSC implements the compressed sparse column format. CSC is used for general rectangular sparse matrices.
COO	COO implements the coordinate format. COO is used for general sparse rectangular matrices.
DIA	DIA implements the sparse diagonal format. DIA is used for sparse band matrices
LoSKY	LoSKY implements the lower skyline format. LoSKY is used for sparse lower triangular matrices and sparse lower band triangular matrices.
UpSKY	UpSKY implements the upper skyline format. UpSKY is used for sparse upper triangular matrices and sparse upper band triangular matrices.

Table 53 *Format components (see Section 10.1.2.2.1.3.6.2 for the explanation of the sparse formats)*

The parameters of the format components are as follows:

ArrFormat[Ext, Diags, Array]
VecFormat[Ext, Diags, ElemVec]
ScalarFormat[Ext, ScalarValue, Config]
CSR[Ext, IndexVec, ElemVec]
CSC[Ext, IndexVec, ElemVec]

COO[Ext, Dict]
DIA[Ext, Diags, Array]
LoSKY[Ext, Diags, IndexVec, ElemVec]
UpSKY[Ext, Diags, IndexVec, ElemVec]

Array is any of the two-dimensional basic containers. ElemVec and IndexVec are any of the one-dimensional basic containers. Ext and Diags need a bit more of explanation.

10.2.4.3.1 Extent and Diagonal Range of a Matrix

The *extent* of a matrix is determined by the number of rows and the number of columns. The *diagonal range* specifies the valid diagonal index range for band matrices. The index of a diagonal is explained in Figure 171. For example, the main diagonal has the index 0. The range -1...1 means that only the diagonals -1, 0, and 1 can contain nonzero elements.

$$\begin{pmatrix} 0 & 1 & & & n-1 \\ -1 & 0 & 1 & \iddots & \\ & -1 & 0 & 1 & \\ & & -1 & 0 & 1 \\ & & & -1 & 0 \\ & \iddots & & & -1 \\ & & & & \\ 1-m & & & & \end{pmatrix}$$

Figure 171 *Numbering of the diagonals of a m-by-n matrix (adapted from [Neu98])*

Table 54 summarizes the extent and diagonal range of some more common matrix shapes.

Shape	Extent	Diagonal range
rectangular	arbitrary m and n	-m...n
diagonal	m = n	0...0
lower triangular	m = n	-m...0
upper triangular	m = n	0...n
symmetric	m = n	-m...n (or –m...0)[150]
band diagonal	m = n	-⌊d/2⌋...⌊d/2⌋, where ⌊ ⌋ denote rounding down
lower band triangular	m = n	1-d...0
upper band triangular	m = n	0...d-1

Table 54 *Extent and diagonal range of some more common matrix shapes (from [Neu98]). d is the number of diagonals.*

Ext

The parameter Ext is used to specify the extent of a matrix, i.e. the number of rows and the number of columns and whether they are determined statically or dynamically. We have a number of small helper components that can be used in place of Ext to specify the extent of rectangular and square matrices. They are described in Figure 172.

Extent Component	Extent	Number of rows / columns
DynExt	rectangular	both dynamic
StatExt	rectangular	both static
DynRowsStatCols	rectangular	row number dynamic, column number static
StatRowsDynCols	rectangular	row number static, column number dynamic
DynSquare	square	both dynamic
StatSquare	square	both static

Figure 172 *Helper components for specifying the extent of a matrix (adapted from [Neu98])*

The extent components have the following parameters:

```
DynExt[IndexType]
StatExt[Rows, Cols]
StatRowsDynCols[Rows]
DynRowsStatCols[Cols]
DynSquare[IndexType]
StatSquare[Rows]
```

Rows specifies the number of rows and Cols the number of columns. IndexType specifies the index type. Rows and Cols can be specified as follows:

```
int_number[Type, value]
```

Diags

The parameter Diags specifies the diagonal range of a matrix format. The helper components which can be used in place of Diags are described in Table 55.

Diags Component	Purpose
DynLo	DynLo specifies lower band triangular shape with dynamic bandwidth. The diagonal range is 1-d...0, where d is specified dynamically.
DynUp	DynUp specifies upper band triangular shape with dynamic bandwidth. The diagonal range is 0...d-1, where d is specified dynamically.
DynBand	DynBand specifies band diagonal shape with dynamic bandwidth. The diagonal range is $-\lfloor d/2 \rfloor ... \lfloor d/2 \rfloor$, where d is specified dynamically.
TriangLo	TriangLo specifies lower triangular shape.
TriangUp	TriangUp specifies upper triangular shape.
Rect	Rect specifies rectangular shape.
StatDiags[FirstDiag, LastDiag]	StatDiags allows to statically specify the band range. The diagonal range is FirstDiag...LastDiag.
StatBand[BandWidth]	StatBand specifies band diagonal shape with static bandwidth. The diagonal range is $-\lfloor BandWidth/2 \rfloor ... \lfloor BandWidth/2 \rfloor$.

Table 55 *Helper components for specifying components the diagonal range of a matrix*

The Diags components have the following parameters:

```
DynLo[IndexType]
DynUp[IndexType]
DynBand[IndexType]
TriangLo[IndexType]
TriangUp[IndexType]
Rect[IndexType]
StatDiags[FirstDiag, LastDiag]
```

10.2.4.4 Symmetry Wrapper

As described in Table 20, the elements of a symmetric matrix can be stored in a lower (or upper) triangular matrix. Thus, we do not need a symmetric format storage component. Instead, we implement the symmetry property using a wrapper which we can put on some other format component. The wrapper maps any element access to it onto a corresponding element access to the lower (or upper) half of the wrapped format component. The wrapper takes a format component as its parameter:

```
Symm[Format]
```

The symmetry wrapper can be used with any of the available format components. However, only the lower part of the component is actually used and thus only some combinations are relevant. For example, we would use a lower triangular format to store a symmetric matrix and lower band triangular format to store a symmetric band diagonal matrix.[151] Putting the symmetry wrapper on a diagonal shape is not particularly useful. Please remember that the shape of a format component depends on the component, the extent, and the diagonal range.

10.2.4.5 Error Checking Components

Currently, we provide components for bounds checking (see Section 10.1.2.2.1.4.1), compatibility checking (see Section 10.1.2.2.1.4.2), and memory allocation error checking (the latter are used by the dynamic basic containers).

Bounds checking is best implemented in the form of a wrapper since it involves precondition checking and thus we only need to wrap the element access methods of the underlying format component. Thus, BoundsChecker takes format as its only parameter:

```
BoundsChecker[FormatOrSymm]
```

The compatibility checking and memory allocation error checking components, on the other hand, are called from within methods of other components, i.e. they involve intracondition checking. Thus, we do not implement them as wrappers. They are implemented as stand alone components and we provide them to the configuration through the MallocErrorChecker and CompatibilityChecker variables of the configuration repository (i.e. Config). Any component that needs them can retrieve them from the repository.

The compatibility checking and memory allocation error checking components are described in Table 56 and Table 57. They all take Config as their parameter.

CompatibilityChecker Component	**Purpose**
CompatChecker	CompatChecker is used to check the compatibility of matrices for addition, subtraction, and multiplication. It is called from the code implementing the matrix operations.
EmptyCompatChecker	EmptyCompatChecker implements its error checking methods as empty methods (please note that inlining will eliminate any overhead). If used, no compatibility checking is done.

Table 56 *Compatibility checking components*

MallocErrorChecker Component	**Purpose**
MallocErrChecker	MallocErrChecker is used to check for memory allocation errors in the dynamic basic containers.
EmptyMallocErrChecker	EmptyMallocErrChecker implements its error checking methods as empty methods. If used, no memory allocation error checking is done.

Table 57 *Memory allocation error checking component*

10.2.4.6 Top-Level Wrapper

Each matrix configuration is wrapped into Matrix. This wrapper is used to express the type commonality of all matrix configurations. This is particularly useful for generic operations so that they can check whether the type they operate on is a matrix type or not.

10.2.4.7 Comma Initializers

Comma initializers implement matrix initialization using comma separated lists of numbers (as implemented in the Blitz++ library [Vel97]). These components are specific to the C++ implementation.

Using a comma initializer, we can initialize a dense matrix by listing its elements (the matrix already knows its extent), e.g.:

```
matrix= 3, 0, 0, 8, 7,
        0, 2, 1, 2, 4,
        6, 0, 2, 4, 5;
```

For sparse matrices, the initialization format is different. We specify the value of an element followed by its indices, e.g.:

```
sparseMatrix= 3, 0, 0,
              1, 1, 2,
              2, 1, 1,
              6, 2, 0,
              2, 2, 2;
```

In general, a dense m-by-n matrix is initialized as follows [Neu98]:

$$A= \begin{array}{llll} a_{11}, & a_{12}, & \cdots & a_{1n}, \\ a_{21}, & \ddots & & a_{2n}, \\ \vdots & & \ddots & \vdots \\ a_{m1}, & \ldots & \ldots & a_{mn}; \end{array}$$

and a sparse m-by-n matrix:

$$A= \begin{array}{lll} a_{i_1 j_1}, & i_1, & j_1, \\ a_{i_2 j_2}, & i_2, & j_2, \\ \vdots & & \\ a_{i_k j_k}, & i_k, & j_k; \end{array}$$

The available comma initialization components are summarized in Table 58.

CommaInitializer Component	Purpose
DenseCCommaInitializer	DenseCCommaInitializer is used for initializing dense matrices with row-wise storage.
DenseFCommaInitializer	DenseFCommaInitializer is used for initializing dense matrices with column-wise storage.
SparseCommaInitializer	SparseCommaInitializer is used for initializing sparse matrices.

Table 58 *Comma initialization components*

The comma initializer currently used by a configuration is published in the configuration repository in the CommaInitializer component variable.

10.2.4.8 Matrix ICCL Grammar

The complete ICCL grammar is specified in the GenVoca-like notation in Figure 173.

MatrixType:	Matrix[OptBoundsCheckedMatrix]
OptBoundsCheckedMatrix:	OptSymmetricMatrix | BoundsChecker[OptSymmetricMatrix]
OptSymmetricMatrix:	Format | Symm[Format]
Format:	ArrFormat[Ext, Diags, Array] | VecFormat[Ext, Diags, ElemVec] | ScalarFormat[Ext, ScalarValue, Config] | CSR[Ext, IndexVec, ElemVec] | CSC[Ext, IndexVec, ElemVec] | COO[Ext, Dict] | DIA[Ext, Diags, Array] | LoSKY[Ext, Diags, IndexVec, ElemVec] | UpSKY[Ext, Diags, IndexVec, ElemVec]
Dict:	HashDictionary[VerticalContainer, HorizontalContainer, HashFunction] | ListDictionary[IndexVec, ElemVec]
Array:	Dyn2DCContainer[Config] | Fix2DCContainer[Size, Config] | Dyn2DFContainer[Config] | Fix2DFContainer[Size, Config]
HorizontalContainer:	ListDictionary[IndexVector, ElementVector]
VerticalContainer:	Dyn1DContainer[HorizPointer, Ratio, Growing, Config] | Fix1DContainer[HorizPointer, Size, Config]
IndexVec:	Dyn1DContainer[IndexType, Ratio, Growing, Config] | Fix1DContainer[IndexType, Size, Config]
ElemVec:	Dyn1DContainer[ElementType, Ratio, Growing, Config] | Fix1DContainer[ElementType, Size, Config]
HashFunction:	SimpleHashFunction[HashWidth]
Ext:	DynExt[IndexType] | StatExt[Rows, Cols] | DynSquare[IndexType] | StatSquare[Rows] | StatRowsDynCols[Rows] | DynRowsStatCols[Cols]
Diags:	DynLo[IndexType] | DynUp[IndexType] | DynBand[IndexType] | TriangLo[IndexType] | TriangUp[IndexType] | Rect[IndexType] | StatDiags[FirstDiag, LastDiag] | StatBand[BandWidth]
ScalarValue:	DynVal[ElementType] | StatVal[Val]
MallocErrorChecker:	EmptyMallocErrChecker[Config] | MallocErrChecker[Config]
CompatibilityChecker:	EmptyCompatChecker[Config] | CompatChecker[Config]
CommaInitializer:	DenseCCommaInitializer[Config] | DenseFCommaInitializer[Config] | SparseCommaInitializer[Config]
Ratio, Growing:	float_number[Type, value]
Rows, Cols, FirstDiag, LastDiag, BandWidth, Size, HashWidth:	int_number[Type, value]
Val:	int_number[Type, value] | float_number[Type, value]
ElementType:	float | double | long double | short | int | long | unsigned short | unsigned int | unsigned long
IndexType:	char | short | int | long | unsigned char | unsigned short | unsigned int | unsigned long | signed char
SignedIndexType:	char | short | int | long
HorizPointer:	HorizontalContainer*
Config:	Configuration (contains the component variables: DSLFeatures, MatrixType, MallocErrorChecker, CompatibilityChecker, CommaInitializer, ElementType, IndexType, Ext, Diags)
DSLFeatures:	this is the “flat” configuration

Figure 173 *Matrix ICCL [Neu98]*

Here is an example of a valid ICCL expression:

```
Matrix[
  BoundsChecker[
    ArrFormat[
      DynExt[unsigned int],
      Rect[unsigned int],
      Dyn2DCContainer[Config]
    ]
  ]
]
```

10.2.5 Mapping from Matrix Configuration DSL to Matrix ICCL

Now that we completely specified the Matrix Configuration DSL and the Matrix ICCL, we need to specify how to translate a Matrix Configuration DSL expression into a Matrix ICCL expression. We will use the dependency tables introduced in Section 10.2.3 for this purpose. Each table will specify how to compute an ICCL parameter based on some DSL parameters from the flat configuration description (Section 10.2.3.5).

10.2.5.1 Basic Containers

We start with the basic containers. The mapping for Array is shown in Table 59. The value of Array depends on the DSL features Malloc and ArrOrder.

Malloc	ArrOrder	Array
fix	cLike	Fix2DCContainer
	fortranLike	Fix2DFContainer
dyn	cLike	Dyn2DCContainer
	fortranLike	Dyn2DFContainer

Table 59 *Table for computing the ICCL parameter Array*

Table 60 specifies how to compute IndexVec, ElemVec, and VerticalContainer.

Malloc	IndexVec, ElemVec, VerticalContainer
fix	Fix1Dcontainer
dyn	Dyn1Dcontainer

Table 60 *Table for computing the ICCL parameters IndexVec, ElemVec, and VerticalContainer [Neu98]*

The following parameters of the basic containers are determined directly from the DSL parameters:

Size (ICCL)	= Size (DSL)
Ratio (ICCL)	= Ratio (DSL)
Growing (ICCL)	= Growing (DSL)
ElementType (ICCL)	= ElementType (DSL)
IndexType (ICCL)	= IndexType (DSL)

Since some DSL parameters have the same name as the corresponding ICCL parameters, we indicate whether we refer to a DSL parameter or to a ICCL parameter by an extra annotation.

The signed index type (SignedIndexType) is used for the diagonal indices in the DIA format component. SignedIndexType is determined based on IndexType. This is specified in Table 61. Please note that this mapping may be problematic for very large matrices with unsigned index type and using DIA since only the have of the row (or column) index range is available for indexing the diagonals.

IndexType	SignedIndexType
unsigned char	signed char
unsigned short	short
unsigned int	int
unsigned long	long
*	=IndexType

Table 61 *Table for computing the ICCL parameter SignedIndexType [Neu98]*

10.2.5.2 Dictionaries

A dictionary component is selected based on the DSL feature DictFormat (see Table 62).

DictFormat	Dict
hashDictionary	HashDictionary
listDictionary	ListDictionary

Table 62 *Table for computing the ICCL parameter Dict [Neu98]*

Currently, two of the parameters of HashDictionary use direct defaults:

HorizontalContainer	= ListDictionary

```
HashFunction     = SimpleHashFunction[HashWidth]
```

Additionally, we have the following dependencies:

```
HashWidth (ICCL)    = HashWidth (DSL)
HorizPointer     = HorizontalContainer*    (C++ notation for pointers)
```

10.2.5.3 Formats

The ICCL parameter Format is determined based on DSL features Shape and Format. The mapping is given in Table 63. Please note that the DSL feature Density was already used for computing the flat configuration DSL parameter Format (see Section 10.2.3.4.3).

Shape	Format (DSL)	Format (ICCL)
diag	---	VecFormat
scalar	---	ScalarFormat
ident	---	ScalarFormat
zero	---	ScalarFormat
*	array	ArrFormat
	vector	VecFormat
	CSR	CSR
	CSC	CSC
	COO	COO
	DIA	DIA
lowerTriang symm lowerBandTriang	SKY	LoSKY
upperTriang upperBandTriang	SKY	UpSKY

Table 63 *Table for computing the ICCL parameter Format [Neu98]*

The remaining tables specify the mapping for the various parameters of the format components.

Shape			Ext
	Rows (DSL)	**Cols (DSL)**	
rect	dynVal	dynVal	DynExt
	dynVal	statVal	StatRowsDynCols
	statVal	dynVal	DynRowsStatCols
	statVal	statVal	StatExt
	Order		
*	dynVal		DynSquare
	statVal		StatSquare

Table 64 *Table for computing the ICCL parameter Ext [Neu98]*

Shape	Diags (DSL)	Diags (ICCL)
rect	---	Rect
diag	---	StatBand
scalar	---	StatBand
ident	---	StatBand
zero	---	StatBand
lowerTriang	---	TriangLo
upperTriang	---	TriangUp
symm	---	TriangLo
bandDiag	dynVal	DynBand
	statVal	StatBand
lowerBandTriang	dynVal	DynLo
	statVal	StatDiags
upperBandTriang	dynVal	DynUp
	statVal	StatDiags

Table 65 *Table for computing the ICCL parameter* Diags *[Neu98]*

Shape	Diags (DSL)	FirstDiag	LastDiag
lowerBandTriang	statVal	=1-DiagsNumber	number[IndexType,0]
upperBandTriang	statVal	number[IndexType,0]	=DiagsNumber-1

Table 66 *Table for computing the ICCL parameters* FirstDiag *and* LastDiag *[Neu98]*

Shape	Diags (DSL)	BandWidth
diag scalar ident zero	---	number[IndexType,1]
bandDiag	statVal	DiagsNumber

Table 67 *Table for computing the ICCL parameter* BandWidth *[Neu98]*

ScalarValue (DSL)	ScalarValue (ICCL)
dynVal	DynVal
statVal	StatVal

Table 68 *Table for computing the ICCL parameter* ScalarValue *[Neu98]*

The ICCL parameter Val is computed as follows:

Val =ScalarValueNumber

Shape	Rows (ICCL)	Cols (ICCL)
rect	= RowsNumber	= ColsNumber
*	= OrderNumber	= OrderNumber

Table 69 *Table for computing the ICCL parameters Rows and Cols [Neu98]*

Finally, Shape determines whether we use the Symm wrapper or not.

Shape	OptSymmetricMatrix
symm	Symm
*	= Format (ICCL)

Table 70 *Table for computing the ICCL parameter OptSymmetricMatrix [Neu98]*

10.2.5.4 Error Checking Components

BoundsChecking	OptBoundsCheckedMatrix
checkBounds	BoundsChecker
noBoundsChecking	= OptSymmetricMatrix

Table 71 *Table for computing the ICCL parameter OptBoundsCheckedMatrix [Neu98]*

MallocErrChecking	MallocErrorChecker
checkMallocErr	MallocErrChecker
noMallocErrChecking	EmptyMallocErrChecker

Table 72 *Table for computing the ICCL parameter MallocErrorChecker [Neu98]*

Shape	CompatibilityChecker
checkCompat	CompatChecker
noCompatChecking	EmptyCompatChecker

Table 73 *Table for computing the ICCL parameter CompatibilityChecker [Neu98]*

10.2.5.5 Comma Initializer

Density	ArrOrder	CommaInitializer
dense	cLike	DenseCCommaInitializer
	fortranLike	DenseFCommaInitializer
sparse	---	SparseCommaInitializer

Table 74 *Table for computing the ICCL parameter CommaInitializer [Neu98]*

10.2.6 Matrix Expression DSL

We will only consider matrix expressions containing matrix-matrix addition, subtraction, and multiplication, e.g. A+B+C-D*E or (A-B)*(C+D+E*F). Of course, the operand matrices have to have compatible dimensions (see Section 10.1.2.2.1.4.2).

We can assign a matrix expression to another matrix, e.g.:

R= A+B+C-D*E.

In the following sections, we discuss the question of what code should be generated for matrix expressions and matrix assignment.

10.2.6.1 Evaluating Matrix Expressions

Let us assume that A, B, C, D, and E are matrices and that they are compatible to be used in the following assignment statement:

E= (A + B) * (C + D)

There are two principal ways to compute this assignment [Neu98]:

1. with intermediate results (i.e. with temporaries)
   1.1. temp1= A + B
   1.2. temp2= C + D
   1.3. temp3= temp1 * temp2
   1.4. E= temp3
2. without temporaries (i.e. lazy)

   All elements of E are computed one after another from the corresponding elements of A, B, C, and D, i.e.

   $e_{11}= \quad (a_{11} + b_{11}) * (c_{11} + d_{11}) +$

   $\qquad (a_{12} + b_{12}) * (c_{21} + d_{21}) +$

   $\qquad \vdots$

   $\qquad \vdots$

   $e_{21}= \ldots$

   $\vdots$

   $\vdots$

   $e_{mn}= \ldots$

Each of these two approaches has its advantages and disadvantages. The first approach is simple to implement using overloaded binary operators. Unfortunately, the initialization of the temporaries, the separate loops for each binary operation, and the final assignment incur a significant overhead.

The second (i.e. lazy) approach is particularly useful if we want to compute only some of the elements of a matrix expression (remember that the first approach computes all elements of each subexpression). Furthermore, it is also superior if the elements of the argument matrices (or subexpressions) are accessed only once during the whole computation. This is the case for expressions (or subexpressions) consisting of matrix additions only. In this case, approach two allows us to evaluate such expression very efficiently: we use two nested loops to iterate over the nonzero region of the resulting matrix and in each iteration we assign to the current element of the resulting matrix the sum of the corresponding elements from all the argument matrices. Thus, we do not need any temporaries and extra loops as in the first approach.[152]

Unfortunately, the second approach is inefficient for matrix multiplication since matrix multiplication accesses the elements of the argument matrices more than once. This is illustrated in Figure 174.

$$\begin{pmatrix} a_{11} & a_{21} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \bullet \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} & a_{21}b_{13} + a_{22}b_{23} \\ a_{31}b_{11} + a_{32}b_{21} & a_{31}b_{12} + a_{32}b_{22} & a_{31}b_{13} + a_{32}b_{23} \end{pmatrix}$$

Figure 174 *Elementwise computation of the multiplication of two sample matrices*

The elementwise computation of the multiplication of the two sample matrices requires each element of the argument matrices to be accessed three times. If the argument matrices are actually matrix expressions, each element of the argument expressions will be computed three times (asumming that both matices do not have nay special shape), which causes an unnecessary overhead.

The negative behavior of the lazy approach in the case of multiplication can be addressed in two ways:

- *Lazy with temporaries*: Whenever one of the arguments to a matrix multiplication is an expression, we create a temporary matrix and assign the expression to it. In effect, we use the lazy approach for matrix addition only.

- *Lazy with cache*: Instead of creating the full temporary for the expression arguments to a matrix multiplication at once (i.e. as above), we can create a cache matrix and compute any element of the argument expression on first access only. In other words, we use the lazy approach for both addition and multiplication and avoid recomputing elements of subexpressions by storing them in a cache.

Both approaches avoid the creation of temporaries for assignment (e.g. temp3 in our original example) and for arguments to matrix addition. However, they also have significant differences. Compared to the first one, the second approach has the overhead of the caching (i.e. store the computed element on first access and do the extra check whether the element has been already computed or not on each access). The effect of this overhead depends on many factors such as element type (i.e. precision, real or complex), number of operation per access, access time for the storage formats used. On the other hand, the lazy-with-cache approach is superior if we want to compute only some but not all of the expression elements. For example, if we are only interested in elements (1,1), (1,2), and (1,3) of the result matrix in Figure 174 and the argument matrices are actually expressions, the lazy-with-cache approach is likely to be faster than the lazy-with-temporaries one.

In the following discussion, we will only consider the lazy-with-cache approach.

10.2.6.2 Implementing the Lazy Approach

An elegant thought model for the lazy approach is to think of the matrix expressions (and also its subexpressions) as objects. If you need to know some element of the expression, you ask the expression for it. An expression object implements only one matrix operation (i.e. addition, subtraction, or multiplication) itself and delegates the rest of the work to its subexpressions.

For example, consider the matrix expression A+B+C. Its object-oriented interpretation is shown in Figure 175. We have one addition expression object pointing to A and to another addition expression objects, which in turn points to B and C. When we ask the top-level addition expression for the element (i,j) by sending it the message getElement(i,j), it executes its getElement() method code shown in the box. This execution involves sending getElement(i,j) to its operands. The same happens for the second addition expression. Finally, each of the argument matrices gets the message getElement(i,j).

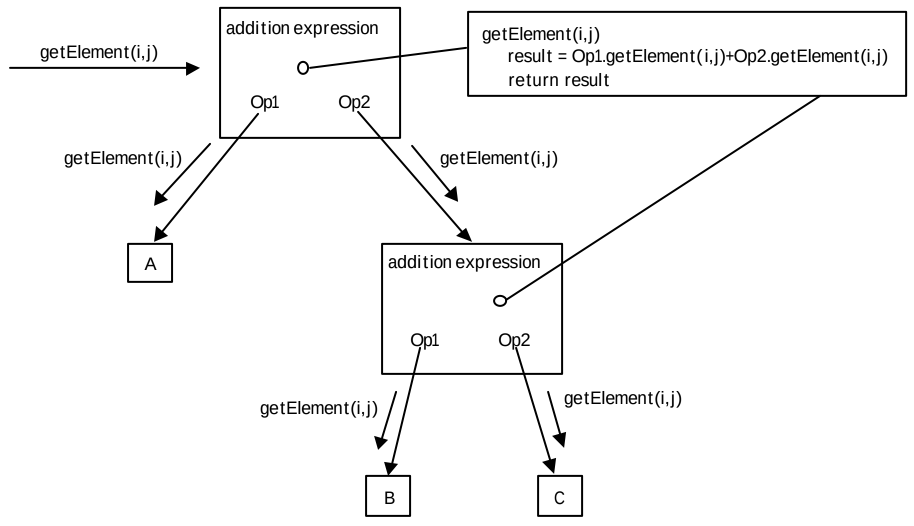


Figure 175 *Computing A+B+C using expression objects*

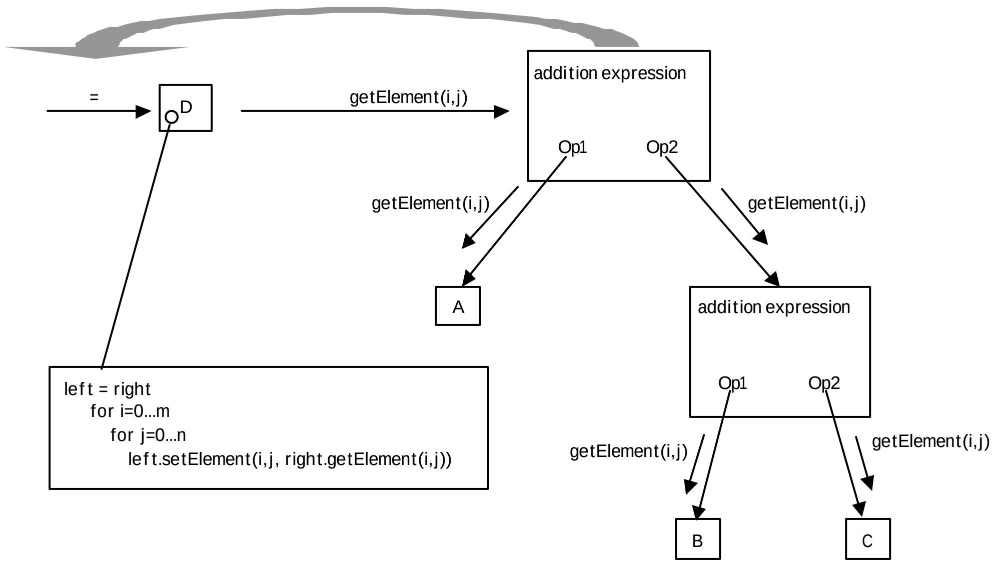
If the calls to getElement() are statically bound and we use inlining, the code generated for the top-level method getElement() looks like this:

```
A.getElement(i,j) + B.getElement(i,j) + C.getElement(i,j)
```

Now, assume that we want to assign our expression A+B+C to D:

```
D = A+B+C
```

This scenario is illustrated in Figure 176. We send the message "=" to D with A+B+C as its parameter. The assignment code (shown in the box) iterates through all elements of D and assigns each of them the value computed by sending getElement() to the expression.



```
left = right
   for i=0...m
      for j=0...n
         left.setElement(i,j, right.getElement(i,j))
```

Figure 176 *Computing D = A+B+C using expression objects*

If we use inlining, the code generated for D = A+B+C will look like this:

```
for i=0...m
   for j=0...n
      D.setElement(A.getElement(i,j) + B.getElement(i,j) + C.getElement(i,j))
```

This is as efficient as we can get for general matrices.

The expression objects for the matrix expression (A+B)*(C+D) are shown in Figure 177. The getElement() method of the multiplication expression accesses Op1 and Op2 through caching wrappers. They make sure that no element of the addition expressions is computed more than once.

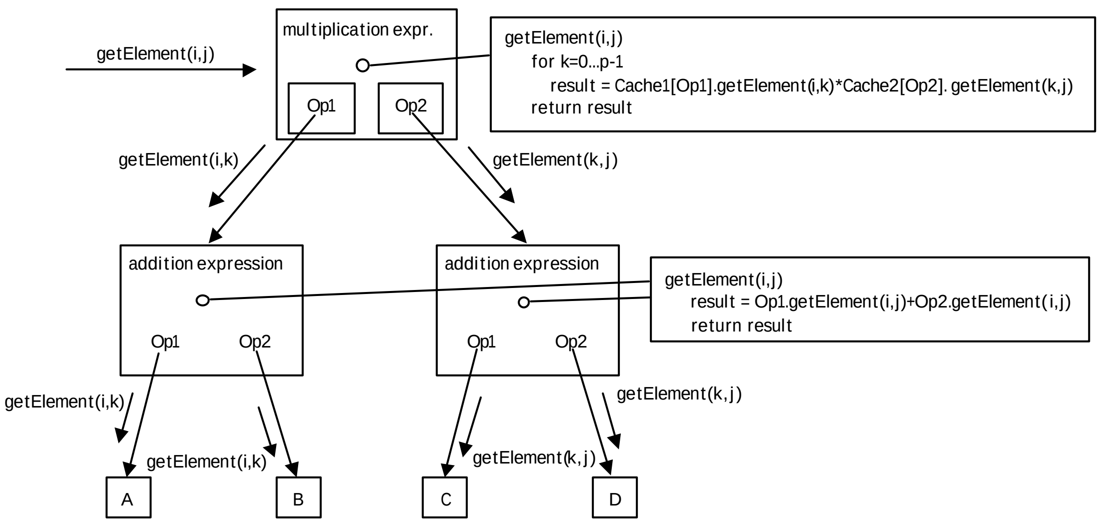


Figure 177 *Computing (A+B)*(C+D) using expression objects*

The sample code for the getElement() method and the assignment operation did not assume any special shape of the argument matrices. However, if the arguments have some special shape, we can usually select a faster implementation for getElement() and the assignment.

For example, if the left operand of a matrix multiplication expression has diagonal shape, we can use the following implementation of getElement():

```
getElement(i,j)
  result = Op1.getElement(i,i) * Op2.getElement(i,j)
  return result
```

Please note that this code is much simpler than the one in Figure 177. First, we do not need a loop. Second, we do not even need the caching wrappers since no element of the operands is accessed more than once.

Similarly, if the resulting shape of an expression is smaller than rectangular (e.g. triangular or diagonal), we only need to iterate over the nonzero region of the expression when assigning it to another matrix. In general, we can implement the assignment by initializing the target matrix and iterating over the nonzero part of the expression and assigning the computed elements to the corresponding elements of the matrix.

Thus, the job of the generator for the Matrix Expression DSL will be to select the appropriate code for the assignments and the calls to getElement() based on the computed type of the argument expressions. The operations themselves will be glued statically by inlining.

Next we will specify the available variant implementation of assignment and getElement() and when to select which. Finally, in Section 10.2.7, we specify how to compute the result type of a matrix expressions.

10.2.6.3 Assignment

The available assignment variants are listed in Table 75. A variant is selected based on the shape and the density of the source matrix in an assignment (see Table 76).

Assignment Component	Explanation
ZeroAssignment	Implements the assignment of a zero matrix by calling initElements on the target matrix.
DiagAssignment	Implements the assignment of a diagonal matrix (see Section 10.2.6.2).
RectAssignment	Implements the assignment of a rectangular matrix (see Section 10.2.6.2).
BandAssignment	Implements the assignment of a band matrix.
SparseAssignment	Implements the assignment of a sparse matrix. The algorithm uses iterators to iterate through the nonzero elements of the sparse source matrix.

Table 75 *Available assignment algorithms*

Shape	Density	Assignment
zero	*	ZeroAssignment
ident scalar diag	*	DiagAssignment
rect	dense	RectAssignment
*	sparse	SparseAssignment
*	*	BandAssignment

Table 76 *Selecting the assignment algorithm*

10.2.6.4 getElement()

The implementation of getElement() depends on the shape of the arguments. The selection table for matrix addition is shown in Table 77 and for matrix multiplication in Table 78.

Shape1	Shape2	GetElement
zero	zero	ZeroGetElement
ident	zero	IdentGetElement
zero	ident	IdentGetElement
zero	*	GetRightElement
*	zero	GetLeftElement
scalar ident	scalar ident	ScalarAddGetElement
rect	rect	RectGetAddElement
*	*	BandAddGetElement

Table 77 *Selecting the implementation of getElement() for matrix addition*

Shape1	Shape2	GetElement
zero	*	ZeroGetElement
*	zero	ZeroGetElement
ident	ident	IdentGetElement
ident	*	GetRightElement
*	ident	GetLeftElement
scalar diag	scalar diag	DiagMultiplyGetElement
scalar diag	*	DiagXMultiplyGetElement
*	scalar diag	XDiagMultiplyGetElement
rect	rect	RectMultiplyGetElement
*	*	BandMultiplyGetElement

Table 78 *Selecting the implementation of* getElement() *for matrix addition*

10.2.7 Computing Result Types of Operations

In order to be able to declare variables for the intermediate results (or for the caches) and also select the optimal implementations of getElement() and assignment during the code generation for matrix expressions, we have to be able to compute the result type of the operations based on the operation argument types. Since a flat configuration represents an abstract and complete description of a matrix type (see Section 10.2.3.5), we do the result type computation at the level of flat configurations. All we have to do is to retrieve the flat configurations from the configuration repositories of the argument matrix types and give them to a metafunction which will compute the resulting flat configuration. Next, the resulting flat configuration can be given to the matrix generator in order to generate the resulting matrix type.

The goal of this section is to specify how to compute the resulting flat configuration from the flat configurations of the arguments for different operations. We only need to specify the mapping for addition and multiplication since addition and subtraction use the almost same mapping.[153]

The computation of the mathematical properties of the resulting matrix can be based on the mathematical theory of matrices. For example, the shape of a matrix resulting from the multiplication or the addition of two lower-triangular matrices is also lower triangular. On the other hand, the result of combining properties such as error checking, optimization flag, etc., is not that obvious. Our current strategy for the latter kind of properties is to return the same property if the argument properties are equal or return unspecified otherwise (see Table 83). If we give a flat configuration with some unspecified features to the matrix generator, it will assume the feature defaults described in Sections 10.2.3.3 and 10.2.3.4. For example, if OptFlag of both argument matrices is speed, OptFlag of the resulting matrix will also be speed. If, on the other hand, one of them is speed and the other one space, then we assume the resulting OptFlag to be unspecified_DSL_feature. According to Table 42, the generator will use space for the resulting OptFlag.

It is worth noting that the result type computation specifies static, domain-specific type inference. This is so since the computation operates on static descriptions of properties of the matrices. Indeed, this inference and the flat configurations define a domain-specific type system for matrices.

Now, we specify the result computation for each of the flat configuration DSL parameters for addition and multiplication. Please note that the following dependency tables give one possible solution and other solutions could be equally good or better. We certainly do not use all possible mathematical dependencies between the features and the choices for the more arbitrary features might be disputable. Nevertheless, these functions appear to be usable and, most importantly, they can be extended and improved any time later.

10.2.7.1 Element Type and Index Type

ElementType and IndexType are numeric types. Thus, the resulting type has to be the larger one. In C++, we achieve this with a type promotion metafunction which returns the type with a larger exponent or, in case the exponents are equal, the one with larger precision. The code is shown in Figure 178. The C++ code uses the standard C++ traits template numeric_limits<>, which describes the properties of built-in types.

```
#include <limits>
using namespace std;

template<class A, class B>
struct PROMOTE_NUMERIC_TYPE
{
   typedef IF<
      numeric_limits<A>::max_exponent10 < numeric_limits<B>::max_exponent10
      ||
      (numeric_limits<A>::max_exponent10==numeric_limits<B>::max_exponent10
         &&
       numeric_limits<A>::digits < numeric_limits<B>::digits),

      B,
      A>::RET RET;
};
```

Figure 178 *C++ meatafunction for promoting nummeric types*

10.2.7.2 Shape

Shape1	Shape2	Shape Result
zero	*	zero
*	zero	zero
ident scalar	(shape2)	=(shape2)
(shape1)	ident scalar	=(shape1)
diag	symm	rect
symm	diag	rect
diag	(shape2)	=(shape2)
(shape1)	diag	=(shape1)
bandDiag	bandDiag	bandDiag
lowerBandTriang	lowerBandTriang	lowerBandTriang
upperBandTriang	upperBandTriang	upperBandTriang
lowerTriang lowerBandTriang	lowerTriang lowerBandTriang	lowerTriang
upperTriang upperBandTriang	upperTriang upperBandTriang	upperTriang
*	*	rect

Table 79 *Resulting Shape for multiplication [Neu98]*

Shape1	Shape2	Shape Result
ident	ident	scalar
zero ident scalar diag	(shape2)	=(shape2)
(shape1)	zero ident scalar diag	=(shape1)
symm	symm	symm
bandDiag	bandDiag	bandDiag
lowerBandTriang	lowerBandTriang	lowerBandTriang
upperBandTriang	upperBandTriang	upperBandTriang
lowerTriang lowerBandTriang	lowerTriang lowerBandTriang	lowerTriang
upperTriang upperBandTriang	upperTriang upperBandTriang	upperTriang
*	*	rect

Table 80 *Resulting* Shape *for addition [Neu98]*

10.2.7.3 Format

Shape Result	Format1	Format2	Format Result
rect	vect	vect	unspecified_DSL_feature
rect	DIA	DIA	unspecified_DSL_feature
rect	SKY	SKY	unspecified_DSL_feature
*	(value)	(value)	=(value)
*	*	*	unspecified_DSL_feature

Table 81 *Resulting* Format *(addition and multiplication)*

10.2.7.4 Density

Density1	Density2	Density Result
sparse	sparse	sparse
*	*	dense

Table 82 *Resulting* Density *(addition and multiplication) [Neu98]*

10.2.7.5 Malloc, OptFlag, BoundsChecking, CompatChecking, MallocErrChecking, DictFormat, and ArrOrder

Feature1	Feature2	Result
(value)	(value)	=(value)
*	*	unspecified_DSL_feature

Table 83 *General formula for computing results of non-mathematical properties (addition and multiplication) [Neu98]*

10.2.7.6 Size and HashWidth

Size1	Size2	Size Result
(value1)	(value2)	=Max((value1),(value2))

Table 84 *Resulting* Size *(addition and multiplication) [Neu98]*

HashWidth1	HashWidth2	HashWidth Result
(value1)	(value2)	=Max((value1),(value2))

Table 85 *Resulting* HashWidth *(addition and multiplication) [Neu98]*

10.2.7.7 Rows and RowsNumber (Multiplication)

Rows1	Order1	Rows Result	RowsNumber Result
stat_val	*	stat_val	=RowsNumber1
*	stat_val	stat_val	=OrderNumber1
*	*	dyn_val	unspecified_DSL_feature

Table 86 *Resulting* RowsNumber *(multiplication) [Neu98]*

10.2.7.8 Cols and ColsNumber (Multiplication)

Cols2	Order2	Cols Result	ColsNumber Result
stat_val	*	stat_val	=ColsNumber2
*	stat_val	stat_val	=OrderNumber2
*	*	dyn_val	unspecified_DSL_feature

Table 87 *Resulting* ColsNumber *(multiplication) [Neu98]*

10.2.7.9 Order and OrderNumber (Multiplication)

Order1	Order2	Order Result	OrderNumber Result
stat_val	stat_val	stat_val	=OrderNumber1
*	*	dyn_val	unspecified_DSL_feature

Table 88 *Resulting* OrderNumber *(multiplication) [Neu98]*

10.2.7.10 Diags and DiagsNumber (Multiplication)

Diags1	Diags2	Diags Result
stat_val	stat_val	stat_val
*	*	dyn_val

Table 89 *Resulting Diags (multiplication) [Neu98]*

Shape	Diags Result	DiagsNumber Result
lowerBandTriang upperBandTriang bandDiag	stat_val	=DiagsNumber1 + DiagsNumber2 - 1
*	*	unspecified_DSL_feature

Table 90 *Resulting DiagsNumber (multiplication) [Neu98]*

10.2.7.11 ScalarValue (Multiplication)

ScalarValue1	ScalarValue2	ScalarValue Result	ScalarValueNumber Result
stat_val	stat_val	stat_val	=ScalarValueNumber1 * ScalarValueNumber2
*	*	dyn_val	unspecified_DSL_feature

Table 91 *Resulting ScalarValue (multiplication) [Neu98]*

10.2.7.12 Ratio and Growing (Multiplication)

We do not compute the result of Ratio and Growing but rather set them to unspecified_DSL_feature. This will cause the matrix generator to assume the default values for both parameters.

10.2.7.13 Rows, RowsNumber, Cols, and ColsNumber (Addition)

Rows1	Rows2	Rows Result	RowsNumber Result
stat_val	*	stat_val	=RowsNumber1
*	stat_val	stat_val	=RowsNumber2
*	*	dyn_val	unspecified_DSL_feature

Table 92 *Resulting RowsNumber (addition) [Neu98]*

Cols1	Cols2	Cols Result	ColsNumber Result
stat_val	*	stat_val	=ColsNumber1
*	stat_val	stat_val	=ColsNumber2
*	*	dyn_val	unspecified_DSL_feature

Table 93 *Resulting ColsNumber (addition) [Neu98]*

10.2.7.14 Order and OrderNumber (Addition)

Order1	Order2	Order Result	OrderNumber Result
stat_val	*	stat_val	=OrderNumber1
*	stat_val	stat_val	=OrderNumber2
*	*	dyn_val	unspecified_DSL_feature

Table 94 *Resulting OrderNumber (addition) [Neu98]*

Additionally, we have to treat the following case as an exception:

Shape == rect and Order == stat_val

In this case, the values for Rows, Cols, RowsNumber, and ColsNumber computed based on other tables have to be overridden as follows:

Rows= stat_val
Cols= stat_val
RowsNumber= OrderNumber
ColsNumber= OrderNumber

10.2.7.15 Diags and DiagsNumber (Addition)

Diags1	Diags2	Diags Result
stat_val	stat_val	stat_val
*	*	dyn_val

Table 95 *Resulting Diags (addition) [Neu98]*

Shape	Diags	DiagsNumber Result
lowerBandTriang upperBandTriang bandDiag	stat_val	=Max(DiagsNumber1, DiagsNumber2)
*	*	unspecified_DSL_feature

Table 96 *Resulting DiagsNumber (addition) [Neu98]*

10.2.7.16 ScalarValue (Addition)

ScalarValue1	ScalarValue2	ScalarValue Result	ScalarValueNumber Result
stat_val	stat_val	stat_val	=ScalarValueNumber1 + ScalarValueNumber2
*	*	dyn_val	unspecified_DSL_feature

Table 97 *Resulting ScalarValue (addition) [Neu98]*

10.2.7.17 Ratio and Growing (Addition)

Ratio1	Ratio2	Ratio Result
(value1)	(value2)	=ArithmeticAverage((value1),(value2))

Table 98 *Resulting Ratio (addition)*

Growing1	Growing2	Growing Result
(value1)	(value2)	=ArithmeticAverage((value1),(value2))

Table 99 *Resulting Growing (addition)*

10.3 Domain Implementation

This section covers the implementation of the matrix component specified in the domain implementation section (i.e. Section 10.2). First, we discuss how to implement the component in C++. For demonstration purpose, we will show the implementation of a very small matrix component, which nonetheless, allows us to explain most of the relevant implementation techniques. The full C++ implementation of the matrix component is described in Section 10.3.1.8.

In Section 10.3.2, we summarize the experience we made during the implementation of a subset of the matrix component in the Intentional Programming System.

10.3.1 C++ Implementation

10.3.1.1 Architecture Overview

The architecture of the C++ implementation of the matrix component is shown in Figure 179. The pipeline on the left compiles matrix configuration expressions, e.g.

```
matrix<double, structure<rect<dyn_val<>, dyn_val<>, CSR<> >, sparse<> > >
```

whereas the pipeline on the right compiles matrix expressions, e.g.

```
M5 = M1+M2*(M3-M4);
```

The compilation of a matrix configuration expression involves parsing, computing defaults for the unspecified features, assembling the GenVoca-like components according to the feature values, and storing the feature values in the configuration repository, which becomes part of the produced matrix type. Features are encoded as types (numbers are encoded using enumeration types). We group them by putting them into a class (or struct) as its member types. This is exactly how the configuration repository is implemented.

A matrix expression is parsed using *expression templates* [Vel95]. We will explain this idea in Section 10.3.1.7.1. The parsing result is a tree of expression objects (or rather the type of the tree to be instantiated at runtime) much in the style shown in Figure 175 and Figure 177. We also need to compute the matrix type of the expression and of all its subexpressions, so that the code generation templates can select the appropriate implementations for the assignment and the getElement() methods (see Section 10.2.6.2) and ask the component assembler to generate the matrix types for the intermediate results (or matrix caches; see Section 10.2.6.1) if any needed.[154]

Any part of the architecture requiring compile-time execution (e.g. computing feature defaults, assembling components, computing result types) is implemented as template metafunctions (we discussed template metafunctions in Chapter 8).

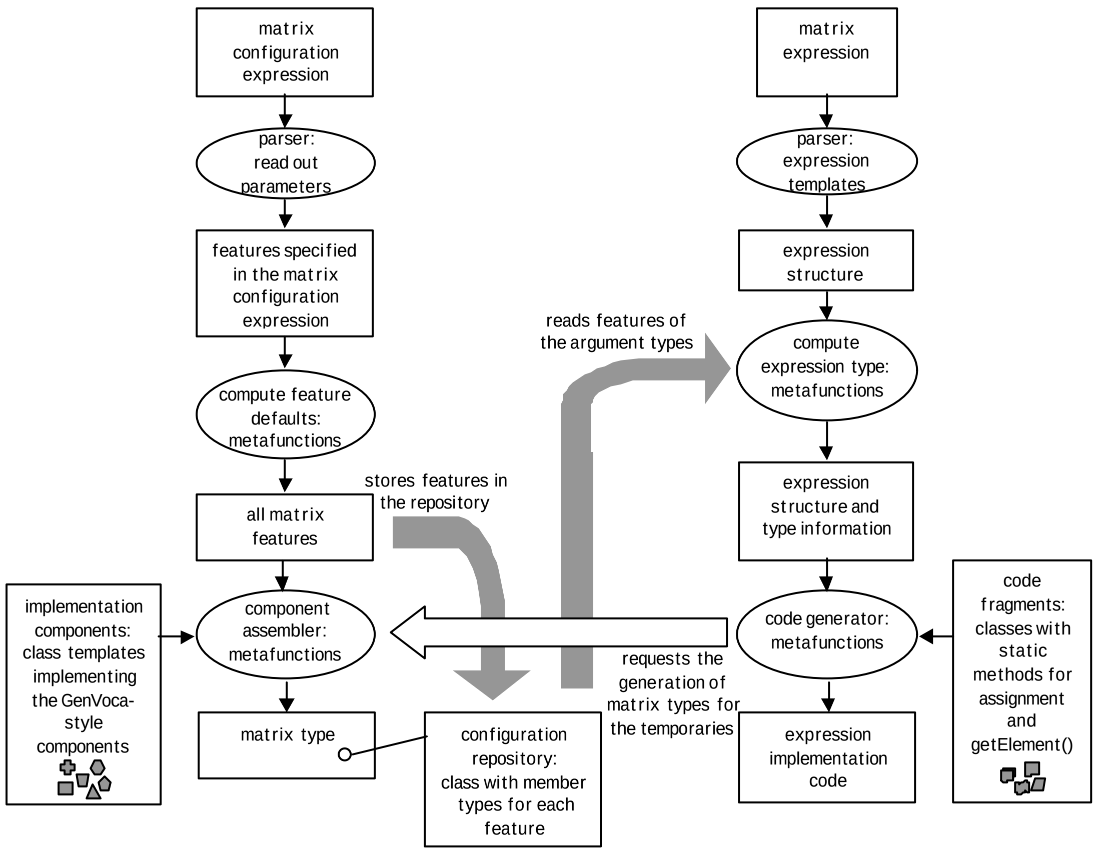
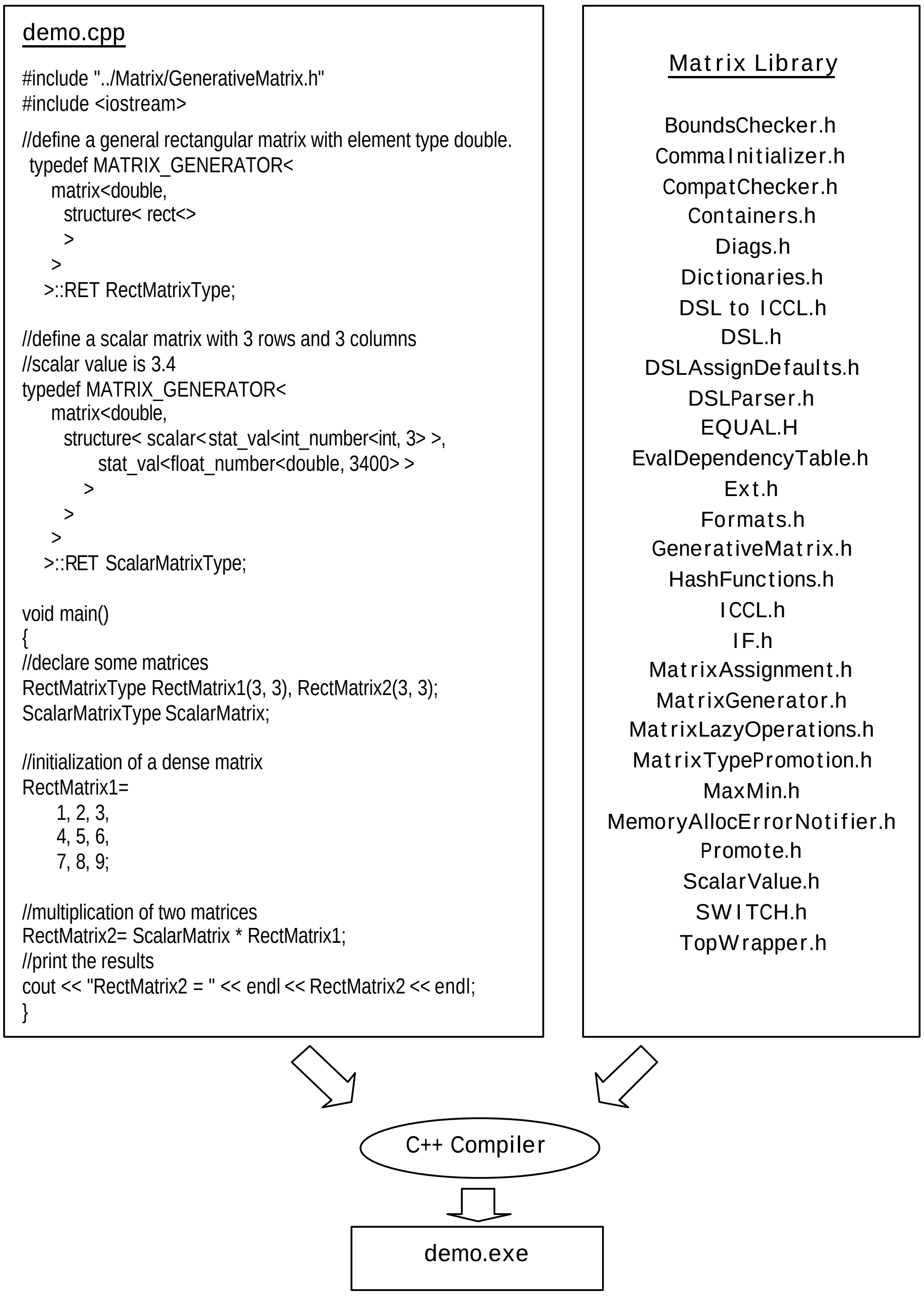

matrix configuration expression
parser: read out parameters
features specified in the matrix configuration expression
compute feature defaults: metafunctions
all matrix features
stores features in the repository
implementation components: class templates implementing the GenVoca-style components
component assembler: metafunctions
matrix type
configuration repository: class with member types for each feature
reads features of the argument types
matrix expression
parser: expression templates
expression structure
compute expression type: metafunctions
expression structure and type information
code generator: metafunctions
requests the generation of matrix types for the temporaries
expression implementation code
code fragments: classes with static methods for assignment and getElement()

Figure 179 *Architecture of the C++ implementation of the matrix component*

The whole matrix component is implemented as a C++ template library. The compilation of a client program including this library is done entirely by a C++ compiler (i.e. we neither use preprocessors nor any generation tools). This is possible since template metaprograms are interpreted by the compiler at compile time. Figure 180 illustrates the compilation of some demo client code using the matrix library.



```
#include "../Matrix/GenerativeMatrix.h"
#include <iostream>

//define a general rectangular matrix with element type double.
 typedef MATRIX_GENERATOR<
    matrix<double,
      structure< rect<>
      >
    >
  >::RET RectMatrixType;

//define a scalar matrix with 3 rows and 3 columns
//scalar value is 3.4
typedef MATRIX_GENERATOR<
    matrix<double,
      structure< scalar<stat_val<int_number<int, 3> >,
           stat_val<float_number<double, 3400> >
        >
      >
    >
  >::RET ScalarMatrixType;

void main()
{
//declare some matrices
RectMatrixType RectMatrix1(3, 3), RectMatrix2(3, 3);
ScalarMatrixType ScalarMatrix;

//initialization of a dense matrix
RectMatrix1=
    1, 2, 3,
    4, 5, 6,
    7, 8, 9;

//multiplication of two matrices
RectMatrix2= ScalarMatrix * RectMatrix1;
//print the results
cout << "RectMatrix2 = " << endl << RectMatrix2 << endl;
}
```

Figure 180 *Compilation scenario of a matrix demo program*

The following sections describe the C++ implementation techniques used to implement the generator. We demonstrate theses techniques using a smaller matrix component, which we describe as next.

10.3.1.2 Overview of the Demo Implementation

The C++ implementation of the demo matrix component consists of a number of C++ modules listed in Table 100. The modules are grouped into five categories. The following sections cover the implementation of each module in the order they are listed in the table. Each section contains the full module source code. The beginning of a new module is marked by a gray side box indicating its name. The source code itself is marked by a vertical ruler to its left. This helps to distinguish it from explanations and code examples (the latter do not have the vertical ruler).

Category	Contents	Modules	Contents	Section
Matrix Configuration DSL	specification and implementation of the Matrix Configuration DSL	DSL.h		10.3.1.3
Matrix ICCL	specification and implementation of the matrix implementation components	ICCL.h	specification of the matrix ICCL	10.3.1.4
		Containers.h	basic containers	10.3.1.4.1
		Formats.h	formats	10.3.1.4.2
		BoundsChecker.h	bounds checker	10.3.1.4.3
		TopWrapper.h	matrix top wrapper	10.3.1.4.4
		CommaInitializer.h	comma initializer	
Matrix Configuration Generator	implementation of the matrix configuration generator	MatrixGenerator.h	matrix configuration generator	10.3.1.5
		DSLParser.h	parse matrix configuration DSL	10.3.1.5.1
		DSLAssignDefaults.h	assign defaults to unspecified DSL features	10.3.1.5.2
		AssembleComponents.h	assemble implementation components	10.3.1.5.3
Matrix Operations	implementation of the matrix operations (Matrix Expression DSL)	MatrixOperTemplates.h	operator templates for the matrix operations	10.3.1.7.1
		MatrixExprTemplates.h	matrix operation expression class templates	10.3.1.7.2
		MatrixCache.h	matrix cache used for matrix multiplication	10.3.1.7.3
		GetElement.h	different implementations of getElement() method for addition and multiplication expressions of different shapes	10.3.1.7.4
		ComputeResultType.h	compute the matrix result type of matrix operations	10.3.1.7.5
		Assignment.h	different implementations of assignment for different matrix shapes	10.3.1.7.6
Auxiliary		GenerativeMatrix.h	general include file	
		Promote.h	promoting numeric types	10.2.7.1
		IF.h	meta if	8.2
		SWITCH.h	meta switch	8.13
		equal.h	auxiliary metafunction to be used with meta if	

Table 100 *Overview of the C++ implementation of the demo matrix component*

10.3.1.3 Matrix Configuration DSL

The matrix configuration DSL that we are going to implement is shown in Figure 181 as a feature diagram. This is an extremely simplified version of the full matrix configuration DSL described in Section 10.2. The DSL allows us to specify matrix element type, matrix shape, storage format, whether to optimize for speed or space, whether to do bounds checking or not, and index type.

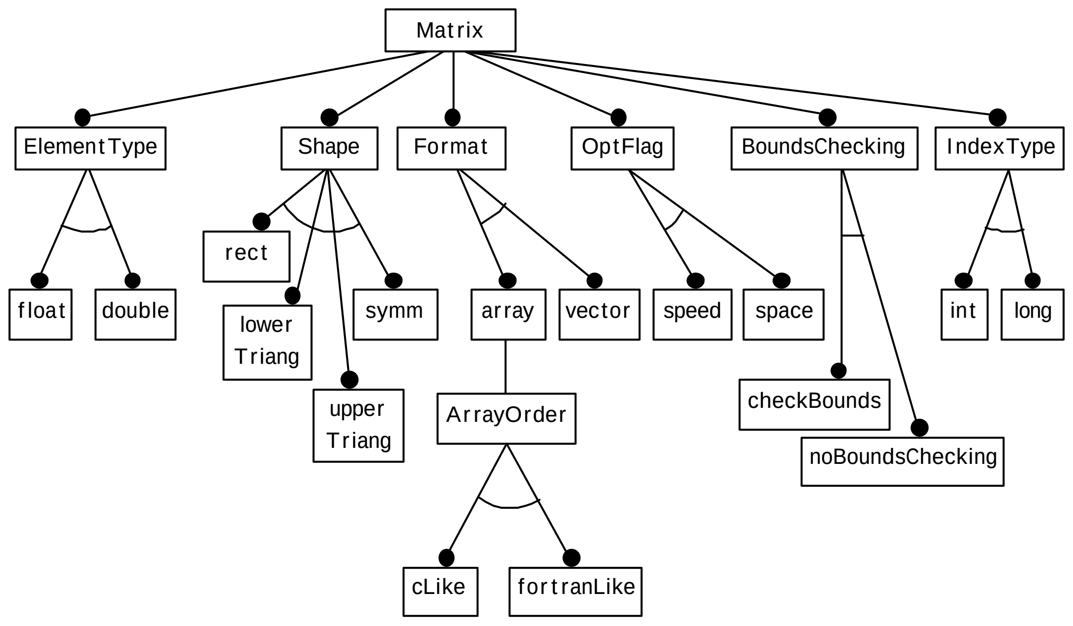


Figure 181 *Configuration DSL of the demo matrix component (Please note that not all possible element and index types are shown; cf. Figure 182)*

The grammar specification of the configuration DSL is given in Figure 182.

Matrix:	matrix[ElementType, Shape, Format, OptFlag, BoundsChecking, IndexType]
ElementType:	float | double | long double | int | long | ...
Shape:	rect | lowerTriang | upperTriang | symm
Format:	array[ArrayOrder] | vector
ArrayOrder:	cLike | fortranLike
OptFlag:	speed | space
BoundsChecking:	checkBounds | noBoundsChecking
IndexType:	char | short | int | long | unsigned int | ...

Figure 182 *Grammar specification of the demo matrix configuration DSL*

We implement the DSL using nested class templates in the module DSL.h.

First thing to do is to copy the grammar definition in Figure 182 and paste it in DSL.h as a comment. Next, under each grammar production, we directly type in the corresponding template declarations:

Module: DSL.h

```
namespace MatrixDSL {

//Matrix: matrix[ElementType, Shape, Format, OptFlag, BoundsChecking, IndexType ]
template<
  class ElementType, class Shape, class Format, class OptFlag,  class BoundsChecking, class IndexType
>struct matrix;

//ElementType :  float | double | long double | short | int | long | unsigned short |
//        unsigned int | unsigned long
//built-in types – nothing to declare

//Shape:  rect | upperTriang | lowerTriang | symm
template<class dummy>struct rect;
template<class dummy>struct lower_triang;
template<class dummy>struct upper_riang;
template<class dummy>struct symm;
```

Alternatively, we could have implemented the various shape values simply as structs rather than struct templates. However, by implementing them as templates we keep the DSL more extendible. This is so since we could add subfeatures to the values without having to change existing client code. All we have to do is to make sure that each template parameter has a default. In this case, if a client writes

```
rect<>
```

we really do not know how many parameters the rect template has.

We continue with the remaining grammar productions in a similar fashion:

```
//Format:  array[ArrayOrder] | vector
template<class ArrOrder>struct array;
template<class dummy>struct vector;


//ArrOrder:    cLike | fortranLike
template<class dummy>struct c_like;
template<class dummy>struct fortran_like;


//OptFlag :     speed | space
template<class dummy>struct speed;
template<class dummy>struct space;


//BoundsChecking : checkBounds | noBoundsChecking
template<class dummy>struct check_bounds;
template<class dummy>struct no_bounds_checking;


//IndexType : char | short | int | long | unsigned char | unsigned short |
//         unsigned int | unsigned long | signed char


//type denoting "value unspecified"
struct unspecified_DSL_feature;
```

The last struct is used as a feature value denoting “value unspecified”. We can use this feature in matrix configuration expressions if we do not want to specify some feature in the middle of a parameter list, e.g.

```
matrix<float,lower_triang,unspecified_DSL_feature,speed>
```

As you have probably already anticipated, we do not have to specify the last two parameters (i.e. bounds checking and index type) since the matrix template defines appropriate defaults.

Now, let us take a look at the implementation of the DSL features. We start with the implementation of unspecified_DSL_feature that we have just mentioned above. As stated, it is used to denote “value unspecified”. However, this is not its only purpose. We also use it as the superclass for all other DSL feature values. This has the following reason: unspecified_DSL_feature defines identification numbers for all DSL features. Identification numbers, or IDs, are used to test types for equality. This is necessary since there is no other means in C++ to do it.

First, we define unspecified_DSL_feature:

```
struct unspecified_DSL_feature
{
  enum {
    unspecified_DSL_feature_id = -1,

    // IDs of Shape values
    rect_id,
    lower_triang_id,
    upper_triang_id,
    symm_id,

    //IDs of Format values
    array_id,
    vector_id,

    //IDs of ArrOrder values
    c_like_id,
    fortran_like_id,

    //IDs of OptFlag values
    speed_id,
    space_id,

    //IDs of BoundsChecking values
    check_bounds_id,
    no_bounds_checking_id,

    //my own ID
    id=unspecified_DSL_feature_id };
};
```

Here is the implementation of the first DSL production:

```
//Matrix: matrix[ElementType, Shape, Format, OptFlag, BoundsChecking, IndexType ]
template<
  class ElementType = unspecified_DSL_feature,
  class Shape = unspecified_DSL_feature,
  class Format = unspecified_DSL_feature,
  class OptFlag = unspecified_DSL_feature,
  class BoundsChecking = unspecified_DSL_feature,
  class IndexType = unspecified_DSL_feature >
struct matrix
{
  typedef ElementType elementType;
  typedef Shape shape;
  typedef Format format;
  typedef OptFlag optFlag;
  typedef BoundsChecking boundsChecking;
  typedef IndexType indexType;
};
```

Please note that we use unspecified_DSL_feature as the default value for each parameter. Of course, we will assign some more useful default values later in the generator. The reason for not assigning the defaults here is that we want to assign all defaults (i.e. both direct and computed) in one place, which will be the matrix generator.

The next thing to point out is that matrix<> defines each parameter type as its member type. We say that it *publishes* its parameters. This is so since now we can access its parameters as follows:

```
matrix<Foo1,Foo2>::shape //this is equivalent to Foo2
```

The final detail is the reason why we use a struct and not a class: all members of matrix are public and by using a struct we do not have to write the extra access modifier public:.

The next production consists of a number of alternative values:

```
//Shape :    rect | lowerTriang | upperTriang | symm
template<class dummy = unspecified_DSL_feature>
struct rect : unspecified_DSL_feature
{  enum { id=rect_id };
};

template<class dummy = unspecified_DSL_feature>
struct lower_triang : unspecified_DSL_feature
{  enum { id=lower_triang_id };
};

template<class dummy = unspecified_DSL_feature>
```

```
struct upper_triang : unspecified_DSL_feature
{ enum { id=upper_triang_id };
};

template<class dummy = unspecified_DSL_feature>
struct symm : unspecified_DSL_feature
{ enum { id=symm_id };
};
```

Each "dummy" parameter has unspecified_DSL_feature as its default. As you remember, the purpose of this parameter was to make values without subfeatures templates, so that new subfeatures can be added without having to modify existing client code. Each value "publishes" its ID using an enum declaration. The initialization values for the IDs were defined in unspecified_DSL_feature, the superclass of all feature values. The following example demonstrates the use of IDs:

```
typedef rect<> Shape1;
typedef upper_triang<> Shape2;
typedef upper_triang<> Shape3;

cout << (Shape1::id == Shape2::id);  //prints: 0
cout << (Shape2::id == Shape3::id);  //prints: 1
```

The approach with the IDs allows us for even a finer testing than just type equality: we can test if two types were instantiated from the same class template even if the types are not equal (i.e. different parameters were used):

```
typedef array<c_like<> > Format1; //array<> and c_like are defined below
typedef array<fortran_like<> > Format2; //fortran_like<> is defined below

cout << (Format1::id == Format2::id); //prints: 1
```

The remaining DSL features are specified in a similar way:

```

//Format   :  array[ArrOrder] | vector
template<class ArrOrder= unspecified_DSL_feature>
struct array : unspecified_DSL_feature
{ enum {id= array_id};
  typedef ArrOrder arr_order;
};

template<class dummy= unspecified_DSL_feature>
struct vector : unspecified_DSL_feature
{ enum {id= vector_id};
};


//ArrOrder:    cLike | fortranLike
template<class dummy = unspecified_DSL_feature>
struct c_like : unspecified_DSL_feature
{ enum { id= c_like_id };
};

template<class dummy= unspecified_DSL_feature>
struct fortran_like : unspecified_DSL_feature
{ enum {id= fortran_like_id};
};


//OptFlag :     speed | space
template<class dummy = unspecified_DSL_feature>
struct speed : unspecified_DSL_feature
{ enum { id=speed_id };
};

template<class dummy = unspecified_DSL_feature>
struct space : unspecified_DSL_feature
{ enum { id=space_id };
};


//BoundsChecking : checkBounds | noBoundsChecking
template<class dummy = unspecified_DSL_feature>
struct check_bounds : unspecified_DSL_feature
{ enum { id=check_bounds_id };
};

template<class dummy = unspecified_DSL_feature>
struct no_bounds_checking : unspecified_DSL_feature
{ enum { id=no_bounds_checking_id };
};

} //namespace MatrixDSL
```

This concludes the implementation of the matrix configuration DSL.

10.3.1.4 Matrix Implementation Components and the ICCL[155]

Figure 183 shows the GenVoca-like component architecture implementing the functionality scope specified in the previous section. The box at the bottom is the configuration repository (Config), which contains all the DSL features of a configuration and some extra types to be accessed by other components. The basic containers layer provides the containers for storing matrix elements: Dyn1DContainer (one-dimensional), Dyn2DCContainer (two-dimensional, row-wise storage) and Dyn2DFContainer (two-dimensional, column-wise storage). All containers allocate memory dynamically. On the top of the containers, we have three alternative format components: ArrFormat (used to store rectangular and triangular matrices), LoTriangVecFormat (used for lower triangular matrices), and UpTriangVecFormat (used for upper triangular matrices).[156] Symm is an optional wrapper for implementing the symmetry property of a matrix. BoundsChecker is another optional wrapper. It implements bounds checking. Finally, Matrix is the top-level wrapper of all matrices.

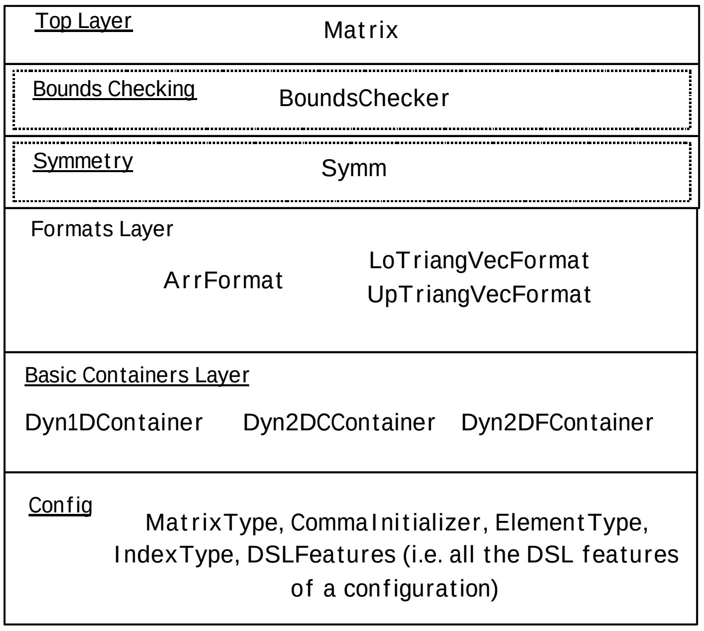


Figure 183 *Layers of the matrix implementation components*

The configurability of the matrix implementation components is specified by the ICCL grammar in Figure 184.

MatrixType:	Matrix[OptBoundsCheckedMatrix]
OptBoundsCheckedMatrix:	OptSymmetricMatrix | BoundsChecker[OptSymmetricMatrix]
OptSymmetricMatrix:	Format | Symm[Format]
Format:	ArrFormat[Array] | LoTriangVecFormat[Vector] | UpTriangVecFormat[Vector]
Array:	Dyn2DCContainer[Config] | Dyn2DFContainer[Config]
Vector:	Dyn1DContainer[Config]
Config:	Config contains: MatrixType, CommaInitializer, ElementType, IndexType, and DSLFeatures (i.e. all the DSL parameters of a configuration)

Figure 184 *ICCL grammar for the demo matrix package*

Each of the matrix implementation components will be implemented by a class template. But before we show the implementation of the components, we first need a file declaring all of the components. You can think of this file as the ICCL specification.

We start as in the case of the DSL specification by copying the ICCL grammar from Figure 184 into ICCL.h as a comment. We type in the component declarations under the corresponding ICCL production:

Module: ICCL.h

```
namespace MatrixICCL {

//Matrix :     Matrix [OptBoundsCheckedMatrix]
template<class OptBoundsCheckedMatrix>class Matrix;

//OptBoundsCheckedMatrix:   OptSymmetricMatrix | BoundsChecker[OptSymmetricMatrix]
template<class OptSymmetricMatrix>class BoundsChecker;

//OptSymmetricMatrix:     MatrixContainer | Symm[MatrixContainer]
template<class MatrixContainer>class Symm;

//Format:     ArrFormat[Array] | LoTriangVecFormat[Vector] | UpTriangVecFormat[Vector]
template< class Array>class ArrFormat;
template< class Vector>class LoTriangVecFormat;
template< class Vector>class UpTriangVecFormat;

//Array:   Dyn2DCContainer[Config] | Dyn2DFContainer[Config]
template<class Generator>class Dyn2DCContainer;
template<class Generator>class Dyn2DFContainer;
```

At this point, we need to explain one implementation detail. As you know, Config is the configuration repository which is always passed to the components in the bottom layer of a GenVoca architecture. But the two last template declarations take Generator as their parameter instead. This is not a problem, however, since Config is a member type of Generator. This detail has to do with some C++ compiler problems (specifically VC++5.0) which are not relevant here. Here are the remaining declarations:

```

//Vector:     Dyn1DContainer[Config]
template<class Generator> class Dyn1DContainer;

//CommaInitializer:   DenseCCommaInitializer | DenseFCommaInitializer
template<class MatrixType>class DenseCCommaInitializer;
template<class MatrixType>class DenseFCommaInitializer;

} //namespace MatrixICCL
```

We use CommaInitializer to provide matrix initialization by comma-separated lists of numbers.

Now we give you the implementation of the components. Each component group is treated in a separate section.

In case that you wonder where the Config is: Config is defined as a member class of the matrix component assembler discussed in Section 10.3.1.5.3.

10.3.1.4.1 Basic Containers

As shown in Figure 183, we need to implement three containers: Dyn1DContainer, Dyn2DCContainer, and Dyn2DFContainer. We start with Dyn1DContainer (in Containers.h):

Module: Containers.h

```
namespace MatrixICCL{

template<class Generator>
class Dyn1DContainer
{   public:
      typedef Generator::Config Config;
      typedef Config::ElementType ElementType;
      typedef Config::IndexType IndexType;
```

As already stated, Generator is expected to provide Config as its member type. Config, in turn, provides element type and index type. All matrix components can access these types in this fashion. Dyn1DContainer allocates the memory for storing its elements from the heap. The size is specified in the constructor:

```
    protected:
      IndexType size_;
      ElementType * pContainer;

    public:
      Dyn1DContainer(const IndexType& l)
        : size_(l)
      { assert(size()>0);
        pContainer = new ElementType [size()];
        assert( pContainer != NULL );
      }

      ~Dyn1DContainer() {delete [] pContainer;}

      void setElement(const IndexType& i, const ElementType& v)
      { checkBounds( i );
        pContainer[ i ] = v;
      }

      const ElementType & getElement(const IndexType& i) const
      { checkBounds( i );
        return pContainer[ i ];
      }

      const IndexType& size() const {return size_;}

      void initElements(const ElementType& v)
      { for( IndexType i = size(); i--; )
        setElement( i, v );
      }

    protected:
      void checkBounds(const IndexType& i) const
      { assert(i>=0 && i<size());
      }
};
```

Dyn2DCContainer is a two dimensional container storing its elements row-wise:

```
template<class Generator>
class Dyn2DCContainer
{
  public:
    typedef Generator::Config Config;
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;
  protected:
    IndexType r_, c_;
    ElementType* elements_;
    ElementType** rows_;

  public:
    Dyn2DCContainer(const IndexType& r, const IndexType& c)
      : r_(r), c_(c)
    {
      assert(r_>0);
      assert(c_>0);

      elements_ = new ElementType[r*c];
      rows_ = new ElementType*[r];
      assert(elements_ != NULL);
      assert(rows_ != NULL);

      ElementType* p= elements_;
      for (IndexType i= 0; i<r; i++, p+= c) rows_[i]= p;
    }

    ~Dyn2DCContainer()
    { delete [] elements_;
      delete [] rows_;
    }

    void setElement(const IndexType& i, const IndexType& j, const ElementType& v)
    { checkBounds(i, j);
      rows_[i][j] = v;
    }

    const ElementType& getElement(const IndexType& i, const IndexType& j) const
    { checkBounds(i, j);
      return rows_[i][j];
    }

    const IndexType& rows() const { return r_; }
    const IndexType& cols() const { return c_; }

    void initElements(const ElementType& v)
    {   for(IndexType i = rows(); i--;)
          for(IndexType j = cols(); j--;)
            setElement(i, j, v);
    }

  protected:
    void checkBounds(const IndexType& i, const IndexType& j) const
    { assert(i>=0 && i<rows());
      assert(j>=0 && j<cols());
    }
};
```

Dyn2DFContainer is a two dimensional container storing its elements column-wise. We can easily derive its implementation from Dyn2DCContainer by inheritance. All we have to do is to override setElement() and getElement() to swap the argument indices and also override rows() and cols() to call the base cols() and rows(), respectively:

```
template<class Generator>
class Dyn2DFContainer : public Dyn2DCContainer<Generator>
{
  private:
    typedef Dyn2DCContainer<Generator> BaseClass;

  public:
    Dyn2DFContainer(const IndexType& r, const IndexType& c)
      : BaseClass(c, r)
    {}
    void setElement(const IndexType& i, const IndexType& j, const ElementType& v)
    {BaseClass::setElement(j, i, v);}

    const ElementType & getElement(const IndexType& i, const IndexType& j) const
    {return BaseClass::getElement(j, i);}

    const IndexType& rows() const {return BaseClass::cols();}
    const IndexType& cols() const {return BaseClass::rows();}
};

} //namespace MatrixICCL
```

10.3.1.4.2 Formats

For our demo matrix, we implement three formats: ArrFormat, LoTriangVecFormat, and UpTriangVecFormat. ArrFormat stores matrix elements directly in a two-dimensional container. This format is appropriate for storing rectangular and triangular matrices. In the latter case, only half of the allocated container memory is used. On the other hand, accessing the elements of a triangular matrix stored in ArrFormat does not involve any explicit index arithmetic. In case you prefer a more space-saving variant, you can use LoTriangVecFormat for lower triangular matrices and UpTriangVecFormat for upper triangular matrices. Each of the latter formats uses a one-dimensional container to store its elements. The elements of a symmetric matrix are stored the same way as the elements of a lower triangular matrix. The only difference is that, for a symmetric matrix, we wrap the format in Symm, which maps any access to the elements above the main diagonal to the elements of the lower half of the underlying format.

We start with ArrFormat. Since ArrFormat stores matrix elements in a two-dimensional container directly, there is hardly any difference between storing the elements of rectangular and triangular matrices. The only detail we have to do differently for triangular matrices is to directly return zero for their zero halves rather than accessing the corresponding container elements. We will encapsulate this detail in the function nonZeroRegion(), which takes the indices i and j and returns true if they address an element within the nonzero region of a matrix and false otherwise. We will have three different implementations of this function: one for rectangular matrices, one for lower triangular matrices, and one for upper triangular matrices. The implementation to be used in a given configuration of matrix components will be selected at compile time according to the value of the shape feature stored in Config. We will implement each variant of the function as a static function of a separate struct and use a metafunction to select the appropriate struct based on the current shape. Here is the implementation of nonZeroRegion for rectangular matrices (in Formats.h):

Module: Formats.h

```
namespace MatrixICCL{

struct RectNonZeroRegion
{ template<class M>
  static bool nonZeroRegion(const M* m, const M::Config::IndexType& i, const M::Config::IndexType& j)
  { return true;
  }
};
```

nonZeroRegion() takes a number of parameters which are not relevant here: we always return true since any of the elements of a rectangular matrix could be a nonzero element. This is different for a lower triangular matrix:

```
struct LowerTriangNonZeroRegion
{ template<class M>
  static bool nonZeroRegion(const M* m, const M::Config::IndexType& i, const M::Config::IndexType& j)
  { return i>=j;
  }
};
```

The first parameter of nonZeroRegion() is a pointer to the matrix format calling this function. We will see the point of call later. The only purpose of this parameter is to provide type information:

we retrieve the index type from its Config. The nonzero region of an upper triangular matrix is just the negation of the above:

```
struct UpperTriangNonZeroRegion
{ template<class M>
  static bool nonZeroRegion(const M* m, const M::Config::IndexType& i, const M::Config::IndexType& j)
  {    return i<=j;
  }
};
```

The following is the metafunction selecting the appropriate implementation of nonZeroRegion() based on the matrix shape:

```
template<class MatrixType>
struct FORMAT_NON_ZERO_REGION
{ typedef MatrixType::Config Config;
  typedef Config::DSLFeatures DSLFeatures;
  typedef DSLFeatures::Shape Shape;

  typedef IF< EQUAL<Shape::id, Shape::lower_triang_id>::RET ||
          EQUAL<Shape::id, Shape::symm_id>::RET,
            LowerTriangNonZeroRegion,

      IF<EQUAL<Shape::id, Shape::upper_triang_id>::RET,
            UpperTriangNonZeroRegion,

      RectNonZeroRegion>::RET>::RET RET;
};
```

Thus, the metafunction uses a nested meta IF to select the appropriate implementation (we discussed metafunctions in Section 8.2). Now, we can implement ArrFormat as follows:

```
template<class Array>
class ArrFormat
{ public:
    typedef Array::Config Config;
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;

  private:
    Array elements_;

  public:
    ArrFormat(const IndexType& r, const IndexType& c)
      : elements_(r, c)
    {}

    const IndexType& rows() const { return elements_.rows(); }
    const IndexType& cols() const { return elements_.cols(); }

    void setElement(const IndexType & i, const IndexType & j, const ElementType & v)
    { if (nonZeroRegion(i, j)) elements_.setElement(i, j, v);
      else assert(v == ElementType( 0 ));
    }

    ElementType getElement(const IndexType & i, const IndexType & j) const
    { return nonZeroRegion(i, j) ? elements_.getElement(i, j) : ElementType(0);
    }

    void initElements(const ElementType & v)
    { elements_.initElements(v);
    }

  protected:
    bool nonZeroRegion(const IndexType& i, const IndexType& j) const
    { return FORMAT_NON_ZERO_REGION<Config::MatrixType>::RET::nonZeroRegion(this,i, j);
    }
};
```

The last return demonstrates the call to nonZeroRegion(). The struct containing the appropriate function implementation is returned by the metafunction FORMAT_NON_ZERO_REGION<>. Since we declared all implementations of nonZeroRegion() as static, inline functions of the structs, the C++ compiler should be able to inline this function to eliminate any overhead. This technique represents the static alternative to virtual functions.

LoTriangVecFormat stores the elements of a lower triangular matrix row-wise in a vector:

```
template<class Vector>
class LoTriangVecFormat
{ public:
    typedef Vector::Config Config;
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;

  private:
    const order_; //number of rows and columns
    Vector elements_;

  public:
    LoTriangVecFormat(const IndexType& r,const IndexType& c)
      : order_(r), elements_(rows() * (rows() + 1) * 0.5)
    { assert(rows()==cols());
    }

    const IndexType& rows() const { return order_; }
    const IndexType& cols() const { return order_; }

    void setElement(const IndexType & i, const IndexType & j, const ElementType & v)
    { if (i >= j) elements_.setElement(getIndex(i, j), v);
      else assert(v == ElementType( 0 ));
    }

    ElementType getElement(const IndexType & i, const IndexType & j) const
    { return i >= j ? elements_.getElement(getIndex(i, j))
                    : ElementType(0);
    }

    void initElements(const ElementType & v)
    { elements_.initElements(v);
    }

  protected:
    IndexType getIndex(const IndexType& i, const IndexType& j) const
    { return (i + 1) * i * 0.5 + j;
    }
};
```

UpTriangVecFormat can be easily derived from LoTriangVecFormat. We only need to override setElement() and getElement() in order to swap the row and column index:

```
template<class Vector>
class UpTriangVecFormat : public LoTriangVecFormat<Vector>
{ public:
    UpTriangVecFormat(const IndexType & r,const IndexType & c)
      : LoTriangVecFormat(r, c)
    {}

    void setElement(const IndexType & i, const IndexType & j, const ElementType & v)
    { LoTriangVecFormat::setElement(j, i, v);
    }

    ElementType getElement(const IndexType & i, const IndexType & j) const
    { return LoTriangVecFormat::getElement(j, i);
    }
};
```

Finally, we need to implement Symm. Symm takes a lower triangular format as its parameter and turns it into a symmetric one. Symm is derived from its parameter and overrides setElement() and getElement():

```

template<class Format>
class Symm : public Format
{ public:
    typedef Format::Config Config;
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;

    Symm(const IndexType& rows,const IndexType& cols)
      : Format(rows, cols)
    {}

    void setElement(const IndexType & i, const IndexType & j, const ElementType & v)
    { if( i >= j ) Format::setElement(i, j, v);
      else Format::setElement(j, i, v);
    }

    ElementType getElement(const IndexType & i, const IndexType & j) const
    { return ( i >= j ) ?
        Format::getElement(i, j) :
        Format::getElement(j, i);
    }
};

} //namespace MatrixICCL
```

10.3.1.4.3 Bounds Checking

Bounds checking is implemented by a wrapper similar to Symm discussed above. Here is the implementation (in BoundsChecker.h):

Module: BoundsChecker.h

```
namespace MatrixICCL{

template<class OptSymmMatrix>
class BoundsChecker : public BaseClass
{ public:
    typedef BaseClass::Config Config;
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;

    BoundsChecker(const IndexType& r, const IndexType& c)
      : OptSymmMatrix(r, c)
    {}

    void setElement(const IndexType& i, const IndexType& j, const ElementType& v)
    { checkBounds(i, j);
      OptSymmMatrix::setElement(i, j, v);
    }

    ElementType getElement(const IndexType& i, const IndexType& j) const
    { checkBounds(i, j);
      return OptSymmMatrix::getElement(i, j);
    }

  protected:
    void checkBounds(const IndexType & i, const IndexType & j) const
    { if ( i < 0 || i >= rows() ||
          j < 0 || j >= cols() )
        throw "subscript(s) out of bounds";
    }
};

} //namespace MatrixICCL
```

10.3.1.4.4 Matrix Wrapper

The top-level matrix component is Matrix<>. It defines a number of useful functions for all matrices: assignment operator for initializing a matrix using a comma-separated list of numbers, assignment operator for assigning binary expressions, assignment operator for assigning matrices to matrices, and a function for printing the contents of a matrix.

Module: TopWrapper.h

```
//declare BinaryExpression
template<class ExpressionType> class BinaryExpression;

namespace MatrixICCL{

template<class OptBoundsCheckedMatrix>
class Matrix : public OptBoundsCheckedMatrix
{ public:
    typedef OptBoundsCheckedMatrix::Config Config;

    typedef Config::IndexType IndexType;
    typedef Config::ElementType ElementType;
    typedef Config::CommaInitializer CommaInitializer;

    Matrix(IndexType rows= 0, IndexType cols= 0, ElementType InitElem = ElementType(0) )
      : OptBoundsCheckedMatrix(rows, cols)
    { initElements(InitElem);
    }

    //initialization by a comma-separated list of numbers
    CommaInitializer operator=(const ElementType& v)
    { return CommaInitializer(*this, v);
    }
```

The following assignment operator allows us to assign binary expressions to a matrix (we will discuss the class template BinaryExpression<> later):

```
    //assignment operator for binary expressions
    template <class Expr>
    Matrix& operator=(const BinaryExpression<Expr>& expr)
    { expr.assign(this);
      return *this;
    }
```

Finally, we have an assignment for assigning matrices to matrices. The implementation code depends on the shape of the source matrix. Thus, we use a similar technique as in the case of nonZeroRegion(): we select the appropriate implementation using a metafunction (the code for the metafunction and for the different assignment variants is given later):

```
    //matrix assignment
    template<class A>
    Matrix& operator=(const Matrix<A>& m)
    {
      MATRIX_ASSIGNMENT<A>::RET::assign(this, &m);
      return *this;
    }

    //assignment operators for other kinds of expressions
    //...

    //print matrix to ostream
    ostream& display(ostream& out) const
    { for( IndexType i = 0; i < rows(); ++i )
```

```
      { for( IndexType j = 0; j < cols(); ++j )
          out << getElement( i, j ) << "   ";
        out << endl;
      }
      return out;
    }
};

} //namespace MatrixICCL

//output operator for printing a matrix to a stream
template <class A>
ostream& operator<<(ostream& out, const Matrix<A>& m)
{
  return m.display(out);
}
```

10.3.1.5 Matrix Configuration Generator

The matrix configuration generator takes a matrix configuration description, e.g. matrix<double,rect<> >, and returns a matrix type with the properties specified in the configuration description. Given the above example configuration description, it generates the following matrix type:

```
Matrix<BoundsChecker<ArrFormat<Dyn2DCContainer<MATRIX_ASSEMBLE_COMPONENTS<...> > > > >
```

MATRIX_ASSEMBLE_COMPONENTS<> is the Generator parameter of Dyn2DCContainer we mentioned earlier. It has some parameters itself, but they are not relevant at this point (we indicated them by three dots). The only thing that matters here is that it contains Config, i.e. the configuration repository, as its member type and that Dyn2DCContainer<> can access it.

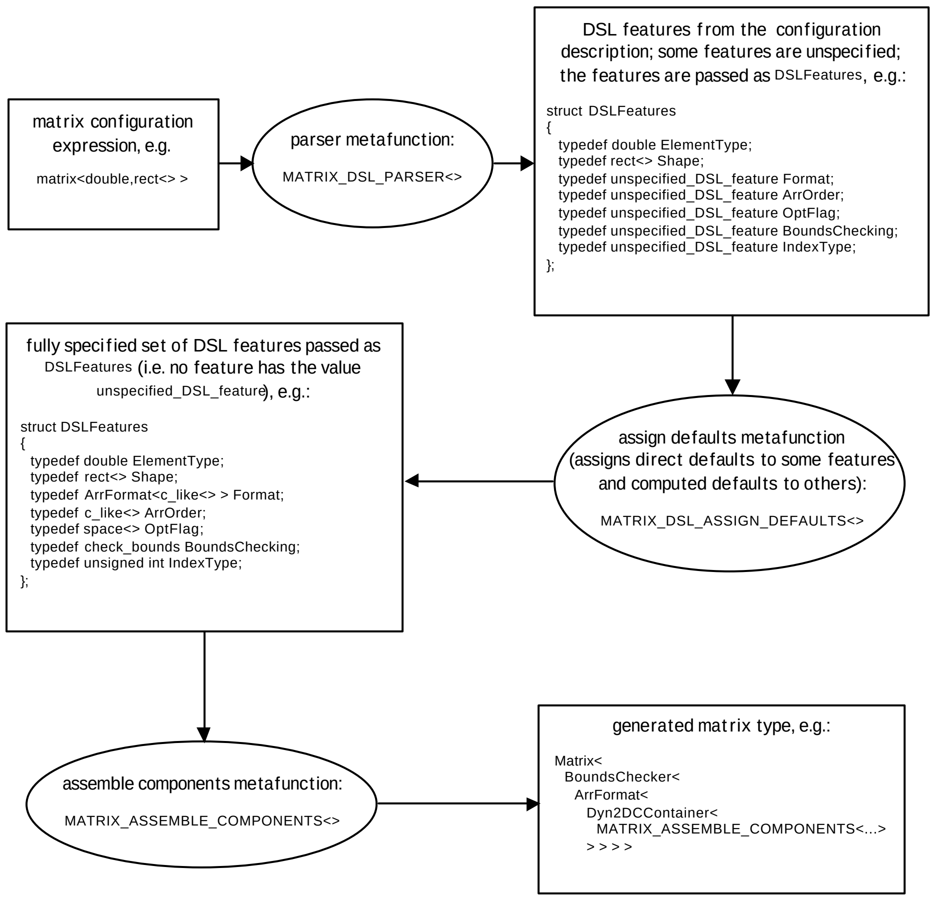
The generator performs three steps:

1. parsing the configuration description by reading out the nested DSL features;

2. assigning defaults to the unspecified DSL features;

3. assembling the matrix implementation components according to the DSL features.

These steps are shown in Figure 185. The processing steps and the corresponding metafunctions implementing them are enclosed in ellipses and the intermediate results are displayed in square boxes.


matrix configuration expression, e.g.
matrix<double,rect<> >
parser metafunction:
MATRIX_DSL_PARSER<>
DSL features from the configuration description; some features are unspecified; the features are passed as DSLFeatures, e.g.:
assign defaults metafunction (assigns direct defaults to some features and computed defaults to others):
MATRIX_DSL_ASSIGN_DEFAULTS<>
fully specified set of DSL features passed as DSLFeatures (i.e. no feature has the value unspecified_DSL_feature), e.g.:
assemble components metafunction:
MATRIX_ASSEMBLE_COMPONENTS<>
generated matrix type, e.g.:

```
struct DSLFeatures
{
   typedef double ElementType;
   typedef rect<> Shape;
   typedef unspecified_DSL_feature Format;
   typedef unspecified_DSL_feature ArrOrder;
   typedef unspecified_DSL_feature OptFlag;
   typedef unspecified_DSL_feature BoundsChecking;
   typedef unspecified_DSL_feature IndexType;
};
```

```
struct DSLFeatures
{
  typedef double ElementType;
  typedef rect<> Shape;
  typedef ArrFormat<c_like<> > Format;
  typedef c_like<> ArrOrder;
  typedef space<> OptFlag;
  typedef check_bounds BoundsChecking;
  typedef unsigned int IndexType;
};
```

```
Matrix<
  BoundsChecker<
    ArrFormat<
      Dyn2DCContainer<
        MATRIX_ASSEMBLE_COMPONENTS<...>
      > > > >
```

Figure 185 *Processing steps of the configuration generator*

Please note that the information about the matrix configuration passed between the processing steps is encoded by the DSLFeatures struct. This class contains a member type representing each of the DSL parameters (cf. Figure 182):

ElementType
Shape
Format
ArrayOrder
OptFlag
BoundsChecking
IndexType

This set of DSL parameters completely describes a matrix configuration. In other words, DSLFeatures represents the *type description record* of a matrix. We will use it later for selecting operation implementation and computing type records of expressions. We often refer to the DSLFeatures as the "flat" configuration class of a matrix.

Now we take a look at the implementation of the matrix generator. We implement it as a metafunction which takes two parameters: the matrix configuration description and a flag specifying what to do. The latter has the following purpose: We want to be able to pass not only a configuration DSL expression to the generator, but also DSLFeatures with some unspecified features and DSLFeatures with all features specified. In other words, if you take a look at Figure 185, we want to be able to enter the generation process just before any of the three processing steps. The reason for this is that we will sometimes already have DSLFeatures and just need to generate the corresponding matrix type. This is the case, for example, when we compute the type record of a matrix expression. We will see this later.

Thus, we first need to implement the flags: do_all, defaults_and_assemble, assemble_components. They have the following meaning:

- do_all: do parsing, assigning defaults, and component assembly; the generator expects a configuration DSL expression;

- defaults_and_assemble: do assigning defaults and component assembly; the generator expects DSLFeatures whose features do not have to be fully specified;

- assemble_components: do component assembly; the generator expects DSLFeatures whose features have to be fully specified;

We implement the flags as integer constants (in MatrixGenerator.h):

Module: MatrixGenerator.h

```
enum {
  do_all,
  defaults_and_assemble,
  assemble_components
};
```

Now, we can implement our configuration generator. As indicated in Figure 185, the generator delegates all the work to three other metafunctions, each of them defining one processing step:

```
MATRIX_DSL_PARSER<>
MATRIX_DSL_ASSIGN_DEFAULTS<>
MATRIX_ASSEMBLE_COMPONENTS<>
```

The generator returns (in its public section) the generated matrix type (as RET). The computation in the generator involves reading the "what to do" flag and calling the appropriate metafunctions:[157]

```
template< class InputDSL = matrix<>, int WhatToDo= do_all >
class MATRIX_GENERATOR
{
  // parse InputDSL (or dummy)
  typedef SWITCH< WhatToDo
              , CASE< assemble_components,     matrix<>          // dummy
              , CASE< defaults_and_assemble,    matrix<>          // dummy
              , DEFAULT<                        InputDSL
         > > > >::RET DSL_Description;
  typedef MATRIX_DSL_PARSER< DSL_Description >::RET ParsedDSL__;
```

Please note that we have to use the dummies since MATRIX_DSL_ASSIGN_DEFAULTS<> below will be "executed" in any case. We use the same calling pattern for the remaining two steps:

```
  // assign defaults to DSL (or to dummy)
  typedef SWITCH< WhatToDo
              , CASE< assemble_components,     ParsedDSL__  // dummy
              , CASE< defaults_and_assemble,    InputDSL
              , DEFAULT<                        ParsedDSL__
         > > > >::RET ParsedDSL_;
  typedef MATRIX_DSL_ASSIGN_DEFAULTS< ParsedDSL_ >::RET CompleteDSL__;

  // assemble components
  typedef SWITCH< WhatToDo
              , CASE< assemble_components,     InputDSL
              , CASE< defaults_and_assemble,    CompleteDSL__
              , DEFAULT<                        CompleteDSL__
         > > > >::RET CompleteDSL_;
  typedef MATRIX_ASSEMBLE_COMPONENTS< CompleteDSL_ > Result;
```

Finally, we have our public return:

```
public:
  typedef Result::RET     RET;
};
```

The following three sections describe the metafunctions implementing the three generation steps.

10.3.1.5.1 Configuration DSL Parser

The configuration DSL parser takes a matrix configuration DSL expression and produces DSLFeatures containing the values of the features explicitly specified in the configuration expression (the unspecified features have the value unspecified_DSL_feature; see Figure 185).

Before we show you the parser code, we need a small helper metafunction for testing whether a DLS feature is unspecified or not:

Module: DSLParser.h

```
using namespace MatrixDSL;

template<class TYPE> struct IsUnspecifiedDSLFeature {enum { RET=0 };};
template<> struct IsUnspecifiedDSLFeature<unspecified_DSL_feature>{enum { RET=1 };};
```

The parser retrieves all the features for the DSLFeatures class one at a time from the matrix configuration (i.e. its parameter DSLDescription) and returns the DSLFeatures:

```
template<class DSLDescription>
class MATRIX_DSL_PARSER
{ private:

    //ElementType
    typedef DSLDescription::elementType ElementType;

    //Shape
    typedef DSLDescription::shape Shape;

    //Format
    typedef DSLDescription::format Format;
```

The retrieval code for ArrOrder looks slightly different since this feature is nested only in the one value array<> of Format. Thus, we first have to check to see if the value of Format is array<...> and if this is not the case we simply return array<>, otherwise we return the actual value of Format. Then we use this intermediate result (i.e. ArrayFormat_) to read out the arr_order member. The intermediate result (i.e. ArrayFormat_) always contains arr_order, but its value is only relevant if the value of Feature is array<...>. We have to go through all this trouble since both types passed to a meta IF are actually built and we have to make sure that _ArrayOrder has arr_order as its member type in all cases. Here is the code:

```

    //ArrOrder
    typedef IF<EQUAL<Format::id, Format::array_id>::RET,
          Format,
          array<> >::RET ArrayFormat_;
    typedef IF<EQUAL<Format::id, Format::array_id>::RET,
          ArrayFormat_::arr_order,
          unspecified_DSL_feature>::RET ArrOrder;
```

The remaining parameters are simple to retrieve:

```

    //OptFlag
    typedef DSLDescription::optFlag OptFlag;

    //BoundsChecking
    typedef DSLDescription::boundsChecking BoundsChecking;

    //IndexType
    typedef DSLDescription::indexType IndexType;
```

Finally, we define DSLFeatures with all the DSL parameters as its member types and return it:

```

  public:
    struct DSLFeatures
    {
      typedef ElementType ElementType;
      typedef Shape Shape;
```

```
        typedef Format Format;
        typedef ArrOrder ArrOrder;
        typedef OptFlag OptFlag;
        typedef BoundsChecking BoundsChecking;
        typedef IndexType IndexType;
    };

    typedef DSLFeatures RET;
};
```

10.3.1.5.2 Assigning Defaults to DSL Features

The second generation step is to assign default values to the unspecified features in DSLFeatures returned by the parser (see Figure 185). Some features are assigned direct default values and some computed default values. Thus, we first need to specify the direct and the computed defaults for our DSL.

ElementType:	double
Shape:	rect
ArrayOrder:	cLike
OptFlag:	space
BoundsChecking:	checkBounds
IndexType:	unsigned int

Table 101 *Direct feature defaults for the demo matrix package*

The direct defaults are listed in Table 101. As already discussed, some choices might be a matter of taste, but we have to make them in any case (see discussion in Section 10.2.3).

The only computed default is Format and the computation formula is given in Table 102. According to this table, Format depends on Shape and OptFlag. We only use vector for triangular and symmetric matrices if the optimization flag is set to space.

Shape	OptFlag	Format
rect	*	array
lowerTriang upperTriang symm	speed	array
	space	vector

Table 102 *Computing default value for* Format

There is one more detail we have to specify: It is illegal to set Shape to rect and Format to vector at the same time. This combination does not seem to be useful and we choose to forbid it explicitly. This is specified in Table 103.

Shape	Format
rect	vector

Table 103 *Illegal feature combination*

Given the above specifications, we can move to the C++ implementation now. First, we implement the table with the direct feature defaults (i.e. Table 101) in a struct. This way we have all direct defaults in one place, which is desirable for maintenance reasons (in DSLAssignDefaults.h):

Module: DSLAssignDefaults.h

```
using namespace MatrixDSL;

//DSLFeatureDefaults implements Table 101 (i.e. direct feature defaults)
struct DSLFeatureDefaults
{ typedef double               ElementType;
  typedef rect<>               Shape;
  typedef c_like<>             ArrOrder;
  typedef space<>              OptFlag;
  typedef check_bounds<>     BoundsChecking;
  typedef unsigned int           IndexType;
};
```

The metafunction for assigning feature defaults does more than just assigning defaults: it also checks to make sure that the specified feature values are correct and that the feature combinations are correct. Thus, for example, if we specified the shape of a matrix as speed<>, this would be the place to catch this error.

The implementation of error checking is a bit tricky. A major deficiency of template metaprogramming is that we do not have any means to report a string to the programmer during compilation. In our code, we use the following partial solution to this problem: If we want to report an error, we access the nonexistent member SOME_MEMBER of some type SOME_TYPE which, of course, causes the compiler to issue a compilation error that says something like: “'SOME_MEMBER' is not a member of 'SOME_TYPE'”. Now, the idea is to use the name of the kind of error we want to report as SOME_TYPE and to encode the error text in the name of SOME_MEMBER.

Here is the implementation of the “SOME_TYPE”, which, in our case, we call DSL_FEATURE_ERROR:

```
struct DSL_FEATURE_ERROR {};
```

DSL_FEATURE_ERROR is usually returned by a meta IF. Thus, we also need some type to return if there is no error:

```
struct nil {}; // nil is just some other type

struct DSL_FEATURE_OK
{
  typedef nil WRONG_SHAPE;
  typedef nil WRONG_FORMAT_OR_FORMAT_SHAPE_COMBINATION;
  typedef nil WRONG_ARR_ORDER;
  typedef nil WRONG_OPT_FLAG;
  typedef nil WRONG_BOUNDS_CHECKING;
};
```

As you see, DSL_FEATURE_OK encodes error strings as member type names.

Now, for each DSL parameter, we implement a checking metafunction, which checks if the value of the parameter is one of the valid values according to the DSL grammar (see Figure 182):

```
template<class Shape>
struct CheckShape
{
  typedef IF< EQUAL<Shape::id, Shape::rect_id>::RET ||
        EQUAL<Shape::id, Shape::lower_triang_id>::RET ||
        EQUAL<Shape::id, Shape::upper_triang_id>::RET ||
        EQUAL<Shape::id, Shape::symm_id>::RET,
          DSL_FEATURE_OK,
          DSL_FEATURE_ERROR>::RET::WRONG_SHAPE RET;
};
```

Thus, if Shape is neither rect<>, nor lower_triang<>, nor upper_triang<>, nor symm<>, the meta IF returns DSL_FEATURE_ERROR and trying to access WRONG_SHAPE of DSL_FEATURE_ERROR results in a compilation error. In the other case, we return DSL_FEATURE_OK, which, as we already saw, defines WRONG_SHAPE as its member.

In practice, our error reporting approach looks as follows: If we try to compile the following line:

```
typedef MATRIX_GENERATOR<  matrix< double, speed<>  > >::RET MyMatrixType;
```

the C++ compiler (in our case VC++5.0) will issue the following error:

```
error C2039: 'WRONG_SHAPE' : is not a member of 'DSL_FEATURE_ERROR'
```

This error indicates that the second parameter to MATRIX_GENERATOR<> is not a valid shape value. This is a useful hint. Unfortunately, it does not tell us the source line where we used the wrong parameter value but rather the line where the erroneous member access occurred.

The checking of Format is a bit more complex since we need to implement Table 103. This table stated that we do not want to allow a rectangular matrix to be stored in a vector. Thus, the following metafunction takes Format and Shape as its parameters:

```
template<class Format, class Shape>
struct CheckFormatAndFormatShapeCombination
{
  typedef IF<(EQUAL<Shape::id, Shape::rect_id>::RET &&
           EQUAL<Format::id, Format::array_id>::RET) ||

         ((EQUAL<Shape::id, Shape::lower_triang_id>::RET ||
          EQUAL<Shape::id, Shape::upper_triang_id>::RET ||
          EQUAL<Shape::id, Shape::symm_id>::RET) &&
           (EQUAL<Format::id, Format::vector_id>::RET ||
            EQUAL<Format::id, Format::array_id>::RET)),

          DSL_FEATURE_OK,
          DSL_FEATURE_ERROR>::RET::WRONG_FORMAT_OR_FORMAT_SHAPE_COMBINATION RET;
};
```

Here are the checking metafunctions for the remaining three DSL parameters:

```
template<class ArrOrder>
struct CheckArrOrder
{
  typedef IF< EQUAL<ArrOrder::id, ArrOrder::c_like_id>::RET ||
          EQUAL<ArrOrder::id, ArrOrder::fortran_like_id>::RET,
            DSL_FEATURE_OK,
            DSL_FEATURE_ERROR>::RET::WRONG_ARR_ORDER RET;
};


template<class OptFlag>
struct CheckOptFlag
{
  typedef IF< EQUAL<OptFlag::id, OptFlag::speed_id>::RET ||
          EQUAL<OptFlag::id, OptFlag::space_id>::RET,
            DSL_FEATURE_OK,
            DSL_FEATURE_ERROR>::RET::WRONG_OPT_FLAG RET;
};


template<class BoundsChecking>
struct CheckBoundsChecking
{
  typedef IF< EQUAL<BoundsChecking::id, BoundsChecking::check_bounds_id>::RET ||
          EQUAL<BoundsChecking::id, BoundsChecking::no_bounds_checking_id>::RET,
            DSL_FEATURE_OK,
            DSL_FEATURE_ERROR>::RET::WRONG_BOUNDS_CHECKING RET;
};
```

Please note that we do not provide checking for element type and index type. The reason is that we do not want to unnecessarily limit the number of types that can be used in their place. This is particularly true of element type since we also would like to be able to create matrices of some user-defined types. If we use matrix expressions, the element type will also have to implement numeric operators such as +, *, -, +=, etc. If it does not, the C++ compiler will report an error from within the matrix operator code.

Finally, we are ready to implement our metafunction for assigning defaults:

```
template<class ParsedDSLDescription>
class MATRIX_DSL_ASSIGN_DEFAULTS
{ private:
    //define a short alias for the parameter
    typedef ParsedDSLDescription ParsedDSL;

    //ElementType
    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::ElementType>::RET,
          DSLFeatureDefaults::ElementType,
          ParsedDSL::ElementType>::RET ElementType;
```

If the element type is unspecified, the code above assigns it the direct default from Table 101. We do the same for IndexType, Shape, ArrOrder, OptFlag, and BoundsChecking:

```
    //IndexType
    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::IndexType>::RET,
          DSLFeatureDefaults::IndexType,
          ParsedDSL::IndexType>::RET IndexType;

    //Shape
    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::Shape>::RET,
          DSLFeatureDefaults::Shape,
          ParsedDSL::Shape>::RET Shape;
    typedef CheckShape<Shape>::RET check_shape_;
```

The last typedef calls the checking metafunction for Shape. We always call the checking metafunction after assigning the default value:

```
    //ArrOrder
    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::ArrOrder>::RET,
          DSLFeatureDefaults::ArrOrder,
          ParsedDSL::ArrOrder>::RET ArrOrder;
    typedef CheckArrOrder<ArrOrder>::RET check_arr_order_;

    //OptFlag
    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::OptFlag>::RET,
          DSLFeatureDefaults::OptFlag,
          ParsedDSL::OptFlag>::RET OptFlag;
    typedef CheckOptFlag<OptFlag>::RET check_opt_flag_;

    //BoundsChecking
    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::BoundsChecking>::RET,
          DSLFeatureDefaults::BoundsChecking,
          ParsedDSL::BoundsChecking>::RET BoundsChecking;
    typedef CheckBoundsChecking<BoundsChecking>::RET check_bounds_checking_;
```

Format is a special case since it does not have a direct default value. Its default value is determined based on Shape and OptFlag, as specified in Table 102:

```
    //Format
    typedef
      IF< (EQUAL<Shape::id, Shape::lower_triang_id>::RET ||
         EQUAL<Shape::id, Shape::upper_triang_id>::RET ||
         EQUAL<Shape::id, Shape::symm_id>::RET) &&
          EQUAL<OptFlag::id, OptFlag::space_id>::RET,

          vector<>,
          array<> >::RET ComputedFormat_;

    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::Format>::RET,
      ComputedFormat_,
      ParsedDSL::Format>::RET Format;
```

Next, we need to check the format-shape combination (cf. Table 103):

```
  typedef CheckFormatAndFormatShapeCombination<Format, Shape>::RET
    check_format_and_format_shape_combination_;
```

Finally, we return the DSLFeatures containing all DSL parameters:

```

public:
  struct DSLFeatures
  {
      typedef ElementType ElementType;
      typedef Shape Shape;
      typedef Format Format;
      typedef ArrOrder ArrOrder;
      typedef OptFlag OptFlag;
      typedef BoundsChecking BoundsChecking;
      typedef IndexType IndexType;
  };

  typedef DSLFeatures RET;
};
```

10.3.1.5.3 Matrix Component Assembler

The final step in the matrix type generation is assembling the matrix implementation components according to the matrix type record (i.e. DSLFeatures) produced in the earlier stages (see Figure 185).

First, we need to specify how to compute the ICCL parameters from the DSL parameters. According to the ICCL grammar in Figure 184, we have the following ICCL parameters with alternative values:

Array
Format
OptSymmetricMatrix
OptBoundsCheckedMatrix

Their computation from the DSL parameters is specified in Table 104 through Table 107.

ArrOrder	Array
cLike	Dyn2DCContainer
fortranLike	Dyn2DFContainer

Table 104 *Table for computing the ICCL parameter* Array

Shape	Format (DSL)	Format (ICCL)
*	array	ArrFormat
lowerTriangular symmetric	vector	LoTriangVecFormat
upperTriangular	vector	UpTriangVecFormat

Table 105 *Table for computing the ICCL parameter* Format

Shape	OptSymmetricMatrix
symm	Symm
*	= Format (ICCL)

Table 106 *Table for computing the ICCL parameter* OptSymmetricMatrix

BoundsChecking	**OptBoundsCheckedMatrix**
checkBounds	BoundsChecker
noBoundsChecking	= OptSymmetricMatrix

Table 107 *Table for computing the ICCL parameter* OptBoundsCheckedMatrix

According to the ICCL grammar, the value of Vector is Dyn1DContainer[Config]. Finally, the types provided by the configuration repository (i.e. Config, see Figure 183) are determined using Table 108 and Table 109.

ElementType (ICCL)	= ElementType (DSL)
IndexType (ICCL)	= IndexType (DSL)

Table 108 *Table for computing the ICCL parameters* ElementType *and* IndexType

ArrOrder	**CommaInitializer**
cLike	DenseCCommaInitializer
fortranLike	DenseFCommaInitializer

Table 109 *Table for computing the ICCL parameter* CommaInitializer

The metafunction for assembling components is implemented as follows (in AssembleComponents.h):

Module: AssembleComponents.h

```
using namespace MatrixDSL;
using namespace MatrixICCL;

template<class CompleteDSLDescription>
class MATRIX_ASSEMBLE_COMPONENTS
{ private:
    //introduce the alias Generator for itself
    typedef MATRIX_ASSEMBLE_COMPONENTS<CompleteDSLDescription> Generator;
    //introduce short alias for the parameter
    typedef CompleteDSLDescription DSLFeatures;
```

Now, each of the following typedefs implements one ICCL parameter (we cite the corresponding specification tables in the comments):

```
    //ElementType (see Table 108)
    typedef DSLFeatures::ElementType ElementType;

    //IndexType (see Table 108)
    typedef DSLFeatures::IndexType IndexType;

    //Vector (see Figure 184)
    typedef Dyn1DContainer<Generator> Vector;

    //Array (see Table 104)
    typedef IF<EQUAL<DSLFeatures::ArrOrder::id, DSLFeatures::ArrOrder::c_like_id>::RET,
          Dyn2DCContainer<Generator>,
          Dyn2DFContainer<Generator> >::RET Array;
```

Please note that we passed Generator to the basic containers in the above typedefs for Vector and Array. Here are the remaining ICCL parameters:

```
    //Format (see Table 105)
    typedef
    IF< (EQUAL<DSLFeatures::Shape::id, DSLFeatures::Shape::lower_triang_id>::RET ||
          EQUAL<DSLFeatures::Shape::id, DSLFeatures::Shape::symm_id>::RET) &&
          EQUAL<DSLFeatures::Format::id, DSLFeatures::Format::vector_id>::RET,
            LoTriangVecFormat<Vector>,

    IF< EQUAL<DSLFeatures::Shape::id, DSLFeatures::Shape::upper_triang_id>::RET &&
          EQUAL<DSLFeatures::Format::id, DSLFeatures::Format::vector_id>::RET,
            UpTriangVecFormat<Vector>,

    ArrFormat<Array> >::RET>::RET Format;

  //OptSymmetricMatrix (see Table 106)
  typedef IF<EQUAL<DSLFeatures::Shape::id, DSLFeatures::Shape::symm_id>::RET,
      Symm<Format>,
      Format>::RET OptSymmetricMatrix;

  //OptBoundsCheckedMatrix (see Table 107)
  typedef IF<EQUAL<DSLFeatures::BoundsChecking::id,
                        DSLFeatures::BoundsChecking::check_bounds_id>::RET,
      BoundsChecker<OptSymmetricMatrix>,
      OptSymmetricMatrix>::RET OptBoundsCheckedMatrix;

  //MatrixType;
  typedef Matrix<OptBoundsCheckedMatrix> MatrixType;

  //CommaInitializer (see Table 109)
  typedef

    IF<EQUAL<DSLFeatures::ArrOrder::id, DSLFeatures::ArrOrder::c_like_id>::RET,
      DenseCCommaInitializer<Generator>,
      DenseFCommaInitializer<Generator> >::RET CommaInitializer;
```

Finally, we return the Config and the generated matrix type. Since we passed Generator to the basic containers, they have access to Config. They also pass it to the components of the upper layers.

```
public:
  struct Config
  {
    //DSL features
    typedef DSLFeatures DSLFeatures;

    //ICCL features
    typedef ElementType ElementType;
    typedef IndexType IndexType;

    //MatrixType
    typedef MatrixType MatrixType;

    typedef CommaInitializer CommaInitializer;
  };

  typedef MatrixType RET; //here is our generated matrix type!!!
};
```

10.3.1.6 A More Intentional Alternative to Nested IFs

As you saw in the previous two sections, the implementation of assigning defaults and assembling components involves implementing the dependency tables for computing defaults and for computing ICCL parameters. The technique we used for implementing those tables were nested meta IFs. This technique was satisfactory for our demo matrix component, but the tables for the full matrix component specified in Section 10.2 are much bigger.

We demonstrate our point using the still relatively simple table for computing the ICCL parameter Array from Section 10.2.5.1 (i.e. Table 59):

Malloc	**ArrOrder**	**Array**
fix	cLike	Fix2DCContainer
	fortranLike	Fix2DFContainer
dyn	cLike	Dyn2DCContainer
	fortranLike	Dyn2DFContainer

The nested-meta-IF implementation of this table looks like this:

```
//Arr
  typedef IF<EQUAL<DSLFeatures::Malloc::id, DSLFeatures::Malloc::fix_id>::RET,
       IF<EQUAL<DSLFeatures::ArrOrder::id, DSLFeatures::ArrOrder::c_like_id>::RET,
         Fix2DCContainer<Size, Generator>,
       IF<EQUAL<DSLFeatures::ArrOrder::id, DSLFeatures::ArrOrder::fortran_like_id>::RET,
         Fix2DFContainer<Size, Generator>,
       invalid_ICCL_feature>::RET>::RET,

     IF<EQUAL<DSLFeatures::Malloc::id, DSLFeatures::Malloc::dyn_id>::RET,
       IF<EQUAL<DSLFeatures::ArrOrder::id, DSLFeatures::ArrOrder::c_like_id>::RET,
         Dyn2DCContainer<Generator>,
       IF<EQUAL<DSLFeatures::ArrOrder::id, DSLFeatures::ArrOrder::fortran_like_id>::RET,
         Dyn2DFContainer<Generator>,
       invalid_ICCL_feature>::RET>::RET,

     invalid_ICCL_feature>::RET>::RET Arr;
```

This is certainly not very readable. We can improve the situation by providing a metafunction allowing us to encode the tables more directly. This metafunction takes two parameters:

```
EVAL_DEPENDENCY_TABLE<HeadRow, TableBody>
```

HeadRow represents the head row of the dependency table (i.e. the first, gray row) and TableBody represents the remaining rows. Both parameters are implemented as lists (by nesting class templates). Here is an example:

```
typedef   EVAL_DEPENDENCY_TABLE

          <       CELL< 1,   CELL< 2                    > >

          , ROW< CELL< 4,   CELL< 3,   RET< Foo1   > > >
          , ROW< CELL< 1,   CELL< 5,   RET< Foo2   > > >
          , ROW< CELL< 1,   CELL< 2,   RET< Foo3   > > >
          , ROW< CELL< 2,   CELL< 3,   RET< Foo4   > > >
           > > > >  >::RET result;                         // result is Foo3
```

EVAL_DEPENDENCY_TABLE<> does the following: It looks for a row in TableBody whose cells match the cells of TableHead starting with the first row in TableBody. If it finds a matching row, it returns the corresponding return type (i.e. the type wrapped in RET<>; you can nest RETs in order to return more than one type, e.g. RET<Foo1, RET<Foo2> >). If it does not find a matching row, it reports an error. You can use anyValue as an asterisk (i.e. a value that matches anything).

Using this metafunction, we can rewrite our table as follows:

```
typedef DSLFeatures::Malloc     Malloc_;
typedef DSLFeatures::ArrOrder   ArrOrder_;
typedef invalid_ICCL_feature    invalid;

enum {
  mallocID     = Malloc_::id,
  dyn          = Malloc_::dyn_id,
  fix          = Malloc_::fix_id,

  arrOrdID     = ArrOrder_::id,
  cLike        = ArrOrder_::c_like_id,
  fortranLike  = ArrOrder_::fortran_like_id
}

// Arr
  typedef EVAL_DEPENDENCY_TABLE                       // tables 16, 20
  //****************************************************************************
  <         CELL< mallocID,  CELL< arrOrdID                                    > >

  , ROW< CELL< fix,          CELL< cLike,         RET< Fix2DCContainer< Size, Generator > > > >
  , ROW< CELL< fix,          CELL< fortranL,      RET< Fix2DFContainer< Size, Generator > > > >
  , ROW< CELL< dyn,            CELL< cLike,         RET< Dyn2DCContainer< Generator >     > > >
  , ROW< CELL< dyn,            CELL< fortranL,      RET< Dyn2DFContainer< Generator >     > > >
  , ROW< CELL< anyValue,     CELL< anyValue,      RET< invalid                          > > >
  //****************************************************************************
  > > > > > >::RET Arr;
```

The implementation of this function is given in Section 10.5.

Unfortunately, compared to the nested-IFs solution, the use of EVAL_DEPENDENCY_TABLE<> increases compilation times of the matrix component by an order of magnitude (EVAL_DEPENDENCY_TABLE<> uses recursion).

10.3.1.7 Matrix Operations

We implement matrix operations such as multiplication and addition using the technique of *expression templates* [Vel95]. We already explained the basic idea behind this implementation in Sections 10.2.6.1 and 10.2.6.2. We give you the C++ code in six parts:

1. *matrix operator templates*: operator templates implementing the operators + and * for matrices;

2. *matrix expression templates*: class templates for representing addition and multiplication expressions;

3. *matrix cache*: cache for implementing matrix multiplication (see the lazy-with-cache variant in Section 10.2.6.1);

4. getElement() *for expressions*: getElement() returns one element of a matrix expression; there will be different implementations for different shape combinations of the argument matrices;

5. *metafunctions for computing result types of expressions*: metafunctions for computing the matrix type of an expression, i.e. the type which is appropriate for storing the result of evaluating the expression;

6. *assignment functions for assigning expressions and matrices to matrices*: there will be different implementation of the assignment functions for different shapes of the source matrix (or expression).

We only consider the implementation of matrix addition and multiplication since matrix subtraction is similar to matrix addition.

10.3.1.7.1 Matrix Operator Templates

The main idea behind expression templates is the following: if you add two matrices, e.g. A+B, you do not return the result matrix, but an object representing the addition expression instead. If you have a more complicated expression, e.g. (A+B)*(C+D), you return a nested expression object (see Figure 177). This is done as follows: You first execute the two plus operators. They both return a matrix addition expressions. Finally, you execute the multiplication operator, which returns a multiplication expression pointing to the other two addition expressions as its argument objects. An expression object can be accessed using getElement() – just as any matrix.

Thus, in any case, you end up with an expression object. Since we implement the operators + and * using overloaded operator templates, we will know the complete type of a complex expression at compile time. The expression type describes the structure of the expression and we can pass it to metafunctions analyzing the expression structure and generating optimized code for the methods of the expression. However, the matrix expression optimization we implement later will involve the inspection of two expression argument types at a time rather than analyzing whole expression structures (the latter could be necessary for other kinds of optimizations). Depending on the shape combination of the arguments, we will select different implementations of the getElement() (i.e. the function for computing expression elements).

According to the approach outlined above, we need the following operator implementations:

- + for two matrices, e.g. A+B
- + for a matrix and an addition expression, e.g. A+(B+C)
- + for an addition expression and a matrix, e.g. (A+B)+C
- + for two addition expressions, e.g. (A+B)+(C+D)

Furthermore, we would need a similar set of four implementations of * and implementations for all the combinations of addition and multiplication expressions, e.g. (A+B)*C, (A+B)*(C+D), etc.

We can avoid this combinatorial explosion by wrapping the multiplication and the addition expressions into binary expressions. In this case, we only need four implementations of + and four implementations of *. The C++ implementation looks as follows (in MatrixOperTemplates.h):

Module: MatrixOperTemplates.h

```
/*** Addition **/

//Matrix + Matrix
template <class M1, class M2>
inline BinaryExpression<AdditionExpression<Matrix<M1>, Matrix<M2> > >
operator+(const Matrix<M1>& m1, const Matrix<M2>& m2)
{ return BinaryExpression<AdditionExpression<Matrix<M1>, Matrix<M2> > >(m1, m2);
}

//Expression + Matrix
template <class Expr, class M>
inline BinaryExpression<AdditionExpression<BinaryExpression<Expr>, Matrix<M> > >
operator+(const BinaryExpression<Expr>& expr, const Matrix<M>& m)
{ return BinaryExpression<AdditionExpression<BinaryExpression<Expr>, Matrix<M> > >(expr, m);
}

//Matrix + Expression
template <class M, class Expr>
inline BinaryExpression<AdditionExpression<Matrix<M>, BinaryExpression<Expr> > >
operator+(const Matrix<M>& m, const BinaryExpression<Expr>& expr)
{ return BinaryExpression<AdditionExpression<Matrix<M>, BinaryExpression<Expr> > >(m, expr);
}

//Expression + Expression
template <class Expr1, class Expr2>
```

```
inline BinaryExpression<AdditionExpression<BinaryExpression<Expr1>, BinaryExpression<Expr2> > >
operator+(const BinaryExpression<Expr1>& expr1, const BinaryExpression<Expr2>& expr2)
{ return BinaryExpression<AdditionExpression< BinaryExpression<Expr1>,
                                              BinaryExpression<Expr2> > >(expr1, expr2);
}


/*** Multiplication **/

//Matrix * Matrix
template <class M1, class M2>
inline BinaryExpression<MultiplicationExpression<Matrix<M1>, Matrix<M2> > >
operator*(const Matrix<M1>& m1, const Matrix<M2>& m2)
{ return BinaryExpression<MultiplicationExpression<Matrix<M1>, Matrix<M2> > >(m1, m2);
}

//Expression * Matrix
template <class Expr, class M>
inline BinaryExpression<MultiplicationExpression<BinaryExpression<Expr>, Matrix<M> > >
operator*(const BinaryExpression<Expr>& expr, const Matrix<M>& m)
{ return BinaryExpression<MultiplicationExpression<BinaryExpression<Expr>, Matrix<M> > >(expr, m);
}

//Matrix * Expression
template <class M, class Expr>
inline BinaryExpression<MultiplicationExpression<Matrix<M>, BinaryExpression<Expr> > >
operator*(const Matrix<M>& m, const BinaryExpression<Expr>& expr)
{ return BinaryExpression<MultiplicationExpression<Matrix<M>, BinaryExpression<Expr> > >(m, expr);
}

//Expression * Expression
template <class Expr1, class Expr2>
inline BinaryExpression<MultiplicationExpression<BinaryExpression<Expr1>, BinaryExpression<Expr2> >
>
operator*(const BinaryExpression<Expr1>& expr1, const BinaryExpression<Expr2>& expr2)
{ return BinaryExpression<MultiplicationExpression<  BinaryExpression<Expr1>,
                                                     BinaryExpression<Expr2> > >(expr1, expr2);
}
```

The operator templates can be thought of as a parsing facility. For example, given the above operator templates and the two matrices RectMatrix1 and RectMatrix2 of type

```
Matrix<BoundsChecker<ArrFormat<Dyn2DCContainer<MATRIX_ASSEMBLE_COMPONENTS<...>>>>>
```

the C++ compiler derives for the following expression

```
(RectMatrix1 + RectMatrix2)*(RectMatrix1 + RectMatrix2)
```

the following type:

```
BinaryExpression<
  MultiplicationExpression<
    BinaryExpression<
      AdditionExpression<
        Matrix<BoundsChecker<ArrFormat<Dyn2DCContainer<MATRIX_ASSEMBLE_COMPONENTS<...>>>>
      >,
        Matrix<BoundsChecker<ArrFormat<Dyn2DCContainer<MATRIX_ASSEMBLE_COMPONENTS<...>>>>
      >
      >
    >,
    BinaryExpression<
      AdditionExpression<
        Matrix<BoundsChecker<ArrFormat<Dyn2DCContainer<MATRIX_ASSEMBLE_COMPONENTS<...>>>>
      >,
        Matrix<BoundsChecker<ArrFormat<Dyn2DCContainer<MATRIX_ASSEMBLE_COMPONENTS<...>>>>
      >
      >
    >
  >
>
```

10.3.1.7.2 Matrix Expression Templates

The class template AdditionExpression<> represents the addition of two arguments, MultiplicationExpression<> represents the multiplication of two operands, and BinaryExpression<> is used to wrap the previous two to make them look alike.

We start with the implementation of AdditionExpression<> (in MatrixExprTemplates.h):

Module: MatrixExprTemplates.h

```
template<class A, class B>
class AdditionExpression
{ public:
    typedef A LeftType;
    typedef B RightType;
```

Any of LeftType and RightType can be either a matrix type or a binary expression type. Next, we need to compute the result matrix type of the addition expression, i.e. matrix type which would be appropriate for storing the result of the evaluation of this expression. We compute the result type using the metafunction ADD_RESULT_TYPE<>, which we discuss later. We publish the configuration repository of the result type in the member Config of AdditionExpression<>. Thus, an addition expression has a Config describing its matrix type – just as any matrix type does. Indeed, ADD_RESULT_TYPE<> uses the Config of the operands in order to compute the Config of the result:

```
    typedef ADD_RESULT_TYPE<LeftType, RightType>::RET::Config Config;
```

Next, we read out the element type and the index type for this expression from the resulting Config:

```
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;
```

The addition expression needs two variables to point to its operands:

```
  private:
    const LeftType& _left;
    const RightType& _right;

  protected:
    const IndexType rows_, cols_;
```

The constructor initializes the expression variables and checks if the dimensions of the operands are compatible:

```
  public:
    AdditionExpression(const LeftType& m1, const RightType& m2)
      : _left(m1), _right(m2),
      rows_(m1.rows()), cols_(m1.cols())
    { if (m1.cols() != m2.cols() || m1.rows() != m2.rows())
        throw "argument matrices are incompatible";
    }
```

The addition expression defines a getElement() method for accessing its matrix elements as any matrix does. However, in this case, each element is computed from the operands rather than stored directly. We use a metafunction to select the most appropriate implementation of getElement() based on the shape of the operands (we will discuss this function later):

```
    ElementType getElement( const IndexType & i, const IndexType & j ) const
    { return MATRIX_ADD_GET_ELEMENT<LeftType, RightType>::RET::getElement(i, j, this, _left, _right);
    }

    IndexType rows() const {return rows_;}
    IndexType cols() const {return cols_;}
};
```

The beginning of MultiplicationExpression looks similarly to AdditionExpression, except that we additionally read out the left and the right matrix type to be used for deriving operand caches later:

```
template<class A, class B>
class MultiplicationExpression
{ public:
    typedef A LeftType;
    typedef B RightType;
    typedef LeftType::Config::MatrixType LeftMatrixType;
    typedef RightType::Config::MatrixType RightMatrixType;

    typedef MULTIPLY_RESULT_TYPE<LeftType, RightType>::RET::Config Config;

    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;
```

As we explained in Section 10.2.6.1, in the case of matrix multiplication, we need to use caches for those operands which are expressions themselves. The reason was that matrix multiplication accesses each element of the operands more than one time and by using a cache we avoid the recalculation of the elements of the operand expression on each access. The type of the caches is computed from the corresponding operand types by a metafunction, which we discuss later. Finally, we provide variables for keeping track of the operands and the caches (if any):

```
  private:
    typedef CACHE_MATRIX_TYPE<LeftMatrixType>::RET LeftCacheMatrixType;
    typedef CACHE_MATRIX_TYPE<RightMatrixType>::RET RightCacheMatrixType;

    const LeftType& _left;
    const RightType& _right;

    LeftCacheMatrixType* _left_cache_matrix;
    RightCacheMatrixType* _right_cache_matrix;

  protected:
    const IndexType rows_, cols_;
```

The multiplication expression needs four constructors, each of them for one of the following combinations:

- both operands are simple matrices;
- left operand is a matrix and right operand is an expression;
- right operand is an expression and left operand is a matrix;
- both operands are expressions.

We start with two matrices. In this case we do not need any caches:

```
  public:
    template<class M1, class M2>
    MultiplicationExpression(const Matrix<M1>& m1, const Matrix<M2>& m2)
      : _left(m1), _right(m2),
      _left_cache_matrix(NULL), _right_cache_matrix(NULL),
      rows_(m1.rows()), cols_(m2.cols())
    { ParameterCheck(m1, m2);
    }
```

The following two constructors have to create a cache for one of the two operands:

```
    template<class Expr, class M2>
    MultiplicationExpression(const BinaryExpression<Expr>& expr, const Matrix<M2>& m)
      : _left(expr), _right(m),
      _right_cache_matrix(NULL),
      rows_(expr.rows()), cols_(m.cols())
    { ParameterCheck(expr, m);
      _left_cache_matrix = new LeftCacheMatrixType(expr.rows(), expr.cols());
    }

    template<class M, class Expr>
    MultiplicationExpression(const Matrix<M>& m, const BinaryExpression<Expr>& expr)
      :_left(m),  _right(expr),
      _left_cache_matrix(NULL),
      rows_(m.rows()), cols_(expr.cols())
    { ParameterCheck(m, expr);
      _right_cache_matrix = new RightCacheMatrixType(expr.rows(), expr.cols());
    }
```

Finally, the fourth constructor creates two caches, each one for one of its operands:

```
    template<class Expr1, class Expr2>
    MultiplicationExpression(const BinaryExpression<Expr1>& expr1, const BinaryExpression<Expr2>&
expr2)
      :_left(expr1),   _right(expr2),
      rows_(expr1.rows()), cols_(expr2.cols())
    { ParameterCheck(expr1, expr2);
      _left_cache_matrix = new LeftCacheMatrixType(expr1.rows(), expr1.cols());
      _right_cache_matrix = new RightCacheMatrixType(expr2.rows(), expr2.cols());
    }
```

Since the expressions are returned by the operator templates by copy, we need to implement a copy constructor for the multiplication expression. When the expression is copied, the cache variables of the new copy will point to the caches of the old expression. Thus, we need to reset the cache variables in the old expression to NULL, so that its destructor does not destroy the caches:

```
    MultiplicationExpression(MultiplicationExpression& old)
      : _left(old._left), _right(old._right),
      _left_cache_matrix(old._left_cache_matrix),
      _right_cache_matrix(old._right_cache_matrix),
      rows_(old.rows()), cols_(old.cols())
    { old._left_cache_matrix= NULL;
      old._right_cache_matrix= NULL;
    }
```

The destructor deletes the caches, if any:

```
    ~MultiplicationExpression()
    {
      delete _left_cache_matrix;
      delete _right_cache_matrix;
    }
```

Finally, we have the getElement() function which also uses a metafunction to select the most appropriate implementation based on the shape of the operands:

```
    ElementType getElement(const IndexType & i, const IndexType & j) const
    { return MATRIX_MULTIPLY_GET_ELEMENT<LeftType, RightType>::RET::getElement(i, j,
        this, _left, _right, _left_cache_matrix, _right_cache_matrix);
    }

    IndexType rows() const {return rows_;}
    IndexType cols() const {return cols_;}

  private:
    void ParameterCheck(const A& m1, const B& m2)
    { if (m1.cols() != m2.rows())
        throw "argument matrices are incompatible";
    }
```

```
};
```

The last class template is BinaryExpression<>. It is derived from its parameter, i.e. ExpressionTemplate. Thus, it inherits Config and the getElement() method from the expression it wraps.

```
template<class ExpressionType>
class BinaryExpression : public ExpressionType
{ public:
    typedef ExpressionType::LeftType LeftType;
    typedef ExpressionType::RightType RightType;
    typedef ExpressionType::Config::MatrixType MatrixType;
    typedef ExpressionType::IndexType IndexType;

    BinaryExpression(const LeftType& __op1, const RightType& __op2)
      : ExpressionType(__op1, __op2)
    {}
```

The following method implements assignment and is called from the assignment operator implementation in Matrix<> (Section 10.3.1.4.4):

```
    template<class Res>
    Matrix<Res>* assign(Matrix<Res>* const result) const
    { MATRIX_ASSIGNMENT<MatrixType>::RET::assign(result, this);
      return result;
    }

    ostream& display(ostream& out) const
    { IndexType r= rows(), c= cols();
      for( IndexType i = 0; i < r; ++i )
      { for( IndexType j = 0; j < c; ++j )
           out << getElement( i, j ) << "   ";
        out << endl;
      }
      return out;
    }
};

template <class Expr>
ostream& operator<<(ostream& out, const BinaryExpression<Expr>& expr)
{ return expr.display(out);
}
```

10.3.1.7.3 Matrix Cache

As stated, matrix multiplication uses a cache to avoid recomputing the elements of an operand expression. We implement the cache as a matrix whose elements are cache elements rather than numbers. A cache element has a variable for storing the cached element value and a flag indicating if the value is in cache or not (in MatrixCache.h):

Module: MatrixCache.h

```
template<class ElementType>
struct CacheElementType
{ bool valid;     //if true, the value is already cached (cache-hit); if false, it isn't
  ElementType element;

  CacheElementType() : element(ElementType(0)), valid(false) {}

  CacheElementType(const ElementType& elem)
    : element(elem), valid(false) {}

  bool operator==(const CacheElementType& scnd) const
  {return (valid == scnd.valid && element == scnd.element);}

  bool operator!=(const CacheElementType& scnd) const
  {return (valid != scnd.valid || element != scnd.element);}
  ostream& display(ostream& out) const
  { out << "(" << element << "; " << valid << ")";
    return out;
  }
};

template <class A>
ostream& operator<<(ostream& out, const CacheElementType<A>& elem)
{ return elem.display(out);
}
```

Next, we implement a metafunction which takes a matrix type and returns the corresponding matrix cache type. The only difference between these two types is the element type: the element type of the cache is CacheElementType<> parameterized with the element type of the original matrix type. The cache type derivation involves reading out the description of the matrix type, i.e. DSLFeatures, and deriving a new type from this description, which overrides the inherited element type with the new cache element type, and finally passing DSLFeatures to the matrix generator. In order to be able to do the derivation, we need a little workaround since it is not possible to derive a class from a typename defined by a typedef:

```
struct DerivedDSLFeatures : public DSLFeatures
{};
```

And here is the metafunction:

```
template<class MatrixType>
struct CACHE_MATRIX_TYPE
{ private:
    typedef MatrixType::Config Config;
    typedef Config::DSLFeatures DSLFeatures;

  public:
    //override ElementType:
    struct CachedMatrixDSL : public DerivedDSLFeatures<DSLFeatures>
    { typedef CacheElementType<DSLFeatures::ElementType> ElementType;
    };

    typedef MATRIX_GENERATOR<CachedMatrixDSL, assemble_components>::RET RET;
};
```

10.3.1.7.4 *Implementation of* ***getElement()***

The implementation of getElement() for the addition expression and for the multiplication expression depends on the shape of the operands. Therefore, we will use the same technique as in the case of nonZeroRegion() of ArrFormat in Section 10.3.1.4.2: we implement the method variants as static methods of separate structs and select the structs using metafunctions.

In the case of addition, we provide three algorithms: one general for adding rectangular matrices (which also works in all other cases), one for adding lower triangular matrices, and one for adding upper triangular matrices (in GetElement.h):

Module: GetElement.h

```
struct RectAddGetElement
{ template<class IndexType, class ResultType, class LeftType, class RightType>
  static ResultType::ElementType
  getElement(const IndexType& i, const IndexType& j,
    const ResultType* res, const LeftType& left, const RightType& right)
  {
    return left.getElement(i, j) + right.getElement(i, j);
  }
};

struct LowerTriangAddGetElement
{
  template<class IndexType, class ResultType, class LeftType, class RightType>
  static ResultType::ElementType
  getElement(const IndexType& i, const IndexType& j,
    const ResultType* res, const LeftType& left, const RightType& right)
  {
    return i >= j ? left.getElement(i, j) + right.getElement(i, j)
            : ResultType::ElementType(0);
  }
};

struct UpperTriangAddGetElement
{
  template<class IndexType, class ResultType, class LeftType, class RightType>
  static ResultType::ElementType
  getElement(const IndexType& i, const IndexType& j,
    const ResultType* res, const LeftType& left, const RightType& right)
  {
    return i <= j ? left.getElement(i, j) + right.getElement(i, j)
            : ResultType::ElementType(0);
  }
};
```

The following metafunction selects the appropriate algorithm: it takes LowerTriangAddGetElement for two lower triangular matrices, UpperTriangAddGetElement for two upper triangular matrices, and RectAddGetElement for all the other combinations:

```
template<class Matrix1, class Matrix2>
struct MATRIX_ADD_GET_ELEMENT
{
  typedef Matrix1::Config::DSLFeatures::Shape Shape1;
  typedef Matrix2::Config::DSLFeatures::Shape Shape2;

  typedef IF< EQUAL<Shape1::id, Shape1::lower_triang_id>::RET &&
         EQUAL<Shape2::id, Shape2::lower_triang_id>::RET,
           LowerTriangAddGetElement,

       IF< EQUAL<Shape1::id, Shape1::upper_triang_id>::RET &&
         EQUAL<Shape2::id, Shape2::upper_triang_id>::RET,
           UpperTriangAddGetElement,

       RectAddGetElement>::RET>::RET RET;
};
```

The following is the getElement() for the multiplication of two rectangular matrices:

```
struct RectMultiplyGetElement
{
  template<class _IndexType,
    class ResultType, class LeftType, class RightType,
    class LeftCacheType, class RightCacheType>
  static ResultType::ElementType getElement(const _IndexType& i, const _IndexType& j,
    const ResultType* res, const LeftType& left, const RightType& right,
    LeftCacheType* left_cache= NULL, RightCacheType* right_cache= NULL)
  {
    typedef ResultType::Config Config;
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;

    ElementType result= ElementType(0);

    for(IndexType k= left.cols(); k--;)
      result+= getCachedElement(i, k, left, left_cache) * getCachedElement(k, j, right, right_cache);
    return result;
  }

  private:
    template<class IndexType, class MatrixType, class CacheType>
    static MatrixType::ElementType
    getCachedElement(const IndexType& i, const IndexType& j,
      const MatrixType& matrix, CacheType* cache)
    {
      if (cache == NULL) return matrix.getElement(i, j);
      else
      {
        CacheType::ElementType tmpCacheElem= cache->getElement(i, j);
        if (!tmpCacheElem.valid)
        {
          tmpCacheElem.element= matrix.getElement(i, j);
          tmpCacheElem.valid= true;
          cache->setElement(i, j, tmpCacheElem);
        }

        return tmpCacheElem.element;
      }
    }
};
```

The variant of getElement() for multiplying two lower triangular matrices and the other variant for two upper triangular matrices are analogous and not shown here.

Here is the metafunction for selecting the algorithms:

```
/****************************** Selecting algorithms for multiplication ******************************/
template<class Matrix1, class Matrix2>
struct MATRIX_MULTIPLY_GET_ELEMENT
{
  typedef Matrix1::Config::DSLFeatures::Shape Shape1;
  typedef Matrix2::Config::DSLFeatures::Shape Shape2;

  typedef IF< EQUAL<Shape1::id, Shape1::lower_triang_id>::RET &&
        EQUAL<Shape2::id, Shape2::lower_triang_id>::RET,
          LowerTriangMultiplyGetElement,

      IF< EQUAL<Shape1::id, Shape1::upper_triang_id>::RET &&
        EQUAL<Shape2::id, Shape2::upper_triang_id>::RET,
          UpperTriangMultiplyGetElement,

      RectMultiplyGetElement>::RET>::RET RET;
};
```

10.3.1.7.5 Metafunctions for Computing Result Types of Expressions

As you remember, the addition expression and multiplication expression class templates call the metafunctions for computing the matrix result type of addition and multiplication:

```
ADD_RESULT_TYPE<class class MatrixType1, class MatrixType2>
MULTIPLY_RESULT_TYPE<class MatrixType1, class MatrixType2>
```

The metafunctions work as follows: They take the DSLFeatures from each of the operands, compute the result DSLFeatures and call the matrix generator with the result DSLFeatures to generate the result matrix type.

We first need to specify how to compute the result DSLFeatures from the argument DSLFeatures. As you remember, DSLFeatures contains the following features:

```
ElementType
Shape
Format
ArrayOrder
OptFlag
BoundsChecking
IndexType
```

Thus, we need to compute these features for the result matrix from the features of the argument matrices. The following features are computed in the same way for both addition and multiplication:

ElementType
Format
ArrayOrder
OptFlag
BoundsChecking
IndexType

ElementType and IndexType are computed using the numeric promote metafunction shown in Figure 178. The table for computing Format is given later. The other features are computed as follows: If the value of the given feature in one argument matrix is equal to the value of the same feature in the other argument matrix, the resulting feature value is equal to the other two values. If this is not the case, the resulting feature value is unspecified_DSL_feature (see Table 110). This is useful, since we can let the matrix generator assign the default value for the unspecified features.

Feature1	Feature2	Result
(value)	(value)	=(value)
*	*	unspecified_DSL_feature

Table 110 *General formula for computing result values of nonmathematical DSL features*

The resulting shape is computed differently for addition and for multiplication. This is shown in Table 111 and Table 112.

Shape1	Shape2	Shape Result
symm	symm	symm
lowerTriang	lowerTriang	lowerTriang
upperTriang	upperTriang	upperTriang
*	*	rect

Table 111 *Computing the resulting shape for matrix addition*

Shape1	Shape2	Shape Result
lowerTriang	lowerTriang	lowerTriang
upperTriang	upperTriang	upperTriang
*	*	rect

Table 112 *Computing the resulting shape for matrix multiplication*

Format depends not only on the Format of the operands, but also on the shape of the result. In particular, adding a lower-triangular matrix to an upper-triangular one yields a rectangular matrix. If the format of both operands is vector, we cannot simply assume vector for the result since a rectangular matrix is better stored in an array. This is specified in Table 113.

Shape Result	Format1	Format2	Format Result
rect	vector	vector	unspecified_DSL_feature
*	(value)	(value)	=(value)
*	*	*	unspecified_DSL_feature

Table 113 *Computing the resulting format*

We start with a metafunction implementing the general formula from Table 110 (in ComputeResultType.h):

Module: ComputeResultType.h

```
template<class Feature1, class Feature2>
struct DEFAULT_RESULT
{ typedef
    IF< EQUAL<Feature1::id, Feature2::id>::RET,
          Feature1,
          unspecified_DSL_feature>::RET RET;
};
```

RESULT_FORMAT<> implements Table 113:

```
template<class Shape, class Format1, class Format2>
struct RESULT_FORMAT
{
  typedef
    IF< EQUAL<Shape::id, Shape::rect_id>::RET &&
          EQUAL<Format1::id, Format1::vector_id>::RET &&
          EQUAL<Format2::id, Format2::vector_id>::RET,
          unspecified_DSL_feature,

          IF< EQUAL<Format1::id, Format2::id>::RET,
          Format1,

    unspecified_DSL_feature>::RET>::RET RET;
};
```

The following metafunction computes result shape for matrix addition (Table 111):

```

//compute result shape for addition (
template<class Shape1, class Shape2>
struct ADD_RESULT_SHAPE
{

  typedef
    IF< EQUAL<Shape1::id, Shape1::symm_id>::RET &&
          EQUAL<Shape2::id, Shape2::symm_id>::RET,
          symm<>,

    IF< EQUAL<Shape1::id, Shape1::lower_triang_id>::RET &&
          EQUAL<Shape2::id, Shape2::lower_triang_id>::RET,
          lower_triang<>,

    IF< EQUAL<Shape1::id, Shape1::upper_triang_id>::RET &&
          EQUAL<Shape2::id, Shape2::upper_triang_id>::RET,
          upper_triang<>,

    rect<> >::RET>::RET>::RET RET;

};
```

ADD_RESULT_DSL_FEATURES<> computes result DSLFeatures from two argument DSLFeatures. We refer to the resulting DSLFeatures as ParsedDSL since it may contain some unspecified features (thus, it has the same form as the DSLFeatures returned by the DSLParser).

```

template<class DSLFeatures1, class DSLFeatures2>
class ADD_RESULT_DSL_FEATURES
{
  private:
    //ElementType (PROMOTE_NUMERIC_TYPE<> is shown in Figure 178)
    typedef PROMOTE_NUMERIC_TYPE<DSLFeatures1::ElementType, DSLFeatures2::ElementType>::RET
        ElementType;

    //IndexType
    typedef PROMOTE_NUMERIC_TYPE<DSLFeatures1::IndexType, DSLFeatures2::IndexType>::RET
        IndexType;

    //Shape
    typedef ADD_RESULT_SHAPE<DSLFeatures1::Shape, DSLFeatures2::Shape>::RET Shape;

    //OptFlag
    typedef DEFAULT_RESULT<DSLFeatures1::OptFlag, DSLFeatures2::OptFlag>::RET OptFlag;

    //BoundsChecking
    typedef DEFAULT_RESULT<DSLFeatures1::BoundsChecking, DSLFeatures2::BoundsChecking>::RET
        BoundsChecking;

    //Format
    typedef RESULT_FORMAT<Shape, DSLFeatures1::Format, DSLFeatures2::Format>::RET Format;

    //ArrOrder
    typedef DEFAULT_RESULT<DSLFeatures1::ArrOrder, DSLFeatures2::ArrOrder>::RET ArrOrder;

  public:
    struct ParsedDSL
    {
      typedef ElementType ElementType;
      typedef Shape Shape;
      typedef Format Format;
      typedef ArrOrder ArrOrder;
      typedef OptFlag OptFlag;
      typedef BoundsChecking BoundsChecking;
      typedef IndexType IndexType;
    };

    typedef ParsedDSL RET;
};
```

ADD_RESULT_TYPE<> returns the result matrix type for addition. It calls the above metafunction in order to compute the resulting DSL features and the matrix generator to generate the matrix type:

```
template<class MatrixType1, class MatrixType2>
struct ADD_RESULT_TYPE
{
  typedef ADD_RESULT_DSL_FEATURES <  MatrixType1::Config::DSLFeatures,
                                      MatrixType2::Config::DSLFeatures> BaseClass;
  typedef MATRIX_GENERATOR<BaseClass::ParsedDSL, defaults_and_assemble>::RET RET;
};
```

The code for computing the result type of multiplication is similar:

```
//this function implements Table 112
template<class Shape1, class Shape2>
struct MULTIPLY_RESULT_SHAPE
{
  typedef

    IF< EQUAL<Shape1::id, Shape1::lower_triang_id>::RET &&
      EQUAL<Shape2::id, Shape2::lower_triang_id>::RET,
      lower_triang<>,

    IF< EQUAL<Shape1::id, Shape1::upper_triang_id>::RET &&
      EQUAL<Shape2::id, Shape2::upper_triang_id>::RET,
      upper_triang<>,

    rect<> >::RET>::RET RET;
};

template<class DSLFeatures1, class DSLFeatures2>
struct MULTIPLY_RESULT_DSL_FEATURES
{
  private:
    //ElementType
    typedef PROMOTE_NUMERIC_TYPE<DSLFeatures1::ElementType, DSLFeatures2::ElementType>::RET
      ElementType;
    //IndexType
    typedef PROMOTE_NUMERIC_TYPE<DSLFeatures1::IndexType, DSLFeatures2::IndexType>::RET
      IndexType;

    //Shape
    typedef MULTIPLY_RESULT_SHAPE<DSLFeatures1::Shape, DSLFeatures2::Shape>::RET Shape;

    //OptFlag
    typedef DEFAULT_RESULT<DSLFeatures1::OptFlag, DSLFeatures2::OptFlag>::RET OptFlag;

    //BoundsChecking
    typedef DEFAULT_RESULT<DSLFeatures1::BoundsChecking, DSLFeatures2::BoundsChecking>::RET
      BoundsChecking;

    //Format
    typedef RESULT_FORMAT<Shape, DSLFeatures1::Format, DSLFeatures2::Format>::RET Format;

    //ArrOrder
    typedef DEFAULT_RESULT<DSLFeatures1::ArrOrder, DSLFeatures2::ArrOrder>::RET ArrOrder;

  public:
    struct ParsedDSL
    {
      typedef ElementType ElementType;
      typedef Shape Shape;
      typedef Format Format;
      typedef ArrOrder ArrOrder;
      typedef OptFlag OptFlag;
      typedef BoundsChecking BoundsChecking;
      typedef IndexType IndexType;
    };

    typedef ParsedDSL RET;
};

template<class MatrixType1, class MatrixType2>
struct MULTIPLY_RESULT_TYPE
{
  typedef MULTIPLY_RESULT_DSL_FEATURES < MatrixType1::Config::DSLFeatures,
                                          MatrixType2::Config::DSLFeatures> BaseClass;
  typedef MATRIX_GENERATOR<BaseClass::ParsedDSL, defaults_and_assemble>::RET RET;
};
```

10.3.1.7.6 Matrix Assignment

The implementation of matrix assignment depends on the shape of the source matrix (or source expression). In general, our assignment implementations perform two steps: initializing the target matrix with zero elements and assigning the nonzero elements from the source matrix (thus, we have to iterate over the nonzero region of the source matrix). We provide an assignment implementation for assigning a rectangular matrix, lower triangular, and upper triangular (the symmetric case is covered by the rectangular case; see Table 114).

Shape (source matrix)	**Assignment**
lower_triang	LowerTriangAssignment
upper_triang	UpperTriangAssignment
*	RectAssignment

Table 114 *Selecting the assignment algorithm*

Here is the C++ code for the assignment variants (in Assignment.h):

Module: Assignment.h

```
struct RectAssignment
{
  template<class Res, class M>
  static void assign(Res* res, M* m)
  {
    typedef Res::Config::IndexType IndexType;

    for (IndexType i= m->rows(); i--;)
      for (IndexType j= m->cols(); j--;)
        res->setElement(i, j, m->getElement(i, j));
  }
};

struct LowerTriangAssignment
{
  template<class Res, class M>
  static void assign(Res* res, M* m)
  {
    typedef Res::Config::IndexType IndexType;

    for(IndexType i= 0; i< res->rows(); ++i)
      for(IndexType j= 0; j<=i; ++j)
        res->setElement(i, j, m->getElement(i, j));
  }
};

struct UpperTriangAssignment
{
  template<class Res, class M>
  static void assign(Res* res, M* m)
  {
    typedef Res::Config::IndexType IndexType;

    for(IndexType i= 0; i< res->rows(); ++i)
      for(IndexType j= i; j< res->rows(); ++j)
        res->setElement(i, j, m->getElement(i, j));
  }
};
```

The following metafunction implements Table 114:

```
template<class RightMatrixType>
struct MATRIX_ASSIGNMENT
{ typedef RightMatrixType::Config::DSLFeatures::Shape Shape;

  typedef
      IF<EQUAL<Shape::id, Shape::lower_triang_id>::RET,
        LowerTriangAssignment,

      IF< EQUAL<Shape::id, Shape::upper_triang_id>::RET,
        UpperTriangAssignment,

      RectAssignment>::RET>::RET RET;
};
```

This concludes the implementation of the demo matrix component.

10.3.1.8 Full Implementation of the Matrix Component

The full implementation of the matrix component specified in Section 10.2 comprises 7500 lines of C++ code (6000 lines for the configuration generator and the matrix components and 1500 lines for the operations).[158] The full implementation illustrated a number of important points:

- *Separation between problem and solution space*: The use of a configuration DSL implemented by a separate set of templates hides the internal library architecture (i.e. the ICCL) from the application programmer. It is possible to change the ICCL (e.g. add new components or change the structure of the ICCL) without having to modify the existing client code. All we have to do is to map the existing DSL onto the new ICCL structure, which requires changes in the configuration generator. To a certain extent, we can even extend the configuration DSL without invalidating the existing client code. The kind of possible changes include

- enlarging the value scope of existing DSL parameters (e.g. adding new matrix shapes)
- appending new parameters to existing parameter lists (e.g. we could add new parameters at the end of the matrix parameter list and the structure parameter list; furthermore, we could also add new subfeatures to any of the leave nodes of the DSL feature diagram since all DSL features are implemented as templates).

In fact, during the performance optimization phase in the development of the matrix component, we had to modify the structure of the ICCL by splitting some components and merging others. Since the functionality scope did not change, we did not have to modify the DSL at all. If we used a library design without a configuration DSL (e.g. as in the STL), we would have to modify the existing application using the library since the applications would hardwire ICCL expressions in their code. Thus, the separation into the configuration DSL and the ICCL supports software evolution.

- *More declarative specification*: The client code can request a matrix using a configuration DSL expression which specifies exactly as much detail as the client wishes to specify. Since the configuration DSL tries to derive reasonable feature defaults from the explicitly specified features, the client code does not have to specify the details that do not concern it directly. Consider the following situation as an analogy: When you buy a car, you do not have to specify all the bolts and wires. There is a complex machinery between you and the car factory to figure out how to satisfy your abstract needs. On the other hand, if you need the car for a race, you may actually want to request some specially-customized parts (provided you have the technical knowledge to do so). The same applies to a configuration DSL: You should be able to specify any detail about the ICCL expression you want. Being forced to specify too much detail (this is the case if you do not have a configuration DSL), makes you dependent on the implementation of the library. If the client code, on the other hand, cannot specify all the details provided by the ICCL, it may well happen that the code will run slower compared to what is possible giving its context knowledge. For example, if you know that the shape of your matrix will not exceed lower triangular shape during the execution, you can specify this property at compile time. If the matrix configuration DSL does not let you do this, the generated component will certainly not run the more efficient lower-triangular algorithms on the matrix data.

- *Minimal redundancy within the library*: Parameterizing out differences allows you to avoid situations where two components contain largely the same code except for a few details that are different. Furthermore, there is a good chance that the “little” components implementing these details can be reused as parameters of more than one parameterized component. Thanks to an aggressive parameterization, we were able to cover the functionality of the matrix component specified in Section 10.2 with only 7500 lines of C++ code. The use of a configuration DSL enables us to hide some of the fragmentation caused by this parameterization from the user.

- *Coverage of a large number of variants*: The matrix configuration DSL covers some 1840 different kinds of matrices. This number does not take element type, index type, and all the number-valued parameters such as number of rows or the static scalar value into account. Given the 7500 lines of code implementing the matrix component, the average number of lines of code per matrix variant is four. If you count the nine different element types and the nine different index types, the number of matrix variants increases to 149 040. The different possible values for extension, diagonal range, scalar value, etc. (they all can be specified statically or dynamically) increase this number even more.

- *Very good performance*: Despite the large number of provided matrix variants, the performance of the generated code is comparable with the performance of manually coded variants. This is achieved by the exclusive use of static binding, which is often combined with inlining. We did not implement any special matrix optimizations such as register blocking or cache blocking. However, the work in [SL98b] demonstrates that by implementing blocking optimizations using template metaprogramming, it is possible to parallel the performance of highly tuned Fortran 90 matrix libraries.

Unfortunately, the presented C++ techniques also have a number of problems:

- *Debugging*: Debugging template metaprograms is hard. There is no such thing as a debugger for the C++ compilation process. Over time, we had to develop a number of tricks and strategies for getting the information we need (e.g. requesting a nonexistent member of a type in order to force the compiler to print out the contents of a typename in an error report; this corresponds to inspecting the value of a variable in a regular program). Unfortunately, they are not always effective. For example, many compilers would not show you the entire type in the error report if the name exceeds a certain number of characters (e.g. 255). The latter situation is very common in template metaprogramming, where template nesting depths may quickly reach very large numbers (see Figure 186).

- *Error reporting*: There is no way for a template metaprogram to output a string during compilation. On the other hand, template metaprograms are used for structure checking (e.g. parsing the configuration DSL) and other compilation tasks, and we need to be able to report problems about the data the metaprogram works on. In Section 10.3.1.5.2, we saw just a partial solution. This solution is unsatisfactory since there is no way to specify the place where the logical error occurred. In order to do this, we would need access to the internal parse tree of the compiler.

- *Readability of the code*: The readability of template metacode is not very high. We were able to improve on this point by providing explicit control structures (see Section 8.6) and using specialized metafunctions such as the table evaluation metafunction in Section 10.3.1.6. Despite all of this, the code remains still quite peculiar and obscure. Template metaprogramming is not a result of a well-thought out metalanguage design, but rather an accident.

- *Compilation times*: Template metaprograms may extend compilation times by orders of magnitude. The compilation time of particular metacode depends on its complexity and the programming style. However, in any case, the conclusion is that template metaprogramming greatly extends compilation times. There are at least two reasons for this situation:

  - template metacode is interpreted rather than compiled,

  - C++ compilers are not tuned for this kind of use (or abuse).

- *Compiler limits*: Since, in template metaprogramming, the computation is done quasi “as a byproduct” of type construction and inference, complex computations quickly lead to very complex types. The complexity and size limits of different compilers are different, but, in general, the limits are quickly reached. Thus, the complexity of the computations is also limited (e.g. certain loops cannot iterate more than some limited number of times).

- *Portability*: Template metaprogramming is based on many advanced C++ language features, which are not (yet) widely supported by many compilers. There are even differences in how some of these features are supported by a given compiler. Thus, currently, template metaprograms have a very limited portability. This situation will hopefully change over the next few years when more and more compiler vendors start to support the newly completed ISO C++ standard.

In general, we conclude that the complexity of template metaprograms is limited by compiler limits, compilation times, and debugging problems.

```
MultiplicationExpression<class LazyBinaryExpression<class AdditionExpression<class
MatrixICCL::Matrix<class MatrixICCL::BoundsChecker<class MatrixICCL::ArrFormat<class
MatrixICCL::StatExt<struct MatrixDSL::int_number<int,7>,struct MatrixDSL::int_number<int,7>>,class
MatrixICCL::Rect<class MatrixICCL::StatExt<struct MatrixDSL::int_number<int,7>,struct
MatrixDSL::int_number<int,7>>>,class MatrixICCL::Dyn2DCContainer<class
MATRIX_ASSEMBLE_COMPONENTS<class MATRIX_DSL_ASSIGN_DEFAULTS<class
MATRIX_DSL_PARSER<struct MatrixDSL::matrix<int,struct MatrixDSL::structure<struct
MatrixDSL::rect<struct MatrixDSL::stat_val<struct MatrixDSL::int_number<int,7>>,struct
MatrixDSL::stat_val<struct MatrixDSL::int_number<int,7>>,struct
MatrixDSL::unspecified_DSL_feature>,struct MatrixDSL::dense<struct
MatrixDSL::unspecified_DSL_feature>,struct MatrixDSL::dyn<struct
MatrixDSL::unspecified_DSL_feature>>,struct MatrixDSL::speed<struct
MatrixDSL::unspecified_DSL_feature>,struct MatrixDSL::unspecified_DSL_feature,struct
MatrixDSL::unspecified_DSL_feature,struct MatrixDSL::unspecified_DSL_feature,struct
MatrixDSL::unspecified_DSL_feature>>::DSLConfig>::DSLConfig>>>>>,class MatrixICCL::Matrix<class
MatrixICCL::BoundsChecker<class MatrixICCL::ArrFormat<class MatrixICCL::StatExt<struct
MatrixDSL::int_number<int,7>,struct MatrixDSL::int_number<int,7>>,class MatrixICCL::Rect<class
MatrixICCL::StatExt<struct MatrixDSL::int_number<int,7>,struct MatrixDSL::int_number<int,7>>>,class
MatrixICCL::Dyn2DCContainer<class MATRIX_ASSEMBLE_COMPONENTS<class
MATRIX_DSL_ASSIGN_DEFAULTS<class MATRIX_DSL_PARSER<struct MatrixDSL::matrix<int,struct
MatrixDSL::structure<struct MatrixDSL::rect<struct MatrixDSL::stat_val<struct
MatrixDSL::int_number<int,7>>,struct MatrixDSL::stat_val<struct MatrixDSL::int_number<int,7>>,struct
MatrixDSL::unspecified_DSL_feature>,struct MatrixDSL::dense<struct
MatrixDSL::unspecified_DSL_feature>,struct Ma...
```

Figure 186 *Fraction of the type generated by the C++ compiler for the matrix expression (A+B)*C*

10.3.2 Implementing the Matrix Component in IP

This section gives a short overview of a prototype implementation of the matrix component in the Intentional Programming System (see Section 6.4.3).[159] The prototype covers only a small subset of the functionality specified in Section 10.2. Its scope is comparable to the scope of the demo matrix component (Sections 10.3.1.2 through 10.3.1.7). More precisely, the prototype covers the following matrix parameters: element type, shape (rectangular, diagonal, lower triangular, upper triangular, symmetric, and identity), format (array and vector), and dynamic or static row and column numbers.

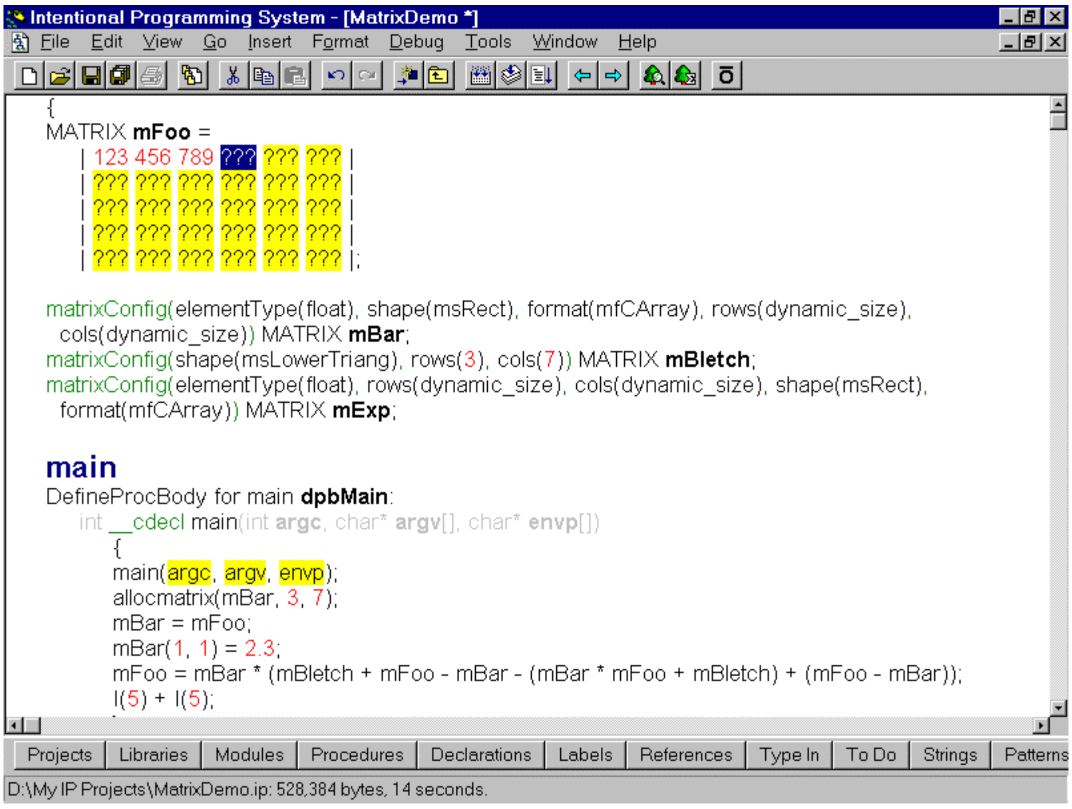
The implementation consists of two IP files:

- interface file containing the declaration of the matrix intentions (e.g. matrix type, matrix operations, configuration DSL parameters and values) and

- implementation file containing implementation modules with rendering, editing, and transforming methods.

The implementation file is compiled into an extension DLL. The interface file and the DLL are given to the application programmer who wants to write some matrix code.

Figure 187 shows some matrix application code written using the intentions defined in our prototype matrix library. The code displayed in the editor is rendered by the rendering methods contained in the extension DLL. The DLL also provides editing methods, which, for example, allow us to tab through the elements of the matrix literal used to initialize the matrix variable mFoo (i.e. it behaves like a spreadsheet).


Intentional Programming System - [MatrixDemo *]
File Edit View Go Insert Format Debug Tools Window Help
Projects Libraries Modules Procedures Declarations Labels References Type In To Do Strings Patterns

```
{
MATRIX mFoo =
    | 123 456 789 ??? ??? ??? |
    | ??? ??? ??? ??? ??? ??? |
    | ??? ??? ??? ??? ??? ??? |
    | ??? ??? ??? ??? ??? ??? |
    | ??? ??? ??? ??? ??? ??? |;

matrixConfig(elementType(float), shape(msRect), format(mfCArray), rows(dynamic_size),
  cols(dynamic_size)) MATRIX mBar;
matrixConfig(shape(msLowerTriang), rows(3), cols(7)) MATRIX mBletch;
matrixConfig(elementType(float), rows(dynamic_size), cols(dynamic_size), shape(msRect),
  format(mfCArray)) MATRIX mExp;

main
DefineProcBody for main dpbMain:
    int __cdecl main(int argc, char* argv[], char* envp[])
        {
        main(argc, argv, envp);
        allocmatrix(mBar, 3, 7);
        mBar = mFoo;
        mBar(1, 1) = 2.3;
        mFoo = mBar * (mBletch + mFoo - mBar - (mBar * mFoo + mBletch) + (mFoo - mBar));
        I(5) + I(5);
```

D:\My IP Projects\MatrixDemo.ip: 528,384 bytes, 14 seconds.

Figure 187 *Sample matrix program using the IP implementation of the matrix component*

In order to appreciate the effect of the displaying methods, Figure 188 demonstrates how the same matrix code looks like when the extension library is not loaded. In this case, the source tree is displayed using the default rendering methods, which are provided by the system. These methods render the source tree in a functional style.

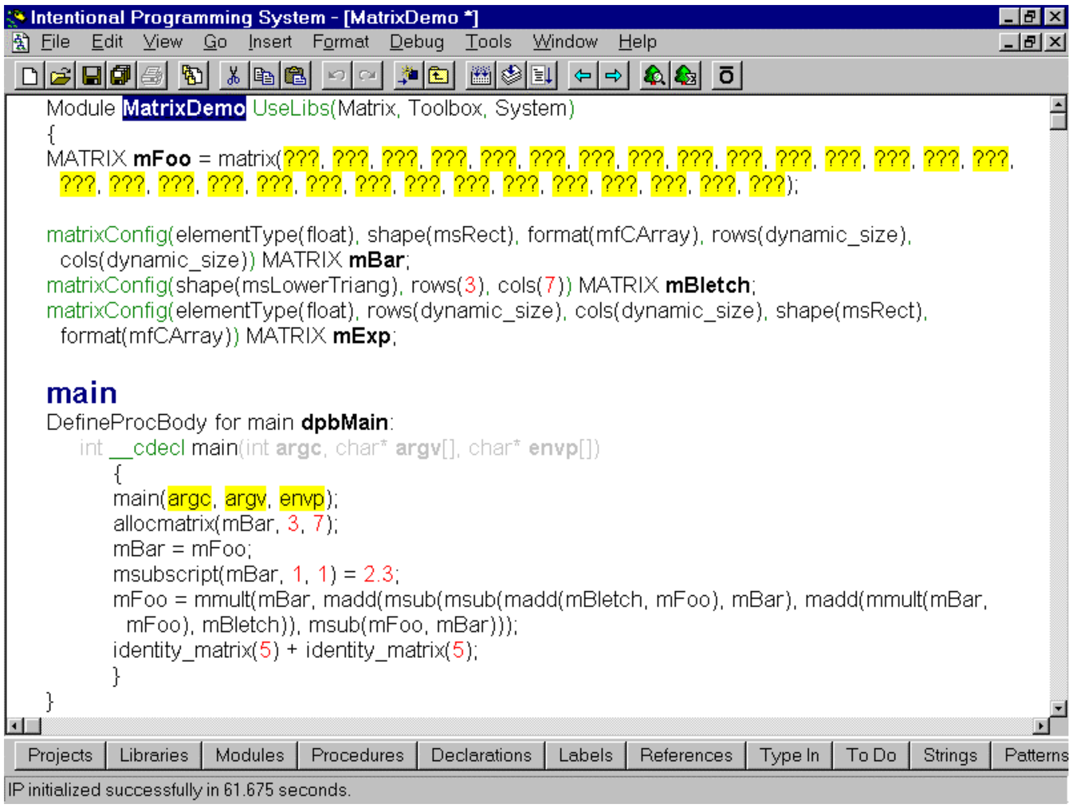
The prototype allows you to declare a matrix as follows:

```
matrixConfig(elementType(float), shape(msRect), format(mfCArray), rows(dynamic_size),
cols(dynamic_size)) MATRIX aRectangularMatrix;
```

MATRIX denotes the matrix type. It is annotated by matrixConfig, which specifies matrix parameters. This declaration is quite similar to what we did in the C++ implementation. The main difference is that, in contrast to the C++ template solution, the parameter values are specified by name in any order.

A matrix expression looks exactly like in C++, e.g.:

```
mFoo = mBar * (mBletch + mFoo - mBar - (mBar * mFoo + mBletch) + (mFoo - mBar));
```


Intentional Programming System - [MatrixDemo *]
File Edit View Go Insert Format Debug Tools Window Help
Projects Libraries Modules Procedures Declarations Labels References Type In To Do Strings Patterns

```
Module MatrixDemo UseLibs(Matrix, Toolbox, System)
{
MATRIX mFoo = matrix(???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???,
  ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???, ???);

matrixConfig(elementType(float), shape(msRect), format(mfCArray), rows(dynamic_size),
  cols(dynamic_size)) MATRIX mBar;
matrixConfig(shape(msLowerTriang), rows(3), cols(7)) MATRIX mBletch;
matrixConfig(elementType(float), rows(dynamic_size), cols(dynamic_size), shape(msRect),
  format(mfCArray)) MATRIX mExp;

main
DefineProcBody for main dpbMain:
   int __cdecl main(int argc, char* argv[], char* envp[])
      {
      main(argc, argv, envp);
      allocmatrix(mBar, 3, 7);
      mBar = mFoo;
      msubscript(mBar, 1, 1) = 2.3;
      mFoo = mmult(mBar, madd(msub(msub(madd(mBletch, mFoo), mBar), madd(mmult(mBar,
        mFoo), mBletch)), msub(mFoo, mBar)));
      identity_matrix(5) + identity_matrix(5);
      }
}
```

IP initialized successfully in 61.675 seconds.

Figure 188 *Sample matrix program (from Figure 187) displayed using the default rendering (mmult is the name of the intention representing matrix multiplication and madd the name of the intention representing matrix addition)*

Table 115 gives an overview of the library implementation.

IP document (i.e. IP file)	Module	Contents
matrix interface (Matrix.ip)	ADTs	declares intentions representing matrix type, matrix literal, and all the matrix operations (multiplication, addition, subtraction, number of rows, number of columns, subscripts, initialization, etc.)
	Config	declares intentions representing the DSL parameters and parameter values (e.g. elementType, shape, format, rect, lowerTriang, etc.)
	VIs	declares virtual intentions for the matrix type (see Section 6.4.3.3)
matrix implementation (MatrixImpl.ip)	RenderingAndEditing	implements the rendering and editing methods for matrix type, matrix literal, matrix operations, and matrix configuration DSL specifications
	MatrixTypeTransforms	implements processing of a matrix configuration (structure check, computing defaults, providing a flat configuration record), generating the data structures for a matrix according (e.g. C structs) to the configuration record
	OperationTransforms	implements checking and transforming matrix expressions, computing the result type of expressions, and generating operation implementations in C
	UI	implements a number of dialog boxes, e.g. for entering matrix configuration descriptions using radio buttons (optional) and for setting the number of rows and columns of a matrix literal

Table 115 *Overview of the IP implementation of the matrix library*

The functionality covered by the module MatrixTypeTransforms corresponds to the configuration generator and the matrix configuration components. However, there are two main differences:

- The generating part in MatrixTypeTransforms uses regular C code (mainly if and case statements) to do the configuration-DSL-to-ICCL mapping.

- Rather than being able to implement the matrix implementation components as class templates (which are convenient to compose), the MatrixTypeTransforms has reduce matrix application code to C, which is much more complex to implement (at the time of developing the prototype, the implementation of C++ in IP was not complete).

The module OperationTransforms implements the functionality covered in the operations section of the C++ implementation (i.e. Section 10.3.1.7). There are functions for computing result types of expressions, for checking the structure and optimizing an expression, and for generating C code.

RenderingAndEditing is the only implementation module that does not have its counterpart in the C++ implementation. This is so since C++ matrix programs are represented by ASCII text and no special support is needed to display them. On the other hand, the IP prototype provides a more natural intention for a matrix literal (i.e. the spreadsheet-like initializer of mFoo) than the textual and passive comma-separated list of numbers provided by the C++ solution (see Section 10.2.5.5). This might seem like a detail, but if we consider adding some specialized mathematical symbols (e.g. Figure 90), the advantages of the IP editing approach outweigh the simplicity of the textual, ASCII-based approach.

Compared to template metaprogramming, implementing the matrix library in IP had the following advantages:

- Metacode is easier to write since you can write it as usual C code.

- Debugging is easier since you can debug metacode using a debugger. You can also debug the generated code at different levels of reduction.

- You can easily issue error reports and warnings. They can be attached to the source tree node that is next to where the error occurred.

- It allows you to design new syntax for domain specific abstractions (e.g. the matrix literal).

There were also two main points on the negative side:

- The transformation scheduling protocol of IP used to be quite limiting (see Section 6.4.3.4) and required complicated and obscure tricks to resolve circular dependencies between intentions. However, as of writing, the scheduling part of IP has been completely reworked to remove these limitations.

- The IP APIs for rendering, editing, and transforming are still quite low level. Also, the fact that we had to reduce to C added a lot of complexity.

A more detailed comparison between the template metaprogramming and the IP approach is given in Table 116. Please note that the comparison applies to the current C++ compilers and the current IP System and may look differently in future. Also, this comparison is done in the context of our programming experiment. There are other features of IP such as code refactoring and code reengineering which are not considered here.

Criterion	Template Metaprogramming	Intentional Programming
complexity limits	The complexity of metaprograms is limited by the limits of current C++ compilers in handling very deeply nested templates.	There are no such limits.
debugging support	There is no debugging support.	Metacode can be debugged using a debugger. Also the generated code can be debugged at different levels of reduction.
error reporting	Inadequate.	Error reports and warnings can be attached directly to source tree nodes close to the locations where the problems occur.
programming effort	Due to the lack of debugging support, error-prone syntax (e.g. lots of angle brackets), and current compiler limits, template metaprograms may require significant programming effort. On the other hand, the code to be generated can be represented as easy-to-configure class templates.	The current IP system also requires significant programming effort. This is due to the low level APIs, e.g. tree editing, displaying, etc. The system provides only few declarative mechanisms (e.g. tree quote and simple pattern matching). Also the unavailability of the C++ mechanisms (classes, templates, etc.) adds programming complexity. On the other hand, the system can be extended with declarative mechanisms at any time. This is more scalable than the template metaprogramming approach.
readability of the metacode	Low.	Due to the (currently) low-level APIs, the readability is also low.
compilation speed	Larger metaprograms (esp. with recursion) may have unacceptable compilation times. Template metaprograms are interpreted.	Since IP has been designed for supporting metaprogramming, there are no such problems as with template metaprogramming (e.g. IP metaprograms are compiled). Nevertheless, compiling a C program using a commercial C compiler is much faster than compiling it using the current version of IP. This situation is expected to improve in future versions of IP.
portability/ availability	Potentially wide available, but better support for the C++ standard is required. The same applies to portability.	The IP system is not yet commercially available. In future, the portability will depend on the availability of such systems and interoperability standards.
performance of the generated code	Comparable to manually written code. The complexity of optimizations is limited by the complexity limits of template metaprograms.	Comparable to manually written code or better. This is so since very complex optimizations are possible.
displaying and editing	ASCII	Supports two-dimensional displaying, bitmaps, special symbols, graphics, etc.

Table 116 *Comparison between template metaprogramming and IP*

10.4 Appendix: Glossary of Matrix Computation Terms

Banded A banded matrix has its nonzero elements within a ‘band’ about the diagonal. The bandwidth of a matrix A is defined as the maximum of |i-j| for which a_{ij} is nonzero. The upper bandwidth is the maximum j-i for which a_{ij} is nonzero and j>i. See diagonal, tridiagonal and triangular matrices as particular cases. [MM]

Condition number The condition number of a matrix A is the quantity $||A||^2 * ||A\text{-}1||^2$. It is a measure of the sensitivity of the solution of Ax=b to perturbations of A or b. If the condition number of A is ‘large’, A is said to be ill-conditioned. If the condition number is one, A is said to be perfectly conditioned. The Matrix Market provides condition number estimates based on Matlab's condest() function which uses Higham's modification of Hager's one-norm method. [MM]

Defective A defective matrix has at least one defective eigenvalue, i.e. one whose algebraic multiplicity is greater than its geometric multiplicity. A defective matrix cannot be transformed to a diagonal matrix using similarity transformations. [MM]

Definiteness A matrix A is positive definite if x^T A x > 0 for all nonzero x. Positive definite matrices have other interesting properties such as being nonsingular, having its largest element on the diagonal, and having all positive diagonal elements. Like diagonal dominance, positive definiteness obviates the need for pivoting in Gaussian elimination. A positive semidefinite matrix has x^T A x >= 0 for all nonzero x. Negative definite and negative semidefinite matrices have the inequality signs reversed above. [MM]

Dense A dense matrix or vector contains a relatively large number of nonzero elements.

Diagonal A diagonal matrix has its only nonzero elements on the main diagonal.

Diagonal Dominance A matrix is diagonally dominant if the absolute value of each diagonal element is greater than the sum of the absolute values of the other elements in its row (or column). Pivoting in Gaussian elimination is not necessary for a diagonally dominant matrix. [MM]

Hankel A matrix A is a Hankel matrix if the anti-diagonals are constant, that is, $a_{ij} = f_{i+j}$ for some vector f. [MM]

Hessenberg A Hessenberg matrix is 'almost' triangular, that is, it is (upper or lower) triangular with one additional off-diagonal band (immediately adjacent to the main diagonal). A unsymmetric matrix can always be reduced to Hessenberg form by a finite sequence of similarity transformations. [MM]

Hermitian A Hermitian matrix A is self adjoint, that is AH = A, where AH, the adjoint, is the complex conjugate of the transpose of A. [MM]

Hilbert The Hilbert matrix A has elements $a_{ij} = 1/(i+j-1)$. It is symmetric, positive definite, totally positive, and a Hankel matrix. [MM]

Idempotent A matrix is idempotent if $A^2 = A$. [MM]

Ill conditioned An ill-conditioned matrix is one where the solution to Ax=b is overly sensitive to perturbations in A or b. See condition number. [MM]

Involutary A matrix is involutary if $A^2 = I$. [MM]

Jordan block The Jordan normal form of a matrix is a block diagonal form where the blocks are Jordan blocks. A Jordan block has its nonzeros on the diagonal and the first upper off diagonal. Any matrix may be transformed to Jordan normal form via a similarity transformation. [MM]

M-matrix A matrix is an M-matrix if a_{ij} <= 0 for all i different from j and all the eigenvalues of A have nonnegative real part. Equivalently, a matrix is an M-matrix if a_{ij} <= 0 for all i different from j and all the elements of A-1 are nonnegative. [MM]

Nilpotent A matrix is nilpotent if there is some k such that Ak = 0. [MM]

Normal A matrix is normal if $A A^H = A^H A$, where AH is the conjugate transpose of A. For real A this is equivalent to $A A^T = A^T A$. Note that a complex matrix is normal if and only if there is a unitary Q such that $Q^H A Q$ is diagonal. [MM]

Orthogonal A matrix is orthogonal if $A^T A = I$. The columns of such a matrix form an orthogonal basis. [MM]

Rank The rank of a matrix is the maximum number of independent rows or columns. A matrix of order n is rank deficient if it has rank < n. [MM]

Singular A singular matrix has no inverse. Singular matrices have zero determinants. [MM]

Sparse A sparse matrix or vector contains only a relatively small number of nonzero elements (often less than 1%).

Symmetric/ Skew-symmetric A symmetric matrix has the same elements above the diagonal as below it, that is, $a_{ij} = a_{ji}$, or $A = A^T$. A skew-symmetric matrix has $a_{ij} = -a_{ji}$, or $A = -A^T$; consequently, its diagonal elements are zero. [MM]

Toeplitz A matrix A is a Toeplitz if its diagonals are constant; that is, aij = f_{j-i} for some vector f. [MM]

Totally Positive/Negative A matrix is totally positive (or negative, or non-negative) if the determinant of every submatrix is positive (or negative, or non-negative). [MM]

Triangular An upper triangular matrix has its only nonzero elements on or above the main diagonal, that is $a_{ij}=0$ if i>j. Similarly, a lower triangular matrix has its nonzero elements on or below the diagonal, that is $a_{ij}=0$ if i<j. [MM]

Tridiagonal A tridiagonal matrix has its only nonzero elements on the main diagonal or the off-diagonal immediately to either side of the diagonal. A symmetric matrix can always be reduced to a symmetric tridiagonal form by a finite sequence of similarity transformations.

Unitary A unitary matrix has A^H = A-1. [MM]

10.5 Appendix: Metafunction for Evaluating Dependency Tables

The following metafunction evaluates dependency tables (see Section 10.3.1.6). This implementation does not use partial template specialization.

```
/*******************************************************************************
    file:       table.h

    author:     Krzysztof Czarnecki, Johannes Knaupp
    date:         August 25, 1998

    contents:     declaration of meta function EVAL_DEPENDENCY_TABLE which
                  finds the first matching entry in a selection table.
*******************************************************************************/

#ifndef TABLE_H
#define TABLE_H

#pragma warning( disable : 4786 )       // disable warning: identifier shortened
                                        // to 255 chars in debug information

#include "IF.H"


//****************************** helper structs ********************************

struct Nil {};

//endValue is used to mark the end of a list
//anyValue represents a value that matches anything
enum {  endValue = ~(~0u >> 1),         // least signed integer value
```

```
         anyValue = endValue + 1            // second least int
    };

//End represents the end of a list
struct End
{
   enum    { value = endValue };
   typedef End Head;
   typedef End Tail;
};

// ResultOfRowEval is used a struct for returning two result values
template< int found_, class ResultList_ >
struct ResultOfRowEval
{
   enum    { found = found_ };
   typedef ResultList_ ResultList;
};

//helper struct for error reporting
struct ERROR__NoMatchingTableRow
{
   typedef ERROR__NoMatchingTableRow ResultType;
};

//****************************** syntax elements ********************************

template< class ThisRow, class FollowingRows = End >
struct ROW
{
   typedef ThisRow             Head;
   typedef FollowingRows   Tail;
};

template< int value_, class FurtherCells = End >
struct CELL
{
   enum    { value = value_ };
   typedef FurtherCells  Tail;
};

template< class ThisResultType, class FurtherResultTypes = End >
struct RET
{
   typedef ThisResultType           ResultType;
   typedef FurtherResultTypes      Tail;
   enum    { value = endValue };
};

//************************* metafunction for evaluating a single row ***************************

template< class HeadRow, class TestRow >
class EVAL_ROW
{  //replace later by a case statement
      typedef HeadRow::Tail HeadTail;
      typedef TestRow::Tail RowTail;

      enum { headValue = HeadRow::value,
                 testValue = TestRow::value,
                 isLast    = (HeadTail::value == endValue),
                 isMatch   = (testValue == anyValue) || (testValue == headValue)
          };

      typedef IF<   isLast,
                        ResultOfRowEval< true, RowTail >,
                        EVAL_ROW< HeadTail, RowTail >::RET
              >::RET ResultOfFollowingCols;
```

```
  public:
    typedef IF<  isMatch,
                 ResultOfFollowingCols,
                 ResultOfRowEval< false, ERROR__NoMatchingTableRow >
         >::RET RET;
};

template<>
class EVAL_ROW< End, End >
{
public:
  typedef Nil RET;
};

//****************** meta function EVAL_DEPENDENCY_TABLE *********************

template< class HeadRow, class TableBody >
class EVAL_DEPENDENCY_TABLE
{
    typedef EVAL_ROW< HeadRow, TableBody::Head >::RET RowResult;
    typedef RowResult::ResultList      ResultList;
    typedef TableBody::Tail            FurtherRows;

    enum { found          = RowResult::found,
           isLastRow    = (FurtherRows::Head::value == endValue)
      };

    //this IF is used in order to map the recursion termination case onto <End, End>.
    typedef IF<  isLastRow, End, HeadRow >::RET HeadRow_;

    typedef IF<  isLastRow,
                 ERROR__NoMatchingTableRow,
                 EVAL_DEPENDENCY_TABLE< HeadRow_, FurtherRows >::RET_List
         >::RET NextTry;

  public:
    //returns the whole result row (i.e. the RET cells of the matching row)
    typedef IF< found, ResultList, NextTry >::RET RET_List;

    //returns the first RET cell of the matching row
    typedef RET_List::ResultType RET;
};

template<>
class EVAL_DEPENDENCY_TABLE< End, End >
{
public:
  typedef Nil RET_List;
};

#endif   // #ifndef TABLE_H
```

The following file demonstrates the use of EVAL_DEPENDENCY_TABLE:

```
/******************************************************************************
       File:        table.cpp
       Author:    Krzysztof Czarnecki, Johannes Knaupp
       Date:        August 25, 1998

       Contents:   test of meta function EVAL_DEPENDENCY_TABLE which
                finds the first matching entry in a selection table.
******************************************************************************/

#include "iostream.h"
#include "table.h"
```

```
//a couple of types for testing
template< int val_ >
struct Num
{
  enum { val = val_ };
};

typedef Num< 1 > One;
typedef Num< 2 > Two;
typedef Num< 3 > Three;
typedef Num< 4 > Four;


void main ()
{
  // test with a single return type
  typedef EVAL_DEPENDENCY_TABLE
       //****************************************************************
       <          CELL<    1,   CELL<   2                              > >

       , ROW< CELL<    1,   CELL<   3,     RET<      One     > > >
       , ROW< CELL<    1,   CELL<   3,     RET<      Two     > > >
       , ROW< CELL<    1,   CELL<   2,     RET<      Three   > > >
       , ROW< CELL<    1,   CELL<   3,     RET<      Four    > > >
       //****************************************************************
       > > > > >::RET Table_1;

  cout << Table_1::val << endl; //prints "3"


  // test with two return types
  typedef EVAL_DEPENDENCY_TABLE
       //**************************************************************************
       <          CELL<    4,        CELL<   7                                          > >

       , ROW< CELL<    3,        CELL<   7,           RET< Three,    RET<  Four > > > >
       , ROW< CELL<    4,          CELL<   5,           RET<  Four,    RET<  Three   > > > >
       , ROW< CELL< anyValue, CELL<   7,           RET<  One,    RET< Two     > > > >
       , ROW< CELL< anyValue, CELL< anyValue, RET<  Two,      RET< One   > > > >
       //**************************************************************************
       > > > > >::RET_List ResultRow;

  typedef ResultRow::       ResultType ReturnType_1;
  typedef ResultRow::Tail::ResultType ReturnType_2;

  cout << ReturnType_1::val << '\t' << ReturnType_2::val << endl; //prints "1     2"

}
```

10.6 References

[ABB+94] E. Anderson, Z. Bai, C. Bischof, J. W. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1994, see http://www.netlib.org/lapack/lug/lapack_lug.html

[BLAS97] Basic Linear Algebra Subprograms: A Quick Reference Guide. University of Tennessee, Oak Ridge National Labolatory, and Numerical Algorithms Group Ltd., May 11, 1997, see http://www.netlib.org/blas/blasqr.ps

[BN94] J. : Barton and L. R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Applications*. Addison-Wesley, 1994

[Bre98] U. Breymann. *Designing Components with the C++ STL: A new approach to programming*. Addison Wesley Longman, 1998

[CHL+96] S. Carney, M. Heroux, G. Li, R. Pozo, K. Remington, and K. Wu. A Revised Proposal for a Sparse BLAS Tookit. SPARKER Working Note #3, January 1996, see http://www.cray.com/PUBLIC/APPS/ SERVICES/ALGRITHMS/spblastk.ps

[DBMS79] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1979

[DDDH90] J. Dongarra, I. Duff, J. DuCroz, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. In *ACM Transactions on Mathematical Software*, vol. 16, 1990, pp. 1-17, see http://www.netlib.org/blas/blas2-paper.ps

[DDHH88] J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. In *ACM Transactions on Mathematical Software*, vol. 14, no. 1, 1988, pp. 1-32, see http://www.netlib.org/blas/blas3-paper.ps

[DLPRJ94] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A Sparse Matrix Library in C++ for High Performance Architectures. In *Proceedings of the Second Object Oriented Numerics Conference*, 1994, pp. 214-218, see ftp://gams.nist.gov/pub/pozo/papers/sparse.ps.Z

[DLPR96] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. IML++ v. 1.2 Iterative Method Library Reference Guide, 1996, see ftp://gams.nist.gov/pub/pozo/docs/iml.ps.gz

[DPW93] J. Dongarra, R. Pozo, and D. Walker. LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra. In *Proceedings of Supercomputing '93*, IEEE Computer Society Press, 1993, pp. 162-171, see http://math.nist.gov/lapack++/

[Ede91] A. Edelman. The first annual large dense linear system survey. In *SIGNUM Newsletter*, no. 26, October 1991, pp. 6-12, see ftp://theory.lcs.mit.edu/pub/people/edelman/parallel/survey1991.ps

[Ede93] A. Edelman. Large Dense Numerical Linear Algebra in 1993: The Parallel Computing Influence by A. Edelman. In *Journal of Supercomputing Applications*, vol. 7, 1993, pp. 113-128, see ftp:// theory.lcs.mit.edu/pub/people/edelman/parallel/1993.ps

[Ede94] A. Edelman. Large Numerical Linear Algebra in 1994: The Continuing Influence of Parallel Computing. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 781-787, see ftp://theory.lcs.mit.edu/pub/people/ edelman/parallel/large94.ps

[FS97] F. Gomes and D. Sorensen. C++: A c++ implementation of ARPACK eigenvalue package. User's Manual, draft version, August 7, 1997, see http://www.caam.rice.edu/software/ARPACK/arpackpp.ps.gz

[GL96] G. Golub and C. van Loan. *Matrix Computations*. Third edition. The John Hopkins University Press, Baltimore and London, 1996

[Hig96] N. Higham. Recent Developments in Dense Numerical Linear Algebra. Numerical Analysis Report No. 288, Manchester Centre for Computational Mathematics, University of Manchester, August 1996; to appear in *The State of the Art in Numerical Analysis*, I. Duff and G. Watson, (Eds.), Oxford University Press, 1997, see ftp://ftp.ma.man.ac.uk/pub/narep/narep288.ps.gz

[ILG+97] J. Irwin, J.-M. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-Oriented Programming of Sparse Matrix Code. XEROX PARC Technical Report SPL97-007 P9710045, February 1997, see http://www.parc.xerox.com/aop

[JK93] A. Jennings and J. McKeown. *Matrix Computation*. Second edition, John Wiley & Sons Ltd., 1993

[KL98] K. Kreft and A. Langer. Allocator Types. In *C++ Report*, vol. 10, no. 6, June 1998, pp. 54-61

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. In *ACM Transactions on Mathematical Software*, vol. 5, 1979, pp. 308-325

[LSY98] R. Lehoucq, D. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. Society for Industrial and Applied Mathematics (SIAM), 1998, see http://www.caam.rice.edu/software/ARPACK/usergd.html

[MM] Matrix Market at http://math.nist.gov/MatrixMarket/, an on-line repository of matrices and matrix generation tools, Mathematical and Computational Science Division within the Information Technology Laboratory of the National Institute of Standards and Technology (NIST)

[MS96] D. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, Massachusetts, 1996

[Neu98] T. Neubert. Anwendung von generativen Programmiertechniken am Beispiel der Matrixalgebra. Diplomarbeit, Technische Universität Chemnitz, 1998

[OONP] The Object-Oriented Numerics Page (maintained by T. Veldhuizen) at http://monet.uwaterloo.ca/oon/

[Poz96] R. Pozo. Template Numerical Toolkit for Linear Algebra: high performance programming with C++ and the Standard Template Library. Presented at the Workshop on Environments and Tools For Parallel Scientific Computing III, held on August 21-23, 1996 at Domaine de Faverges-de-la-Tour near Lyon, France, 1996, see http://math.nist.gov/tnt/

[Pra95] R. Pratap. *Getting Started with Matlab*. Sounders College Publishing, Fort Worth, Texas, 1995

[SBD+76] B. Smith, J. Boyle, J. Dongarra, B. Garbow, Y. Ikebe, V. Klema, and C. Moler. *Matrix Eigensystem Routines – EISPACK Guide*. Second edition, vol. 6 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1976

[Sch89] U. Schendel. *Sparse Matrices: numerical aspects with applications for scientists and engineers*. Ellies Horwood Ltd., Chichester, West Sussex, England, 1989

[SL98a] J. G. Siek and A. Lumsdaine. The Matrix Template Library: A Unifying Framework for Numerical Linear Algebra. In *Proceedings of the ECOOP'98 Workshop on Parallel Object-Oriented Computing (POOSC'98)*, 1998, see http://www.lsc.nd.edu/

[SL98b] J. G. Siek and A. Lumsdaine. A Rational Approach to Portable High Performance: The Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (FAST) Library. In *Proceedings of the ECOOP'98 Workshop on Parallel Object-Oriented Computing (POOSC'98)*, 1998, see http://www.lsc.nd.edu/

[Vel95] T. Veldhuizen. Expression Templates. In *C++ Report*, vol. 7 no. 5, June 1995, pp. 26-31, see http://monet.uwaterloo.ca/blitz/

[Vel97] T. Veldhuizen. Scientific Computing: C++ versus Fortran. In *Dr. Dobb's Journal*, November 1997, pp. 34-41, see http://monet.uwaterloo.ca/blitz/

[Wil61] J. Wilkinson. Error Analysis of Direct Methods of Matrix Inversion. In *Journal of the ACM*, vol. 8, 1961, pp. 281-330

Chapter 11 Conclusions and Outlook

The goal of this work was both to develop the concepts behind Generative Programming (GP) and to test them on a case study. However, it should be noted that GP is in its early development stage and much more theoretical and practical work is required. Nonetheless, this thesis provides a number of useful results and conclusions:

- Currently the only methods adequately addressing the issue of development for reuse are Domain Engineering (DE) methods. Furthermore, most existing DE methods have essentially the same structure. We found Organization Domain Modeling (Section 3.7.2) to be a well-documented, publicly available DE method which integrates aspects of many other DE methods.

- The main difference between OOA/D methods and DE methods is that the first focus on developing single systems while the latter on developing families of systems.

- DE methods and OOA/D methods are perfect candidates for integration. OOA/D methods provide useful notations and techniques for system engineering, but they do not address development for reuse. The integration of DE and OOA/D provides us with effective engineering methods for reuse that have the benefits of OOA/D methods.

- Feature modeling is the main contribution of Domain Engineering to OOA/D. Feature models represent the configurability aspect of reusable software at an abstract level, i.e. without committing to any particular implementation technique such as inheritance, aggregation, or parameterized classes. Developers construct the initial models of the reusable software in the form of feature models and use them to guide the design and implementation. To a reuser, on the other hand, feature models represent an overview of the functionality of the reusable software and a guide to configuring it for a specific usage context.

- Achieving high intentionality, reusability, and performance in the implementation requires capabilities traditionally absent in programming languages such as domain-specific optimizations, domain-specific error handling, domain-specific syntax extensions, new composition mechanisms, etc. Modularly extensible compilers and programming environments allow programmers to incorporate these capabilities into the compilation process. Some tools also support domain-specific debugging, displaying, and editing.

- Template metaprogramming in C++ represents a practical alternative to extensible compilers. This approach allows us to build a wide range of generators in C++ without the need to modify the compiler. Template metaprogramming can be combined with many different generic and component-based C++ programming techniques based on templates. Unfortunately, the complexity of template metaprograms is limited by current compiler limits, compilation times, and debugging problems. The problems with compiler limits and portability will decrease as more and more compilers will conform to the ANSI/ISO standard.

- Aspect-Oriented Programming (AOP) provides concepts and techniques for achieving a better separation of concerns both in the design and in the implementation of a piece of software. AOP can be incorporated into Domain Analysis by means of feature starter sets. On the implementation side, AOP fits very well with the idea of modularly extendible compilers and programming environments.

- We found that different categories of domains will require different, specialized DE approaches. In this thesis, we developed DEMRAL, a concrete approach for the category of algorithmic domains.

- The Domain Engineering Method for Reusable Algorithmic Libraries (DEMRAL) covers the analysis, design, and implementation of algorithmic libraries such as numerical libraries, image processing libraries, and container libraries. It is based on novel concepts such as feature starter sets and configuration and expression DSLs.

- DEMRAL was tested by applying it to the domain of matrix computation libraries. The result is the Generative Matrix Computation Library (GMCL). The C++ implementation of the matrix component (which is a part of C++ GMCL) comprises only 7500 lines of C++ code, but it is capable of generating more than 1840 different kinds of matrices. Despite the large number of provided matrix variants, the performance of the generated code is comparable with the performance of manually coded variants. The application of template metaprogramming allowed a highly intentional library API and a highly efficient library implementation at the same time. The implementation of GMCL within the Intentional Programming system (IP) demonstrates the advantages of IP, particularly in the area of debugging and displaying, and shows the need for providing higher-level programming abstractions in IP.

Areas for future work include the following:

- Analysis and Design for GP;

  - methods for different categories of domains (e.g. distributed and business information systems);

  - testing DEMRAL on other algorithmic domains (e.g. image processing);

- industrial strength extendible compilers and programming environments;

- testing techniques and metrics for GP.