Object-Oriented Frameworks

A survey of methodological issues

Michael Mattsson

CODEN: LUTEDX/(TECS-3066)/1-130/(1996)

LU-CS-TR: 96-167

Michael Mattsson

Department of Computer Science and Business Administration
University College of Karlskrona/Ronneby
S - 372 25 Ronneby, Sweden
E-mail: Michael.Mattsson@ide.hk-r.se
WWW: http://www.pt.hk-r.se/~michaelm/

Department of Computer Science
Lund University
Box 118
S - 221 00 Lund
SWEDEN

Preface

The work with this thesis was carried out within the Department of Computer Science and Business Administration at University College of Karlskrona/Ronneby.

First, I would like to thank my advisors, Boris Magnusson and Lennart Ohlson, for their guidance and patience during my studies.

Then, I would like to thank my colleagues at the University College of Karlskrona/Ronneby, in particular Jan Bosch, Peter Molin, Bertil Ekdahl, Conny Johansson and Rune Gustavsson.

Finally, a special thank to Christina Magnusson, for reading the first version of this thesis, and to Martin Fuller, for commenting and improving the english in the final version.

This work is supported in part by Blekinge Forskningsstiftelse.

Michael Mattsson
Ronneby, February 1996

Contents

1 Introduction

Software productivity has increased the last three decades [You92]. However, the gap between the demands on the software industry and what state-of-the-practice can deliver is still large. Several decades of research in software engineering left few alternatives but software reuse as the (only) realistic approach to bring about the gains in productivity and quality that the software industry needs [Mil95].

With the introduction of object-oriented technology in the software development process a number of qualities were introduced that enable reuse [Mil95].

- Problem-orientation. The object-oriented models are described in terms of the problem domain. The seamlessness from analysis to programming makes it simpler to communicate the models between the user and developer and thereby deliver a better software product.
- Resilience to evolution. The processes in an application domain change more often then the entities in the domain. Thus, a model that is structured around the entities are more stable to changes.
- Domain analysis. Object-oriented analysis is natural to extend to domain analysis (versus single application analysis). Domain analysis can be viewed as a broader and more extensive analysis that tries to capture the requirements of the complete problem domain including future requirements. For a more thorough description of domain analysis, see [Sch94b].

When introducing software reuse in an organisation this implies changing existing software life cycles into new ones:

- Development-for-reuse and,
- Development-with-reuse

The development-for-reuse life cycle has the focus to develop reusable assets intended to be reused in the development-with-reuse life cycle. In the

development-with-reuse life cycle reusable assets are searched for and used, if suitable, in the software development.

Reusable assets in object-orientation can be found in a continuum ranging from objects and classes to design patterns and object-oriented frameworks, the latter which represent the state-of-the-art in object-oriented reusable assets.

Historically, reusable software components have been procedure and function libraries (the 1960s) and in the late 60's Simula 67 [Bir73] introduced objects, classes and inheritance which resulted in class libraries. Both the procedural and the class libraries are mainly focused on reuse of code. However, reuse of design would be more beneficial in economical terms. This since design is the main intellectual content of software and it is more difficult to create and re-create then programming code.

With the integration of data and operations into objects and classes an increase in reuse was achieved. The classes were packaged together into class libraries. These class libraries often consist of classes for different data structures, such as lists and queues. The class libraries were further structured using inheritance to facilitate specialisation of existing library classes. The class libraries delivered software reuse beyond traditional procedural libraries.

The problems with reusing class libraries are that they do not deliver enough software reuse. Software developers still have to develop a lot of application code themselves. Limitations in reusing classes and objects from class libraries comprise [Cot95]:

- Complexity. In large and complex system the class hierarchies can be very confusing. If the developer has no access to detailed documentation it can be tricky to figure out the designers intention with the hierarchy.
- Duplication of effort. Class libraries allow the developers to reuse classes which can be put together by different developers in different ways. This may result in different solutions to the same kind of problems and may cause maintenance problems.
- Flow of control. When reusing class libraries in application development, the program is still responsible for the flow of control, i.e. the control of interactions among all the objects created from the library. This can be a problem since the developer has the responsibility of

deciding in which order objects and operations have to be performed and errors can be made.

The striving to avoid the above mentioned problems in combination with the desire to reuse designs has resulted in o*bject-oriented frameworks*. An object-oriented framework is a reusable design for an application, or subsystem, represented by a set of abstract classes and the way they collaborate. Thus, a framework outlines the main architecture for the application to be built. Together with the framework there are accompanying code that provide a default implementation for a "standard" application (or subsystem). A framework then implies reuse of both code and *design*. A framework particularly emphasizes those parts of the application domain that will remain stable and the relationships and interactions among those parts, i.e the framework is in charge of the flow of control. The framework will also point out those parts that are likely to be customized for the actual application under development.

The first examples of the object-oriented framework concept appeared in the mid-eighties [Gol84], [Sch86]. Frameworks attained more interest when the Interviews [Lin89] and ET++ [Wei89] user interface frameworks were developed in the C++ language. However, the interest was on the software itself, a reusable set of classes that reduced an effort consuming task; user interface development. No, or little, attention was paid to the development process of the frameworks or the structure of the frameworks. The user interface frameworks were reused by software engineers that mastered the C++ language well, so the size of the frameworks and insufficient documentation were no major hindrance.

Proposed object-oriented methods, [Boo86], [Boo91], [Rum91], do not address or emphasize important design elements necessary to increase the reusability of an object-oriented framework [Mat95]. Thus, there is a need for methods for framework development.

Due to the size of a framework, issues such as describing the framework's structure and other documentation aspects are difficult to communicate to the framework re-user. In the late eighties and early nineties, the notion of patterns arose. In 1987 Beck and Cunningham presented five patterns at an OOPLSA (Conference on Object-Oriented Programming Systems, Languages, and Applications) workshop [Bec87] which described how to design windows-based user interfaces. In parallel Gamma was writing about

reusable object-oriented software for ET++, a user interface framework [Wei89]. During the OOPSLA conferences 1990 and 1991 a number of people meet at the architecture workshops and the writing of an pattern catalogue, which now is available [Gam95], started. A few pattern papers were published which addressed issues such as design structure and the documentation of frameworks. For example, [Joh92] describes how in small steps to document a framework. In [Gam93a] the concept of object-oriented design patterns were introduced which can be used for describing small parts of a larger object-oriented design. Now, in the mid-nineties, a pattern movement is seen (e.g in journals and conferences about object-orientation) and this may be the first step towards a better understanding of the framework concept and associated methodological issues, such as documentation and application development based on frameworks.

1.1 Thesis Outline

This thesis will give a survey and discuss the state-of-the-art reuse assets, frameworks and patterns, in object-oriented software development. The focus will be on concepts and methodological issues related to object-oriented frameworks. The three main activities examined are:

- Framework Development
- Documentation of Frameworks
- Using Frameworks

Since the notion of patterns has influence on the framework concept and related issues, as depicted in figure 1.1, patterns are thoroughly discussed, too.

Issues which are often discussed in the context of software reuse, besides technical issues, are organizational, economical, managerial and tool issues [Kar95].

- Organizational issues discuss how the development and reuse of the assets should be organized. In the case of frameworks this has been briefly discussed in [Bir94], which argues that the development should be divided into two groups: One framework maintenance group which is responsible for the development and maintenance of the framework,

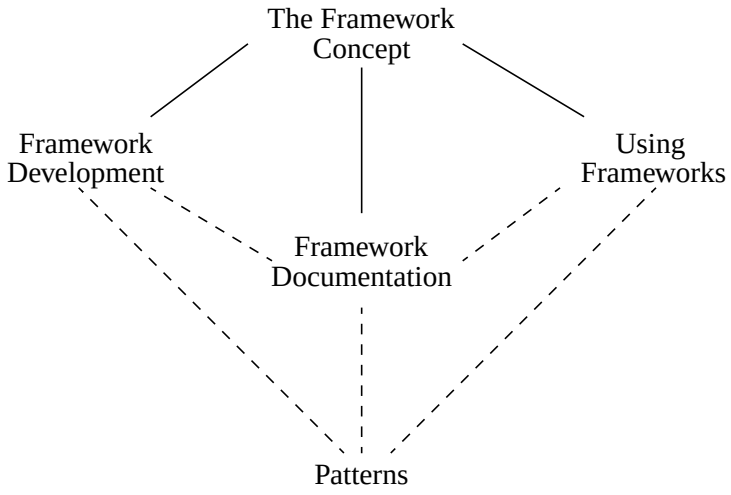


Figure 1.1: *Influences of patterns on frameworks*

and one other group that consists of the application developers who reuse the framework. The argument for this division is: only through knowing the requirements of many applications is it possible to identify the parts of the designs that are general enough to be incorporated in the framework. The argument against a separate framework group is that the framework designers must be close to the application development in order to get a better understanding of both the application domain and the problems with reusing the framework [Joh91].

- Economical issues discuss cost models for developing reusable software. Regarding economical issues for object-oriented frameworks, little work has been done. In [Kih95] a method for deciding the business value to an organisation that will use framework technology is discussed, and a cost model for framework-based development is presented. Currently, this method has not been applied in an industrial setting.
- Managerial issues deal with how to encourage the staff to develop and reuse the assets. In the case of object-oriented frameworks the importance of other issues, such as good documentation and marketing of the

reusable framework inside the development organisation, have been emphasized [Tal94].

- Tool issues focus on supporting software for the asset re-user and the asset repository manager. Typical tools are search and retrieval tools, adaptation tools and reverse engineering tools. Some categories of tools may not be important for object-oriented frameworks due to the size of the framework, e.g retrieval tools. However, tools that support version and configuration management, can be more useful. This since a number of applications based on one framework will be developed and later maintained (due to new application requirements or internal changes in the frameworks). Controlling and mastering these kinds of situations will require effective tool support.

As the emphasis of this thesis will be on the technical aspects of frameworks and patterns, we will not further discuss the mentioned reuse issues unless it is necessary in terms of context.

This thesis is structured in the following chapters: Patterns, Framework Development, Using Frameworks, Documentation of Frameworks, Conclusions and Future Work.

Chapter 2, Patterns, discusses the concept of patterns. The origin of the pattern idea in object-orientation and the two main categories of patterns, generative and non-generative patterns, are introduced and reviewed. Variants of the generative and non-generative patterns are also presented. Existing classification schemes and selection approaches for using patterns are discussed. Finally, experiences considering both advantages and problems of using patterns is discussed and areas that need further investigation are outlined.

Framework Development, Chapter 3, focuses on how to develop an object-oriented framework. The concept itself is discussed and compared to related concepts to avoid misunderstandings. The advantages and disadvantages together with different characterisations of frameworks are presented. The framework development process, guidelines and important design elements for the development thereof are described. The maintenance aspect of a framework, especially restructuring, is also addressed. The chapter ends with a discussion of a number of topics which are of interest to researchers in the area of framework development.

Chapter 4, Using Frameworks, addresses the development-with-reuse process. We propose a method for framework-based development. The method's relation to the phases of the software development process is also discussed. Using frameworks is, from a methodological perspective, rarely investigated. Therefore a number of issues related to methods for framework-based development are proposed as research areas.

Chapter 5, Documentation of Frameworks, concerns one of the most crucial aspects for successful reuse: documentation. If the documentation does not fulfil the requirements of the different framework users it will not be reused. The different kinds of audiences for framework documentation together with their needs are described. Existing approaches for the documentation together with comments on how the approaches meet the needs are discussed. Finally, areas that need further research in framework documentation are commented on.

In Chapter 6, Conclusions, the earlier chapters are summoned up and research areas in the field of patterns and object-oriented frameworks are highlighted.

In Chapter 7, Future Work, we describe a case study to develop and evaluate a documentation technique for framework documentation to pursue as the next step in our research.

Appendix A introduces the notation used for the class diagrams in the figures.The notation is an enhanced variant of the Object Modelling Technique notation [Rum91], which allows implementation details to be shown.

Appendices B and C give an introductory description to two existing sets of design patterns [Bus94a],[Gam95].

Finally, Appendix D gives a complete description of the Prototype design pattern [Gam95].

2 Patterns

2.1 Introduction

The notion of patterns has its origin in the work by Christopher Alexander, an architect that developed the idea of a pattern language to enable people to design their own homes and communities [Ale77]. The idea of patterns has been adopted by software engineers in object-orientation, who transformed the ideas of Alexander to the area of software development. The main approach taken by the software engineers is to develop patterns that are solutions to small-scale design problems.

In the field of building a *pattern language* is a set of patterns, each of which describes how to solve a particular problem. A pattern describes how to deal with a certain architectural design issue in a structured way, for example Parallel roads and Sequence of sitting spaces. Alexander patterns vary in their level of detail, in fact they start on a global level; how the World should be broken down to countries, Countries be broken down to regions, and down to Office connections and Placement of windows in kitchens.

In Alexander's book *A Pattern Language* [Ale77] 253 patterns were presented, all conforming to a structure with the following headings [Ale77], [Lea94a]:

- *Name*. A well-known, descriptive phrase or name. The name often gives a hint of the solution rather than the problem or the context.
- *Example*. Photos, diagrams and descriptions that illustrate the patterns application.
- *Context*. Under which circumstances the pattern can be applied. It explains why the pattern exists and how general the pattern is.
- *Problem*. A description of the relevant forces and constraints, and how they interact.

- *Solution*. Relationships and dynamic rules that describe how to construct artifacts in accordance with the pattern. Variants and similar patterns are referenced. References to both higher- and lower-level patterns are included.

In addition to these headings, a pattern must fulfil some other properties in order to be useful, such as encapsulation, generativity, equilibrium, abstraction, openness and composibility [Lea94a]. Ideally, a pattern has all these properties.

- *Encapsulation*. A well-defined problem/solution is encapsulated by the pattern. The pattern has to be specific and independent. It has to be precisely formulated to make clear when it can be applied and whether it captures real issues.
- *Generativity.* Each pattern contains a description of the process that explains how to realize the solution. The patterns are written to be usable for all participants in the construction work, not just experienced designers.
- *Equilibrium*. The pattern has a solution space with an invariant that minimizes the conflict between constraints and forces.
- *Abstraction*. Experiences are captured and the pattern represents the knowledge captured.
- *Openness*. Openness means the ability for patterns to be extended down to an arbitrary level of detail.
- *Composibility.* The patterns are related to each other hierarchically in the way that coarse grained patterns are layered on top of, relate, and constrain fine grained ones. Patterns can be composed and arranged as a kind of language.

This structure of Alexander's patterns and the properties described have influenced software engineers to create analogous patterns in object-oriented analysis and design (e.g [Coad92], [Joh92], [Gam93b]) of software. Two main variants of patterns are identified, *non-generative* and *generative patterns*, of which each, in turn, has a number of variants.

Non-generative patterns are patterns that can be observed in already built systems, describe the relationships between objects and answer the question "*What of a design or analysis"* that has been documented. These kinds of patterns are descriptive and passive. Although we can find these patterns, there is no well-articulated rationale behind them. What one is striving after

is to capture the rationale behind these "good" non-generative patterns so the developers of a software system can apply the patterns to generate the system or part of the system. The rationales are captured and described as *generative patterns*, which are prescriptive and active patterns. They can be viewed as an answer to the question *"Why a design or analysis has this structure"*. Thereby, providing a rationale for a design or analysis after the fact.

2.2 A Pattern Example

To give an idea of what a pattern is we will give a simple design example and discuss how it can evolve into a design pattern.

Consider the well-known class hierarchy about graphic figures for graphical applications such as a drawing editor, figure 2.1. In this kind of application there is always a need to build more complex figures out of simpler figures. A simple implementation would be to define a class which acts as a container for the simpler graphic figures and which may contain other container classes, see figure 2.2.

The problem with the approach in figure 2.2 is that the classes that use these solutions must treat simple graphic figure objects and complex graphic figure objects differently. Handling this difference makes the application more

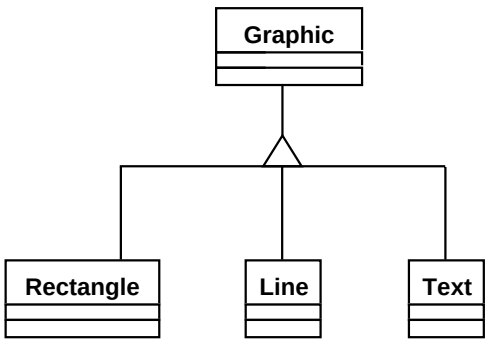


Figure 2.1: *The Graphics Figure Hierarchy*

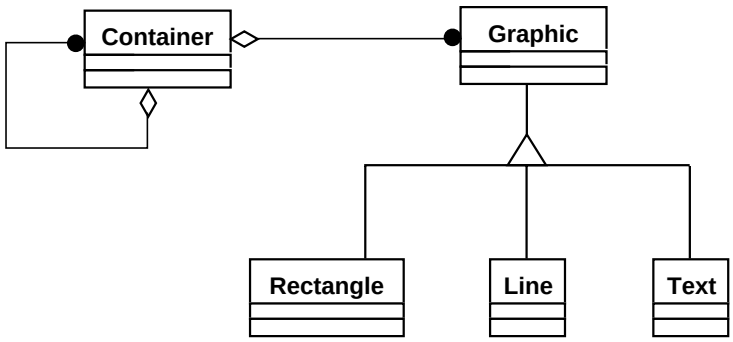


Figure 2.2: *The Container Solution*

complex. To avoid this problem the Composite design pattern [Gam95] can be used.

The Composite pattern composes objects into tree structures to represent part-whole hierarchies and lets clients treat individual objects and compositions of objects uniformly [Gam95].

The solution proposed by the Composite pattern is to introduce an abstract class that represents both the simple graphic figure and the complex one, see figure 2.3. The class Graphic is an abstract class and declares a Draw operation that is specific to graphic objects. It also declares operations for accessing and managing its children. The Picture class aggregates Graphic objects and implements the Draw operation to call Draw on its children. Since the Picture interface conforms to the Graphic interface, objects of the Picture class can be composed of other Picture objects recursively. The solution proposed in figure 2.3 and an argument like the one above are parts of the Composite pattern.

Besides these parts, a design pattern has sections that describe the applicability of the pattern (in our case representing whole-part hierarchies of objects and treat the composite and simple objects uniformly), a general class diagram for the pattern (see figure 2.4), a description of the participating classes and their collaboration. There are also discussions on the consequences of using the pattern and implementation issues.

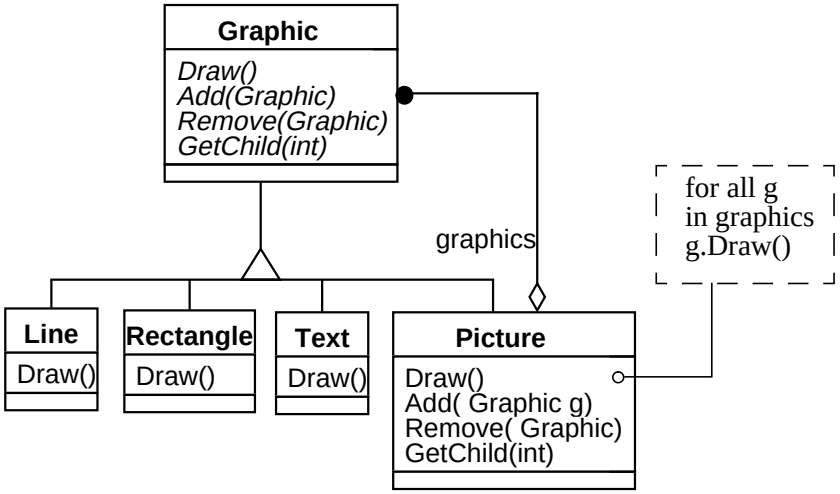


Figure 2.3: *The Composite pattern solution*

2.3 Non-Generative Patterns

A *non-generative pattern* focuses on solution(s) rather than when to use it or the rationale behind the pattern. As in Alexander's work, where patterns for different architectural phases exist, there exist non-generative patterns for different software development phases, i.e analysis and design.

The object-oriented analysis patterns are defined as: *An object-oriented (*analysis) *pattern is an abstraction of a doublet, triplet, or other small grouping of classes that is likely to be helpful again and again in object-oriented development [Coa92].*

The object-oriented analysis patterns [Coa92] comprise three headings and a diagram. The headings are; *The pattern name* with a short description of the pattern and the diagram describing the classes and the relationships between the classes; *An example* of the analysis patterns, and *Guidelines for using* the analysis pattern.

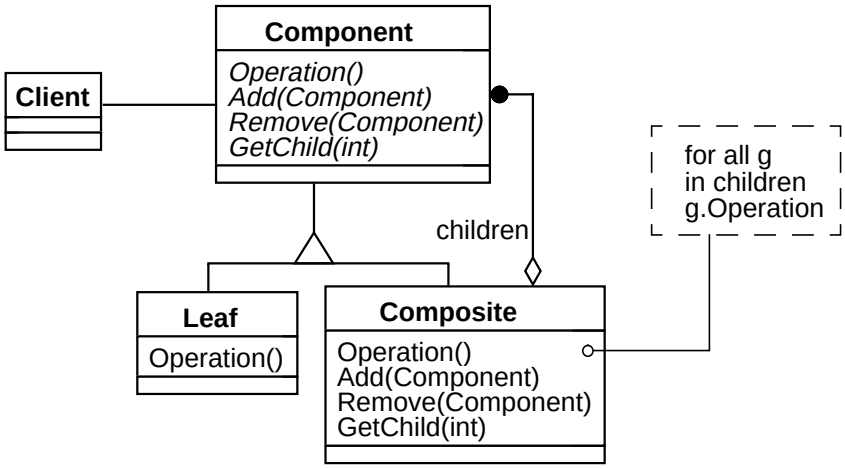


Figure 2.4: *The structure of the Composite pattern*

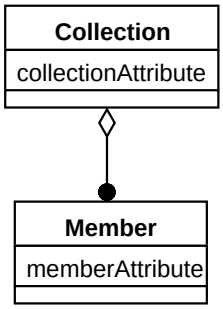
The definition of an object-oriented analysis pattern is imprecise. The headings of the pattern will give too little information to make the pattern useful. Furthermore, the examples given are very general, for example the pattern "State Across a Collection", figure 2.5, which describes how to use aggregation, will occur in almost every analysis task.

In the design phase a number of non-generative patterns on different granularity levels can be identified. The granularity levels identified are *architectural frameworks*, (object-oriented) *design patterns*, and *idioms* [Bus94a], where the architectural framework has the highest granularity and the idioms the lowest.

Software architectures are built following some overall structuring principles described by an architectural framework. The software architecture is composed of a number of smaller units which are described by design patterns. Finally, idioms deal with the implementation and realization of particular design issues.

Idioms

Idioms are the lowest-level of non-generative patterns. They describe how to implement particular components, their functionality, or their relationships to other components in a given design. The idioms are always closely con-



The pattern. A "collection" object knows its state; this state applies to the collection and may also apply to its parts by physical or temporal proximity. Each "member" object also has a state of its own.

An example. An aircraft is an assembly of engines: in other words, an "aircraft" object may know about a number of "engine" objects. Each "engine" object knows about its performance; this particular attribute value applies to the whole, and also to its parts, by physical proximity.

Guidelines for use. Use this pattern whenever there is a whole-part in a business domain or implementation domain, and one or more attributes apply to the whole (the collection). [Coa92]

Figure 2.5: *The State Across a Collection pattern [Coa92]*

nected to a programming language. Idioms in C++ [Ell90] and Smalltalk [Gol83] can be found in [Cop92] and [Bec94+] respectively. Since idioms are at the lowest level and deal with source code, they are the final link between the design phase and the implementation.

An idiom describes how to implement particular components, (part of) their functionality, or their relationships to other components with a given design. They often are specific for a particular programming language. [Bus94a]

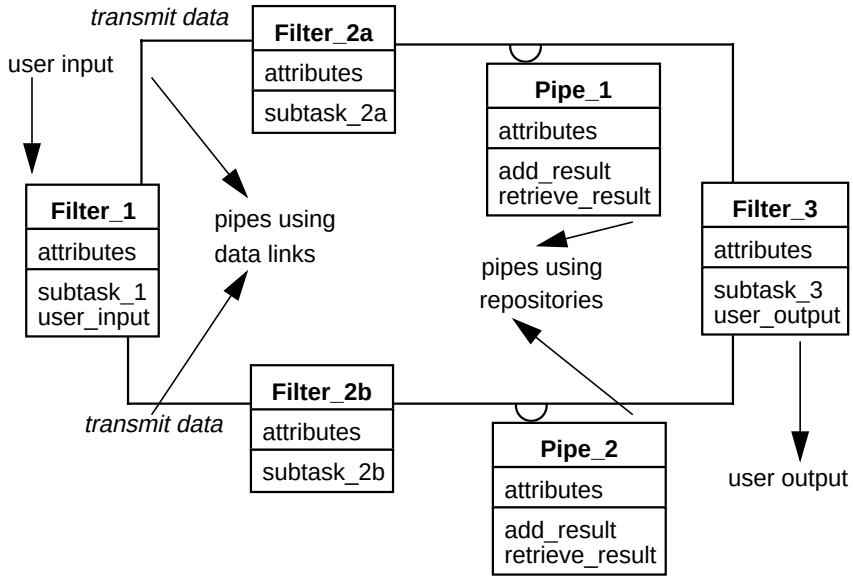
An example of a C++ specific idiom is the reference counting idiom [Cop92], which describes how to implement reference counting in C++ in order to handle deallocation of multiple referenced objects. In Smalltalk this idiom is not necessary since a garbage collector is available. This example shows that idioms are sometimes closely related to a programming language.

Architectural Frameworks

An architectural framework expresses a fundamental paradigm for structuring software. It provides a set of predefined subsystems as well as rules and guidelines for organizing the relationships between them. [Bus94a]

The architectural framework determines the basic structure of an application and can be viewed as a template for concrete software architectures. Examples of architectural frameworks are the structuring principles of the Pipes and Filters [Meu95], and the Model-View-Controller [Kra88] architectures.

The applicability and a class diagram, in OMT (Object Modelling Technique) [Rum91] notation, for the Pipes and Filter architectural framework is given in figure 2.6.



Applicability: The Pipes and Filters architecture is particularly useful for software systems where different subtasks can be identified and attached to separate components which: 1) are by themselves complete, 2) are independent from other components of the system, 3) only cooperate with each other by using the output of one component as input to another, and 4) are to be executed in a precisely determined sequential or parallel order.

Figure 2.6: *The Pipes and Filters architectural framework [Meu95]*

Object-Oriented Design Patterns

Descriptions of existing object-oriented design pattern are nearly always written in the *Alexandrian form*, or a variant of the Alexandrian form. The patterns in the Alexandrian form all try to capture the properties of Alexander's architectural (i.e building) patterns (see section 2.1).

The existing design pattern descriptions cover, at least, the following headings [Cop94a]. (More detailed design pattern descriptions are discussed in section 2.5).

- *The pattern name*, by which the pattern is called.
- *The problem* the pattern is trying to solve.
- *Context*, the context for which the pattern solves the problem.
- *Forces*, or trade-offs, this section makes clear the complexities of the problem.
- *Solution*, structure, behaviour etc. of the solutions described.
- *Examples*, a number of examples in which the pattern can be applied.
- *Force resolution*, what forces are not solved or what other patterns may be combined with this one.
- *Design rationale*, a description of where the pattern comes from, why it works and why experts use it.

Object-oriented design patterns, sometimes called Gamma patterns after the work of Erich Gamma [Gam92], have been discussed in [Gam93a], [Bus93a], [Gam94]. An introductory description of the set of Gamma patterns [Gam93a] can be found in appendix B and a description of Buschmann's set of design patterns [Bus94a] can be found in appendix C.

An object-oriented design pattern is a structure for capturing and expressing design experience. They identify, name and abstract common themes in object-oriented design. They preserve design information by capturing the intent behind a design. They identify classes, instances, their roles, collaborations and the distribution of responsibilities [Gam93a].

A design pattern is not a complete application or subsystem. It is a set of classes and instances that collaborate to solve a specific kind of design problem and they are not described in any programming language. This makes the design patterns reusable since they can be applied over and over again.

A design pattern consists of three major parts [Gam95]:

- An abstract description of a class or object collaboration and its structure.
- The issue in system design addressed by the abstract structure.
- The consequences of applying the abstract structure to a system's architecture.

As an example of an object-oriented design pattern, see the Prototype pattern [Gam95] in appendix D.

A more concise definition of an object-oriented design pattern is:

A design pattern describes a basic scheme for structuring subsystems and components of a software architecture as well as their relationships. It identifies, names and abstract a common design principle by describing its different parts, their collaboration and responsibilities [Gam93a].

The Gamma patterns have the following properties [Gam93a][Bus94a]:

- design experience is captured and described.
- a description of how to realize a particular functional or structural principle. The description comprises the patterns different parts and their collaborations and responsibilities.
- abstractions above the level of classes and objects are identified, named and specified.
- a technique for dealing with the complexity of software.
- a common vocabulary for designers for communicating, documenting and exploring design alternatives.
- application independence.
- a reusable design block in software development.
- addresses both functional and non-functional design aspects.

Metapatterns

If we examine the existing design patterns three sorts of patterns can be observed [Pre94b]:

- patterns relying on abstract coupling,
- patterns based on recursive structures,
- and other patterns, i.e patterns that do not make use of the above techniques.

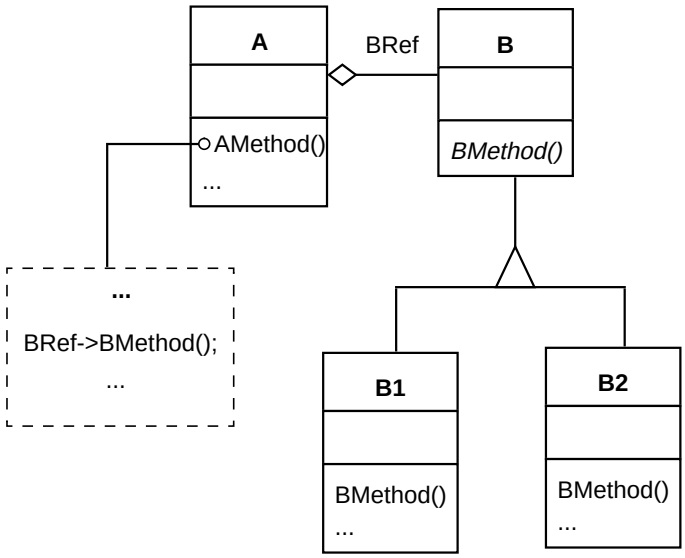


Figure 2.7: *Abstract coupling of class A and abstract class B*

Given a class A that maintains a reference to an abstract class B, class A is said to be abstractly coupled to B. This is called *abstract coupling*. See figure 2.7.

A pattern based on a recursive structure enables an object of a superclass to contain objects of the descendant classes. See figure 2.8.

If the concepts of abstract coupling and recursive structures are combined with each other and the notion of multiplicity is taken into account, this will end up in seven combinations which Pree [Pre94a] calls *metapatterns*.

When discussing metapatterns, the following classification of methods [Joh91] is used:

- template methods which are based on
- hook methods, which can be
  - abstract methods,
  - regular methods, or
  - template methods.

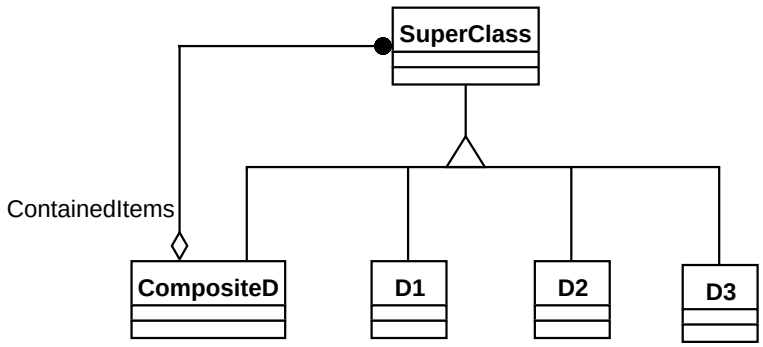


Figure 2.8: *A class diagram which enables recursive object structures*

A *template method*, T(), is a method which is implemented using elementary methods, called *hook methods*, H(). For some hook methods only the method interface can be defined, not the implementation. These methods are called *abstract methods*, A(). Hook methods that have an implementation are called *regular methods*, R(). See figure 2.9.

Analogously, a *template class* is a class which has a template method and a *hook class* is class which has a hook method. Note that a class can be both a template class and a hook class depending on the viewpoint taken. In figure 2.10 the classes are named T, H and TH, respectively. Here class TH repre-

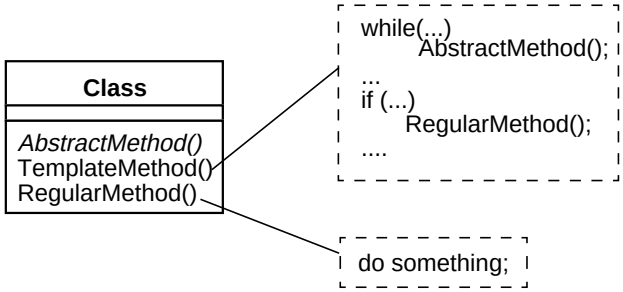


Figure 2.9: *Different kinds of methods [Pre94c]*

sent the degenerated case where the template and hook methods are in the same class.

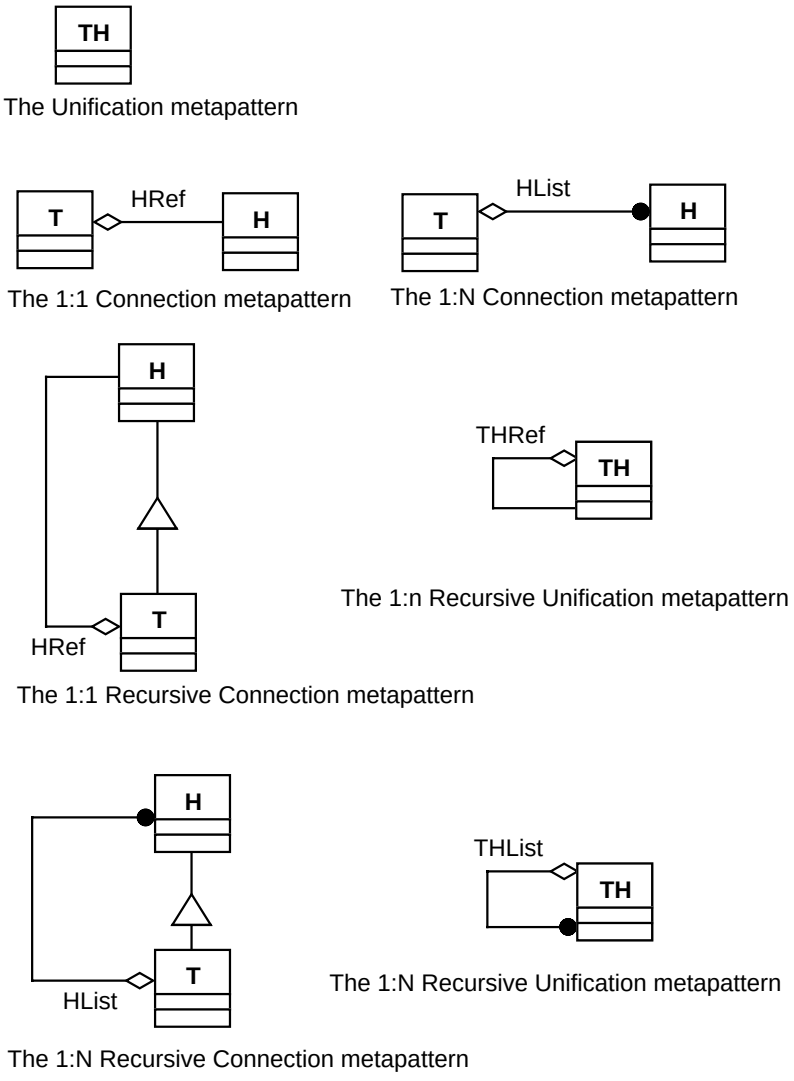


Figure 2.10: *The seven metapatterns [Pre94b]*

Note that the concept of template method and template class must not be confused with the C++ template construct [Ell90].

Meta-patterns can be viewed as the foundation for object-oriented design patterns since they capture the essence of existing design patterns. The seven meta-patterns that makes use of abstract coupling and recursive structures can be found in figure 2.10.

Meta-patterns can be used when developing tools that have to exhibit the parts of a design which are allowed to vary i.e where template methods call hook methods which are abstract methods. Meta-patterns may also be useful as a foundation when developing new design patterns.

2.4 Generative Patterns

Rather then describing the solution to a problem (non-generative patterns), *generative patterns* describe how to come to the solution. Generative patterns exist for a number of different phases and situations related to software development. There are generative design patterns [Bec94], motifs (i.e documentation patterns) [Joh92] and maintenance patterns [Foo94].

The structure of a set of generative patterns implies a non-cyclic graph where each generative pattern represents a vertex and the forward reference in the pattern represents the edge. If the structure of the generative pattern graph is performed carefully it will end up as a pattern language.

A *Generative Design Pattern* describes how to go from a design decision to a solution in the form of an object-oriented design pattern. Thus, a generative design pattern provides the rational for ending up in a design solution. The generative design patterns are very closely related to the object-oriented design patterns and can be seen as an extension to the Intent section of an object-oriented design pattern. The generative design patterns have the following format [Bec93]:

- *Preconditions*, what requirements and generative design patterns have to be satisfied before applying this one.
- *Problem*, what design problem is addressed by the generative pattern.
- *Constraints*, which are the conflicting forces to any solution to the problem.

- *Solution*, a description of the solution to the problem

The conditions under which the patterns apply, are the aspects most emphasized by generative design patterns.

Patterns that discuss maintenance aspects, *Maintenance Patterns*, are focused on restructuring techniques for object-oriented software, particularly frameworks, and are further described in chapter 3.

The *Motifs* is a kind of pattern developed for documentation of object-oriented applications and frameworks and may be used for learning and understanding a framework. They will be described in more detail in chapter 5.

Pattern Languages

A *pattern language in software development* is a structured collection of generative patterns, where the patterns are related to each other, that can be used to transform the needs and constraints into an architecture. Note, that the term pattern language used here is not a formal language such as a context-free language.

A good pattern language helps the designer to end up with a useful architecture and to avoid ”bad” architectures. Most existing pattern languages, [Cun94] for example, present the patterns in a top-down fashion. Thus allowing the early patterns to set the context for later ones. The disadvantage is that a lot of forward references occur in the description. If the pattern names are carefully chosen this will be no major problem. Pattern languages teach the designer how to do the design. A pattern language is a flexible construct so it is easy to integrate new patterns and thereby incorporate new design knowledge. Pattern languages exist for domains such as avionics control systems [Lea94b], distributed processing [DeB94] and input data error detection [Cun94]. However, the question, “Is the pattern language complete?”, seems impossible to answer for most of the domains. Maybe it is possible to give a positive answer for very small and narrow domains.

It is important to observe, as in [Cop94b], that when designers become experts they discard the patterns and pattern languages, as they no longer need a guide. They have become skilled and the designs have become intuitive for them. However, they will still use the vocabulary introduced. Designers at this experienced level will hopefully start to develop new patterns and pattern languages on a new level of abstraction.

2.5 Description Templates

Each object-oriented design pattern is described using a standard template to make it easier to understand. The template for the Gamma patterns is described in figure 2.11, (for a complete pattern description see appendix D where the Prototype pattern [Gam95] is described) and the Buschmann template [Bus94a] in figure 2.12.

The two templates are nearly identical and the main difference is that Buschmann has three additional sections:

- Methodology, the methodological steps for constructing the pattern are listed.
- Variants, possible variants of the design patterns are described.
- Dynamic behaviour, the dynamic behaviour of the pattern is illustrated

The Master-Slave pattern [Bus94b] is used to exemplify the additional sections. The pattern handles the computation of replicated services within a software system in order to achieve fault tolerance and robustness. It separates independent components which all provide the same service, (the slaves), from a component, (the master), being responsible for invoking them and for selecting a particular result for further use out of the ones they return [Bus94b].We give a condensed excerpt of the Methodology section for the Master-Slave pattern as an example, figure 2.13, together with the pattern's class diagram.

Variants. In some patterns it is possible to identify variants of the pattern. For example it might be possible to combine two classes into one to make the design simpler.

Dynamic behaviour. The dynamic behaviour of the objects is described using interaction diagrams to show the collaboration. See figure 2.14 for an example using the Master-Slave pattern.

2.6 Using Object-Oriented Design Patterns

To make it easier for a user of design patterns to decide which design pattern to use, it is necessary that the design patterns are easily identified and a suit-

DESIGN PATTERN Jurisdiction Characterization

What is the pattern's name and classification? The name should convey the pattern's essence succinctly. A good name is vital, as it will become part of the design vocabulary.

Intent What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Motivation A scenario in which the pattern is applicable, the particular design problem or issue the pattern addresses, and the class and object structures that address this issue. This information will help the reader understand the more abstract description of the pattern that follows.

Applicability What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can one recognise these situations?

Structure A graphical representation of the pattern using a notation based on the Object Modelling Technique (OMT) [Rum91], to which pseudo-code has been added.

Participants Describe the classes and/or objects participating in the design pattern and their responsibilities using CRC [Wir90a] conventions.

Collaborations Describe how the participants collaborate to carry out their responsibilities.

Consequences How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What does the pattern objectify? What aspect of system structure does it allow to be varied independently?

Implementation What pitfalls, hints, or techniques should one be aware of when implementing the pattern? Are there language specific issues?

Sample Code Include code fragments that illustrate how one might implement the pattern in a mainstream language such as C++ or Smalltalk.

Known Uses This section provides examples of the pattern's use in real systems. We try to include at least two examples from different domains.

Related Patterns What design patterns have closely related intent? What are the important differences? With which other patterns should this one be used?

Figure 2.11: *The Gamma patterns Description Template [Gam95]*

able selection process exists. A few different classification schemes for design patterns have seen the light and depending on the classification system used, the selection process may be different. The different pattern classification schemes and selection processes are described below.

Name The name of a pattern should convey its essence succinctly. A good name is vital, as it will become part of the design vocabulary.

Rationale The motivation for the pattern is presented or, in other words, what particular design issues it addresses.

Applicability A rule is presented stating when to use the pattern

Classification The pattern is classified according to the classification scheme, (Granularity, Functionality, Structural principles).

Diagram The participants and collaborators of the pattern are described as well as their responsibilities and relationships among each other.

Diagram A graphical representation of the pattern's structure is given.

Dynamic behaviour Where appropriate, the dynamic behaviour of a pattern is illustrated.

Methodology The methodological steps for constructing the pattern are listed.

Implementation Guidelines for implementing the pattern are presented. When appropriate, the guidelines are illustrated with pseudo-code and a concrete C++ code example.

Variants Possible variants of the patterns are listed and described

Examples Examples for the use of the pattern are presented

Discussion The constraints of applying the pattern are discussed

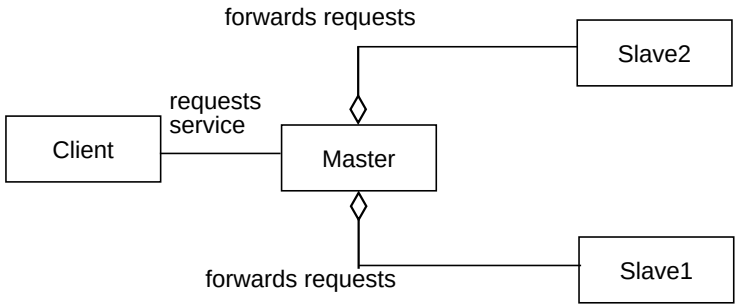
See also References to related patterns are given.

Figure 2.12: *The Buschmann patterns Description Template [Bus94a]*

2.7 Pattern Classification

2.7.1 The Reuse Error Classification

The simplest classification scheme uses the different kinds of reuse errors that a pattern solves as the classification principle. Since the design patterns are intended to be reusable it is important that the system developer knows what kind of reuse errors a design pattern solves. A *reuse error* occurs when a class is not reusable without major effort or modification. Experienced



Methodology. In the first step, every service which should be solved with the help of the Master-Slave pattern introduces a master component. These components must offer the requested service in its public interface. In the second step, a number of slave components are to be specified, each of them implementing the service provided by the master. The slave components have to be completely independent of each other. In the third step, the master component is to be associated with the slaves to which the concrete execution of the service the master provides is delegated. Finally, select which strategy is used by the master for selecting a particular result from those returned by its slaves. This result is to be returned by the master to its clients

Figure 2.13: *Methodology description for the Master-Slave pattern [Bus94b]*

software developers have tried to reuse classes for a long time and gained the experience which typical reuse errors that may occur. The following reuse errors have been identified [Gam93b]:

- Dependence on a particular implementation class when creating an object. Here a commitment to the implementation has been used instead of a commitment to a particular protocol.
- Dependence on particular operations.
- Dependence on operating environment. To achieve portability and applicability in different environments it is important to design away such hardware and software dependencies.

- Dependence on specific representation or implementation. The clients should not be aware of the representation, storing, or implementation of an object.
- Dependence on particular algorithms. Algorithms are likely to be optimized, extended and replaced during time. Algorithms that are likely to be changed should be isolated.
- Dependence on particular clients and inter-object relationships. I.e avoiding classes which are tightly coupled.
- Dependence on subclassing as an extension mechanism. The disadvantages of subclassing is that the internal details of the superclasses are made visible, which implies a commitment to a particular implementation. Inheritance relationships in a vendor class library are not likely to be changed. A more flexible solution is to use run-time composition instead of compile-time inheritance.

2.7.2 The Gamma Classification

The classification proposed by [Gam93a] classifies the design patterns according to two dimensions - *jurisdiction* and *characterization*. Jurisdiction is the domain over which a pattern applies. Here we distinguish between *class*, *object* and *compound jurisdiction*. The characterization

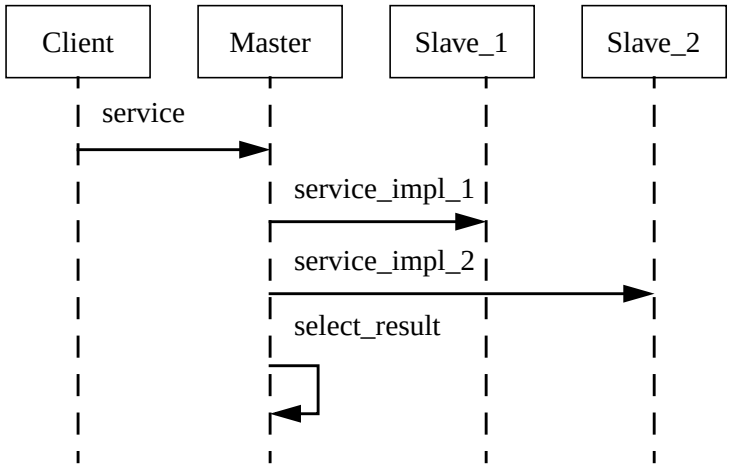


Figure 2.14: *Interaction Diagram for the Master-Slave Pattern {Bus94b]*

dimension can be either *creational*, *structural* or *behavioural*. Creational patterns concern object creation, structural patterns describe the composition of classes or objects. Finally, the behavioural patterns characterize the ways in which classes or objects interact.

Class jurisdiction deals with relationships between base classes and their subclasses. Object jurisdiction concerns relationships between peer objects whilst design patterns with compound jurisdiction deal with recursive object structures. In figure 2.15 we summarize the classification of the Gamma patterns.

Jurisdiction \ Characterization	Creational	Structural	Behavioural
Class	Factory Method	Adapter (class) Bridge (class)	Template Method
Object	Abstract Factory Prototype Singleton	Adapter (object) Bridge (object) Flyweight Glue Proxy	Chain of Responsibility Command Iterator (object) Mediator Memento Observer State Strategy
Compound	Builder	Composite Wrapper	Iterator (compound) Walker

Figure 2.15: *Classification of the Gamma Patterns [Gam93a]*

This classification of the design patterns is however, too broad to be useful for a software designer in a given design situation.

2.7.3 The Buschmann Classification

A third and for the designer more appropriate categorisation of object-oriented design patterns is the one by Buschmann et al. [Bus94a].

They claim that "a*ll patterns together form a system of building blocks that can be used to construct and compose software architectures*". In order to be useful for software development Buschmann et al. state that the system must fulfil a number of conditions and requirements.

- A classification of the patterns in order to make it easier to select a given pattern in a given design situation.
- All the patterns must be described in a uniform way. The description must cover aspects regarding characterization, detailed description, implementation, selection and comparison with other patterns.
- Composition of patterns - which patterns are possible to combine. This is to avoid bad designs and give hints about reusable combinations.
- Supporting the evolution of the system. Some patterns will be born and others die.

The Buschmann classification scheme comprises of three principles, *Granularity*, *Functionality* and *Structural principles*. The Granularity category is argued as necessary since development of a software system covers various levels of abstraction, from basic structures of the application to concrete implementation issues. The levels of granularity depicted are:

- Architectural Frameworks
- Design Patterns
- Idioms

Every software architecture is built to an overall structuring principle. These principles are described by the architectural frameworks.

An architectural framework expresses a fundamental paradigm for structuring software. It provides a set of predefined subsystems as well as rules and guidelines for organizing the relationships between them [Bus94a].

The selection of a particular architectural framework for a software system is a fundamental design decision. The architectural framework determines the basic structure for an application and can be viewed as a template for concrete software architectures.

The smaller software architectural units that a software architecture consists of are described by design patterns. Design patterns can be combined into new, composed, design patterns [Zim94b].

The finest granularity level, idioms, deals with realisation and implementation of particular design issues. Idioms describes how to implement particular components, (part of) their functionality, or their relationships to other components within a given design.

Idioms are the lowest granularity level of patterns and are related to a specific programming language. They comprise aspects of both design and implementation for a particular structure.[Bus94a]

The Functionality category captures a specific kind of functionality for the design pattern. It can be further divided into the following four sub-categories [Bus94a]:

- Creation of objects. The creation of recursive or aggregate object structures.
- Guiding communication between objects. How a set of objects collaborate to communicate.
- Access to objects. Without violating the encapsulation of state and behaviour, accessing the services and states of objects.
- Organizing the computation of complex tasks. Specify how the responsibilities are distributed among the cooperating objects in order to solve a more complex task.

The third category, Structural principles, is used by the patterns to realize their functionality. The structural principles are [Bus94a]:

- Abstraction. A pattern provides an abstract or generalized view upon a particular and often complex entity or task within a software system.
- Encapsulation. A pattern encapsulates details of a particular object, component, or service in order to remove dependencies on it from its clients or to protect such details from access.
- Separation of concerns. A pattern factors out specific responsibilities into separate objects or components in order to solve a particular task or to provide a certain service.
- Coupling and Cohesion. A pattern removes or relaxes the structural and communicational relationships and dependencies between otherwise strongly coupled objects.

Applying these three dimensions, Granularity, Functionality and Structural principles results in the following classification scheme, figure 2.16 [Bus94a]. The patterns in the classification scheme constitute the pattern set

by Buschmann. For an introductory description of the Buschmann patterns, see appendix C

Structural\ Functional	Creation	Access	Communication	Organizing Complex tasks
Abstraction	Abstract Factory	Controller-Command View-Representation	Forward-Receiver Client-Broker-Server Client-Trader-Server	Actor-Supplier Actor-Agent-Supplier Composite-Part Wrapper
Encapsulation		Controller-Command View-Representation	Forward-Receiver Client-Broker-Server Client-Trader-Server	Composite-Part Wrapper
Separation of Concern	Abstract Factory	Proxy-Original Subject-Snapshot Collection-Iterator	Forward-Receiver Client-Broker-Server Client-Trader-Server	Actor-Supplier Actor-Agent-Supplier Composite-Part Master-Slave Worker-Repository

Figure 2.16: *Classification of the Buschmann Patterns [Bus94a]*

Structural\ Functional	Creation	Access	Communica-tion	Organizing Complex tasks
Coupling & Cohesion	Abstract Factory	Envelop-Letter Bridge Handle-Body Proxy-Original Subject-Snap-shot Controller-Com-mand View-Represen-tation Producer-Con-sumer Producer-Repos-itory-Consumer Producer-Sensor-Consumer	Mediator-Worker Adapter Sensor Display Publisher-Sub-scriber Client-Broker-Server Client-Trader-Server	Actor-Supplier Actor-Agent-Supplier

Figure 2.16: *Classification of the Buschmann Patterns [Bus94a]*

2.7.4 The Zimmer Classification

A fourth way of classifying design patterns is presented by Zimmer [Zim94b]. He investigated the set of Gamma patterns [Gam93b] and their relationships to each other. The objectification of behaviour was observed as a major common theme among a number of design patterns and a new design pattern, Objectifier, was created. It is a generalization of those patterns that objectifies the different kinds of behaviour. The design patterns related to the Objectifier design pattern are described in figure 2.17.

The construction of the Objectifier design pattern and the relationships between the Gamma patterns (including Objectifier) form the base for a classification of the patterns in three semantic layers [Zim94b], see figure 2.18.

- Basic design patterns and techniques
- Patterns for typical software problems
- Application domain-specific design patterns

Design Pattern	Objectified Behaviour
Bridge	Implementation of some abstraction
Builder	Creation/Representation of objects
Command	Command dependent behaviour
Iterator	Traversal of object structures
Observer	Context dependant behaviour
State	State dependent behaviour
Strategy	(Complex) Algorithm
Visitor	Type dependent behaviour (types of single objects in compound structure)

Figure 2.17: *Behavioural Design Patterns Generalized by Objectifier [Zim94b]*

The basic design pattern layer comprises those patterns that are used by the design patterns at the other two layers. The problem addressed by the patterns on this level occurs again and again when developing software. The intermediate layer patterns are only used by the application specific layer or patterns in the same layer. Typical software problems addressed by the patterns in this layer are: object creation, traversing of objects, objectification of an operation etc. In the top-level layer, application domain specific patterns, the patterns are the most specific ones. The set of Gamma patterns only include one domain specific pattern, Interpreter, which parses some input, e.g simple languages. Using this layered classification for the design

patterns reduces the number of design patterns that are actual in a design situation on a specific semantic level.

Summary of the Classification Approaches

In his classification Buschmann focuses on the categories that support the choice of a specific pattern in a given design situation. This is in contrast to

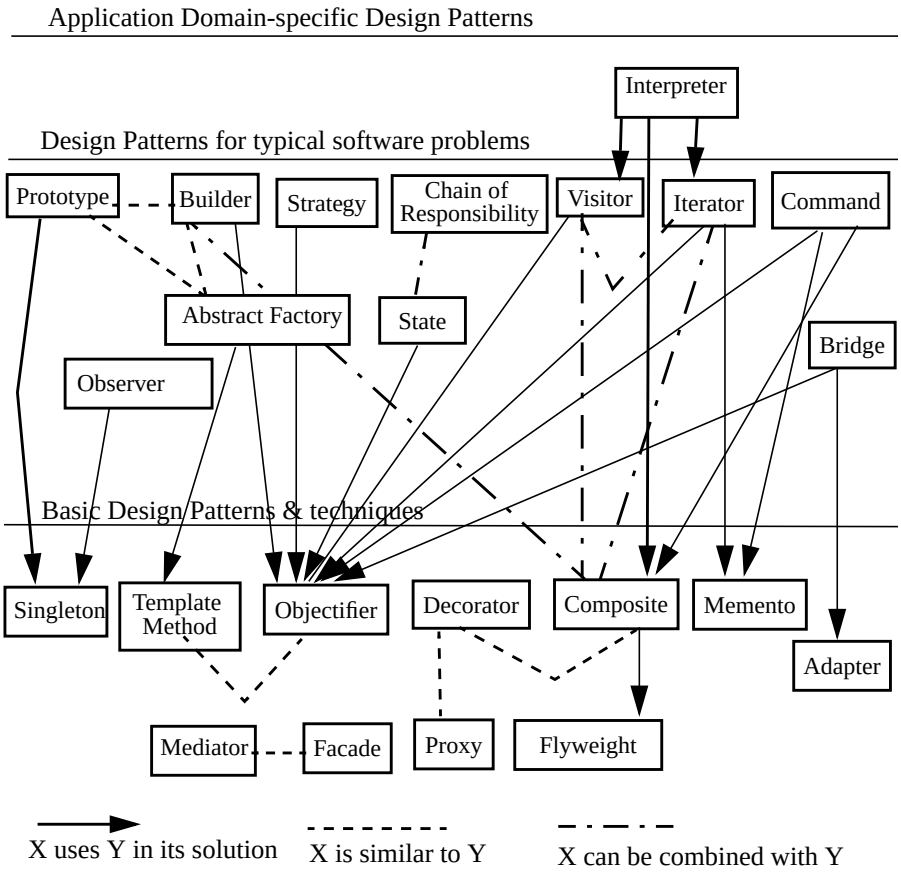


Figure 2.18: *The Layered Classification of the Gamma Patterns [Zim94b]*

the classification of Gamma et al., which classifies the design patterns on major characteristics that are useful to systematize the patterns. The Buschmann approach seems to make it easier to talk about patterns and to understand them.

2.8 Pattern Selection

2.8.1 Selection based on the Reuse Error Classification

One approach to the selection of an appropriate design pattern is to answer the question “Which kind of new requirements or changes has the design to meet in the future?“. Since the design patterns allow some aspects of a system structure to be varied independently of its client they can be mapped to the set of reuse errors (causes of redesign), see figure 2.19. If the question can be answered, there is just a small set of design patterns to select from.

Causes of Redesign	Design Pattern
Dependence on a particular implementation class when creating an object	Abstract Factory Builder Factory Method Prototype
Dependence on particular operations.	Adapter Chain of Responsibility Command Visitor
Dependence on operating environment.	Bridge

Figure 2.19: *Reuse Error (Causes of redesign) versus Design Pattern*

Causes of Redesign	Design Pattern
Dependence on specific representation or implementation.	Abstract Factory Adapter Bridge Composite Facade Memento Proxy
Dependence on particular algorithms.	Builder Iterator Strategy Template Method Visitor
Dependence on particular clients and inter-object relationships.	Bridge Facade Mediator
Dependence on subclassing as extension mechanism.	------

Figure 2.19: *Reuse Error (Causes of redesign) versus Design Pattern*

2.8.2 Selection based on the Buschmann Classification

If the patterns are classified according to Buschmann's classification scheme, the following steps can be suitable when selecting a design pattern [Bus94a]:

- Which granularity is required for the pattern? It is usually simple to decide if the pattern will serve as a concrete implementation to a design problem or, as a scheme for structuring a component or serve as a basic structure for a whole application/subsystem.
- Select the required functionality. In a given design situation it is often clear which kind of functionality is to be considered. A problem that can arise is that it is necessary to combine several functional aspects within a single structure. Then the choice is to select a pattern that captures all the needed functionality or a set of patterns that can be combined together, that offers the needed functionality.

- Which are the desired structural principles? This is the most difficult question to answer since there is often more then one structural solution to choose for realizing a functionality. The division of the structural principles into four different types forces the developer to think about which one is the most important in this situation.

These steps, together with the classification, serve as a guide to the designers when searching for a suitable pattern.

2.8.3 Summary of Selection Approaches

The classification by Gamma et al. into creational, structural and behavioural patterns is relative broad. This sometimes makes it difficult to select the most suitable pattern in the category. It can, however, be useful in combination with other classification systems, for example the one by Zimmer [Zim94b].

Pattern selection based on the Reuse error classification requires that the developer has a good understanding of the patterns that are classified. This knowledge can be difficult to maintain if the number of patterns is high.

The classification by Buschmann seems to be the most appropriate one since it is the most detailed and the selection process significantly reduces the set of patterns that can be actual in a given situation. The level of details in the classification system implies that the designer must have a clear view of which kind of design problems have to be solved.

2.9 Experiences Reported

Only a few cases of experiences resulting from the use of design patterns in industrial development of object-oriented software have been reported [Zim94a], [Laj94], [Bus94a], [Sch95a],[Sch95b].

In [Laj94], design patterns were used as key components for the reuse of design when developing an environment for business modelling and reengi-

neering using the ET++ framework [Wei89], [Wei94]. The experiences proved that design patterns are useful as

- a guiding blueprint to introduce new designs
- a reference to ensure that hierarchy restructuring and class redesign do not affect behavioural collaborations
- a means to understand the framework

One problem observed was pattern identification (i.e which pattern is involved in a specific class) when the developers used patterns for the understanding of and reference to the designs. Their suggested solution is that new implementations of a design include an identification to the guiding design pattern in the component classes. Another improvement made by Lajoie et al. [Laj94] was to include a Contract Example section in the design pattern description. A contract is a construction for explicitly specifying interactions among groups of objects [Hel90]. The use of the contract notion applied best to behavioural design patterns. Lajoie also developed a contract template to support the use of contracts in the development, with the following headings [Laj94]:

- Pertinent Design Patterns, which abstract design patterns are involved in the actual design.
- Participants, which classes are involved in the contract.
- Participants Pseudo Code details, descriptions of class data members and operations.
- Invariants, definition of the invariants the contract's participants will maintain
- Instantiation,i.e. specification of preconditions on participants to establish the contract and methods required for the instantiation of the contract.

Buschmann [Bus94a] reports that the pattern approach has been applied in automation, telecommunication and business domains. All projects gave the positive response that patterns help to develop a system with clearly defined architectures with defined properties e.g changeability and portability.

Design patterns have also been used to reorganize an existing object-oriented application [Zim94a]. Zimmer used design patterns as targets for the reorganization of an existing hypermedia application. The reasons for

selecting design patterns as a major resource for the reorganisation were the promises that design patterns:

- improve the design quality and comprehensibility by reusing well-designed structures,
- provide a common vocabulary for discussions and documentation,
- raise the granularity level of design from classes, methods etc. to larger building blocks.

The conclusions [Zim94a] were that the design pattern approach fulfilled the promises. The major advantage observed was that design patterns introduce a common vocabulary for the designers in discussions and documentation.

In [Sch95a], [Sch95b] Schmidt reports experiences gained through applying design patterns in large-scale software development processes in domains such as network monitoring and image transportation in ATM networks. We can briefly classify the lessons learned in three categories. Experiences that are relevant for the software development organisation, experiences related to the software development process and observations related to the pattern concept.

From the viewpoint of the software development organisation the following experiences are reported in [Sch95a], [Sch95b]:

- Patterns explicitly capture knowledge that experienced developers already understand implicitly. The organization will benefit from this since the experts' knowledge of advanced architectural concepts is documented, discussed and reasoned systematically, thereby preserve intellectual capital to the organization.
- Focus on developing patterns that are strategic to the domain and reuse existing tactical patterns. Existing sets of design patterns [Gam95], Bus93b] are domain-independent, and an organization will benefit from reusing these tactical design patterns. Thereby, focus the effort to develop strategic, domain-specific, design patterns that have major impact on the software architectures.
- Pattern descriptions explicitly record engineering trade-offs and design alternatives. Since patterns explicitly enumerate consequences and implementation trade-offs they become a sort of history that describes why certain design choices were selected and others rejected. This history preserves domain knowledge in the organisation.

- Institutionalize rewards for developing patterns. This activity handles the problems that normally arises when new methods and techniques are being considered as threats to traditional techniques and expertise. The staff that develops new patterns often consider these as a competitive advantage and are reluctant to share their knowledge.
- Integrating patterns into a software development process is a human-intensive activity. This kind of reuse technology costs, as do other reuse technologies. Since very few professionals were equally skilled in both domain experience and the ability to capture general properties of patterns, pattern review sessions were introduced as a means to facilitate reuse of design patterns.

The experiences described in [Sch95a], [Sch95b] that may be relevant to the software development process where:

- Patterns improve communication within and across software development teams. This since they provide a common vocabulary for discussing architecture and thereby raise the abstraction level of the discussions.
- Directly involve pattern authors with application developers and domain experts. If the pattern developers were separated from the application developers, the pattern became too abstract and general and did not capture the actual domain requirements. To circumvent this problem the pattern review session was used to make the patterns more successful.
- Patterns may lead developers to think they know more about the solution to a problem then they actually do. The developers often overestimated the power of the design patterns and were not aware of the problems and effort needed to implement the patterns. The only way to handle this problem was to emphasise that patterns complement, not substitute, design and implementation skills.
- Resist the temptation to recast everything as a pattern. Initially, patterns were developed which may not be described as patterns, for example, building linked lists. This pattern overload quickly became counter productive.

The experiences that concern the pattern, its description and usefulness in [Sch95a], [Sch95b] are:

- Pattern descriptions should contain concrete examples. If the patterns are too abstract, i.e lack source code examples, they become difficult to understand. This since they abstract properties of good design and are not limited to one implementation. If source code accompanied the patterns the developers got a better understanding of how the patterns worked.
- Pattern names should be chosen carefully and used consistently. The vocabulary of the pattern sets must be unambiguous to avoid misunderstanding. However, very short and distinct names, such as Reactor [Sch94a] and Iterator, may be difficult to understand. To handle this problem a one line description/sentence alias was given for each pattern. Reactor for example had the alias “dispatch handlers automatically when events occur from multiple sources”.
- Useful patterns arise from practical experiences. It is necessary to work together with domain experts when looking for new patterns. This implies that patterns are discovered in a ”bottom-up” fashion. If patterns were to be invented in a “top-down” way it could be a sign of pattern overload.
- Patterns are validated by experience rather then by testing. The most effective way to see if a pattern fulfils its promise is to perform periodic reviews. At the reviews people present new patterns. The strengths and weaknesses of each pattern are discussed, experiences from the review members are added as well as improvements in content and style. This implies that patterns can not be validated in the traditional sense of testing software.

The experiences reported all advocate that design patterns fulfil the promises as usable vehicles for communicating designs and design issues. They also confirmed that patterns contribute to the quality of both the software and the software process. A few drawbacks have been reported, most of them belonging to well-known problems in technology transfer situations and reuse initiatives, e.g too high expectations and difficulties in giving away reuse assets, respectively.

2.10 Discussion

The following research areas can be identified in the domain of patterns:

- Combinations of patterns. Does there exist useful combinations of patterns that solve larger design problems than those solved by existing patterns? One example of a pattern that is a combination pattern is the Reactor pattern [Sch94b], which dispatches handlers automatically when events occur from multiple sources. The Reactor pattern makes use of the Observer, Facade, Template Method [Gam95] and Active Object [Sch95c] patterns.
- Development of domain specific patterns and the integration of patterns into pattern languages. Existing sets of design patterns present solutions to common design problems but there will be a need to develop related patterns and pattern languages for specific application domains such as, client-server programming, distributed computing and real-time systems.
- Techniques for discovering and documenting patterns. Existing patterns have been developed from individual developers experiences through discussions with their colleagues. Nearly all patterns, are documented in the Alexandrian form. Does there exist other, better, ways to document patterns? Does there exist any kind of techniques that will make it easier to capture and document new patterns?
- Integration of patterns into existing methods. Patterns do not replace existing object-oriented methods and software process models [Sch95b]. They should be viewed as a complement to the methods and models. In the analysis and design phases patterns help developers in selecting software architectures, large or small, that have proven to be useful. In the implementation and maintenance phases patterns help to describe the strategic choices made in the development. These strategic choices are made at a level higher than ordinary source code and object models. Therefore, a movement of integration of patterns into existing methods and process models is foreseen.

3 Framework Development

3.1 Introduction

Designing software is hard and design of reusable software is even harder. This has been observed and quoted many times, also in object-oriented software development [Opd90], [Opd92], [Bec94]. An object-oriented framework is a kind of reusable software architecture comprising both design and code. The concept of frameworks is expected to raise the productivity of the software engineers.

What is an object-oriented framework? Early examples of the framework concept can be found in literature that has its origins in the Smalltalk environment [Gol84] and Apple Inc. [Sch86]. The Smalltalk-80 user interface framework, Model-View-Controller (MVC), was perhaps the first widely used framework. MVC was used to develop user interfaces and divided the user interface into three parts; models, views and controllers. The model is an application object which is independent of the user interface, the view manages a region of the display and the controller converts user events into operations on its model and view. Apple Inc. developed the MacApp user interface framework which was designed for implementing Macintosh applications. Frameworks are not limited to user interface framework but exist for other domains. For example, operating systems [Rus90] and fire-alarm systems [Kar95]. We see that a framework tries to capture the general flow of control in an application which can then be further specialised for a specific application.

A number of different and similar definitions of frameworks exist. Deutsch [Deu89] states that

"a framework binds certain choices about state partitioning and control flow; the (re)user (of the framework) completes or extends the framework to produce an actual application".

So the work for an application developer will be to extend an existing reusable software architecture.

The definition by Johnson and Foote [Joh88] is the most well-known one:

A framework is a set of classes that embodies an abstract design for solutions to a family of related problems.

A similar definition was made in [Joh91]:

A framework is a set of objects that collaborate to carry out a set of responsibilities for an application or subsystem domain.

In [Fir94] the following is used as a definition:

A framework is a significant collection of collaborating classes that capture both the small-scale patterns and major mechanisms that implement common requirements and design in a specific application domain.

Based on this definitions we define an object-oriented framework as:

A (generative) architecture designed for maximum reuse, represented as a collective set of abstract and concrete classes; encapsulated potential behaviour for subclassed specializations.

The framework can be seen as generative since it is intended to be used as the foundation for the development of a number of applications in the application domain captured by the framework. This is in contrast to the normal way of developing an object-oriented application. The difference in the development process is outlined in figure 3.1.

Given this definition of an object-oriented framework, the differences compared to other concepts such as:

- an object-oriented design pattern
- a pattern language
- a class library
- an architectural framework
- an ordinary object-oriented application

are discussed.

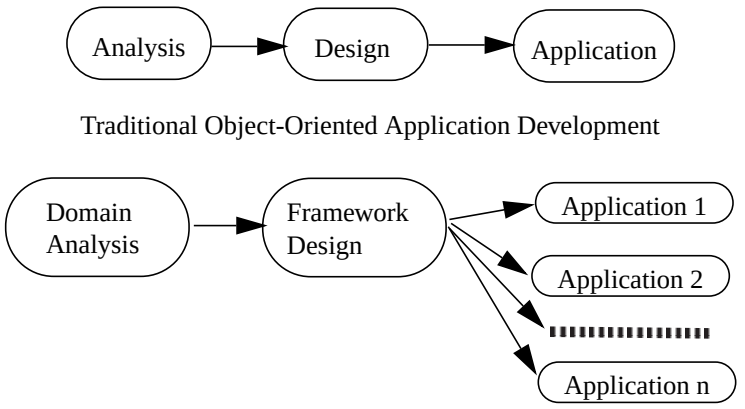


Figure 3.1: *Traditional versus Framework-based Application Development*

An *object-oriented design pattern* differs from a framework in three ways [Gam95]:

- The design patterns are more abstract than a framework. Frameworks are embodied in code. This is not the case for design patterns, where only examples of the design patterns are embodied in code. The design patterns also describe the intent, trade-offs, and consequences of a design, which is not the case for frameworks.
- Design patterns are smaller architectures than frameworks. A framework can contain a number of design patterns, but the opposite is never possible. Thus, the design patterns have no major impact of the application's architecture.
- Frameworks are more specialized than design patterns. Frameworks are always related to a specific application domain, whereas design patterns are general and can be applied in any application domain.

Pattern languages differ from frameworks in the way that a pattern language describes how to make a design where an object-oriented framework is a design. Or described in another way, a pattern language instructs you how to do it, while a framework does it for you. Pattern languages complement a

framework since they can teach software engineers how to use a framework, and describe why it was designed the way it was.

A *class library* is a set of related classes that provides general-purpose functionality. The classes in a class library are often not related to a *specific application domain*, which is the case for classes in an object-oriented framework. The functionality typically covered by class libraries are, for example, collection classes (lists, stacks, sets etc.) and i/o-handling (the C++ iostream library [Tea93]). The difference between a class library and a framework is the degree of reuse and its *impact on the architecture* of the application. Since the framework covers the functionality of the application domain the main architecture is captured by the framework. Thereby, the framework has a major impact on the architecture of the framework-based application developed. The impact on the architecture can be also be identified by the fact that a class in a class library is reused individually and a class in a framework is reused together with other classes in the framework to solve a specific instance of a certain problem.

Architectural frameworks differ from a framework in the way that an object-oriented framework is the implementation of a common architecture for a family of applications. This is in contrast to architectural frameworks, which are language independent. An object-oriented framework is often built using an architectural framework as the foundation for the framework's architecture.

An *object-oriented application* differs from a framework in the way that the application describes a complete executable program that satisfies a requirement specification. This is in contrast to the framework that captures the functionality of the application (and similar applications in the domain), but is not executable because it does not cover the behaviour in the specific application case.

Monolithic versus Fine-Grained Frameworks

Existing literature, [Deu89], [Joh88], [Laj94], [Pre94a], about frameworks describes experiences gained from using monolithic frameworks, i.e object-oriented software architectures for some specific domain. Another approach that can be found in [Cot95] is to identify, small, fine-grained frameworks that collaborate with each other. An application can then be "composed of" a number of these fine-grained frameworks, as depicted in figure 3.2.

Development of fine-grained frameworks will be similar to the development of monolithic frameworks. One major difference is that the fine-grained framework has to be designed for integration because the coverage of the framework is not a complete domain.

3.2 Advantages and Disadvantages

A framework can be seen as a set of cooperating classes that comprises a reusable foundation for a specific application domain. Note the use of application domain instead of application, this since the concept of framework can be applied in finer granularity.

The main goals of a framework are:

- To keep the organisation's knowledge about the application domain within the organisation. This is possible since the organisation has a design to start from, the framework, when developing a new application in the domain
- Minimize the amount of code needed to implement similar applications. This is possible since the common abstractions for the applications are captured in the framework, which reduces the fraction of new code to be developed.
- Optimize generality and leverage. The framework users have to tailor the framework to the applications's specific needs, for example,

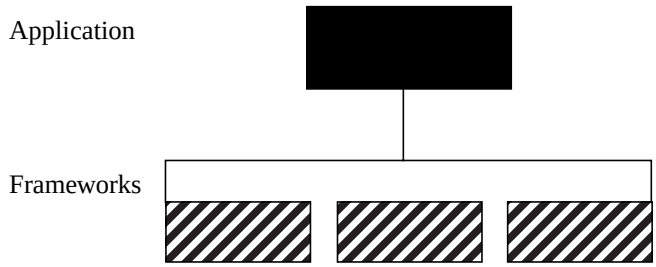


Figure 3.2: *The decomposition of an application based on fine-grained frameworks*

through subclassing. For leverage, it is necessary that the frameworks include a set of predefined subclasses that the framework user can use at once. Thus if the framework is too general, it will not support the framework user and, if it is not general, it will be useable in very few situations.

As in every situation where there are conflicts between the general and the specific, it is possible to identify strengths and weaknesses. Object-oriented frameworks have the following strengths:

- A real decrease in lines of code that have to be developed if the application's required functionality and the functionality captured by the framework are similar to a high degree.
- The framework's code is already written and debugged.
- The framework offers reuse of design and not only code. This reuse of design made by others may transfer design knowledge and experience to the framework user.
- Efficiency aspects of the software can be deployed by the framework developers which can not normally be made within a normal application project budget and schedule.

Additional benefits identified are [Cot95]:

- More focus on area of expertise, less focus on areas of system compatibility. This is true for frameworks in the domains of systems software, which is the case in [Cot95].
- Improved maintenance. When an error is corrected in the framework, the same error in software derived from the framework is corrected.

Potential weaknesses of an object-oriented framework are:

- It is difficult to develop a good framework. Experience in the application domain is necessary when building the framework.
- The documentation of the framework, since it is crucial to the framework user. If the frameworks are not supported by the necessary user documentation, they are not likely to be used. A further discussion on the documentation aspects of frameworks is found in chapter 5.
- Backward compatibility can be difficult to maintain. This since the frameworks evolve and become more mature over time and the applications built on the frameworks must evolve with it.

- The debugging process can be complicated because it is difficult to distinguish bugs in the framework from bugs in the application code. If the bugs are in the framework, it can be impossible for the framework user to fix the bug.
- The generality and flexibility of the framework may work against its efficiency in a particular application.

Even with these disadvantages frameworks will be important, especially in larger object-oriented applications. These will consist of a number of class libraries and frameworks layered on each other and cooperating with each other.

3.3 Characterization of Frameworks

Object-oriented frameworks can be characterised by different dimensions. The most important dimensions are; the problem domain the framework addresses, the internal structure of the framework and how the framework is intended to be used [Tal94].

Framework domains

So called *application frameworks* are frameworks that cover functionality that can be applied to different domains. Examples of application frameworks are frameworks for graphical user interfaces [Wei94].

Domain frameworks capture knowledge and expertise in a particular problem domain. Frameworks for manufacturing control [Sch95d] and multimedia are examples of domain frameworks.

Finally, the *support frameworks* are frameworks that offer low-level system services such as device drivers [And94] and file access.

Support frameworks are typical cases of fine-grained frameworks, whereas application and domain frameworks often are monolithic frameworks.

Framework structures

If the framework's internal structure is described it can make it easier to understand the behaviour of the framework. The internal structure of a framework is related to the concepts of software architectures. In [Gar93] a

number of common software architectures have been identified. Buschmann [Bus94a] calls these “architectural frameworks” since they have been designed in a way that captures the main structure of an object-oriented software architecture. The overall principles for the internal structure of an object-oriented framework are described by its architectural framework. The described architectural frameworks [Bus94a] are:

- The Layered architectural framework
- The Pipes and Filters architectural framework
- The Model-View-Controller architectural framework
- The Presentation-Abstraction-Controller architectural framework
- The Reflective architectural framework
- The Microkernel architectural framework
- The Blackboard architectural framework
- The Broker architectural framework

The *Layered* architectural framework helps to structure applications that can be decomposed into groups of subtasks with different levels of abstraction.

The *Pipes and Filters* architectural framework can be used to structure applications that can be divided into several completely independent subtasks which are to be performed in a strongly determined sequential or parallel order.

The *Model-View-Controller* architectural framework defines a structure for interactive applications that decouples their user interface from their functional core.

The *Presentation-Abstraction-Controller* architectural framework is suitable to structure software systems which are highly interactive with human users, enable multiple controls and presentations of their abstraction models of the system and can be composed out of subfunctions that are independent of each other.

The *Reflective* architectural framework is applicable to applications which need to consider a future adaptation to changing environments, technology, and requirements, but without an explicit modification of their structure and implementation.

The *Microkernel* architectural framework is suitable for software systems which have to provide different views upon their functionality and which

often have to be adapted to new systems requirements, i.e operating systems.

The *Blackboard* architectural framework helps to structure complex applications that involve several specialized subsystems for different domains. These subsystems closely cooperate to build solutions to the problems to be solved.

The *Broker* architectural framework structures distributed software systems in which decoupled components interact via remote operation calls in a client-server fashion.

Using one of the architectural frameworks as the main structuring principle for a framework, means that the classes in that architectural framework are good candidates for applying object-oriented design patterns, and for developing the framework.

Using a Framework

An object-oriented framework can be used in two ways. Either the framework user derives new classes from it or instantiates and combines existing classes [Ada95].

The first approach is called *architecture-driven* or *inheritance-focused* frameworks [Ada95]. The main approach is to develop applications relying on the inheritance mechanism. Framework users make their adaptation of the framework through deriving classes and overriding operations. A problem with frameworks that are architecture-driven is that it can be difficult to adapt them because the framework users have to provide the implementation of the behaviour by themselves, which implies a large amount of code to be written.

The second approach, instantiation and combination, is referred to as *data-driven* or *composition-focused* frameworks [Ada95]. Adapting the frameworks to the specific needs of the application means relaying on object composition. How the objects can be combined are described by the framework, but what the framework does depends on what objects the framework user passes into the framework. A framework that is data-driven is normally easy to use but limiting.

To handle the problems associated with using inheritance-focused and composition focused frameworks, a suitable approach can be to provide the

framework with an architecture-driven base and a data-driven layer so it is both extendable and easy to use [Tal94].

3.4 The Framework Development Process

The development process of a framework depends on the experience of the organisation in the problem domain the framework addresses. A more experienced organisation can select a more advanced development process since they will have fewer problems with the problem domain. A few possible processes for framework development have been proposed [Wil93], [Joh93].

Development Process based on Application Experiences

A pragmatic framework development approach is described in [Wil93]. First develop *n*, say two, applications in the problem domain. When they work correctly the first iteration in the process begins, figure 3.3. Identify the common features in both applications and extract these into a framework. To evaluate whether the extracted features are the right ones, redevelop the two (*n*) applications based on the framework. This should be quite easy if the commonalities identified are the right ones. If it is not easy to redevelop the applications, rewrite the framework as necessary and use the experiences achieved to develop the next version of the framework. This is iteration number two and all the following iterations in figure 3.3. Based on the new version of the framework, new applications can be developed. The experiences gained through developing new applications are also used as inputs for scheduled maintenance of the framework. The whole process is outlined in figure 3.3

Development Process based on Domain Analysis

A more sophisticated approach for the development of a framework can also be found in [Wil93]. The development process is outlined in figure 3.4. The first activity is to analyse the problem domain to identify and understand well-known abstractions in the domain. Analysing the domain requires analysing existing applications (which is a difficult task to perform) and it is only possible if the organisation has applications developed in the domain. The analysis of existing applications will also take a large portion of the

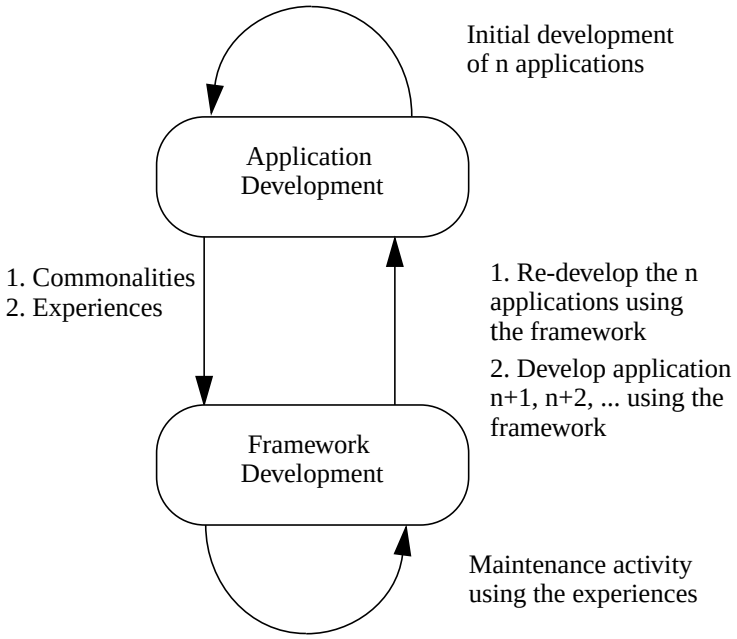


Figure 3.3: *Framework Development Process based on Application Experiences*

budget. After the abstractions have been identified, develop the framework together with a test application, (the Use arrow in figure 3.4), then modify the framework if necessary. Next, develop a second application based on the framework. Modify the framework if necessary and revise the first application so it works with the changes introduced in the framework. Let the framework evolve, through scheduled maintenance activities, while more applications are being developed based on the framework. A similar approach has been proposed by Johnson in [Joh93].

Development Process Utilizing Design Patterns

A pragmatic design pattern focused process for development of a framework is possible to identify as depicted in figure 3.5. First, develop an application

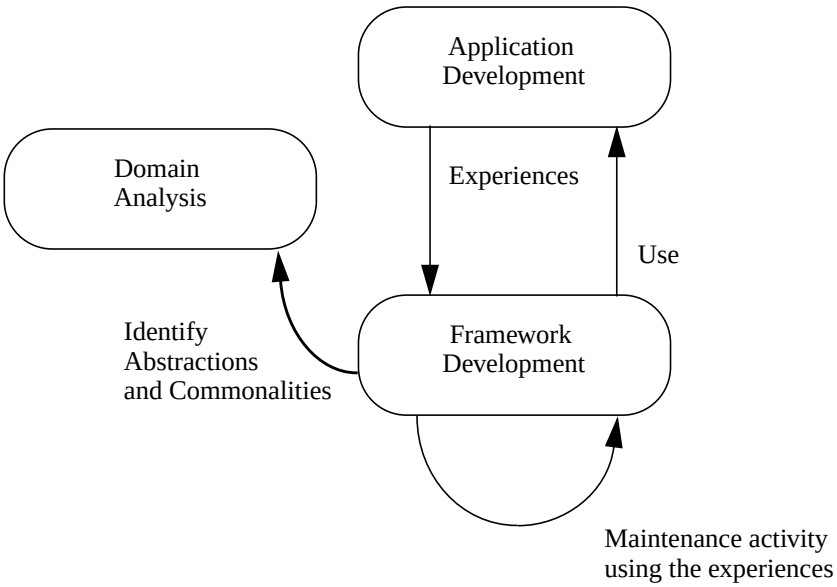


Figure 3.4: *Framework Development Process based on Domain Analysis*

in the problem domain. Second, establish and train the staff in a standard set of design patterns. Take the application and systematically apply the design patterns to create the framework. Then the iteration between application development and framework can take place. Also in this process, scheduled maintenance activities of the framework exist.

The General Framework Development Process

The common elements of the processes for development of a framework are outlined in figure 3.6:

- Analysis of the problem domain. This is performed systematically or through development of one or a few applications in the domain and the key abstractions are identified.
- The first version of the framework is developed utilizing the key abstractions found.

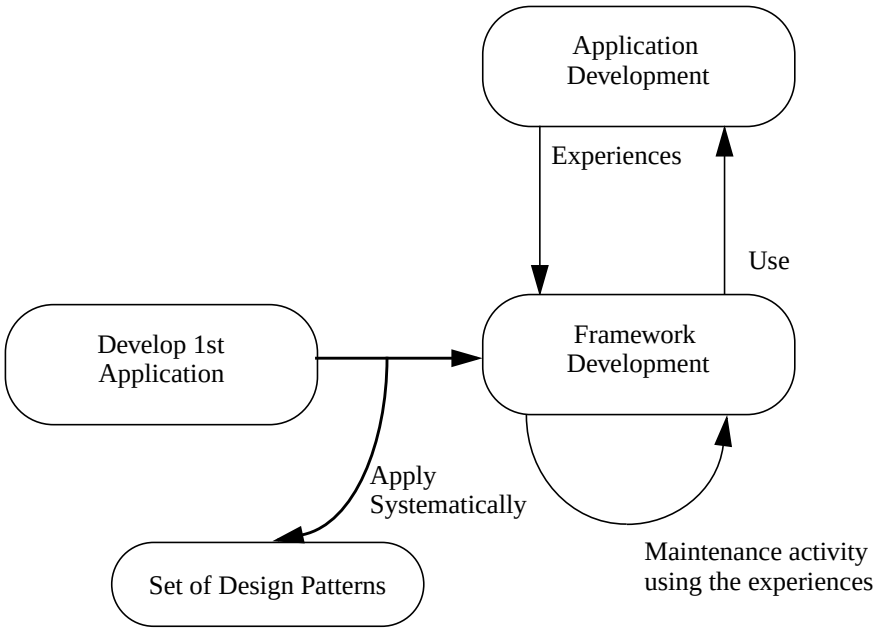


Figure 3.5: *Framework Development Process Utilizing Design Patterns*

- One or possibly a few applications are developed based on the framework. This is the testing activity of the framework. Testing a framework to see if it is reusable is the same activity as developing an application based on the framework
- Problems when using the framework in the development of the applications are captured and solved in the next version of the framework.
- After repeating this cycle a number of times the framework has reached an acceptable maturity level and can be released for multi-user reuse in the organisation.

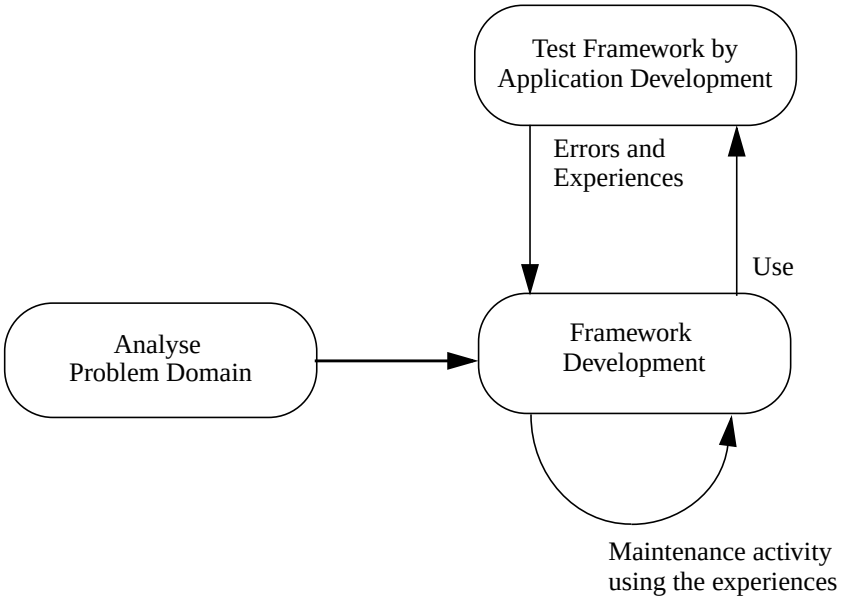


Figure 3.6: *The General Framework Development Process*

3.5 Development Guidelines

A small number of development guidelines for framework development exist [Ada95],[Wil93],[Wei89].

One of the most important guidelines to bear in mind when developing a framework, is how to design the interaction between the framework user and the framework. The interaction must be well-described so it is possible to use the framework with little effort.

The focus has to be on how the *framework user interacts* with the framework. For example, when developing an object-oriented framework for processes that will be performed by some end users (e.g some kind of 2D-

data visualization), it is necessary to determine which steps that will be performed by the framework and which are performed by the framework user. Thus, it is important to identify which classes and operations the framework user has to use. The amount of code that has to be written by the framework user has to be reduced to a minimum. This can be achieved through:

- *concrete implementations provided by the framework* that can be used without any kind of modification [Wil93],
- the number of classes that have to be derived are reduced to a minimum [Ada95],
- operations in the classes that have to be overridden are reduced to a minimum, i.e minimize the public interfaces for the framework users. This is sometimes named the *narrow interface principle* [Wei89].

Issues such as *naming conventions* for classes, operations and how abstract operations are used are important to consider. Using detailed style guidelines is an advantage which will make it clear to the framework user how the code has to be interpreted, one example of a suitable style guide is [Tal94b].

It is necessary to have *control of the use of multiple inheritance* in the framework so there will be no surprises for the framework users.

Keep *problem-domain modelling independent* of the user interface and the platform. One way of achieving this is to use and apply object-oriented design patterns.

The *interaction between the framework classes and the framework user's classes and code* have to be described. Important views to document are the object creation sequence when the framework is called and when the framework users overrides are called by the framework.

3.6 Important Design Elements

When developing a framework, rather than an ordinary object-oriented application, some program constructs, or "design elements", are very important. Such *design elements* are abstract classes, object-oriented design patterns, dynamic binding and contracts. The lack of support for abstract classes, contracts and dynamic binding in existing object-oriented methods has also been reported in [Mat95].

Abstract classes

Abstract classes are distinguished from concrete classes since they can not be used for creating instances. Classes that have this ability are called instantiable or concrete classes. In [Hur94] the so called *Abstract Superclass Rule* (ASC), "*All superclasses must be abstract*" was evaluated from different aspects. Aspects covered in the evaluation were expressiveness, data modelling, evolution, reusability and simplicity.

The expressiveness of an object-oriented design is not reduced if the ASC-rule is applied, since every class structure that contains a concrete superclass can be transformed into an equivalent class structure conforming to the ASC-rule.

Regarding data modelling, three factors were identified where the ASC-rule makes a difference: the notion of inheritance, multiple inheritance and classes as collections of objects. The ASC-rule allows both the "concept-oriented view", i.e is-a, and the "program-oriented view", i.e implementation, of inheritance. Multiple inheritance is motivated by the distinction of *interface inheritance* and *data inheritance*. When a class only inherits a set of interfaces defined in the superclass, this is called interface inheritance and when only a set of instance variables is inherited, it is called data inheritance. These two views of inheritance of necessity means that superclasses are abstract. The aspect of classes as collections of objects motivates the ASC-rule by the following example. Consider the situation at the left in figure 3.7. In that case it is not possible to define a variable that consists of objects of class A excluding objects of classes B and C. This because if objects of class A are allowed, so must object of classes B and C be allowed. However, in the situation at the right, it is possible to define variables that only consist of objects from class A'. Thus, from a data modelling perspective, the ASC-rule is highly motivated.

The evolutionary aspect covers the dynamics of a software system. How will the ASC-rule conform to a changing system? Changes that can take place are that there will be new subclasses, and modifications of properties of existing classes. In the case of subclassing there will be problems if existing class libraries are used and the source for the library is not available. However, the ASC-rule will work properly if it is used in conjunction with the abstract library rule [Hur94]. For avoiding later restructuring of the hierar-

chy in the library, the following four guidelines have to be followed. Depending on the expected use of the class, for each instantiable class:

- If the class is to serve for instantiation only, provide a concrete class
- If the class is to serve for subclassing only, provide an abstract class
- If the class is to serve for both subclassing and instantiation, provide both an abstract superclass and a concrete subclass
- If the expected use of a class cannot be predetermined, it is safer to provide both an abstract and a concrete class.

Using the ASC-rule in conjunction with the abstract library rule will cost extra in maintenance.

Regarding the aspect of modifying class properties, the ASC-rule is useful for the program-oriented view of inheritance, and it also allows for easier change of representation.

The reusability aspect is very well captured by the ASC-rule since both class libraries and object-oriented frameworks are striving to have a high degree of abstraction. One of the goals of library and framework design is to make a generalization of a concept and capture the concept in an abstract class to make it suitable for reuse in later applications.

Finally, the ASC-rule simplifies the object-oriented development process in a number of ways. It is possible to identify which classes can be instantiated or not, adding and deleting properties as well as changing representations.

To summarize, the Abstract Superclass Rule, has both it advantages and disadvantages. In the aspects of data modelling, evolution and reusability (which are highly relevant for framework development), the application of the ASC-rule is likely.

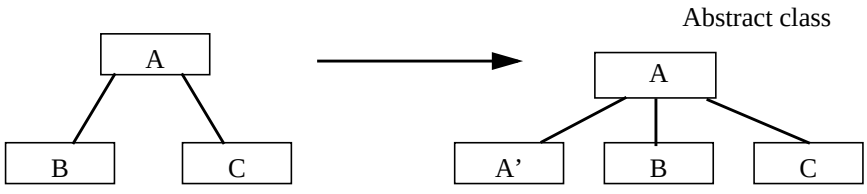


Figure 3.7: *Concrete superclass transformed to abstract superclass*

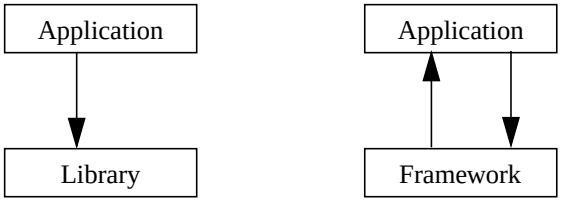


Figure 3.8: *The flip-flop of Control*

Related to the ASC-rule is the idea of object types [Mat94][Ohl93a]. Here two classes are considered to belong to the same object type if they have the same public interface. The rule introduced says that if the public interface has to be extended, the derivation should result in an abstract class and then perform another derivation that will result in a concrete class. Following this principle will make it possible to reuse existing classes and make it easier to utilize the dynamic binding facility.

To summarize, the use of abstract classes is a useful mean for achieving reusability, proper use of inheritance and utilization of dynamic binding. All those aspects are relevant to the design of an object-oriented framework.

Dynamic binding

The concept of dynamic binding is neglected in existing object-oriented methods [Palm93].This is a problem since the full potential of object-orientation comes with inheritance in combination with dynamic binding.

Dynamic binding is identified as one of the key issues for developing object-oriented frameworks [Ohl93b]. A framework can be considered as a library designed for dynamic binding. If we compare a framework with a conventional library we will see the importance of dynamic binding. When a conventional library is used by an application routine, calls are made from the application to the library only. In an object-oriented framework, however, calls can go also in the opposite direction, see figure 3.8. This two-way flow of control is made possible by dynamic binding.

Currently, some fragments in existing methods are seen that point towards the use of dynamic binding as a design element, i.e the notion of abstract classes is one step in that direction.

Contracts

To ease the derivation of a concrete design from an abstract design, e.g a framework instantiation, some intermediate representation can be suitable, i.e. a contract. A contract specify the collaboration between a number of objects. The importance of object behaviour collaboration has been identified in [Wir90b] and contracts formalize these collaborations and behavioural relationships.

Contracts have been proposed on various levels of abstraction and formalism [Helm90], [Mey92].

The contracts by [Helm90] specifies behavioural compositions in terms of: the participating objects, the contractual obligations for each participant, invariants to be maintained by the participants and the operations which instantiate the contract. The behaviour of a participant is then specified by its obligations. The contractual obligations consists of *type obligations* and *causal obligations*. In the case of type obligations, the participant must support certain variables and external interface and, in the case of causal obligations, the participant must perform an ordered sequence of actions, including sending messages to other participants.

The contracts proposed by Meyer [Mey92] are at a lower level of abstraction than the one above, and they have been introduced into an object-oriented programming language, Eiffel [Mey88]. A contract for a class describes the conditions that an object must satisfy during its existence and the contract also includes pre- and postconditions for each class operation. The preconditions must be satisfied by the client of the object otherwise the operation will not be performed. If the precondition is satisfied, the operation will perform its task and it will also guarantee that the defined postconditions for the operations will be true. Here, the contract idea is a way to more formally specify the division of responsibilities between the server class and the client class. An example of a software contract generated by the Eiffel system for a put-operation on a Dictionary class [Man93] is presented below, figure 3.9.

A more informal description of the contract idea has been applied when using the ET++ user interface framework [Wei89] and when developing a business modelling and reengineering prototype environment [Laj94]. The contract template used is presented in figure 3.10.

The advantages of using contracts when developing object-oriented frameworks are [Laj94]:

- A vocabulary is established which helps the application developers to communicate with each other when using the framework.
- Identification of application-specific classes, variables, methods, and hooks for customization, all which are necessary for identifying, maintaining and implementing a specific behaviour are provided to the application developer.
- Better understanding of existing design patterns in the framework, thus improving the understanding of the overall framework.

Object-Oriented Design Patterns

Object-oriented design patterns have not been extensively used when developing frameworks. However, since design patterns represent abstract descriptions of solutions to common and recurring design problems, they will be a useful reference when developing frameworks. This since they are larger building blocks than concrete and abstract classes. Experiences gained through applying design patterns in framework development have recently been reported in [Hun95], where design patterns have been applied successfully when developing an object-oriented framework for network

```
class interface DICTIONARY [ ELEMENT ] feature ;

    put( x: ELEMENT; key: STRING) is
    -- insert x so it's accessible through key
    require
        count <= capacity;
        not key.empty
    ensure
        has(x);
        item(key) = x;
        count = old count + 1
    ... interface specifications of other features
    invariant
        0 <= count ; count <= capacity
    end class interface - DICTIONARY
```

Figure 3.9: *Example of a software contract [Man92]*

protocol software. In figure 3.11, an ideal picture of a future application developed with frameworks and patterns is presented.

3.7 Maintenance

The design of an object-oriented framework is the result of many iterations. Thus there is a need for maintenance techniques that can restructure the framework regularly. This is most important in the very early versions of the framework, since it is the first times the framework is being reused that the need for redesign and restructuring is identified.

CONTRACT NAME
The name of the contract should clearly convey
its intent. Since this name will become part of the design vocabulary, it is
very importnat and must be chosen carefully.

Pertinent Design Pattern
Identify the design pattern abstracting the concrete's design if
one exist.

Participants
A participant may either be Active or Non-Active.
Non-Active participants support type obligations only.
Active participants support both type and casual obligations ,thus
maintaining Non-Active participants.

Participant(s) Pseudo Code Details
Each participant details all type and/or casual obligations i.e the
class data members and methods (in pseudo code) involved in
the contract.

Invariants
Definition of the invariants that participants cooperate to maintain.

Instantiation
Identification of the preconditions necessary to establish the contract
as well as the methods for its instantiation.

Figure 3.10: *A Contract Template [Laj94]*

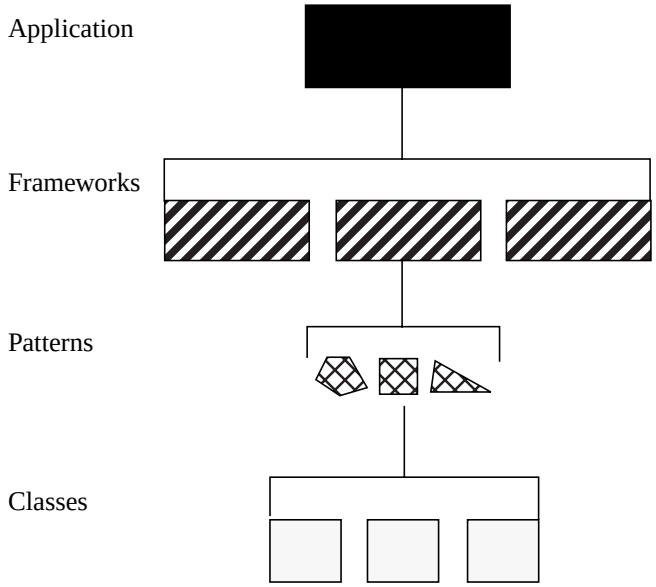


Figure 3.11: *The decomposition of an application based on frameworks and patterns*

Software evolution and maintenance activities are divided into the following categories [Mar94]:

- corrective maintenance, fixing failures to meet system requirements
- perfective maintenance, improving the performance of the system, maintainability and ease of use in ways that do not violate the requirements
- adaptive maintenance, evolving the system to meet changing needs

A set of *refactorings* (restructuring operations), which is one aspect of perfective maintenance, that supports the design and evolution of object-oriented software has been described in [Opd92][Opd93][Joh93b]. Some of the refactoring algorithms require several inputs from the user. Each algorithm has a precondition; if the precondition has met the refactoring it is behaviour preserving and will not introduce any defects in the program.

In [Opd93], three major refactorings were identified that rely upon about 25 low-level refactorings.

- Refactoring to Generalize: Creating an Abstract Superclass
- Refactoring to Specialize: Subclassing, and Simplifying Conditionals
- Refactoring to Capture Aggregation & Components

The three refactorings are further described in the following:

Creating an Abstract Superclass

The purpose of the refactoring is to create an abstract superclass for a couple of classes C1 and C2 [Opd93]. The classes must have a common superclass or no superclasses. Then create an abstract superclass that will be the superclass of the classes C1 and C2. The subsequent steps, or low-level refactorings, are:

- adding function signatures to the superclass protocol after making them compatible in both subclasses
- making function bodies and the variables referenced by them compatible in both superclasses
- migrating common variables to the superclass
- migrating common code to the superclass

Subclassing, and Simplifying Conditionals

The idea is to break down a complex class into a number of smaller concrete classes, and a superclass which represents the general abstraction. The set of flags, tags and conditional statements give a hint of how to decompose the class. For each condition in the original class, create a subclass whose abstraction implies the condition. The result is the same as the State design pattern [Gam95]. This refactoring comprises the following steps:

- choose a conditional whose conditions suggest subclasses
- for each condition in the conditional, create a subclass with a class invariant that matches the condition
- copy the body of the conditional to each subclass and simplify in each subclass the conditional based on the invariant; that being true for the subclasses.
- specialize some or all expressions that create instances of the superclass. This means replacing an expression that creates an instance of

the superclass with an expression that creates an instance of one of the newly created subclasses.

Capture Aggregation & Components

This refactoring is in fact three refactorings that deals with the restructuring of aggregation relationships. An aggregate class (sometimes named container class) contains one or more component members; a component class is a class whose instances are components of another class. The three refactorings identified related to aggregate relationships are:

- moving members from an aggregate class to the class of one of its components
- moving members from a component class to the aggregate classes that contain component members, and which are instances of the component class
- converting a relationship, modelled using the inheritance relationships, into an aggregation relationship

These three refactorings all consist of a number of other low-level refactorings.

A prototypical tool for providing automatic support for the refactorings of a subset of the C++ language has been developed [Opd92] and a tool for the Smalltalk language is under construction [Joh93b].

A subset of all the refactorings has been described in the Alexandrian pattern form, maintenance patterns,[Foo94] and are listed in figure 3.12.

3.8 Discussion

Topics in the field of object-oriented framework development that need further investigation are:

- Integration of design patterns with framework development and methods for framework development. Since frameworks are implementation language dependent and design patterns are not, systematic integration/ application of design pattern into a framework will, probably, result in frameworks that are more reusable. This will require methods/tech-

niques that will help to identify which parts of the application that are suitable for applying design patterns, which parts that are stable in the framework's domain and which may vary. Methods for framework development also have to explicitly support the use of the identified design elements, dynamic binding, contracts, abstract classes and design patterns.

- In the field of software architecture the notion of *architectural style* is being more used [Gar94]. An architectural style expresses the components of a software architecture and relationships between them together with the constraints of their application as a well as associated composition and design rules for their construction. In the context of object-oriented frameworks the concepts of architectural framework and design patterns may be useful for the development of an *architectural method for framework development*. An architectural method for framework development may comprise a set of representations and techniques which can be used to develop a framework with a main structure, given by the architectural framework. This research area will generate a number of architectural methods for framework development. E.g at least one for each architectural framework.
- When developing frameworks special attention is needed to handle non-functional requirements, for example memory management policies, parallel activities and effiency. Another issue is how to design the framework so it can easily be integrated with other frameworks or other systems when developing larger applications. Is there a need for special subframeworks which address some of these non-functional requirements, e.g persistence?
- Tools that support the iteration process of framework development. The development of mature and reusable frameworks requires a lot of iteration between the application development and the framework development. A number of standard restructurings (refactorings) have been identified, but currently there exists no software environment that supports framework iteration.
- Developing a framework in a domain is in itself a research task. This since the development will gain knowledge abut the domain and how it works which is often not well-enough understood.

Category	Refactoring(s)
High Level refactorings	Create abstract superclass Subclass and simplify conditionals Capture aggregations and components
Low Level Refactorings: Create program entity	Create empty class Create member variable Create operation
Delete program entity	Delete unreferenced class Delete unreferenced variable Delete a set of operations
Change program entity	Change class name Change variable name Change operation name Change type of a set of variables and function Change access control mode Add function argument Delete function argument Reorder function argument Add function body Delete function body Convert instance variable to pointer Convert variable references to function calls Replace statement list with function call In-line function call Change superclass
Move member variable	Move member variable to superclass Move member variable to subclass
Composite refactorings	Abstract access to member variable Convert code segment to function Move a class

Figure 3.12: *Refactoring or Maintenance Patterns [Foo94]*

4 Using Frameworks

4.1 Introduction

Application development is often made ad hoc, i.e without reusing previously developed software in a formalized or methodological way. In some cases reuse of existing software, such as class libraries for user interfaces or data structures, is made in the application development.

Developing an application based on existing reusable assets requires new methods that take the reuse aspect into consideration. How will these methods look and how are they related to the software development phases of the software development process? In the case of object-oriented frameworks, we need a method that use the framework as some kind of starting foundation for the application development; a framework-based application development method.

Application development based on object-oriented frameworks differs from application development using class libraries. The differences are primarily two; in an object-oriented framework the framework is in control of the runtime behaviour for the application to be developed, which is not the case when using a class library in the development. Secondly, the framework imposes the overall structure of the application software whereas a class library has minor impact of the software structure.

Currently, there exist no methods for developing applications based on frameworks. In order to make frameworks useful in an industrial setting, it is necessary to have methods that guide the software engineers when developing applications using frameworks as a basis.

In the next section the traditional software development process will be briefly described. In section 4.3 we propose a framework-based application

development method comprising a set of activities which also will be related to the software development process.

4.2 The Software Development Process

Independent of the software life cycle model chosen, (e.g waterfall model [Boe81], spiral model [Boe88]), the software development process comprises the following phases [Bus93b]:

- Domain Analysis
- Requirement Analysis
- System Analysis
- Architectural Design
- Detailed Design
- Implementation
- Testing
- Maintenance

The Domain Analysis or Subject Analysis phase is aimed at getting an understanding of the problem. I.e the real world task that has to be solved by the software system, the environment the task is performed in, existing policies etc. for solving the tasks. Previously captured domain knowledge and domain standards are also scanned to get an understanding of the problem domain. This phase is normally not so extensive and thorough as the discipline of Domain Analysis [Sch94] but can be considered as a first attempt in that discipline. The result of this phase is a description of the task that is to be solved by the software.

Requirement Analysis aims to elicitate and identify both the functional and non-functional requirements for the intended software to be developed. Here both inputs from the subject analysis and earlier experiences from the domain are used to capture the requirements. The result is normally a list of requirements that has to be fulfilled by the software.

The Systems Analysis phase concentrates on what to do in the application. The entities identified in the real world task are mapped to a software model. The system's functionality is mapped to software components and their rela-

tionships and the components responsibilities are defined. The result of this phase is called the *conceptual framework.*

The Architectural or System design phase deals with the main structuring of the application to be built. The architectural framework for the application is chosen and the conceptual framework is mapped to the architectural framework. The subsystems and the relationships between the subsystems are specified. This can be handled by the use of design patterns, for example the decoupling of subsystems using Adapter or Mediator [Gam95]. Also the subsystems are further structured and refined and design patterns may be useful to carry this out. The result of the phase is a model of the application comprising its components, their responsibilities, high-level data structures as well as their relationships.

In the Detailed Design phase the responsibilities and data structures are specified in full detail. Principles and policies for implementation of the data structures and the subsystems are defined.

The final implementation of the system takes place in the Implementation phase using the results from the Architectural and Detailed design phases.

Finally, the functional and non-functional requirements for the application are validated in component tests, system tests, and integration tests.

When the application is in operation it will after some time require modifications due to new requirements or operating environments. These changes are handled in the Maintenance phase.

4.3 A Framework-based Development Method

Application development based on frameworks is still performed ad hoc and no methods supporting this exist. To deliver full-scale reuse of the framework technology, methods that describe how to develop framework-based applications are needed. We will outline a set of activities that must be included in that kind of method and relate them to the software development phases. As a basis for the proposed method, Buschmann's ideas [Bus93b] concerning architectural methods for developing software architectures are used. The activities proposed are general in nature and have to be refined

and specialized depending on the specific object-oriented framework(s) involved.

- Activity 1: *Define the conceptual framework* for the application.This activity is based on a functional decomposition and an analysis of the work organisation of the software's primary matter. It includes the association of functionality to components, the specification of the relationships between them, and the organization of collaboration between them.
- Activity 2: *Select an object-oriented framework* for the application. Based on general properties the software will have, as well as non-functional requirements, the basic structure for the application under development is specified and an appropriate framework is selected.
- Activity 3: *Map the entities* of the conceptual framework for the application into the selected object-oriented framework and subsystems/ subframeworks (if any). The term subframework is used from now on but, if it is decided that a subsystem is not worth generalizing as a subframework, the term subframework should be interpreted as subsystem. Here it will be difficult to find a perfect match, i.e will the application's requirements conform precisely to the functionality offered by the framework.
- Activity 4: *Specify or revise the relationships between the subframeworks* of the application's object-oriented framework. The selected framework may be "composed" of a number of subframeworks with predefined relationships which have to be revised or remain unchanged. The other situation that can occur is that the framework is designed with *hot-spots*, (i.e those places in the framework that have been designed to be flexible), which enable attachments of different subframeworks, e.g a device driver framework or a database access framework. The relationships between the subframeworks describe the communication and collaboration between the subframeworks or their access to subframeworks.
- Activity 5: *Structure the various subframeworks* of the object-oriented framework. The components of the conceptual framework that has been mapped into each subframework are to be further organized with the purpose of achieving the required non-functional properties for the application as a whole. If a suitable subframework exists this will be simple. Otherwise, the mapped components have to be examined and a

suitable combination of architectural frameworks and design patterns has to be designed that addresses the non-functional requirements.
- Activity 6: *Structure the different components of a subframework.*This includes deciding which parts of the subframework will continue to be hot-spots and which parts that will be *frozen spots*, (i.e the part of a design that will not vary). If they are to be frozen spots, the implemented default behaviour of the subframework will be used. For the hot-spot case, concrete classes have to be designed.
- Activity 7: *Implement* the software system. Code all the classes that have been specified.

The activities we have outlined describe general steps that have to be more specialized depending on the actual framework involved. For example, there will be a specific development method for frameworks structured with the Model-View-Controller architectural framework as a basis.

The activities described are not located in any one phase in the software process model but may be mapped to a number of different phases. Figure 4.1 describes the mapping between the software process phases and the activities in the framework-based development method.

The emphasis in the method lies in the Architectural Design phase, which is not surprising since frameworks pay special attention to the structure of the software to make them more reusable.

4.4 Discussion

Using an object-oriented framework is not only to:

- define and subclass any new classes that are needed, and
- configure a set of objects by providing parameters to each object and connecting them.

The process of using frameworks must be put in an industrial context, which implies methodological demands and issues that need further investigation:

- There is a need for the development of more concrete and specific methods for application development based on frameworks. It is not

enough with the outlined method and, again, there will be domain or framework specific methods that have to be developed.

- Development of applications using a number of different fine-grained frameworks also put new demands of the methods to be developed.
- There is also need for tools to support the configuration of framework-based applications.

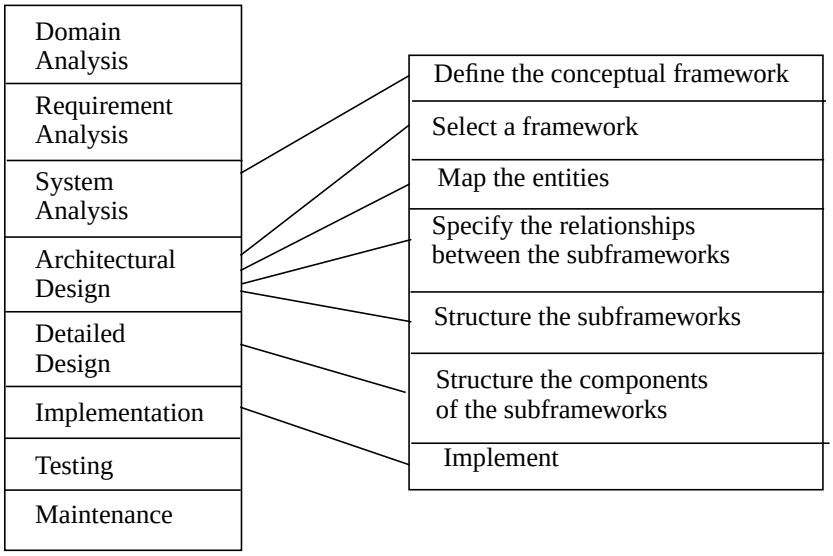


Figure 4.1: *Mapping of the activities of the framework-based method to the software process*

5 Documentation of Frameworks

5.1 Introduction

Suppose there is a need for developing an application in the domain of an existing object-oriented framework. The first problem that will occur is: Can we understand the documentation of the object-oriented framework? Is the description of the object-oriented framework easily understandable?

The documentation of a framework has to suit different audiences and then has to be tailored to meet different needs. At least three different audiences can be identified:

- Software engineers who have to decide which framework to use. Here a short description of the framework's capabilities is sufficient. The documentation has to explain the most important features of the framework, i.e through examples.
- Software engineers who have decided to use the framework. Here the documentation has to describe how the framework is intended to be used. A good way can be to give the software engineers some kind of cookbook that will quickly teach them how to use the framework.
- Software engineers who want go beyond the normal usage of the framework and add new features to it. This kind of usage requires a more detailed and deeper understanding of the framework. The documentation has to describe the abstract algorithms used and the collaborative model of the framework.

Since the framework covers a lot of complexity in the application domain which must be described and explained, there is a need for an understanda-

ble description of the framework. This will require a set of documentation techniques suitable to frameworks.

The documentation for an ordinary object-oriented application has to be described on different levels with varying contents for different categories of software engineers. For class libraries and object-oriented frameworks this is equally true. According to [Lin90] the following information has to be included in the documentation for understanding and using classes in a class library:

- Structural information, such as class name, superclass if any, type and order of any instantiation-time parameters plus similar information for the operations.
- Descriptions, a natural language description of each class that describes the essence of the class and the abstraction it represents.
- Usage, describing if the class is intended to be instantiated in a particular way, or not instantiated at all. For example, instantiated to object, abstract library class or provided for in X-objects (i.e. instances of the class are created by objects of class X).
- Terminology, the terminology introduced regarding the concept the class captures.
- Configuration, description of how classes are related to each other (this is similar to object-oriented design patterns), and intended to be instantiated in certain configurations.
- Assertions, semantic constraints stating preconditions and postconditions for operations and class invariants.
- Operations, for each operation structural documentation such as parameters, results of operations and the corresponding types.

These aspects are all relevant for the documentation of a framework. Besides that, the documentation of a framework must be described on different levels of abstraction since it must address the needs of developers with varying levels of experience. There are different needs for an experienced user of the framework and a first-time user. The framework documentation must encompass [Joh92]:

- the purpose of the framework
- how to use the underlying framework
- the purpose of the application examples
- the design of the framework

The purpose of the framework

First of all the documentation of a framework must describe its purpose, and which problem domain it is developed for. It must be very clear which problems the framework is the solution for in the problem domain covered by the framework. This must be stated early in the documentation because if the framework is not suitable for the framework user/reader of the documentation, then it is not necessary to read any further in the documentation.

How to use the framework

This is the most important documentation aspect for achieving optimal reuse of the framework. Often the documentation tries to explain how the framework works and is structured without explaining how to use it. It is more important to try the other way around. First describe how to use the framework and after that describe how it works.

If the framework can not be documented properly in order to be easy to (re)use, it will not be reused. A lot of effort is needed to make this part of the documentation a quality product. The framework user will not spend a lot of effort in learning the intricacies of the framework. He wants some kind of guide and guidelines on how to use the framework, i.e enough information so he can start to reuse parts of the framework without knowing all the details.This is similar to the minimalist instructions, detailed in [Ros93]. Thus, this part of the documentation has to explain how to use a framework without explaining how it works.

The purpose of the application examples

The application examples in framework documentation are more important than first recognized. Concrete examples make the framework more understandable. The introduction of examples has been used in some object-oriented methods as an important concept, (e.g. use-cases [Jac92]).It also gives the framework user help in deciding if he has understood the framework or not. Reading examples is a good approach to understanding the main structure of a framework. In one sense the set of examples gives the framework user an opinion of the borders or limits of the framework.

The design of the framework

The description of the detailed design of the framework must contain both the classes and their relationships as well as the collaborations between the classes. Since frameworks are large collections of classes, framework users will not know all the details in the beginning. But, the more the framework is understood by the framework user, the more the framework will be reused. This incremental way may be preferable to learning the whole detailed design at one go.

In the following sections a number of different approaches are described which can be or have been applied in documenting object-oriented frameworks.

5.2 Cookbook Approaches

Most framework users are not interested in the details of a framework but are looking for documentation that describes how to use the framework. They are looking for some kind of cookbook, one that guides them in how to use the framework. A few different approaches that use the cookbook idea exist [Kra88][App89][Pre94a].

The Model-View-Controller Cookbook Approach

The Model-View-Controller (MVC) metaphor has its origins from Smalltalk-80, and describes how to decompose an application that will have a user interface. The three components in the decomposition are; the Model that captures the functionality of the domain, the View that displays the application's state, and the Controller that handles the interaction between the model and the view. The MVC metaphor is described and refined in the Observer design pattern [Gam95].

The documentation approach taken by [Kra88] starts with a description of the MVC metaphor implementation on a detailed level and then relies on examples. The examples in the MVC cookbook approach are collected together with the aim of being read as a unit. The examples describe how the whole framework can be used. The major weakness of a cookbook approach is that it describes the normal way to use the framework, but lacks to means

of knowing how the framework will be used in the future. Thus a cookbook cannot describe every use of the framework. This can be one reason why [Kra88] included an informal description of the MVC framework design. However, it is not surprising that a cookbook approach does not support the unexpected use of the framework since this is not the intention of a cookbook.

The MacApp Cookbook Approach

MacApp is a graphical user interface application framework developed by Apple Inc. Both the [App89] and the MVC cookbook approaches are example-based. The difference is that the MacApp approach is more like a set of recipes where each recipe solves a particular problem while, the MVC approach describes the whole solution. There exists some structure in the MacApp documentation since the recipes cross-reference each other. Finally, the MacApp cookbook approach lacks the ability to describe any unforeseen use of the framework which, of course, is not the purpose of a cookbook.

The Active Cookbook Approach

The active cookbook approach utilizes metapatterns as a way of describing an object-oriented framework [Pre94a], [Pre94b]. In each metapattern it is clearly defined what the template method or class is, and which is the hook method or class. A metapattern is attached to those parts in a framework in which the semantics are to be varied. The whole framework design and the accompanying metapatterns are then put into a hypertext system. The hypertext system is then used as an active cookbook that presents for the user those parts of the framework design that are interesting for the user (the ones which have a metapattern attached to it).

Discussion

The cookbook approaches are all good for addressing the purpose of the framework and present examples. The aspect of how to use the framework is described informally through natural language. The detailed design of the frameworks is nearly neglected in these approaches, but can be presented with some standard design notation [Boo94][Rum91]. Only some discussions about a few classes and class hierarchies in the frameworks exist in the cookbook approaches. An exception is the active cookbook approach which

will give a good understanding of the design. However, this approach is very expensive to use and requires a mature framework.

5.3 Pattern Approaches

Patterns exist in a number of shapes and some of them are suitable for the documentation of frameworks. Object-oriented design patterns and motifs seem to be two of the most promising pattern variants for documentation.

The Motif approach

The approach by Johnson [Joh92] uses a kind of documentation pattern, *motif* [Laj94], which describe how to use a framework. Thus, the motifs are in a way related to the activity of application development based on a framework.

The motifs conform to a common structure. First, a description of the problem is given, followed by a discussion about different ways to solve the problem. The discussions include examples and pointers to other parts of the framework. The motif ends with a summary of the solution and pointers to other motifs.

We give an example of a motif, figure 5.2, that is part of a motif description [Joh92] of HotDraw. HotDraw is a framework for semantic graphic editors implemented in Smalltalk. The problem description in the first motif of Hotdraw is found in figure 5.1, and is included to give some background information before reading the complete motif example.

> *HotDraw is a framework for structured drawing editors. It can be used to build editors for specialized two-dimensional drawings such as schematic diagrams, blueprints, music, or program designs. The elements of these drawings can have constraints between them, they can react to commands by the user, and they can be animated. The editors can be complete applications, or they can be a small part of a larger system. [Joh92]*

Figure 5.1: *Problem description for the first HotDraw motif [Joh92]*

Motif 3: Changing drawing element attributes

There are at least three ways to edit a figure's attribute; with a handle, with the figures's pop-up menu, or with a special tool. Each technique is appropriate in different cases.

The size of a figure and other numeric attributes are best edited with handles. Textual attributes like names, or numeric attributes that must be precise like dates, are best edited by displaying the text as part of the figure and letting the user edit it with the TextTool. Use of the TextTool implies that the figure is a CompositeFigure (see **Complex Figures (4)**).

A figure's **handles** methods returns a collection of handles on the figure. The **handles** method for Figure returns resizing handles on the four corners, so it is common for a **handles** method to call it (with super handles) and add more handles. SelectionTrackHandle has class methods that create customized handles. For example **colorOf:** will change the inside color of a figure, **borderColorOf:** will change the color of the border, and so on. For good examples of a **handles** method, see LineFigure or RectangleFigure.
A Figure's menu is returned by its **menu method**. By default, an operation in the menu is sent to the DrawingView. Operations whose selectors are in a collection returned by the **menuBindings** method of a figure are instead sent to the figure. The default menu defined in Figure implements cut and paste. See Figure for a simple example and TextFigure for a more complex example with hierarchical menus.

*List the attributes of the drawing elements that you want to edit. For each attribute, decide whether to edit it with a handle, an operation from the menu, or a tool, and update the **handles** methods, the **menu** method, or the list of tools in the drawing editor.*

To make new kinds of handles, see **Handles(6)**. To make new kinds of tools, see **Tools(8)**.

Figure 5.2: *A complete motif example for the Hotdraw framework [Joh92]*

The Object-Oriented Design Pattern Approach

Object-oriented design patterns describe parts of an object-oriented design. This doesn't make them suitable for presenting the complete design of an object-oriented framework in one view. A class in a framework can have two or more different roles, depending on how many design pattern's it is partic-

ipating in. This means that it is not possible to identify the borderlines between different design patterns in the framework.

However, object-oriented design patterns are useful in the context of describing parts of a framework design through the diagrams for the design patterns and the vocabulary introduced. In some cases where the framework offers the users few entries to adapt the framework, design patterns can be a suitable approach.

Using object-oriented design patterns when writing MacApp-like cookbooks may improve the MacApp cookbook approach significantly.

Discussion

The object-oriented design pattern approach does not address the purpose of the framework aspect. This is easily covered by the motif approach which can start with an introductory motif, describing the purpose of the framework. The set of motifs developed for a framework will be an acyclic graph of motifs, where each motif describes more and more details of the framework. The motif approach can also be suitable for introducing examples which describe how to use the framework.

The aspect of describing the detailed design of the framework can be difficult when using the motif approach. Here the object-oriented design pattern approach will fulfil this requirement and capture the intricacies of the framework design.

Since a framework is the result of many design iterations, the real world concepts have often been lost in the design. This implies a higher learning curve for the framework, which may be reduced if the framework is documented with design patterns and the framework user masters the set of patterns used.

5.4 The Framework Description Language Approach

Wilson and Wilson [Wil93] propose a Framework Description Language (FDL) for the documentation of a framework. Inheritance relationships, ref-

erences and object creation sequences are necessary items to be described when documenting a framework. There must also exist information that answers the following questions:

- Which new classes must be provided for the framework?

This is necessary information to make it possible to develop an application from the framework. There must also be information that tells the application developer which operations have to be written for each class.

- What classes should be used from the framework?

Some classes in the framework may not be subclassed and are intended to be used as they are defined in the framework. Other questions related to this are: Which operations must be called and which operations are often called in the framework?

- What classes must be subclassed?

Since a framework can be viewed as an abstract design of a family of related applications, the framework maybe has described some concept in the application domain as an abstract class. This implies that the application developer has to subclass the abstract class. Related questions that must be answered are: Which operations must be overridden and which operations are often overridden?

To capture the answers to these questions a Framework Description Language (FDL), which consists of a number of keywords and a C++ like class header specification for a framework, is proposed [Wil93]. The FDL interface template is presented in figure 5.3.

The FDL implementation template describes which operations must be or may be overridden in the framework. An outline of the FDL implementation template is presented in figure 5.4.

One improvement to the FDL template will be to include the names of the libraries if several libraries are used. This may be the case if the framework is large. There is also a need to handle classes where there are optional alternatives/implementations for the same class. This is necessary if the framework is portable across different platforms, for example. This raises a need for some block of information that indicates that there are alternatives for the class.

```
framework name {
    mustInstantiate:
        // Set of classes that must be instantiated
    mayInstantiate:
        // Set of classes that may be instantiated
    mustSubclass:
        // Set of classes that must be subclassed
    library:
        // Set of classes that are used from a class library (gives the
        // application developer linking information)
    private:
        // Set of classes that are internal for the framework (and
        // objects of these that classes are instantiated by the framework)
};
```

Figure 5.3: *The Framework Description Langugage (FDL) interface template [Wil93]*

A third improvement is to add a Framework Class Description (FCD) template, which is a variant of the ordinary C++ class header. The FCD template describes for each class which operation must or may be overridden.

To summarize [Wil93], the documentation of a framework must consist of:

- A framework description in FDL
- Class Diagrams
- Sample Programs
- Recipes

The FDL approach together with the class diagrams, sample programs and recipes encompass most of the documentation aspects. The FDL templates give a more detailed description of how the frameworks can be used, through providing information about classes that have to be subclassed, and operations that must be overridden. This approach does not meet the requirement of giving a detailed design description of the framework. With

```
framework name {
    mustOverride:
        TClassA::method1();
        TClassA::method3();
    oftenOverride:
        TClassA::method4();
        TClassC::method3();
};
```

Figure 5.4: *The Framework Description Langugae (FDL) implementation template [Wil93]*

one exception, the FDL description, the documentation techniques proposed in [Wil93] are similar to the Cookbook approaches.

5.5 Discussion

All the documentation techniques presented have their deficiencies. Only the motif approach tries to address the four needs for framework documentation:

- the purpose of the framework
- how to use the underlying framework
- the purpose of the application examples
- the design of the framework

However, the motif approach has problems related to the requirement of describing the design of a framework. Currently, there does not exist any documentation approach that fulfils all the needs. One major drawback with the existing approaches, is that they have been applied to just one single framework, (with the exception of the FDL approach), and seldom applied to different audiences under controlled circumstances.

There is a need to evaluate and improve existing approaches to see if they can fulfil the needs or, if it is necessary, to use different documentation tech-

niques for different audiences. The cost of producing the documentation has to be considered in the evaluation too.

It is also necessary to ascertain if and how well the existing approaches scale up when applied to larger frameworks. This since most of the approaches have only been applied to small frameworks.

Other issues relating to documenting frameworks, which deserve attention, are how to describe the interaction between the objects in the framework, i.e the dynamic aspects. This is necessary to ensure that the framework user gets a good picture of the framework's behaviour.

If the framework is small, there may be a need to integrate different frameworks when developing applications.

Which are the important properties to document, ones that describe relevant integration information for a framework?

At a lower level of framework user documentation, *usage rules* have to be described. A usage rule gives necessary information on how to use the framework and describes the framework's behaviour. A set of operations that must be called in a given sequence for a certain object in the framework, is an example of a usage rule. Other usage rules may describe cardinality of framework objects, creation and destruction of static and dynamic framework objects, instantiation order, synchronization and performance. These usage rules are often implicitly hidden or missing in existing documentation, and there is a large need to elicitate and document them.

Another documentation issue that has to be captured is the maintenance of a history log, one that records the transformation of domain classes into larger design structures. This is especially important for frameworks, since they undergo several design iterations and become more abstract after each iteration.

6 Conclusions

A number of methodological issues regarding object-oriented frameworks have been surveyed and presented. The main focus has been on the following activities:

- Developing Frameworks
- Using Frameworks
- Documenting Frameworks

The concept of patterns can be used in all the three activities and have been presented in a separate chapter. The rest of this chapter makes a condensed summary of the pattern and framework concepts, ending with a discussion about suitable research topics that need further investigation.

Object-oriented design patterns describe solutions to small-scale design problems in a given context, whereas frameworks address solutions to whole application or subsystem domains. Patterns are naturally divided in two main categories, non-generative and generative patterns.

Non-generative patterns focus on the solutions to the given problem rather than on when to use the pattern or the rationale behind the pattern, (which is the case for generative patterns). Generative patterns exist for a number of different activities; such as documentation, software development and restructuring of programming code. Non-generative patterns are mainly concerned with software design (design patterns) and are the most investigated area of patterns.

Today, there exist sets of design patterns [Gam95], [Bus93b]. Most of the existing design patterns are domain independent, but there are domain specific patterns, for example in the distributed systems domain. Documentation templates for describing design patterns, the development of classification systems and selection processes have been proposed [Gam95], [Bus93b]. The most exhaustive classification scheme, by Buschmann et al. [Bus94b], proposes a classification in three dimensions: Granularity, Func-

tionality (object creation, object communication, object access and organisation of computation) and Structural principles (abstraction, encapsulation, separation of concerns, coupling and cohesion). The classification scheme will support the designer in selecting an appropriate pattern in a given design situation.

An object-oriented framework is *"a (generative) architecture designed for maximum reuse, represented as a collective set of abstract and concrete classes; encapsulated potential behaviour for subclassed specializations."*

The major difference between an object-oriented framework and a class library is that the framework calls the application code. Normally the application code calls the class library. This inversion of control is sometimes named the Hollywood principle, "Do not call us, we call You".

Development of a framework requires a number of design iterations in order to make it reusable. Existing object-oriented methods have no explicit support for the development of reusable object-oriented software [Mat95], and particularly not frameworks. Design elements of special importance in the development of a framework are: abstract classes, dynamic binding, contracts and design patterns.

Using a framework is normally described as: define and subclass any new classes that are needed, configuring a set of objects by providing parameters to each object and then connecting them. However, the task is not so simple. If frameworks will be used in an industrial setting, there is a need for methods that support application development based on frameworks. Currently no such methods exist, and an outline of such a method is presented in this thesis.

One major aspect of reusable software is its documentation. If the framework's documentation does not meet the requirements of its intended users, the framework will not be reused. Existing approaches that have been, or are suitable for documenting a framework, are surveyed. The approaches are variants of cookbooks that describe in natural language how the framework works and can be used, pattern approaches which use different variants of patterns, and a Framework Description Language (similar to a C++ header).

A design pattern is more abstract than a framework since it represents a general design solution to a problem. Design patterns are also small design structures: a framework can consists of a number of design patterns, (see fig-

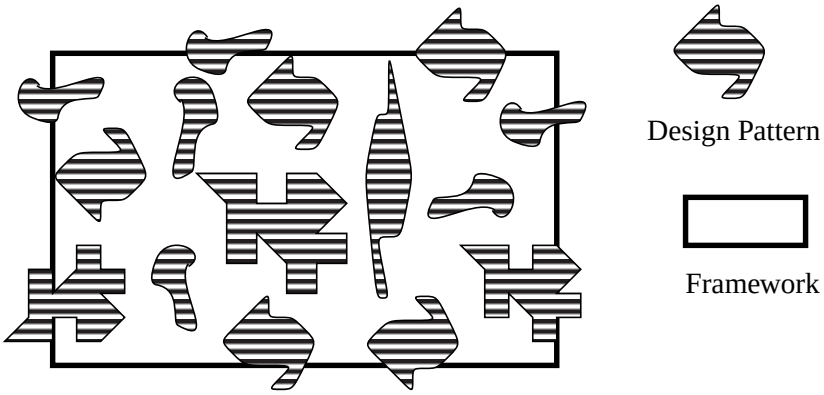


Figure 6.1: *Framework consisting of a number of design patterns*

ure 6.1), but the opposite is not possible. A framework is always related to a specific application domain which is not the case for design patterns which are of a general nature.

A number of research issues in the field of object-oriented frameworks and patterns can be identified. Pure pattern specific topics are:

- combinations of patterns, i.e “Does there exist larger domain-independent design patterns that are combinations of existing design patterns?“,
- development of domain specific patterns and
- techniques for discovering and documenting patterns.

Existing object-oriented methods support reuse only implicitly, through inheritance and use of class libraries. There is thus a need to integrate the concept of patterns into existing methods.

Research in framework development will include development of methods that supports the framework concept. The methods have to incorporate design elements such as design patterns, abstract classes, contracts and dynamic binding. Since a framework will be based on some structural principles, such as a client-server or layered architecture, there will be specific methods for each kind of structuring principle. Other research issues related to framework development are on how non-functional requirements will be

fulfilled in the framework and the integration aspect (i.e how should the framework be designed to be easy to integrate with other software or frameworks).

Regarding the documentation aspect of frameworks, existing approaches have to be evaluated and improved to see if they meet all the needs of the framework users. Not one of the existing documentation approaches have been applied to large frameworks, so the problems of scale have to be investigated. One other major research issue is how to describe and document, for the framework user, the dynamics of the frameworks, i.e the interaction between the objects in the framework.

7 Future Work

During the work with this thesis I have accumulated knowledge and ideas about the frameworks area. I also have identified a number of issues that need further investigation in order to make the concept of frameworks successful.

From my perspective, the most important aspect to investigate is the documentation of frameworks. Without good documentation, a lot of the investment in a framework will be lost.

The existing documentation approaches described in chapter 5 all have their deficiencies. Thus, to improve the documentation of frameworks, I am planning to make a case study to evaluate existing and new techniques for documentation of frameworks. The objective of the study is to find a useful documentation technique for object-oriented frameworks.

7.1 The Framework Documentation Case Study

The Framework Documentation case study will address the documentation aspects for the framework user, the developer who will reuse the framework.

There exists at least two ways to evaluate documentation approaches: The first one will be to document a framework with different approaches, have a a set of software developers use one documentation approach each and let them give feed-back on the documenation.This approach will be difficult to evaluate since the feed-back will be quite diverse and approach-specific.

The second approach, the one I will use, is to collect good ideas from existing methods and documenting approaches; to collate this information and then present it to a number of software developers for application develop-

ment. This approach will generate feed-back which will be easier to draw conclusions from.

The main structure of the case study will be:

- Definition of an eclectic documentation approach to object-oriented frameworks
- Selection and documentation of an appropriate framework
- Application of the documentation approach to an existing framework
- Using the documentation in framework-based application development
- Evaluation of the documentation approach
- Improve the documentation approach
- Iterate

Definition of an eclectic documentation approach for frameworks

This activity will focus on finding techniques for describing the dynamic behaviour, the static structure and the usage of a framework. This will comprise selection and improvement of graphical notations and techniques from existing object-oriented methods and documentation approaches that emphasize the needs of a framework documentation.

Of particular interest are those notations and techniques that describe the dynamic behaviour of the framework since no good technique exists that describes the collaborative model of object-oriented software. In the case of object-oriented frameworks, there is furthermore a need to describe the collaboration between abstract classes (which are not addressed at all in existing object-oriented methods). The documentation of the dynamic behaviour will probably incorporate the concept of contracts and variants of patterns (e.g behavioural design patterns).

Regarding the documentation of the static structure, existing class and object diagrams together with design patterns and architectural frameworks, will serve as a good basis to start from.

Except for documentation dealing with the dynamic behaviour and static structure of the framework, low-level documentation for using the frameworks has to be developed. Currently, this kind of information is missing or hidden in existing documentation approaches and I will capture this knowledge in what I call *usage rules.*

The result of this activity will be an eclectic documentation approach for object-oriented frameworks.

Selection and documentation of an appropriate framework

An object-oriented framework has to be selected and documented with the defined eclectic documentation approach. The framework selected will be a small or medium-sized framework. This is to enable the software developers to develop an application in a relatively short time. The framework will probably be in the domain of user interfaces since it is a quite general domain.

Application of the documentation approach to an existing framework

Here a set of software developers will develop an application based on the selected framework. The framework will be presented with the eclectic documentation approach defined.

Evaluation of the documentation approach

In this activity the developers will give feed-back on the documentation approach and report their experiences. Based on the feed-back, conclusions about the pros and cons of the approach will be drawn.

Improve the documentation approach

Based on the evaluation results the eclectic documentation approach will be improved. The result of this activity will be an improved documentation technique for object-oriented frameworks.

Iterate

If resources are available the case study will be carried out once again, this time with the improved documentation approach.

Expected results

The main results from the Framework Documentation case study will be:

- A complete documentation technique for using object-oriented frameworks that is based on experience.

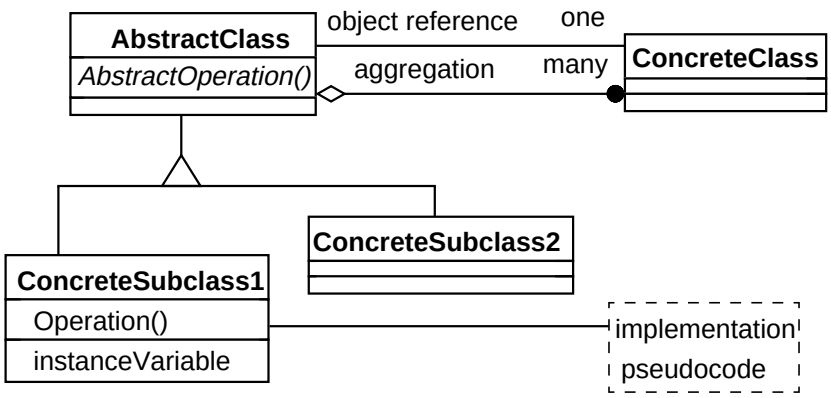
An additional result will be

- A new approach to documenting the necessary low-level information on how to use a framework, comprising a set of usage rules.

Beside the results related to the documentation aspect experiences gained by the software developers, framework-based application development knowledge will be gained and used for further and more detailed descriptions of methods for framework-based developments than that outlined in chapter 4.

Appendix A

The Object Modelling Technique Notation



Appendix B

Introductory Description of the Gamma Patterns

Abstract Factory provides an interface for creating generic product objects. It removes dependencies on concrete product classes from clients that create product objects.

Adapter makes the protocol of one class conform to the protocol of another.

Bridge separates an abstraction from its implementation. The abstraction may vary its implementation transparently and dynamically.

Builder provides a generic interface for incrementally constructing aggregate objects. A Builder hides details of how objects in the aggregate are created, represented, and composed.

Chain of Responsibility defines a hierarchy of objects, typically arranged from more specific to more general, having responsibility for handling a request.

Command objectifies the request for a service. It decouples the creator of the request for a service from the executor of that service.

Composite treats multiple, recursively-composed objects as a single object.

Decorator attaches additional services, properties, or behaviour to objects. decorators can be nested recursively to attach multiple properties to objects.

Facade defines a single point of access to objects in a subsystem. It provides a higher level of encapsulation for objects in the subsystem.

Factory Method lets base classes create instances of subclass-dependent objects.

Flyweight defines how objects can be shared. Flyweight supports object abstraction at the finest granularity.

Interpreter defines how to represent the grammar, abstract syntax tree, and interpretation for simple languages.

Iterator objectifies traversal algorithms over object structures.

Mediator decouples and manages the collaboration between objects.

Memento opaquely encapsulates a snapshot of the internal state of an object and is used to restore the object to its original state.

Observer enforces synchronisation, coordination, and consistency constraints between objects.

Prototype creates new objects by cloning a prototypical instance. Prototypes permit clients to install and configure dynamically instances of particular classes they need to instantiate.

Proxy acts as a convenient surrogate or placeholders for another object. Proxies can restrict, enhance, or alter an object's properties.

Singleton defines a one-of-a-kind object that provides access to unique or well-known services and variables.

State lets an object change its behaviour when its internal state changes, effectively changing its class.

Strategy objectifies an algorithm or behaviour.

Template Method implements an abstract algorithm, deferring specific steps to subclass methods.

Visitor centralizes operations on object structures in one class so that these operations can be changed independently of the classes defining the structure.

Appendix C

Introductory Description of the Buschmann Patterns

Actor-Supplier pattern: Applicable when a complex task can be divided into several simpler, context independent subtasks.

Composite-Part pattern: Can be used to protect components, which are used to compose a complex function, from unauthorized access or to treat a number of cooperating components as a single unit within a software system. Variants: Wrapper pattern

Mediator-Worker pattern: Can be used to decouple cooperating components of a software system and to realize them without having direct dependencies among each other. It can also be applied when services offered by individually developed or already existing components should be composed to realize a more complex functionality.

Abstract Factory pattern: Can be applied, when dependencies and variations on the creation and composition of complex components or subsystems should be removed from a client application or client module.

Envelope-Letter pattern: Can be applied to support multiple implementations of components or to change the implementation or behaviour of an object of a software system without any modification of its clients. The structure of this pattern is related but not identical to the one defined in [Cop91].

Proxy-Original pattern: Applicable whenever there is a need for a representative for a component in a different address space (remote proxy), for a component that should be loaded on demand (virtual proxy), or for a protected access to a component (protected proxy).

Subject-Snapshot pattern: Can be used, when it is necessary to snapshot the state of a component in order to be able to restore it later or when other modules which need information about the state of a component should not obtain access to its implementation or functionality.

Controller-Command pattern: Can be used in situations where the request for a service should be decoupled from its execution, when it is desired to provide functionality related to the execution of services, or when the execution of a service may vary.

View-Representation pattern: Can be applied when the general organization of views upon a software system should be separated from actions necessary to create and administrate a particular one, or when it is necessary to support different views upon the same subject or the same view for various output devices.

Collection-Iterator pattern: Can be used to handle the access to elements of a collection component and to support simultaneous iteration over the same collection. In addition, it can be used to provide multiple traversal strategies for a collection component as well as to support their variation.

Master-Slave pattern: Can be applied whenever it is necessary to handle the computation of replicated services within a software system.

Producer-Consumer pattern: Can be used for organizing the access to results of services provided by active suppliers within a structure of cooperating

Publisher-Subscriber pattern: Can be used to propagate changes and to maintain consistency among cooperating components of a software system.

Forwarder-Receiver pattern: Can be applied to separate and encapsulate inter process communication facilities used within distributed software systems with a statically known distribution of services.

Appendix D

The Prototype Design Pattern

Name PROTOTYPE **Classification** Object Creational

Intent

Specify the kinds of objects to be created dynamically by using a prototypical instance as a specification and creating new objects by copying it.

Motivation

You could build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves. The editor framework may have a palette of tools for adding these music objects to the score. The palette would also include tools for selecting, moving, and otherwise manipulating music objects. Users will click on the quarter-note tool and use it to add quarter notes to the score. Or they can use the move tool to move a note up or down on the staff, thereby changing its pitch.

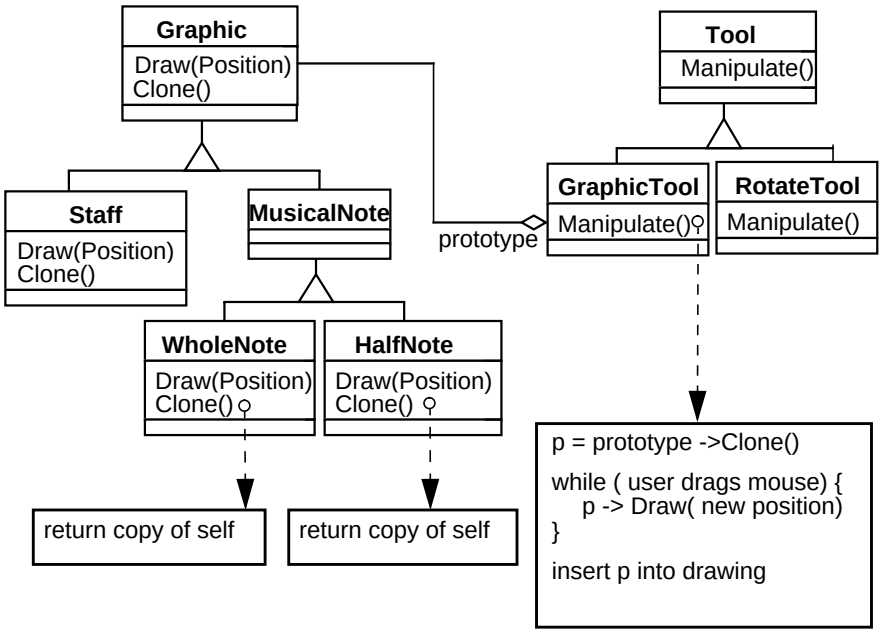
Let's assume the framework provides an abstract Graphic class for graphical components, like notes and staves. Moreover, it'll provide an abstract Tool class for defining tools like those in the palette. The framework also predefines a GraphicTool subclass for tools that create instances of graphical objects and add them to the document.

But GraphicTool presents a problem to the framework designer. The classes for notes and staves are specific to our application, but the GraphicTool class belongs to the framework. GraphicTool doesn't know how to create instances of our music classes to add to the score. We could subclass GraphicTool for each kind of music object, but that would produce lots of subclasses that differ only in the kind of music object they instantiate. We know object composition is a flexible alternative to subclassing. The ques-

tion is, how can the framework use it to parameterize instances of GraphicTool by the *class* of Graphic they're supposed to create?

The solution lies in making GraphicTool create a new Graphic by copying or "cloning" an instance of a Graphic subclass. We call this instance a prototype. GraphicTool is parameterized by the prototype it should clone and add to the document. If all Graphic subclasses support Clone operation, then the GraphicTool can clone any kind of Graphic.

So in our music editor, each tool for creating a music object is an instance of GraphicTool, that's initialized with a different prototype. Each GraphicTool instance will produce a music object by cloning its prototype and adding the clone to the score.



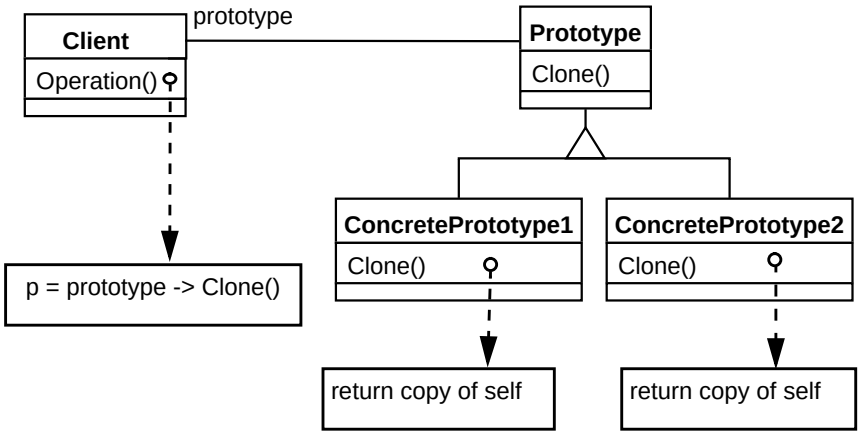
We can use the Prototype pattern to reduce the number of classes even further. We separate classes for whole notes and half notes, but that's probably unnecessary. Instead they could be instances of the same class initialized with different bitmaps and durations. A tool for creating whole notes

becomes just a GraphicTool whose prototype is MusicalNote initialized to be a whole note. This can reduce the number of classes in the system dramatically. It also makes it easier to add a new kind of note to the music editor.

Applicability

Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and

- when the classes to instantiate are specified at run-time, for example, by dynamic loading; or
- to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or
- when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually each time with the appropriate state.



Participants

Prototype (Graphic)

- declares an interface for cloning itself.

ConcretePrototype (Staff, WholeNote, HalfNote)

- implements an operation for cloning itself.

Client (GraphicTool)

- creates a new object by asking a prototype to clone itself.

Collaborations

- A client asks a prototype to clone itself.

Consequences

Prototype has many of the same consequences as Abstract Factory and Builder have: It hides the concrete product classes for the client, thereby reducing the names clients know about. Moreover, these patterns let a client work with applications-specific classes without modification.

Additional benefits of the Prototype pattern are listed below.

1. Adding and removing products at run-time. Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client. That's a bit more flexible than other creational patterns, because a client can install and remove prototypes at run-time.

2. Specifying new objects by varying values. Highly dynamic systems let you define new behaviour through object composition - by specifying values for an object's variables, for example - and not by defining new classes. You effectively define new kinds of objects by instantiating existing classes and registering the instances as prototypes of client objects. A client can exhibit new behaviour by delegating responsibility to the prototype.

This kind of design lets users define new "classes" without programming. In fact, cloning a prototype is similar to instantiating a class. The Prototype pattern can greatly reduce the number of classes a system needs. In our music editor, one GraphicTool class can create a limitless variety of music objects.

3. Specifying new objects by varying structure. Many applications build objects from parts and subparts. Editors for circuit design, for example, build circuits out of subcircuits. For convenience, such applications often let you instantiate complex, user-defined structures, say, to use a specific subcircuit again and again.

The Prototype pattern supports this as well. We simply add this subcircuit as a prototype to the palette of available circuit elements. As long as the composite circuit object implements Clone as a deep copy, circuits with different structures can be prototypes.

4. Reduced subclassing. Factory Method often produces a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern lets you clone a prototype instead of asking a factory method to make a new object. Hence you don't need a Creator class hierarchy at all. This benefit applies primarily to languages like C++ that don't treat classes as first-class objects. Languages that do, like Smalltalk and Objective C, derive less benefit, since you can always use a class object as a creator. Class objects already act like prototypes in these languages.

5. Configuring an application with classes dynamically. Some run-time environments let you load classes into an application dynamically. The Prototype pattern is the key to exploiting such facilities in a language like C++.

An application that wants to create instances of a dynamically loaded class won't be able to reference its constructor statically. Instead, the run-time environment creates an instance of each class automatically when it's loaded, and it registers the instance with a prototype manager. Then the application can ask the prototype manager for instances of newly loaded classes, classes that weren't linked with the program originally. The ET++ application framework has a run-time system that uses this scheme.

The main liability of the Prototype pattern is that each subclass of Prototype must implement the Clone operation, which may be difficult. For example, adding Clone is difficult when the classes under consideration already exist. Implementing clone can be difficult when their internals include objects that don't support copying or have circular references.

Implementation

This section discusses different implementation aspects of the pattern. The discussion is intentionally omitted.

Sample Code

This section presents code fragments that illustrate how to implement the pattern in C++ or Smalltalk.The source code is intentionally omitted.

Known Uses

References to existing applications, where the pattern have been applied, are described here. The references for this pattern are intentionally omitted.

Related Patterns

Abstract Factory: Prototype and Abstract Factory are competing patterns in some ways. They can also be used together, however. An Abstract Factory might store a set of prototypes from which to clone and return product objects.

Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well.

Bibliography

[Ada95] D. Adair, *Building Object-Oriented Frameworks*, AIXpert, February and May 1995

[Ale77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobsen, I. Fiksdahl-King, S. Angel, *A Pattern Language*, Oxford University Press, 1977

[And94] G. Andert, *Object Frameworks in the Taligent OS*, Proceedings of Compcon 94, IEEE CS Press, Los Alamitos, California, 1994

[App89] Apple Computer Inc., *MacAppII Programmer's Guide*, 1989

[Bec87] K. Beck, W. Cunningham, *Using Pattern Langugages for Object-Oriented Programs*, Position Paper for the Specification and Design for Object-Oriented Programming Workshop, The 3rd Conference on Object-Oriented Programming Systems, Languages and Applications, Orlando, USA, 1987

[Bec94] K. Beck, R. Johnson, *Patterns Generate Architectures*, In Proceedings of the 8th European Conference on Object-Oriented Programming, Bologna, Italy, 1994

[Bec94+] K. Beck, *The Smalltalk Idiom Column*, The Smalltalk Report, Vol. 4, All issues, 1994

[Bir73] G.M. Birtwhistle, O-J. Dahl, B. Myrhaugh, K. Nygaard, *Simula Begin*, Petrocelli/Charter, 1973

[Bir 94] A. Birrer, T. Eggenschwiler, *Leveraging Corporate Sofware Development*, Computer Science Research at

UBILAB, Strategy and Projects; Proceedings of the UBILAB '94 Conference, Zurich, September 1994, Universitätsverlag Konstanz, Konstanz, 1994

[Boe81] B.W. Boehm, *Software Engineering Economics*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1981

[Boe88] B.W. Boehm, *A Spiral Model of Software Development and Enhancement*, IEEE Computer, Vol. 21, No. 5, May 1988

[Boo86] G. Booch, *Object-Oriented Development*, IEEE Transactions on Software Engineering, Vol 12. No. 2, February 1986

[Boo91] G. Booch, *Object-Oriented Design with Applications*, Benjamin Cummings, 1991

[Boo94] G. Booch, *Object-Oriented Analysis and Design with Applications* , 2ed, Benjamin Cummings, 1994

[Bus93a] F. Buschmann, *Rational Architectures for Object-Oriented Systems*, Journal of Object-Oriented Programming, Vol. 6, No. 5, September 1993

[Bus93b] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M. Stahl, *Pattern-Oriented Software Architecture*, Draft copy, Siemens AG, 1993

[Bus94a] F. Buschmann, R. Meunier, *A System of Patterns*, Proceedings of the First Conference on Pattern Languages and Programming, Addison-Wesley, 1994

[Bus94b] F. Buschmann, *The Master-Slave Pattern*, Proceedings of the First Conference on Pattern Languages and Programming, Addison-Wesley 1994

[Coa92] P. Coad, *Object-Oriented Patterns*, Communications of the ACM, Vol. 35, No. 9, 1992

[Cop92] J.O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992

[Cop94a] J.O. Coplien, *Software Design Patterns: Common Questions and Answers*, AT&T Bell Laboratories, 1994

[Cop94b] J.O. Coplien, *Generative Pattern Languages: An Emerging Direction of Software Design*, Proceedings of the 5th Annual Borland International Conference, 1994

[Cot95] S. Cotter, M. Potel, *Inside Taligent Technology*, Addison-Wesley, 1995

[Cun94] W. Cunningham, *The CHECKS Patterns Language of Information Integrity*, Proceedings of the First Conference on Pattern Languages and Programming, Addison-Wesley 1994

[DeB94] D.L. DeBruler, J.O. Coplien, *A Generative Pattern Language for Distributed Processing*, Proceedings of the First Conference on Pattern Languages and Programming, Addison-Wesley 1994

[Deu89] L.P Deutsch, *Design Reuse and Frameworks in the Smalltalk-80 system*, In T. J. Biggerstaff and A. J. Perlis, editors, Software Reusability, Vol II, ACM Press, 1989

[Ell90] M. Ellis, B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990

[Fir94] D.G. Firesmith, *Frameworks: the golden path to object Nirvana*, Journal of Object-Oriented Programming, Vol. 7 No. 8, 1994

[Foo92] B. Foote, *A Fractal Model of the Lifecycle of Reusable Objects*, Position Paper for the Reuse Workshop, The 7th Conference on Object-Oriented Programming Systems, Languages and Applications, Vancouver, USA, 1992

[Foo94] B. Foote, W. F. Opdyke, *Consolidation and Aggregation Patterns that Support Evolution and Reuse*, Proceedings of the First Conference on Pattern Languages and Programming, Addison-Wesley, 1994

[Gam92] E. Gamma, *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools* (in German), Springer-Verlag, 1992

[Gam93a] E. Gamma R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Abstraction and Reuse of Object-Oriented Design*, Proceedings of the 7th European Conference on Object-Oriented Programming, Kaiserslautern, Germany, 1993

[Gam93b] E. Gamma R. Helm, R. Johnson, J. Vlissides, *A Catalog of Object-Oriented Design Patterns*, draft copy, September 1993

[Gam95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[Gar93] D. Garlan, M. Shaw, *An Introduction to Software Architecture*, in Advances in Software Engineering and Knowledge Engineering, Vol. 1, World Scientific Publishing Company, 1993

[Gar94] D.Garlan, R. Allen, J. Ockerbloom, *Exploiting Style in Architectural Design Environments*, Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering, December 1994.

[Gol83] A. Goldberg, D. Robson, *Smalltalk-80, The language and its implementation*, Addison-Wesley, 1983

[Gol84] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, 1984

[Hel90] R. Helm, I.M. Holland, D. Gangopadhyay, *Contracts:Specifying Behavioural Compositions in Object-Oriented Systems*, Proceedings of the 4th European Conference on Object-Oriented Programming/The 5th Conference on Object-Oriented Programming Systems, Languages and Applications Conference, Ottawa, Canada, 1990

[Hun95] H. Huni, R. Johnson, R. Engel, *A Framework for Network Protocol Software*, Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications Conference, Austin, USA, 1995

[Hur94] W.L. Hursch, *Should Superclasses be Abstract ?*, Proceedings of the 8th European Conference on Object-Oriented Programming, Bologna, Italy, 1994

[Jac92] I. Jacobsson, M. Christerson, P. Jonsson, G. Övergaard, *Object-Oriented Software Engineering*, Addison-Wesley, 1992

[Joh88] R.E. Johnson, B. Foote, *Designing Reusable Classes*, Journal of Object-Oriented Programming, Vol 1, No. 2, June 1988

[Joh91] R.E. Johnson, *Reusing Object-Oriented Design*, University of Illinois, Technical Report UIUCDCS 91-1696, 1991

[Joh92] R.E. Johnson, *Documenting Frameworks with Patterns*, Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages and Applications, Vancouver, Canada, 1992

[Joh93] R.E. Johnson, *How to Design Frameworks,* Tutorial Notes, 8th Conference on Object-Oriented Programming Systems, Languages and Applications , Washington, USA, 1993

[Joh93b] R.E. Johnson & W.F. Opdyke, *Refactoring and Aggregation,* In Proceedings of ISOTAS '93: International Symposium on Object Technologies for Advanced Software, 1993

[Kar95] E-A. Karlsson, Editor, *Software Reuse - a Holistic Approach*, John Wiley & Sons, 1995

[Kih95] M. Kihl, P. Ströberg, *The Business Value of Software Development with Object-Oriented Frameworks*, Master

Thesis, Department of Business Administration and Computer Science, University of Karlskrona/Ronneby, Sweden, May 1995 (in Swedish)

[Kra88] G. E. Krasner, S. T. Pope, *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*, Journal of Object-Oriented Programming, Vol. 1, No. 3, August-September 1988

[Laj94] R. Lajoie, R. K. Keller, *Design and Reuse in Object-oriented Frameworks: Patterns, Contracts and Motifs in Concert*, Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l'Avancement des Sciences, Montreal, Canada, May 1994

[Lea94a] D. Lea, *Christopher Alexander: An Introduction for Object-Oriented Designers*, Software Engineering Notes Vol 19, No 1, Jan 1994

[Lea94b] D. Lea, *Design Patterns for Avionics Control Systems*, http://g.oswego.edu/dl/acs/acs/acs.html, 1994

[Lin89] M.A. Linton, J.M. Vlissides, P.R. Calder, *Composing User Interfaces with Interviews*, Computer, Vol 22., N0. 2, February 1989

[Lin90] J.L. Lindskov-Knudsen, G. Hedin, B. Magnusson, R.H. Trigg, E. Sörensen-Sandvad, *Documenting Object Oriented Systems*, Proceedings of the Nordic Workshop on Programming Environments Research, Editors O. Solberg, A. Venstrand, Trondheim, Norway, 1990

[Man92] D. Mandrioli, B. Meyer, *Advances in Object-Oriented Software Engineering*, Prentice Hall, 1992

[Mar94] J.J. Marciniak, Editor, *Encyclopedia of Software Engineering*, John Wiley & Sons, 1994

[Mat94] M. Mattsson, L. Ohlsson, *Flexible Class Hierchies using Abstract Classes*, Position Paper for the Pattern Workshop,

8th European Conference on Object-Oriented Programming, Bologna, Italy, 1994

[Mat95] M. Mattsson, *A Comparative Study of Three New Object-Oriented Methods*, Research Report 2/95, University of Karlskrona/Ronneby, Sweden

[Meu95] R. Meunier, *The Pipes and Filters Architecture*, Proceedings of the First Conference on Pattern Languages and Programming , Addison-Wesley, 1994

[Mey88] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988

[Mey92] B. Meyer , *Applying Design by Contract*, IEEE Computer, October 1992

[Mil95] H. Mili, F. Mili, A. Mili, *Reusing Software: Issues and Research Directions*, IEEE Transactions on Software Engineering, Vol. 21, No. 6, June 1995

[Ohl93a] L. Ohlsson, *Using Object Types for Framework Design*, Utilia Consult AB, Technical Report, 1993

[Ohl93b] L. Ohlsson, *The Next Generation of OOD*, Object Magazine, May-June 1993

[Opd90] W.F. Opdyke, R.E. Johnson, *Refactoring: An aid in designing object-oriented application frameworks*, Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications, 1990

[Opd92] W.F. Opdyke, *Refactoring Object-Oriented Frameworks*, PhD thesis, University of Illinois at Urbana-Champaign, 1992

[Opd93] W.F Opdyke, R.E Johnson, *Creating Abstract Super-Classes by Refactoring*, Procedings of CSC'93: The ACM 1993 Computer Science Conference, February 1993

[Pal93] Palmer J., *Object-Oriented Analysis and Polymorfism: Gross concept neglected*, Object Magazine, March 1993

[Pre94a] W. Pree, *Framework Adaption via Active Cockbooks and Framework Construction via Relations*, Demonstration Paper, The 8th European Conference on Object-Oriented Programming, Bologna, Italy, 1994

[Pre94b] W. Pree, *Meta Patterns - A means for capturing the essential of reusable object-oriented design*, Proceedings of the 8th European Conference on Object-Oriented Programming, Bologna, Italy, 1994

[Pre94c] W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley ACM Books, 1994

[Rus90] V. F. Russo. *An Object-Oriented Operating System*, PhD thesis, University of Illinois at Urbana-Champaign, October 1990

[Ros93] M.B. Ross, J.M. Carroll, *Developing Minimalist Education for Object-Oriented Programming and Design*, Tutorial Notes, The 8th Conference on Object-Oriented Programming Systems, Languages and Applications , Washington, USA, 1993

[Rum91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991

[Sch86] K.J. Schmucker, *Object-Oriented Programming for the Macintosh*, Hayden Book Company, 1986

[Sch94a] D.S Schmidt, *Reactor - An Object Behavioral Pattern for Event Demultiplexing and Event Handler Dispatching*, Proceedings of the First Conference on Pattern Languages and Programming, Addison-Wesley, 1994

[Sch94b] W. Schäfer, R. Prieto-Diaz, M. Matsumoto, *Software Reusability*, Ellis-Horwood Ltd., 1994

[Sch95a] D.S. Schmidt, P. Stephenson, *Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms*, Proceedings of the 9th European

Conference on Object-Oriented Programming, Aarhus, Denmark, 1995

[Sch95b] D.S. Schmidt, *Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software*, Communications of the ACM, Vol. 38, No 10, October, 1995

[Sch95c] D.S Schmidt, G. Lavender, A*ctive Object: An Object Behavioral Pattern for Concurrent Programming*, Submitted to the Second Pattern Languages of Programs Conference , Monticello, Illinois, 1995

[Sch95d] H. A. Schmid, *Creating the Architecture of a Manufacturing Framework by Design Patterns*, Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications Conference, Austin, USA, 1995

[Tal93] Taligent Inc., *Leveraging Object-Oriented Frameworks*, A Taligent White Paper, 1993

[Tal94] Taligent Inc., *Building Object-Oriented Frameworks*, A Taligent White Paper, 1994

[Tal94b] Taligent Inc, *Taligent's guide to designing programs - well-mannered object-oriented design in C++*, Addison-Wesley, 1994

[Tea93] S. Teale, *C++ IOStreams handbook*, Addison-Wesley, 1993

[Wei89] A. Weinand, E. Gamma, R. Marty., *Design and Implementation of ET++, a Seamless Object-Oriented Application Framework*, Structured Programming, Vol. 10, No. 2, July 1989

[Wei 94] Weinand, A., Gamma, E., *ET++ - a Portable, Homogenous Class Library and Application Framework*, Computer Science Research at UBILAB, Strategy and Projects; Proceedings of the UBILAB '94 Conference, Zurich, Sep-

tember 1994, Universitätsverlag Konstanz, Konstanz, 1994

[Wil93] D. A. Wilson, S. D. Wilson, *Writing Frameworks - Capturing Your Expertise About a Problem Domain*, Tutorial notes, The 8th Conference on Object-Oriented Programming Systems, Languages and Applications, Washington, 1993

[Wir90a] R. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice-Hall, 1990

[Wir90b] R.J. Wirfs-Brock, R. E. Johnson, *Surveying Current Research in Object-Oriented Design*, Communications of the ACM, Vol. 33, No. 9, September 1990

[You92] E. Yourdon, *Decline & Fall of the American Programmer*, Prentice-Hall, 1992

[Zim94a] W. Zimmer, *Experiences using Design Patterns to Reorganize an Object-Oriented Application*, Position Paper for the Pattern Workshop, The 8th European Conference on Object-Oriented Programming, Bologna, Italy, 1994

[Zim94b] W.Zimmer, *Relationships between Design Patterns*, Proceedings of the First Conference on Pattern Languages and Programming, Addison-Wesley, 1994