Department of Computer Science
Series of Publications C
Report C-2002-47

Automatic Extraction of Framework Specialization Patterns

Jukka Viljamaa

Licentiate Thesis

University of Helsinki
Finland

Contact information

Postal address:

Department of Computer Science
P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 191

Telefax: +358 9 1914 4441

Computing Reviews (1998) Classification: D.2.2, D.2.6, D.2.11, D.2.13
Helsinki 2002
Yliopistopaino
Pikapaino

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Fakultet/Sektion — Faculty Faculty of Science	Laitos — Institution — Department Department of Computer Science

Tekijä — Författare — Author
Jukka Viljamaa

Työn nimi — Arbetets titel — Title
Automatic Extraction of Framework Specialization Patterns

Oppiaine — Läroämne — Subject
Computer science

Työn laji — Arbetets art — Level Licentiate thesis	Aika — Datum — Month and year September 2002	Sivumäärä — Sidoantal — Number of pages 91 + 31

Tiivistelmä — Referat — Abstract

Reuse is an important factor in software development. It enhances productivity and increases product quality. Reuse of software architecture and high-level design has proven to be especially effective. *Object-oriented application frameworks* provide an established mechanism for reusing both functionality and architecture of systems. Usability of frameworks remains a problem, however. The complexity, variability, and abstract nature of frameworks make them difficult to specialize. It is therefore essential that a framework is delivered with proper documentation describing its purpose, usage, overall architecture, and detailed design.

For an application developer it is especially crucial that the framework's *reuse interface* is well documented. *Design patterns* provide a means to express that interface in a systematic way. In the Fred framework engineering environment, a pattern formalism is introduced to enable task-driven assistance for framework specialization. Based on the patterns that are used to annotate the reuse interface, Fred offers code generation, dynamically adjusted user documentation, and checking of architectural constraints.

In this thesis, various *reverse engineering* techniques are analyzed in the context of recovering a framework's architecture from the source code of the framework itself and a set of available example applications. In particular, *framework hot spot* recovery and pattern-based annotation of the framework's reuse interface are discussed. A *concept analysis* algorithm is adapted to produce Fred patterns from Java source code, and the algorithm is implemented within the Fred tool set. In a case study, automatic pattern extraction is conducted for the JUnit testing framework. The effectiveness of the method is assessed through a comparison to a manual annotation of the same framework.

Computing Reviews Classification (1998): D.2.2, D.2.6, D.2.11, D.2.13

Avainsanat — Nyckelord — Keywords
framework, usability, documentation, pattern, reverse engineering, concept analysis

Säilytyspaikka – Förvaringställe — Where deposited
Library of the Department of Computer Science, serial number C-2002-47

Muita tietoja – Övriga uppgifter — Additional information

Acknowledgements

The pattern extraction method and its implementation are a part of the Fred framework engineering environment that was initially developed in the Fred project (Framework Editor for Java — Support for Framework Development and Use) carried out at University of Helsinki, University of Tampere, and Tampere University of Technology in 1997-1999. The Fred environment is currently being maintained and developed further in the JavaFrames project (see [FRE02] for new versions and up-to-date information). The National Technology Agency of Finland (TEKES) and several industrial partners have been financially supporting both Fred and JavaFrames.

I have very much enjoyed working in the Fred team, and I have learned a lot from my colleagues. Most of all I would like to thank professors Jukka Paakki and Kai Koskimies for their valuable comments and advice. My brother, Antti Viljamaa, has also been an excellent co-worker. He never got tired of reading early drafts and providing constructive criticism.

Finally, I would like to thank my Mum, Dad, and Tuija for their endless support.

Automatic Extraction of Framework Specialization Patterns

Contents

1. Introduction

Recent empirical evidence shows that *reuse* is an important factor in enhancing software production [BBM96, MoN96, RiN00, SBF96]. It reduces the cost of development and improves the quality and maintainability of systems. In [RiN00] Rine and Nada define a reference model of factors that affect software reuse. They present an extensive collection of empirical data supporting the hypothesis that systematic reuse increases productivity and product quality and decreases development time and time-to-market. Based on the experiences gained from surveying numerous organizations and conducting nearly 30 detailed case studies they conclude that the most important indicators of successful reuse are product-line approach with usage of standardized architecture, interfaces, and data formats for all applications, reuse of high-level artifacts (e.g. requirements and design), usage of state-of-the-art tools and methods, and committed management.

From the software engineer's point of view, the key to maximal reuse lies in anticipating new requirements and designing software so that it can be changed accordingly. *Object-orientation* in design and programming promises to fulfill the demand for this kind of flexibility [Boo96, BoR96]. However, applying an object-oriented language or design methodology does not automatically guarantee productivity. To enable full-scale reuse and to avoid expensive redesign, flexibility has to be explicitly built into the systems [Cli96, PVR99].

There are various levels of reuse in software production [Deu89]. *Internal reuse* means designing the system's abstractions so that they can be used in many parts of the same application. When general-purpose functions are gathered into libraries that are applicable across several applications and domains, we talk about *code reuse*.

Reuse in object-oriented systems has been mainly achieved through the use of *class libraries*, which are the object-oriented equivalents of traditional *subroutine libraries*. They offer general services, such as basic data structures or stream IO (i.e. code reuse), but seldom *architectural* or *design level reuse*. That is why library usage does not significantly reduce the effort required to produce complex systems. That is also why the object community has shifted its attention from class libraries to *application frameworks* [FaS97, FSJ99a, MaS99].

Object-oriented frameworks represent the state of the art of reusing both code and design to build large applications. A framework defines the main concepts of its application domain as abstract interfaces. It nails down the overall architecture of the applications derived from it by defining the relationships between the main concepts and also the default functionality and algorithms associated with them. It provides a *reuse interface* consisting of *hot spots* (i.e. variation points) where programmers can plug in their concrete application-specific classes [Pre95]. Furthermore, a framework usually comes with a library of ready-made concrete components that the application developer can directly use in her application.

To effectively use a framework in software production, the developer must have clear understanding of the hot spots of the framework. That is why a framework should be accompanied with comprehensive documentation and a set of examples illustrating its purpose and usage. It has been suggested that the documentation should be given in the form of *cookbooks* or *patterns* [Joh92, KrP88].

Unfortunately framework documentation can be inadequate or missing all together. Many times simple example applications are the only assistance provided for the application developer. They offer only a means to get started, however. Advanced and creative framework usage requires always in-depth knowledge of the architecture and also the implementation of the framework. Without documentation the developer is forced to acquire that knowledge from the source code, which is obviously a slow and tedious way to go. The lack of documentation can increase the temptation to use the framework in an *ad-hoc* (or trial-and-error) manner. This, in turn, has implications for the quality of the resulting software, because improper reuse of framework components often results in bugs or incompatibilities at a later stage of software development [CDS97].

We believe that successful framework-based application development requires systematic methods and tool support. In our vision a future framework is accompanied with a wizard that provides dynamically changing goal-oriented documentation and automatic code generation. The tool should also enforce the architectural constraints of the framework. Such a tool can be implemented on top of a *role-based* framework annotation formalism.

Producing and maintaining a framework annotation is, of course, laborious and demanding. It would be of great help for framework developers to have even semi-automatic tool support for producing this kind of organized documentation. *Reverse engineering* is a discipline that tries to extract high-level descriptions of a system from its source code to supplement or replace missing or outdated documentation [Arn92, ChC96, HRY95, WaC94]. The problem of obtaining precise understanding of the reuse interface of a framework resembles the general problem of legacy system maintenance. Therefore we argue that reverse engineering techniques can also be used to recover valuable information for framework annotation.

In this thesis a *concept analysis algorithm* [SiR97, ToA99] is adapted to produce *framework specialization patterns* from Java [Sun02a] source code of the framework itself and a set of available example applications. The algorithm has been implemented within the context of Fred, a framework engineering environment that provides task-driven assistance for framework specialization [FRE02, HHK01a, HHK01b]. Fred offers code generation, dynamically adapting documentation, and checking of architectural constraints based on the patterns that annotate the reuse interface of the framework.

The rest of this thesis is organized as follows. In chapter 2 the basic characteristics of object-oriented application frameworks are introduced. Chapter 3 discusses more extensively the requirements and methods for framework documentation and the possibilities for supporting framework usage. Chapter 4 first introduces reverse engineering and the general properties of program understanding systems. Then, automatic detection of design patterns and framework hot spots is discussed in more detail.

Chapter 5 describes a pattern-based method to annotate framework hot spots. It also shows how concept analysis can be used to automatically produce such annotations from framework implementation or detailed design. The Fred framework engineering environment, and the pattern extraction tool that has been implemented as a constructive part of this work are introduced in chapter 6. A case study where automatic pattern extraction is conducted for the JUnit testing framework is also represented, and the resulting annotation is compared to a hand-written annotation prepared for the same framework. Finally, chapter 7 concludes the thesis.

2. Object-Oriented Application Frameworks

An *object-oriented application framework* is a collection of classes implementing the shared architecture and the common functionality of a family of applications [Deu89, FaS97, FSJ99a, JoF88]. An *application developer* adapts the framework to produce an actual application by customizing, configuring, and instantiating appropriate framework classes produced by the *framework developer*.

There are several benefits to be gained from using a framework. It offers reuse on a higher level than individual classes of a class library because it incorporates a common architecture for the domain, the main interactions between the classes, and a uniform interface to the infrastructure operations. For the developers this kind of standardization means good maintainability and easier communication. For the application users the benefit is that framework usage results in consistent user interfaces. From the management perspective the biggest benefit is increased productivity because development becomes faster and alterations can be made quickly by smaller development teams. Higher percentage of reused code also increases the overall product quality.

The Smalltalk environment was the first programming environment to systematically utilize frameworks [Deu89, GoR89]. In the beginning of 1990s Taligent's work on CommonPoint frameworks [CoP95] attained interest in larger communities. Many of the early frameworks supported construction of GUI applications (e.g. Unidraw [VlL90], HotDraw [Joh92], and VisualWorks [Yel96]). GUI development benefits a lot from a framework-based approach because GUIs are widely used and they consist of complex but relatively repetitive code. By now, there are mature frameworks available for a multitude of domains [FaJ00, FSJ99b, HJE95]. Examples range from operating systems [CIR93], neural networks [BeP99], and compilers [JKN95] to banking [EgG92], cryptographic protocols [NiP99], and 2D action videogames [SaN01].

In [FSJ99a] frameworks are classified by their scope to *system infrastructure frameworks* (e.g. operating system or communication frameworks), *middleware integration frameworks* (e.g. ORB frameworks, transactional databases), and *enterprise application frameworks* (i.e. frameworks that are specific to the business domain of the enterprise, e.g. telecommunications, avionics, manufacturing, or financial services). Another classification is given in [Nem97] where frameworks are grouped to *application frameworks* (e.g. Java Foundation Classes [Sun02b]), *domain frameworks* (e.g. banking systems, alarm systems), and *support frameworks* (e.g. memory management systems, file systems).

In this chapter we will give an overview of framework design (2.1), framework implementation (2.2), and framework-based application development (2.3). A more detailed discussion of how frameworks are used and documented will follow in chapter 3.

2.1 Framework Design and Evolution

Designing frameworks is difficult. This is because in a reusable design there are always many important classes and objects that have no counterparts in the real world and thus cannot be readily deduced from the concepts of the application domain. In fact, as Gamma *et al.* note in [GHJ95]: "*strict* *modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's.*"

Reusable architectures rely on various kinds of *organizational objects* that are often found in the later phases of design after many modifications. These less-obvious abstractions involve, for example, representing an algorithm or an association as a separate object. Other examples would be various kinds of control and policy objects. This kind of *reification* or *objectification* of behavior with additional objects introduces flexibility into designs by separating domain objects from the activities in which they are employed [WoL95, Zim95].

The main problem in framework design is to decide which properties of the system should be left for the user to define, and how this flexibility should be expressed so that the system remains easy to comprehend. Too little flexibility makes the framework hard to customize and to adapt to various needs. On the other hand, too much flexibility can result in a framework that is difficult to understand. Added dynamic behavior and indirection in implementation may also introduce system overhead and inefficiency.

Black-box reuse is defined in [Szy97] as the concept of reusing implementations without relying on anything but their interfaces and specifications. *White-box reuse*, on the other hand, means using a software fragment through its interfaces, while relying on the understanding gained from studying the actual implementation. Framework evolution is usually characterized by a gradual transformation from a *white-box framework* to a *black-box framework*. Immature frameworks depend mainly on inheritance in providing flexibility. Users of this kind of a white-box framework are forced to acquire knowledge about the internals of the classes they have to specialize.

As the framework evolves, *object composition* becomes more dominating making the framework easier to use. Only the external interfaces of the framework classes and objects are exposed to the users. Of course such a black-box framework is harder to design. It usually takes several trials and errors to get the interfaces right. The capabilities of pure black-box frameworks are limited to the functionality provided by the predefined components. For that reason, most frameworks provide also white-box mechanisms to enable advanced ways of reusing the core abstractions [FSJ99a].

In [DuJ96] Durham and Johnson show how black-box frameworks can evolve even further. They note that frameworks are targeted at expert programmers, whereas *domain-specific visual languages* are intended for domain experts. They argue that a black-box framework can be augmented with a special-purpose visual language, and that with such a language a domain expert can directly specify systems without actual programming.

As a case study, Durham and Johnson describe a run-time system framework for programming language implementations. They transformed the framework from a collection of classes representing the domain knowledge into a set of orthogonal components. After that, they built a graphical front end for the framework with different kinds of browsers for easier usage. They gave each component in the system a visual tool and let the user organize the whole system by instantiating and connecting components.

In principle, a framework development process can be divided into *problem generalization* and *framework design* as described in figure 2.1 [KoM95]. Problem generalization starts from a representative application (or preferably from a set of applications) and generalizes it set-by-step into the most general sensible form. During framework design an implementation for each generalization is given in terms of a framework. Framework design typically involves application of appropriate *design patterns* [GHJ95] to attain the required variability.

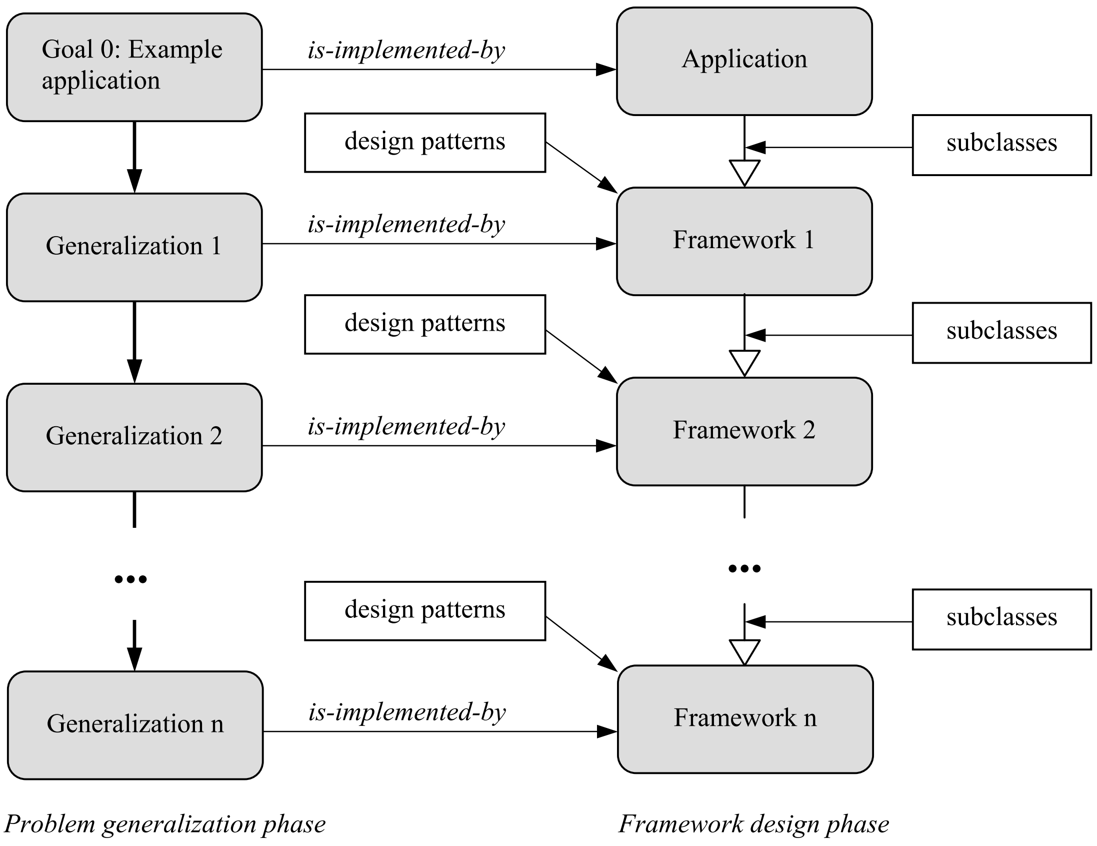


Figure 2.1: Framework design by stepwise generalization [KoM95]

A framework on level *i specializes* the framework on the next general level *i+1*. Framework specialization may require, e.g., adding a few concrete subclasses that implement the missing functionality. This approach leads to a hierarchy of more and more refined frameworks, and the initial example application is derived from the most concrete framework. This demonstrates that the framework is applicable in the representative case that was chosen as the starting point for the problem generalization.

2.2 Framework Implementation

In practice, framework development consists of recurring prototype, expansion, and consolidation phases [FoO95, GuB01, Sch95]. Initially, the design process starts with a simple domain-specific model that results from an analysis of a particular configuration. This configuration may be a set of prototypes and toy programs with which the developers have been exploring the characteristics of the domain or, if the domain is mature enough, a representative set of actual production applications developed earlier and independently from the framework project.

In the expansion phase, a sequence of transformation steps is performed to make the framework's functional logic configuration independent. Each step increases the reusability of the system by applying a suitable object-oriented technique, such as inheritance, to a specific part of the design to turn it into a hot spot. During consolidation some object relationships, originally modeled using inheritance, may evolve into delegation. Also, some variant implementation strategies originally modeled in applications may be generalized and moved to the framework.

A mature framework implementation typically has a layered structure similar to the one described in figure 2.2. On the highest abstraction level the main concepts of the domain are expressed as purely abstract interfaces (e.g. *Figure*, *Tool*, *ToolBar*, *Connection*, and *ConnectionFigure*). This *interface layer* directly reflects the design documents of the system by defining the framework's basic objects and their services.

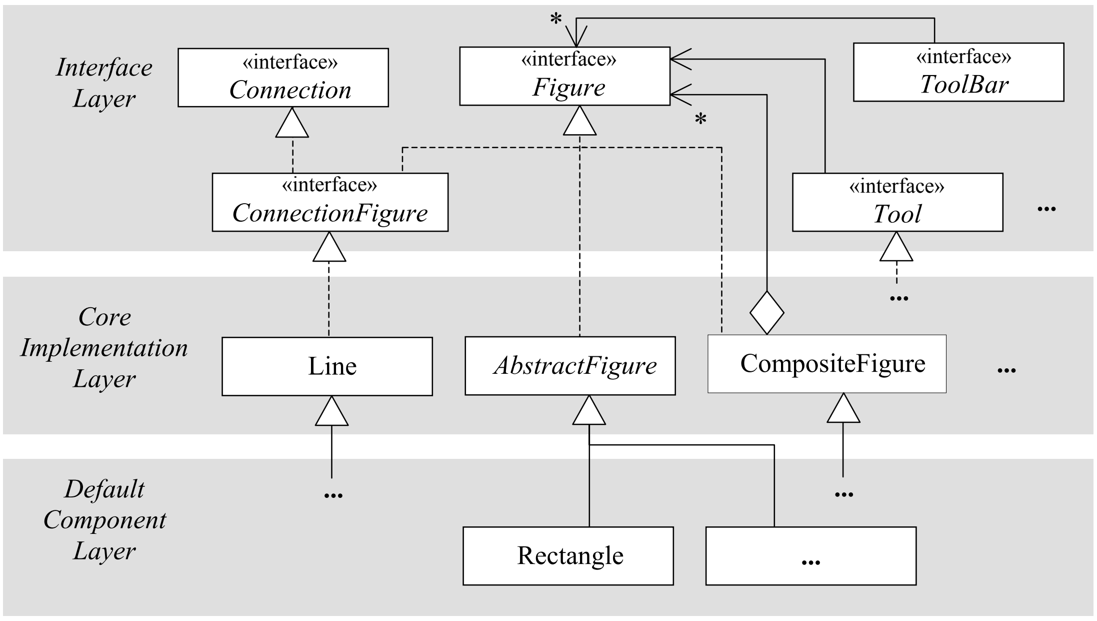


Figure 2.2: Typical layered structure of a framework

Below the interface layer there is a layer of abstract classes that define the main parts of the framework's core algorithms but leave some application-specific parts open (e.g. *AbstractFigure*). This *core implementation layer* supplies the fundamental functionality of the framework by partially implementing the interfaces with abstract classes. It also makes the hot spots of the framework explicit by allowing the user to override some of the functions (*hook methods*), whereas the underlying algorithms calling the hook methods (*template methods*) are usually hidden from the user. Although in principle it is possible to directly implement any interface, the application developer usually specializes classes from the core implementation layer.

On the third layer there is a library of *default components* provided by the framework (e.g. *Rectangle*). It comprises of full default implementations (concrete classes) of the framework's concepts for commonly recurring circumstances. The user can select those classes that she needs and assemble them together in the context of the overall architecture of the framework in order to get a working application.

A typical framework interface (or a class) supports a large number of features, many of which are not needed in most applications. It has been suggested, however, that interfaces should be *narrow* in the sense that they should declare only the most rudimentary features for the concepts they describe. Interfaces should also be *role-oriented*, i.e. each interface should represent a distinct *role* that can be played (i.e. implemented) by a class. A class can naturally implement multiple roles (interfaces), and a role can be implemented by multiple classes. [CaL01, GuB01, RiG98]

When an application developer needs to use (or specialize) a framework concept, she should always use an appropriate narrow interface. This makes the intent of the usage relationship precise thus supporting the overall understandability of the system. Wide interfaces are more complex and harder to use because they comprise more methods, and also because the methods usually have more parameters than methods in narrow interfaces.

In general, using a service through an abstract interface allows the implementation of the referred interface to change — even at run-time. Furthermore, clearly defined interfaces make systems safer by explicitly expressing, which services of each component are supposed to be accessed from outside and by simultaneously hiding the other features. The main advantage of narrow interfaces, in particular, is that they can be combined to support practically any possible combination of features. Whenever a class needs to use another class in many roles (i.e. through multiple interfaces), a new derived interface for that purpose should be defined by combining the desired element roles with *role inheritance* (e.g. *ConnectionFigure* in figure 2.2). In this case the usual problems of multiple inheritance can be avoided since implementations are not inherited (consider, e.g., Java's interfaces that *extend* other interfaces). Role inheritance enables the developer to choose as narrow (or as wide) an interface as is appropriate. Role-oriented interfaces and interface inheritance should be used with care, though, because in some cases they may lead to an unnecessarily large number of small interfaces.

A good framework design strikes a balance between *overfeaturing* in the framework and *code replication* in applications [CDS97]. The problem is that what is appropriate for one application may appear to be overfeaturing for another. As long as a framework provider is forced to make a rigid distinction between framework code and application code, balancing those two factors will remain a compromise.

Modifications in a framework class often lead to modifications in the applications using the class, too. The usage of narrow interfaces helps to keep feature implementations orthogonal so that adding new features can be implemented by adding new framework classes instead of changing existing ones. And if changes are unavoidable, they can be localized to few selected classes, so that only those parts of the system that actually use the modified classes are affected.

When the features of the framework are declared via a set of narrow interfaces, adding new framework classes is as easy as adding new application classes. There is actually no need of making a sharp distinction between application classes and framework classes: application classes can be promoted to be included in the library of framework's predefined components if a need to use the component in multiple applications arises.

2.3 Framework Specialization

For a framework user the most crucial part of the framework is its reuse interface. She uses that interface to derive an application from the framework. This process is called *framework specialization* or sometimes also *framework adaptation* or *framework instantiation* [FSJ99a, GuB01]. The reuse interface consists of hot spots related to specializing or implementing the framework's abstract concepts (*specialization interface*) and calling or combining its concrete default components (*call interface*). Figure 2.3 shows how a simple diagram drawing application could be specialized from an imaginary drawing framework.

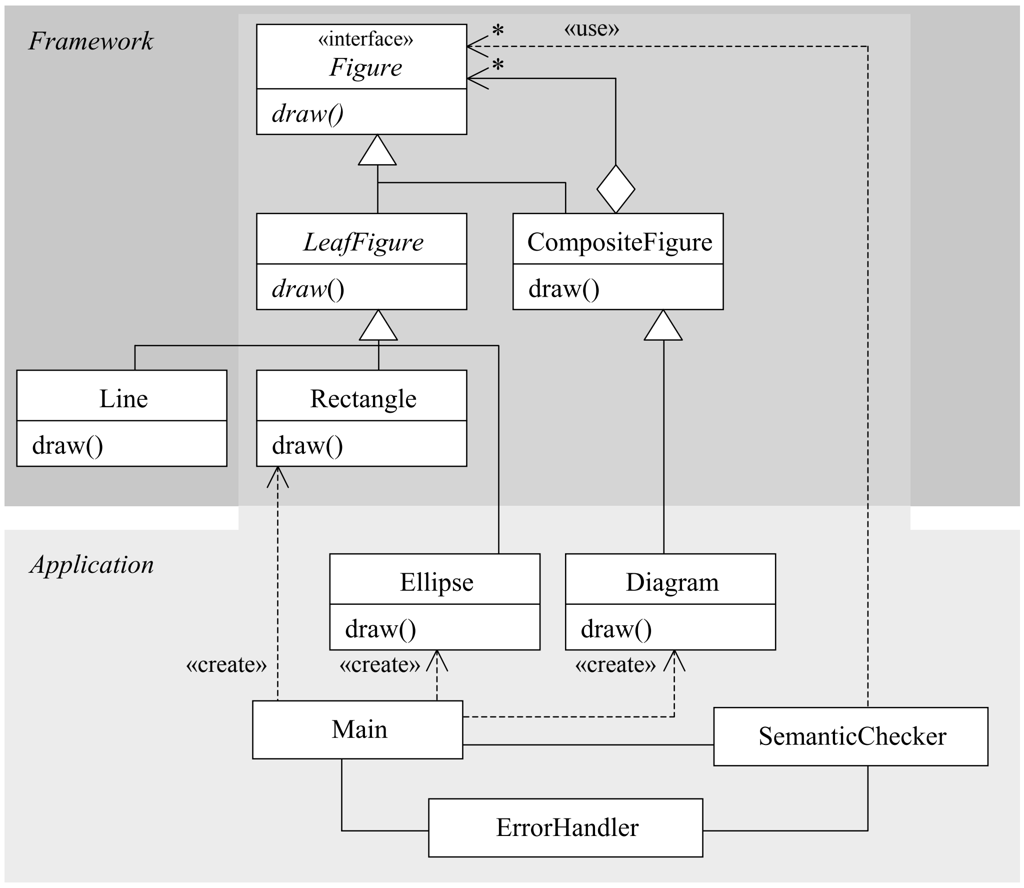


Figure 2.3: An application derived from a framework

Figure 2.3 illustrates how the application consists of interfaces and abstract classes defined in the framework (e.g. *Figure*, *LeafFigure*, *CompositeFigure*), ready-made concrete subclasses from the framework's component library (*Rectangle*), and application-specific subclasses (*Ellipse*) that fill in gaps of the abstract framework classes by providing concrete hook methods (e.g. *draw*), which the framework's template methods call via the use of dynamic binding.

In addition, there is always initialization and glue code to assemble together the pieces of the application (*Main*). Naturally, there can also be application-specific code that does not depend on the framework in any way (*ErrorHandler*) or classes that just call some services of the framework but are not derived from the framework (e.g. *SemanticChecker*).

Frameworks often define the control flow on a framework-calls-application basis (also known as the *Hollywood principle*) [LaN95, Vli96]. It means that the application developer must think in terms of the responsibilities of the objects and write handlers for various events that the framework dispatches. Figure 2.3 gives an example of implementing such reversed control flow. The *Diagram* class is a subclass of *CompositeFigure*. When *Diagram*'s *draw* method (inherited from *CompositeFigure*) is called, it just forwards the call to all *Diagram*'s children that adhere to the common *Figure* interface. Each elementary figure (i.e. subclass of *LeafFigure*) must render itself to screen in an implementation of the *draw* method without knowing who and where actually called it.

From a more abstract point of view framework-based application development can be described as a five-step process (see figure 2.4). First, the requirements for the application are gathered (1), and a *conceptual architecture* (i.e. the main components together with their functionality, relationships, and collaborations) is defined to meet the requirements.

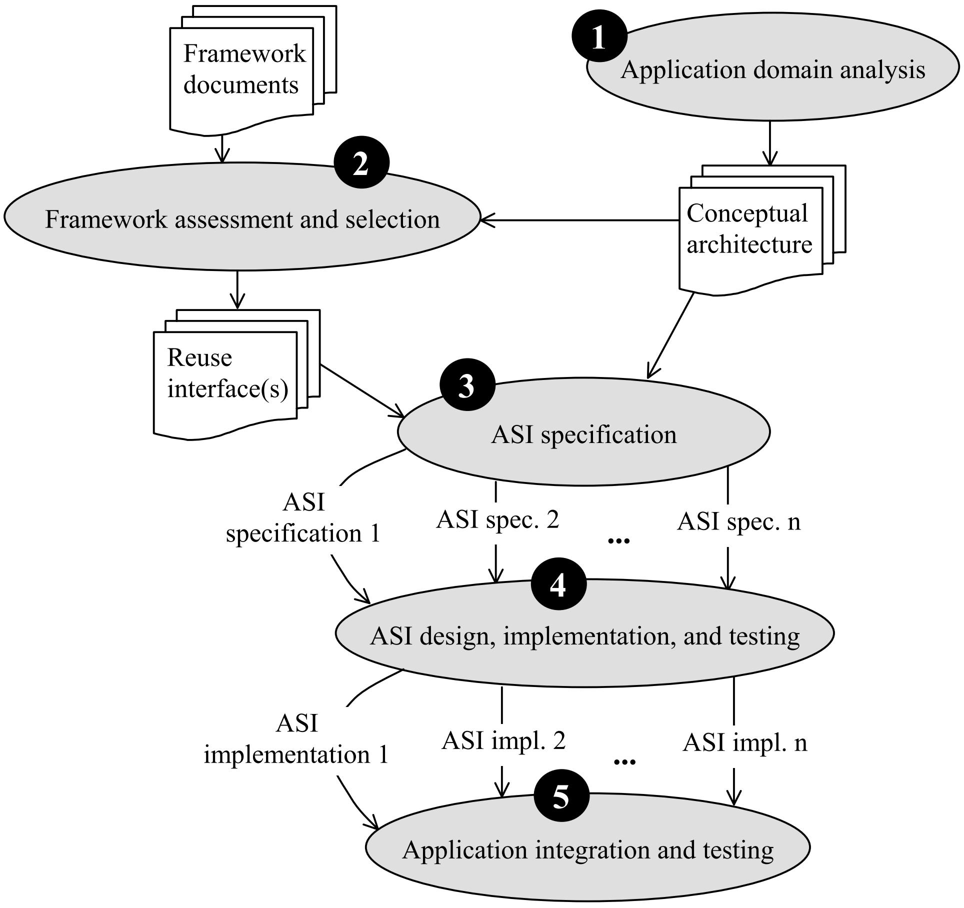


Figure 2.4: Framework-based application development

Next, the candidate frameworks are assessed to select those that provide the concepts and functionality best suited for the application (2). At this point, the application developer must choose between using a framework directly, modifying it to better fit the need (if the source code is available), or constructing the application from scratch. The choice is based on an estimate of how well the application matches the domain of the framework. This analysis should include, not only the functionality provided by the framework, but also non-functional issues, like interoperability and performance.

When a set of suitable frameworks has been selected, it is possible to specify the *application-specific increments* (3). An application-specific increment (ASI) describes a part of the implementation that specializes a framework to fulfill a particular requirement [BMM99]. It is based on the reuse interface of the framework and the requirement documentation. Note that using multiple frameworks may lead into *framework composition problems* if, for instance, several frameworks assume to have the main control loop [MBF99].

After the specification of the ASIs, they must be designed, implemented, and tested (4). This involves gluing together the selected ready-made components from the framework's component library, deriving application-specific components by implementing the selected interfaces directly or by specializing abstract classes, and providing application-specific classes that implement the functionality not covered by the framework. Finally, all ASIs are integrated with the underlying framework(s) and the complete application is tested against the original requirements (5). Note that even though there is no explicit iteration depicted in figure 2.4 it is very likely that backtracking in one form or another will occur also in framework-based application development.

3. The Framework Usage Problem

Most framework research has concentrated on framework design and construction, whereas framework usage has been studied less [SLB00]. This is surprising since framework usability is of great practical significance. There are many difficulties in using frameworks, and it has been estimated that it takes nearly a year for a professional programmer to master a large framework in order to be able to use it effectively [Boo96, FaS97].

The problems are most severe with white-box frameworks that are widely used in the industry. The dominance of white-box frameworks over the more easily usable black-box frameworks is mainly due to the immaturity of the framework technology, the immaturity of the domains it has been applied to, and the generally more flexible nature of white-box frameworks. Systematic methods and tool support are acutely needed to assist white-box framework usage because it is not likely that all (or even most) white-box frameworks would evolve to become (pure) black-box frameworks.

The biggest problem in applying frameworks is the steep learning curve [BMM99, MoN96, Pre95]. The understanding of the concepts of the framework may be required already when choosing the framework, but it becomes an absolute necessity when the application-specific increments are specified and designed. However, the complexity, variability, and abstract nature of framework classes make them difficult to understand. Interfaces and classes of a framework are designed to work together and they must be learned as a whole. That is why learning a framework is harder than learning a regular class library in which one can focus on individual classes one at a time.

The first use of a framework is usually the hardest so it is good if an expert (e.g. a member of the framework's development team) is available to give guidance for novice users. Although *mentoring* is a good way to teach frameworks [Jol99], in many cases it cannot be used. The main problem with mentoring is that the key persons are always needed somewhere else. That is why frameworks must have thorough user documentation.

There are three aspects to application framework documentation: the *purpose* of the framework, the *instructions* on its usage, and the *detailed design* of the framework [Joh92]. Johnson suggests that framework documentation should be arranged hierarchically so that first the framework's application domain is introduced by giving examples of what it can be used for. After that the general functionality of the framework should be explained by giving examples of typical use cases. Finally, the detailed design of the framework must be documented by illustrating how objects collaborate to provide the framework's functionality.

Documentation that describes the inner workings of a framework should concentrate first on conveying the intent of the design and its overall structure [BMM99, FSJ99a]. If the user does not understand the underlying principles of the design, the detailed rules and constraints defined by the framework developers do not make sense. The documentation should focus on the responsibilities of the objects and the interactions between them (both within the framework and especially between the framework and applications). These interactions, after all, are what differentiate frameworks from traditional class libraries.

Regardless of the methods that are being used, framework documentation must support both framework maintenance and specialization. In this thesis we are interested in the usability of frameworks so we will concentrate on the latter aspect of framework documentation. From the framework user's point of view, the documentation should be very goal-oriented and prescriptive. The documentation should be directed to users that might not be domain experts or experienced software engineers.

In chapter 3.1, we take a look at some empirical studies on framework usability and discuss the role of examples, diagrams, and reference manuals in learning frameworks. More advanced ways of documenting frameworks, such as *framework cookbooks* [KrP88], are presented in chapter 3.2. Chapter 3.3 gives a detailed account on using template and hook methods to implement the framework hot spots. In chapter 3.4 we discuss *patterns* [BeJ94, GHJ95, Joh92], which in our view provide the most natural way of annotating a framework reuse interface. Examples of various formalizations and tools for supporting framework usage are given in chapter 3.5. Finally, in chapter 3.6, we discuss general requirements for framework documentation and especially tool support for user guidance.

3.1 Framework Understanding

Software understanding is a central issue in software reuse [SLB00]. A software engineer must understand existing source code and documentation in order to be able to maintain a system or to reuse parts of it in a new system. An application developer using a framework, for example, needs to know the main classes of the framework and the functionality they provide. It is also significant that she is aware of the control flow of the framework and understands how the main objects of the framework interact. The most important thing, however, is that she can identify the hot spots of the framework, and that she understands the constraints and dependencies associated with them.

3.1.1 Example-Based Framework Learning

Many authors emphasize the role of *example applications* in learning frameworks [FSJ99a, Joh92, Jol99, LaN95, MRT98, SLB00]. An example application is a ready-made specialization provided with a framework distribution to illustrate the potential and mechanisms of the framework. Examples are concrete and thus easier to learn than the abstract framework in itself.

Examples are most useful when describing the purpose of the framework. The best way to illustrate a compiler framework's purpose, for example, is of course to show a compiler built with the framework. Examples provide a way to study the internal structure and the dynamics of the framework and, even more importantly, they demonstrate how the framework's classes can be instantiated, specialized, and used in an application. An example application can even be used as a starting point or a template for the very first specializations the application developer produces.

In [SLB00] Shull *et al.* examine how application developers proceed with framework specialization. Their empirical study suggests that an *example-based approach* to teaching frameworks works better than teaching the class hierarchy and object model of the framework. This is true at least when novice users are first learning the features of the framework, especially under time pressure.

The example-based approach allows the developers to start working immediately with a new framework. From the examples they get ideas of how to parameterize the framework classes to gain the required functionality. At the same time they can avoid the difficult process of navigating through class hierarchies to find the correct reusable classes.

The example-based approach to framework usage becomes problematic when the user has several alternative ways of implementing certain functionality. These alternatives might each have different effect on some non-functional aspect of the system, e.g. flexibility, efficiency, or reliability. It can be almost impossible to make a right choice in such a situation without good conception of the framework's architecture.

The example-based approach can sometimes prevent application developers from going beyond the functionality that is explicitly present in the example applications. A related problem is that application developers are tempted to first implement those features of the application that are most directly supported by the framework. This postpones the handling of a potential application-framework mismatch, which can be very expensive if the developers are forced to make radical changes to the application (or to the framework) very late in the implementation phase.

3.1.2 Understanding the Class Hierarchy and the Design Rationale

Even though programmers are usually quite skillful in inferring framework usage protocols from the provided code examples, the example-based approach does not support the construction of more evolved and complex adaptations. This is because the examples do not reveal the structure of a framework in a systematic way. Thus, advanced framework users need an efficient way of browsing through the structural aspects of frameworks in order to utilize them productively.

A *hierarchy-based approach* to teaching a framework is most often based on a variety of diagrams expressing the overall structure of the system and a reference manual for conveying the details [SLB00]. The static aspects of the framework can be expressed with *class diagrams* and *module dependency charts* and the dynamic aspects with *message sequence charts* and *state transition diagrams*, for example. A good way to tie these views together is to provide a set of carefully selected *scenarios* [Kru95], i.e. instances of use cases that show how different parts of the system interact and how this dynamism is reflected in the different views of the system. At the moment UML provides the most popular set of diagrammatic notations for visualizing the structure and behavior of a software system [RJB99].

Reference documentation for an object-oriented framework typically includes at least a hierarchically organized listing of class and method signatures together with a description of their semantics. The biggest problem with reference manuals is that they require a lot of maintenance as the system evolves. That is why *self-documenting code* has been a long sought ideal [Kot96]. A practical way to reach this goal is to generate reference documentation automatically from the implementation [GuB01, MRT98]. A typical example would be to use the Javadoc tool [Sun02d] to produce documentation from the source code and formal comments of a Java framework.

Neither examples nor class diagrams in themselves explain design rationale [BeJ94, Kel99]. Documenting the design rationale is important because it conveys the designers' intentions, their knowledge of the problem domain, and the limits of the framework's applicability. Unless all this information is communicated to the application developers, they cannot use the framework creatively, at the same time staying within the boundaries of its efficient and appropriate usage.

Unfortunately, most current design notations ignore the rationale that has led to the implemented architecture. One way of expressing the design rationale behind a framework would be to document the whole process of its iterative refinement. This would provide an effective way to study the design of the framework in order to learn how to use it.

Framework design processes can be documented using *design decision trees* [Ran96]. A design decision tree expresses the dependence of the design decisions. Each node in the tree corresponds to a design decision, the requirements it satisfies, and the consequences it implies. Links between the nodes are marked with the criteria used to select the appropriate path. Each design decision can thus be taken in the context of all the earlier decisions.

3.2 Framework Cookbooks and Hooks

Framework documentation should be goal-oriented. This means that when an application developer wants to use the framework to solve a problem it should be easy for her to find a solution and to apply the solution to her problem. To achieve this kind of accessibility, framework documentation can be organized as a *framework cookbook* [KrP88]. Each entry in a cookbook is a *recipe* that describes a (common) problem and gives set-by-step guidance for solving it. The recipes provide the user with an easy access to the documentation in the form of predefined navigational paths. On the other hand, they rely on narrative descriptions that may be imprecise or incomplete.

An *active cookbook* contains, not just textual recipes, but also interactive elements that can supply information on demand or help to perform certain subtasks in a semiautomatic way [PPS95]. An active cookbook is an excellent way to link examples and detailed reference manual documentation to the typical use cases or problems that the application developer may come across when specializing a framework.

Reuse documentation in the form of cookbooks is quite constraining, however. Cookbooks describe only a limited set of predefined ways of reusing a framework [CDS97]. Instructions for adapting a framework cannot be adequately expressed as a static and linear step-by-step task list, because a choice made during the specialization process may change the rest of the list completely. That is why cookbooks, although a step to the right direction, are not enough.

An advanced version of cookbooks is the *SmartBooks method* [OCS00, OrC99]. It extends traditional framework documentation with *instantiation rules* describing the necessary tasks to be executed in order to specialize the framework. Using these rules, a tool can generate an *instantiation plan* that reflects the requirements that the user has for her application. The instantiation plan is realized as a sequence of tasks that guide the application developer through the framework specialization process.

Froehlich, Hoover, Liu, and Sorenson suggest semiformal templates for describing specialization points of frameworks in the form of *hooks* [FHL97, FHL99]. A hook is essentially a description of a hot spot, i.e. a point in the framework that is meant to be adapted in some way, such as by filling in parameters or creating a subclass for an abstract class. Hooks are textual descriptions, but more precise and structured than normal documentation. Defining hooks for a framework forces designers to evaluate their design in detail and may thus expose deficiencies or overly complex structures.

Like cookbooks, also hooks emphasize the intended use of the frameworks instead of their design. In particular, hooks show the flexible parts of the framework and describe how they can be used to come up with an application with desired functionality. A hook presents a recipe as an imperative algorithm that is intended to be read, interpreted, and carried out by the application developer. Also tool support has been suggested, but as the solution description within a hook is given in procedural form it may be hard to support the non-linearity of a software engineering process with hooks.

3.3 Identifying Framework Hot Spots: Template and Hook Methods

In order to understand how the framework hot spots can be described precisely, we must identify the various ways of implementing flexibility in software systems. We base our discussion on Pree's observations on combining the basic object-oriented mechanisms (e.g. encapsulation, inheritance, polymorphism, and delegation) to achieve flexibility [Pre95].

In object-oriented systems the classification of operations into template and hook methods provides a fundamental way of expressing hierarchical organization of algorithms [GHJ95, Pre95]. Template methods are high-level function bodies calling lower-level operations (hook methods) that are defined somewhere else. Hook methods can be *abstract* (defined in subclasses) or *delegated* (defined in other objects). They can also recursively be new template methods for other hooks. These variations together with the relative amount of code placed in template and hook methods affect the flexibility and simplicity of the system.

According to the placement of template and hook methods we get *template* and *hook classes*, respectively. A hook class parameterizes its template class. There are three main aspects related to the composition of template and hook classes: *location* of the hook method (in the same class or in another class), the possible *inheritance relationship* between the classes, and the *cardinality* of the association relationship between the classes. The combinations of these aspects yield seven structures, each with distinct properties. These *metapatterns* [Pre95] are shown in figure 3.1 with the template and hook classes denoted by letters *T* and *H*, respectively.

Unification is a special case where template and hook objects are the same. This means that all modifications to the hooks require defining a new subclass for the unified class, and thus application restart is required for the modification to actually take place. On the other hand, if *T* and *H* are separated from each other, it is possible to have several subclasses of *H* that can be plugged into the *T* objects at runtime. *Separation* of template and hook classes forms thus a precondition of runtime adaptability. Table 3.2 summarizes the levels of flexibility provided by unification and separation in terms of desired end user adaptation possibilities and application restarts.

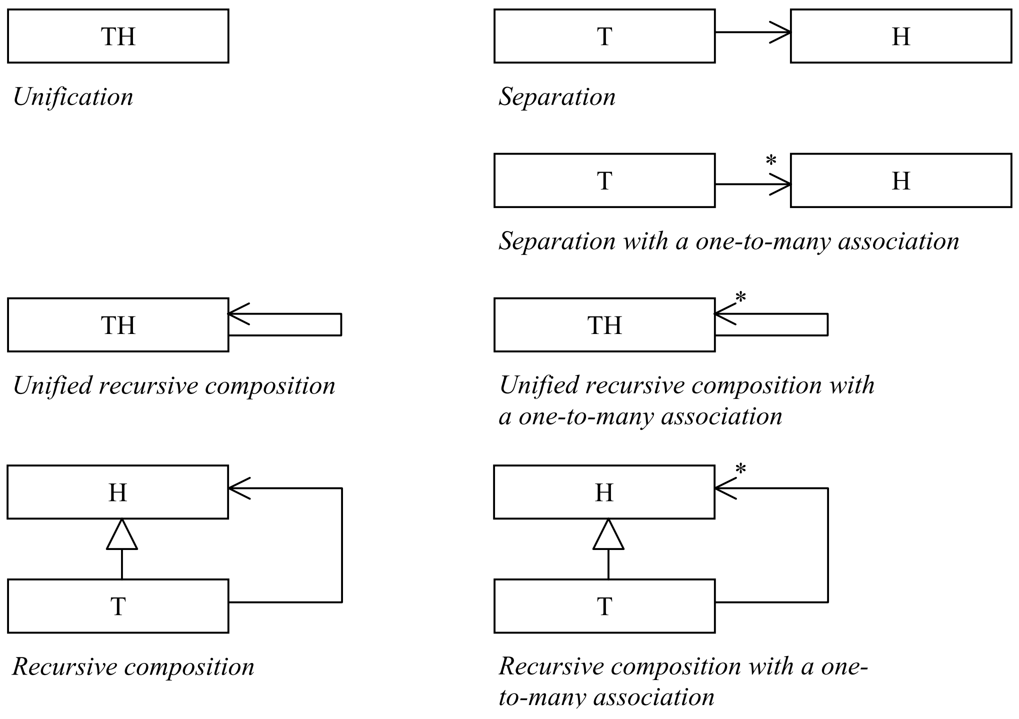

TH
Unification
T
H
Separation
T
*
H
Separation with a one-to-many association
TH
Unified recursive composition
TH
*
Unified recursive composition with a one-to-many association
H
T
Recursive composition
H
*
T
Recursive composition with a one-to-many association

Figure 3.1: Unification, separation, and recursive composition metapatterns [Pre95]

The template class can also be a descendant of the hook class. In the degenerated version, *T* and *H* are unified as the same class, but the caller and the target are different objects, which implies a need for an association from *T* to *H* (as opposed to the *Unification* metapattern). These metapatterns are called *recursive compositions* because they allow building of directed graphs of interconnected objects (either chains or trees depending on the multiplicity of the association). It is also typical for the template method to delegate the action to the hook object(s). For example, a request could be forwarded along a chain of objects until a suitable handler is found.

		Variation selectable by the end user	
		No	**Yes**
Application restart required	**Yes**	*Unification* (creation of an additional hook method)	*Unification* and a static configuration or scripting mechanism
	No	*Separation* (an additional hook method in a separate hook class)	*Separation* and a configuration dialog in the application

Table 3.2: Levels of flexibility offered by unification and separation

3.4 Software Patterns

In an object-oriented framework, the principles of unification, separation, and recursive composition determine the flexibility and variation possibilities offered to the application developers. Knowledge of those principles is therefore essential for successfully identifying the framework's hot spots.

The basic mechanisms of object-oriented programming can be combined in any number of ways in a variety of contexts to produce solutions to commonly occurring design problems. It is inefficient, however, for engineers to start from first principles every time they are trying to come up with a design that meets the given requirements. That is why mature engineering disciplines have handbooks that describe successful solutions to known problems. At the moment there are few such handbooks available for software engineers. In [GHJ95] Gamma *et al.* represent a catalog of 23 *design patterns* as an attempt to help people to reuse successful designs.

A design pattern is a recurring solution to a common design problem. When related patterns are woven together they form a *pattern language* that supports a process for the systematic resolution of software development problems [SJF96]. A pattern language is not a formal language, but rather a collection of interrelated patterns. Both patterns and pattern languages aid experienced developers in communicating their architectural knowledge and help new developers to ignore hidden traps and pitfalls. Patterns teach useful techniques and help designers to reason about their designs. Design patterns as semantically enriched variations of the basic principles of object-oriented design provide also an excellent way to document the frameworks' reuse interfaces.

3.4.1 Definition of Design Pattern

In [GHJ95] a design pattern is defined as "*a description of communicating objects and classes that are customized to solve a general design problem in a particular context*." The four essential elements of patterns are:

- the *name*, which describes the intent of the pattern with few words contributing to the common vocabulary of designers,
- the *problem and its context*, including motivation, symptoms, conditions, and applicability constraints,
- the *solution* that works as a template describing structure and participants (i.e. elements, roles, relationships, responsibilities, and collaborations) presented as text and diagrams, and
- the *consequences*, including results, trade-offs, evaluation of alternatives, costs and benefits, implementation issues, possible pitfalls, as well as impact on reusability, flexibility, and portability.

A pattern should always include concrete examples of its use in an object-oriented programming language as well as references to real-world systems that prove its usability and effectiveness. A pattern description should also make references to other patterns to explain similarities, differences, and cooperation between patterns.

The structure of a pattern is usually illustrated as a class diagram. As an example, figure 3.3 represents the *Composite* design pattern [GHJ95], which is obviously an instance of the recursive composition metapattern. Its intention is to provide a recursive tree structure whose leaves and branch nodes can be treated uniformly. This is achieved by defining a common abstract interface for all nodes (*Component*). The interface is implemented in one or more leaf classes (*Leaf*) as well as in a special class (*Composite*), which defines branch nodes. It manages references to its child nodes and delegates messages to them. The *Composite* pattern has many uses ranging from the internal representations of language processors to the hierarchical ordering of visual components in user interface toolkits and drawing applications (see figure 2.3 for an example).

It should be noted that the class diagram in figure 3.3 does not describe implementation-level elements. The elements that appear to be classes and methods in the figure are really class and method *roles*. What is not explicitly visible in the diagram is that an actual instance of the *Composite* pattern (see figure 2.3, again, for an example) probably contains classes and methods that have different names than the roles they play. Also, there can be many classes playing the *Leaf* role, for instance, and they might have features that have nothing to do with *Leaf*. In fact, the same classes could well be playing some other roles in other patterns.

All these implicit properties of the *Composite* pattern can be inferred by comparing class diagrams of multiple *Composite* instances to the class diagram of the pattern itself or by consulting the textual descriptions associated with the pattern. If we were to give a formal description of the *Composite* pattern, all these properties should be explicitly defined (see chapters 4.3 and 5.2).

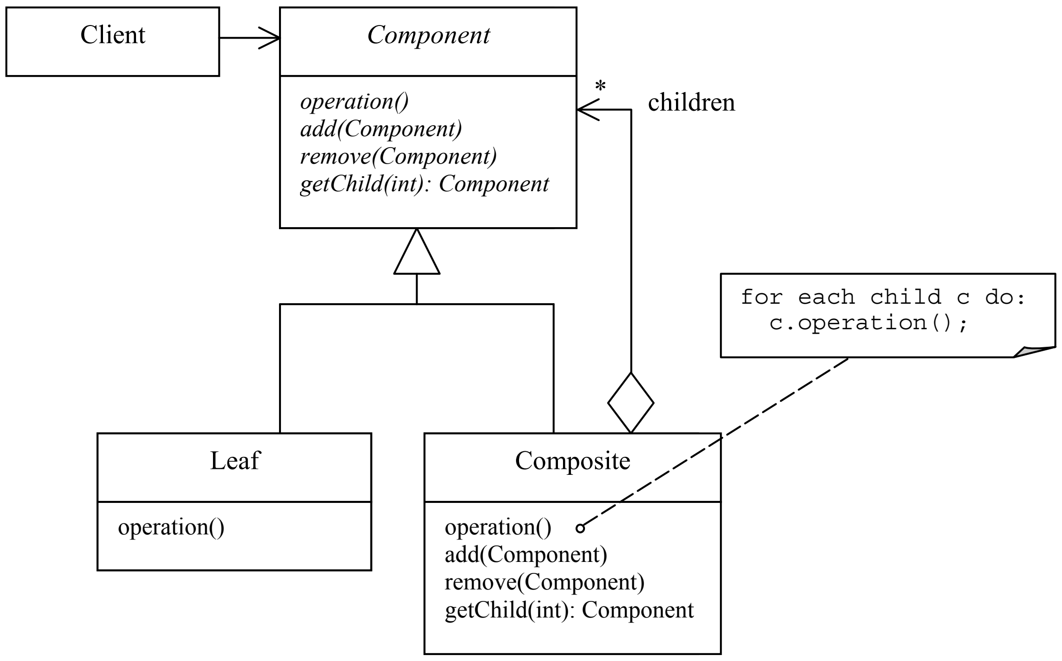


Figure 3.3: The Composite design pattern [GHJ95]

3.4.2 Various Kinds of Software Patterns

The beginning of the pattern movement in software engineering can be dated back to 1987 when Kent Beck and Ward Cunningham became interested in the architect and urban planner Christopher Alexander's work on patterns [Ale79]. He had tried to condense the knowledge of experienced architects, as well as the common sense rules and traditions of builders in various cultures, into a structured set of patterns that could guide designers to achieve the combination of elegance and practicality ("*the quality without a name*") that he thought was common in all great designs. An introduction to Alexander's work and discussion on its relevance to software development can be found in [Lea94].

Beck and Cunningham applied the concept of patterns to the development of graphical user interfaces in Smalltalk. They presented their experiences in OOPSLA‘87 and proposed the general usage of pattern languages in programming [Bec88]. The idea of archetypical design patterns caught the object community by a storm. For over ten years it has remained one of the hottest topics in the field of object-oriented software engineering. Patterns have now even a dedicated annual conference: *Pattern Languages of Program Design* (PLoP) [CoS95, HFR00, MRB97, VCK96]. The most influential of the many books published about design patterns is the one by the "*gang of four*" [GHJ95]. It describes the idea behind the design patterns and represents a very impressive set of general and reusable micro architectures in a compact catalog format.

Since the introduction of the notion of design patterns, a wide variety of different kinds of patterns have been proposed. A commonly accepted classification of patterns is based on their granularity [BMR96, RiZ95, Vil95]. *Architectural patterns* (or *architectural styles* as they are sometimes called) deal with the overall architecture of the system. They clarify the properties of such well-known application organization schemes as *client-server*, *layered architecture*, *meta-level modeling*, and *pipes and filters* [Bus96, Sha96]. Design patterns, on the other hand, deal with class or object relationships [GHJ95]. They are therefore often called *micro architectures*.

While coding, one often uses some language-dependent conventions to avoid pitfalls and to keep implementation understandable. These low-level patterns are called *idioms* or *programming patterns*. James Coplien describes C++-related idioms in [Cop92]. Diverse examples of language-dependent patterns dealing with, e.g. memory management problems of C++ programs as well as efficiency and source code control of Smalltalk programs, can be found in [VCK96].

Other kinds of patterns include *domain-oriented patterns* [Cun95, Lea96], *organizational* and *process patterns* [Cop95, FoO95, Ker95], as well as *anti-patterns* [BMM98].

3.4.3 Design Patterns and Frameworks

Design patterns are closely related to application frameworks. Frameworks and design patterns both address the need for reuse in object-oriented software development. However, there are major differences between design patterns and frameworks. First of all, design patterns are more abstract than frameworks. Frameworks are code, but only instances of patterns can be embodied in code. Design patterns are smaller architectural elements than frameworks. In any reasonable framework there are many instances of design patterns interwoven with each other. A framework has always a distinctive application domain, whereas patterns are usually generally applicable, although there are also some domain specific pattern languages.

Patterns act as a framework design tool and have demonstrated their value especially in structuring extension points of a framework. Most design patterns are based on abstract interfaces that are the key to reuse [GuB01, Deu89, RaL89, Yel96]. They emphasize the use of dynamic binding, object composition, and delegation instead of inheritance of attributes and operations. On the other hand, it can be argued that the positive impact of increased flexibility can be nullified in some cases by the confusion caused by the added layers of indirection [Men97]. That is why design patterns should be applied with care and only when truly needed.

Design patterns provide basis for concise and accurate communication between designers and help in dissimilating expert knowledge to novices [GHJ95, Kot96]. However, in the context of framework-based software engineering, perhaps the most important aspect of patterns is that they provide an effective way to document the architecture and reuse interface of a framework [HJE95, Joh92, KiB96].

3.5 Language and Tool Support for Framework Specialization

In this chapter we have discussed the problems of framework specialization. We have found out that the main obstacle in using a framework is the difficulty of identifying its hot spots, understanding the relationships between them, and following the restrictions and guidelines associated with them. Example applications have been acknowledged as a good way to familiarize application developers with a new framework. Providing them with appropriate documentation is another way of supporting them in application development. Unfortunately, it is difficult to produce and maintain a compact, yet comprehensive set of documentation and example applications. And even if such sources of information were available, they are typically not formal or general enough to precisely define the hot spots of the framework.

Then, what about defining the reuse interface directly in the implementation? There are mechanisms that allow the prevention of some simple framework misuses. In Java, for example, it is possible to declare classes *final*, which can be used to some extent to distinguish the framework's *frozen spots* from the hot spots [Sun02a]. Also, the *class invariant* as well as the *pre-* and *post-condition* mechanisms of the Eiffel language offer possibilities to restrict the behavior of subclasses specializing the framework classes [Mey92]. However, more complex restrictions or requirements, involving interactions of multiple objects for instance, cannot typically be expressed directly with the current implementation languages. That is why a new language (extension) must be provided to define and enforce those restrictions that are not directly supported by the constructs of the framework's implementation language.

While defining a new implementation language and writing a compiler for it may be considered an elegant and general solution to this problem, there are also many advantages in a solution that uses a separate annotation language and a tool that interprets the annotations. If the reuse interface of a framework is defined as a separate annotation, it is possible to make several annotations for different purposes (e.g. simple annotations for beginners and more comprehensive ones for experts). Furthermore, the tool can support multiple popular implementation languages, and it may even be possible to add it as a plug-in component to a range of widely used *integrated development environments* (IDEs).

A tool that supports framework specialization should, first of all, acknowledge framework specialization as a process that is controlled on the other hand by the restrictions posed by the framework's architecture and on the other hand by the application developer's requirements. The tool should guide this process by presenting the application developer with the options that the framework provides, and by letting the developer experiment with the options, make choices, and cancel them if they prove to be unsuitable. At the same time the tool should make sure that the application remains a valid specialization of the framework. The tool may even act as a kind of automatic tutor when it interprets the annotation produced by the framework expert. In this way it is possible to lower the training costs and leverage the knowledge of the experts in the whole organization.

A framework specialization tool should combine the concreteness of examples and the intuitiveness of cookbooks with the generality of formal constraint descriptions. In this way the tool could actively guide its user (something which might be impossible if mere declarative constraint descriptions were used). It should be easy to locate example code fragments based on the problem at hand and to generate appropriately adapted instantiations of them. Simultaneously, the tool should support validation of applications that use and combine the framework's concepts in more complex ways than what is explicitly present in the examples and recipes.

3.5.1 Pattern-Based Formalizations and Tools

To guarantee generality, it is advisable to have a solid formal base for a framework specialization tool. This can be achieved by giving a precise definition for the framework annotation language. Because well designed frameworks use role-oriented interfaces in their implementation (recall chapter 2.1), and patterns that are widely used to document framework designs are based on role modeling, it is natural to describe also the reuse interface of a framework in terms of roles and constraints associated with the roles. On top of the constraint language and its complementary constraint validation mechanisms it is then possible to build also specialized, perhaps even framework-specific tools.

The specification of an architectural unit of a software system as a pattern with roles bound to actual program elements is not a new idea. One of the earliest works in this direction is Holland's thesis [Hol93] where he proposed the notion of a *contract* to describe run-time collaboration of objects. Other similar formalisms and methods that preceded patterns include *clichés* [RiW88], *template-based design* [HaY85], *plans* [HaN90], *formal contracts* [HHG90], and *motifs* [LaK94].

After the introduction of design patterns, various formalizations have been given to design patterns (see, for example, [FMW97], [MDE97], [Mik98], and [Rie00]), often in the context of specifying the hot spots of frameworks. In [EHL99] Eden, Hirshfeld, and Lundqvist present LePUS, a symbolic logic language for the specification of recurring *motifs* (structural solution aspects of patterns) in object-oriented architectures. They have implemented a PROLOG-based prototype tool and show how the tool can utilize LePUS formulas to locate pattern instances, to verify source code structures' compliance with patterns, and even to apply patterns to generate new code. In [ACL96] Alencar, Cowan, and Lucena propose another logic-based formalization of patterns to describe *abstract data views* (a generalization of the MVC concept).

Kim and Benner take a different approach in their POE (Pattern-Oriented Environment) tool [KiB96]. Their primary goal is to ensure that the pattern instances documenting a system (such as a framework) are kept consistent with the implementation. The tool manages the creation, deletion, and verification of pattern instances and role mappings. In POE there are three kinds of components: classes, relations, and operations. They have attributes like name, parent link, optionality, links to other components (properties of a class, end points of a relation, parameters of an operation) with cardinalities, and bindings to the user's implementation classes. The tool implements validation algorithms to ensure that different pattern instances and role bindings are used properly.

Fontoura, Pree, and Rumpe present a UML extension UML-F to explicitly describe various kinds of framework hot spots [FPR00]. They use a UML *tagged value* (a name-value-pair that can be attached to a modeling element to extend its properties) to identify and document the hot spots. Each hot spot type has a dedicated tag. In addition, there are tags for expressing whether or not the variable information is available at compile time as well as for identifying application-specific classes as opposed to classes belonging to the framework. UML-F descriptions can be thought of as a structured cookbook, which could be executed with a wizard-like framework specialization tool.

3.6 Discussion

By definition, frameworks are flexible systems that can be configured and adapted to work as a basic infrastructure for various kinds of applications. Unfortunately, flexibility usually implies also complexity and abstractness. This makes framework usage hard. It can be difficult to know what can, and especially what cannot be done with a framework, so even choosing a right framework for a job may prove to be complicated. From a framework user's point of view, it is essential that the hot spots of the framework are clearly defined in the documentation. The user must be able to figure out how and where she is to put her application-specific code, and what kind of restrictions and dependencies are associated with the hot spots.

It has been argued that it is unrealistic to assume that the framework developer can anticipate future uses of the framework adequately to provide all required documentation [CDS97, SLB00]. In practice framework specialization is almost always more complex than just specializing a fixed number of predefined hot spots. At least the documentation should not assume that the hot spots are going to be used in isolation from each other. Also, in practice modifications to the structure of the framework itself are common and should be somehow supported.

We believe that there is no accidental reusability in frameworks, and that it is indeed possible to define the reuse interface of a framework precisely. Furthermore, there is a possibility to provide tool support both for documenting the framework and for specializing applications from it (as has been demonstrated in [Vil01]).

It is already widely accepted that tool support is practical and useful for aiding the specialization of some particular kinds of frameworks. *Visual builders* for constructing user interfaces have been standard parts of commercial IDEs for quite some time, and there are similar tools for some other mature black-box frameworks, too [DuJ96]. There have even been attempts to generalize the idea of visual builders to allow the composition of applications from arbitrary components as long as they conform to a standardized component interface or protocol (see, for example, [Sun02c]). In our view, these experiences together with our own experiments show that tool support can become a significant factor in framework engineering; not perhaps so much in designing and constructing them, but in documenting them and guiding their usage.

We see framework specialization as a challenging task for which tool support is of vital importance. According to our vision, frameworks should be accompanied with framework-specific programming environments that both guide and control application programmers in creating applications according to the conventions of the framework.

In practice, a framework usage environment should offer at least context-sensitive documentation that dynamically adjusts to the choices the developer makes, as well as code generation to automate the production of skeletal implementations and trivial details, so that the developer can concentrate on those parts of the application that are genuinely application-specific. It is also important to maintain an explicit connection between the constraint annotations and source code, in order to be able to validate code against constraints even if it has not been generated by the tool or it has been modified later on.

We believe that it is possible to describe the intended rules governing the framework's specializations with a precise pattern-based formalism, and further, that it is possible to extract a significant portion of those rules from the source code of the framework itself and from the examples and other available applications using the framework. That is why we will next discuss reverse engineering and framework design recovery, in particular, in chapter 4. An account of our model for framework annotation and a description of the implementation of that model will follow in chapters 5 and 6, respectively.

4. Framework Design Recovery

The problem of framework understanding and usage resembles the general problem of maintaining a software system. That is why *reverse engineering* and *program comprehension* form a natural background for this thesis.

To document a framework, we need to know the framework's architecture and especially its reuse interface, and we must be able to express this knowledge in a form that is easily accessible to application developers. In an ideal case we would have thoroughly and systematically maintained records of the requirements, design, implementation, and modifications of the framework. Then we could follow the whole lifecycle of the framework and collect all relevant information to a concise reference manual. Unfortunately, in practice there seldom is comprehensive and up-to-date documentation available. This is partly due to *architectural erosion* (see chapter 4.1.1).

Architectural erosion eventually leads to a situation where the initial architecture has been incrementally modified to the point where its key properties no longer hold and the architecture as-implemented conflicts the architecture as-documented. To diagnose and repair such a situation we must recover lost design information from the system's source code.

Various reverse engineering methods and techniques for design recovery are presented in chapter 4.1. In chapter 4.2 we discuss the general properties of program comprehension tools emphasizing source code analysis for capturing instances of design structures from implementation. These ideas are then applied to the pattern and hot spot recovery in chapter 4.3.

4.1 Reverse Engineering

In general, reverse engineering means analyzing finished products to improve on them [ChC96]. Within software production, reverse engineering has mainly concentrated on analysis and comprehension of legacy systems. In [Arn92] reverse engineering is defined as "*the process of deriving abstract formal specifications from the source code of a legacy system, where these specifications can be used to forward engineer a new implementation of that system*." This definition emphasizes reverse engineering as a part of a *reengineering* process [Duc99].

In reengineering, an existing legacy system is changed to meet new requirements (see figure 4.1). This involves design recovery of the existing system (1) and analysis of the new requirements (2). Those parts of the legacy system that hinder its cost effective adaptation to meet the new requirements are detected and analyzed (3) to enable reorganization (4). When the system has been updated (e.g. ported to a new environment, a new language, or a new paradigm), the changes may need to be propagated to other systems (5) in order to ensure that it can be reintegrated to the surrounding environment.

There are also other views to reverse engineering. For example, for those who are simply maintaining a software system, reverse engineering tries to find out where and how each function of the system is implemented. At the same time reverse engineering tries to document how each code fragment from the system's implementation relates to the overall architecture of the system and what is the rationale behind it.

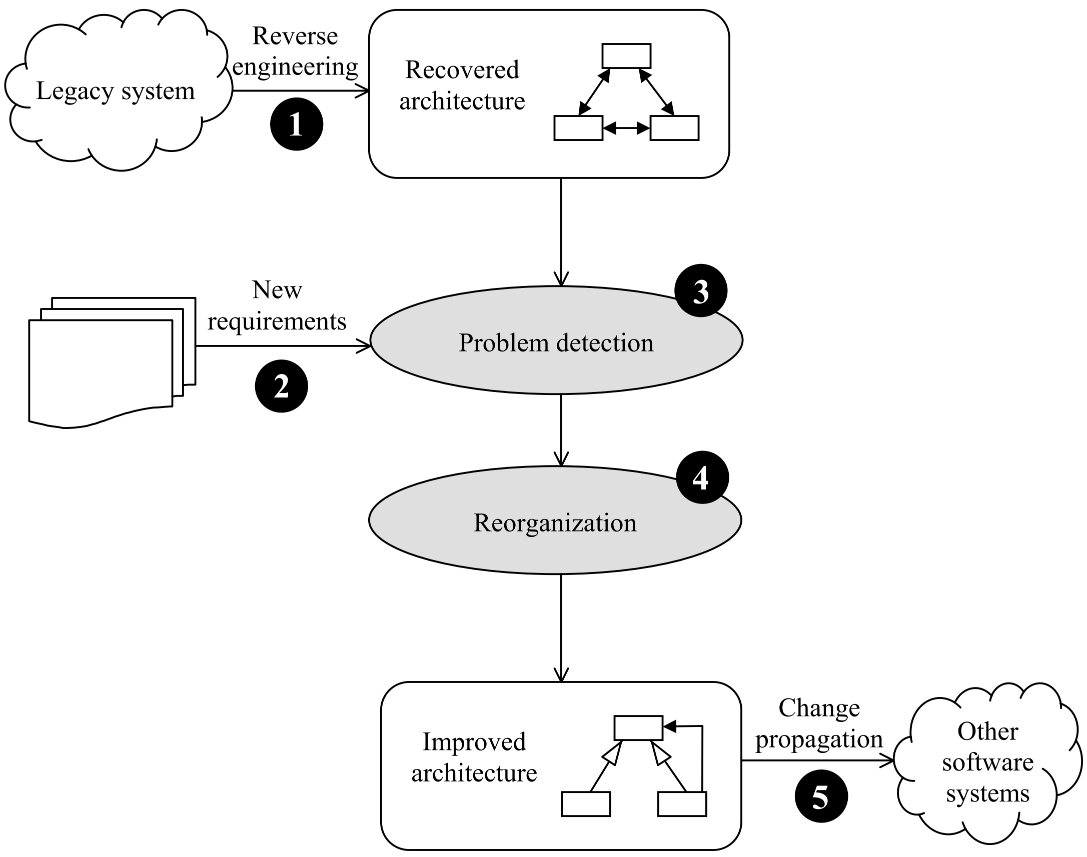


Figure 4.1: Reengineering process

In the context of framework documentation, reverse engineering serves yet another purpose. Instead of aiming at modifications or new implementations of an existing system, we aim at new specializations of an existing framework. For a framework user, reverse engineering may provide valuable insight into the framework's specialization and call interfaces. In this perspective the more general definition of reverse engineering given by Chikofsky and Cross is better. They define reverse engineering as "*the process of analyzing a subject system to (a) identify the system's components and their interrelationships and (b) create representations of the system in another form at a higher level of abstraction*" [ChC96].

4.1.1 Resisting Architectural Erosion

Adding new functions and finding bugs is very efficient when you work on a system with a solid design that is well understood. However, when a complex software system evolves, its implementation tends to diverge from the intended and documented design models [DiM01, Fow99, SSC96b]. This architectural erosion, which happens when people modify the implementation without fully understanding the design objectives behind it, is the main motivation for reverse engineering.

Once a system begins to loose its structure, the code becomes harder to understand and so the chances of cluttering the design further increase. If the developers ignore the architectural constraints of the system they will find themselves in a situation where they have trouble inserting new functions because of unwanted and unexpected side effects. Similarly, modifications to existing features will become harder because of instances of duplicated code. Gradually the system may end up being impossible to maintain or reuse. In general this phenomenon is called *increased software entropy* [JCJ92] or *code decay* [EGK01].

Frequent *compliance checking* helps catching design turnovers before they become irreversible. For instance, Sefika *et al.* have implemented a tool, called Pattern-Lint, for confirming that implementation corresponds to the expected design models, such as a set of design patterns [SSC96b]. Pattern-Lint incorporates static analysis and dynamic visualization. The tool represents program information with hyperlinked diagrams, such as animated method calls, static weighted inter-class call and data sharing graphs, as well as dynamic affinity graphs.

Besides trying to detect and repair consequences of architectural erosion, it is also possible to prevent it by using *refactoring* — a technique to maintain the architectural integrity of implementation code [Bec00, Fow99, Opd92]. The notion of refactoring first emerged in the field of framework development during the 1990s. The basic idea of refactoring is to clean up code in a controlled manner in order to minimize the chances of introducing bugs and to maximize understandability. More precisely, a refactoring can be defined as a "*change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*" [Fow99].

The power of refactoring as a method lies in systematically organized documents, which describe proven techniques for enhancing software safely. They illustrate possible pitfalls and suggest ways to avoid them. Fowler *et al.* have introduced a refactoring catalog where they use a standard format to represent over 70 frequently needed refactorings [Fow99]. Most of them are quite simple step-by-step descriptions of how to transform traditional procedural designs into more object-oriented ones (e.g. *Replace Conditional with Polymorphism* and *Replace Inheritance with Delegation*). The mechanical nature of refactorings means that they are quite straightforward to automate. A good example of a refactoring tool is Smalltalk Refactory [RBJ97].

Refactorings are closely related to design patterns. A design pattern tells you how to solve a recurring design problem in a disciplined manner in a given context. A refactoring, on the other hand, guides you to enhance your existing implementation so that it reflects a better design. Often this "better design" conforms to a design pattern. So, in many cases, you end up applying a set of refactorings to turn a piece of *ad hoc* code into an instance of a design pattern.

Whereas Fowler *et al.* give a catalog of refactorings with detailed how-to-do lists and discussions of suitability to various situations, Cinnéide and Nixon state the desired outcome of the factoring as a design pattern [CiN99]. They define the transformation from the current design to the design goal in terms of small *minitransformations* (partial transformations) corresponding to the *minipatterns* (partial goals) into which the original code can be decomposed.

For each minipattern there is a minitransformation, which includes quite a detailed and technical description consisting of name, arguments, preconditions, algorithm, behavior preservation argument (proof), and post conditions. Based on that information the method can automate the application of the design pattern transformation.

As another approach to fight architectural erosion Ding and Medvidovic propose a method called Focus [DiM01]. Its goal is to support controlled system evolution by recovering the system's architecture and using it as the basis of evolution. Focus emphasizes design recovery over continuous refactoring and architectural compliance checking.

The Focus approach is driven by evolution requirements and is applied iteratively. Each iteration involves an *architecture recovery* and a *system evolution* phase. The architecture recovery of a system begins with identification of its components. First, one generates class diagrams from the source code of those parts of the system that would be affected by a proposed change. A standard CASE tool can be used for this purpose. The generated diagrams do not have to include methods or attributes, but there must be association, generalization, and aggregation relationships present. Next, related classes are grouped together by examining five of their properties: *isolation*, *generalization* (inheritance), *aggregation*, *composition*, and *two-way association* (which implies tight coupling). Finally, the clusters of related classes are packaged together into abstract components. These components can be further grouped to form even larger components.

After the existing components of the system have been identified, they are mapped against the conceptual architecture that has been proposed for the system. Then the component interactions are analyzed by producing UML sequence diagrams for the most important use cases. All inconsistencies with the conceptual architecture and other observations can subsequently be used to drive the system evolution.

4.1.2 Expressing Results of Reverse Engineering: Metrics, Hypertext, and Visualization

In reverse engineering, tool support is essential because of the size and complexity of the systems that are being analyzed [Duc99]. The usefulness of reverse engineering tools is obvious for legacy system maintenance, because they can be used to analyze large amounts of code in short time. With metrics tools, for instance, it is possible to gather abstract and meaningful information about implementation without having to manually analyze all the source code. They make it feasible, for instance, to locate violations of the principles of sound object-oriented design (e.g. the *one-class-one-concept* principle) in large systems.

Object-oriented software metrics usually estimate either *complexity* or *cohesion* within a class, *coupling* between classes, or *characteristics of inheritance hierarchies* [Bau99]. Simple, but useful complexity and cohesion metrics for a class include counting the number of methods in the class (*number of methods*, NOM), adding together the *McCabe cyclomatic complexities* of all methods in the class (*weighted method count*, WMC), and counting the relative number of methods that use at least one common field of the class (*tight class cohesion*, TCC).

Counting the number of fields of the class that refer to another class or abstract data type (*data abstract coupling*, DAC) and calculating how many local methods and methods defined in other classes can be directly invoked by a method defined in the class (*response set for a class*, RFC) are important measures of coupling between classes. Popular inheritance tree metrics, in turn, include the number of direct subclasses of a class (*number of children*, NOC), the number of direct and indirect subclasses of a class (*number of descendants*, NOD), and measuring the longest path in the inheritance tree from the root class to a given class (*depth in inheritance tree*, DIT).

Re-documentation of a complex system, such as a framework, calls for sophisticated tool support that provides different perspectives to the system. One way of communicating the information produced by reverse engineering tools is to use *hypertext*. For example, program dependencies can be represented as hypertextual links like in the HyperSoft model [PSK96]. Also scenarios and other diagrams can be linked to associated reference documentation and source code like in Scene [KoM96].

Visualization is another important aid in understanding framework design. Keller *et al.* use special annotations in class diagrams to visualize instances of *design components* (patterns) [Kel99]. Their method is based on the idea that each abstract design component has a *reference class* that is considered to be the most characteristic for the component. The user selects this main class of the design component.

Upon design recovery incrementally bounding boxes are drawn around those classes in implementations of design components that correspond to the reference classes of *abstract design components*. In that way, those classes that participate as the main class in many instances of design components will have thick bounding boxes and will be clearly visible in diagrams. To get more information, the user can zoom into the particularly interesting classes and examine their details. It is also possible to filter-out certain properties of classes to make diagrams more readable.

The static structure of an object-oriented system, such as a framework, is relatively simple to describe with, e.g., annotated class diagrams. Its *dynamic properties*, on the other hand, are considerably harder to capture. It is useful to join the representation of dynamic aspects (objects and their interactions) with the static aspects (classes and associations) of the system. Dynamic aspects give feeling about the important features of the system, and static information can be used to restrict the output to those features.

To understand the control flow in a framework, it is helpful to have a dynamic view together with trace samples of the execution of the system. *Scenario diagrams* (or message sequence charts as they are sometimes called) are an acknowledged way to describe the message flow in object-oriented systems. Several tools for program execution tracing and for producing animated scenario diagrams have been proposed (see, e.g., [KoM96], [SSC96a], and [LaN95]). The common theme in all of them is the visualization of the dynamic behavior of objects (e.g. message passing) to gain insight of a complex system. They also offer statistics of object lifetimes, active times, processing times, and number of calls. Some of the tools are pattern-oriented: they enable filtering of the behavior to highlight the pattern-specific interaction sequences for finding recurring structures and behavior.

Usually execution tracing tools are used for identifying bottlenecks or problems with coupling and cohesion, for locating incorrect or unnecessary code fragments (e.g. multiple unnecessary calls to some operations, wrong calling orders, use of uninitialized variables, memory leaks), and for comparing designs to the actual implementations. Such tools are also very helpful aids in framework understanding and usage.

4.2 Properties of Reverse Engineering Systems

The goal of reverse engineering is to produce documentation that describes the architecture of a system on a high level of abstraction, communicates the intent of its design, and provides a detailed, yet accessible reference manual that can be used for maintaining or reusing the system. Of course, it is not possible for a reverse engineering tool to comprehend the intent of the implementation. Instead, it gathers, filters, and organizes facts that help a human analyst to form a mental model of the system, which in turn is an aid to gain understanding of the system.

Most current reverse engineering techniques lack effective abstraction mechanisms. They produce diagrams and charts that are cluttered with details. These would typically include *set-use charts* for variables and module or class diagrams expressing the static structure of the system. This kind of information does not directly explain the central architectural properties of the system.

There are some reverse engineering methods that allow abstraction. Software metrics provide a way to condense the details of the implementation to statistics concerning properties like complexity, cohesion, and coupling. Visualization can be used to further abstract these statistics, to illustrate the static structure of systems, or to gain understanding of their dynamic properties. They do not give any direct support for the accurate identification and precise expression of framework hot spots, however.

Harris *et al.* propose integration of reverse engineering and architectural styles for recovering and expressing the architecture of large complex systems [HRY95]. They have constructed a system that automatically recovers instances of architectural styles to verify that specific architectural commitments are actually present in the implementation. On the other hand, starting from the implementation, it is also possible to find out where a particular code fragment falls in the overall architecture.

Extraction of framework documentation from implementation is analogous to architectural design recovery. We need to combine basic reverse engineering techniques, like source code analysis, with design knowledge representations, such as design patterns, to implement automatic recovery of design model instances in the implementation.

4.2.1 Conceptual Architecture and Implemented Architecture

Any tool or method that tries to accomplish design recovery must make a distinction between the conceptual and the implemented architecture of a system [DiM01, HRY95, KNS92, Kru95]. The conceptual architecture is sometimes also called *logical* or *idealized architecture*. It is represented in documentation, for example, with high-level block diagrams. Its purpose is to communicate the intended general structure of the system together with its underlying design principles. The implemented architecture, on the other hand, is defined by the instances of architectural styles or design patterns that actually exist in the system's implementation.

The implemented architecture is always different from the conceptual architecture because it consists of concrete source code elements whereas the conceptual architecture consists of abstract design artifacts. Furthermore, there are usually also some deviations from the commitments of the conceptual architecture in the implemented architecture [BHB99]. They might have been done on purpose, e.g., to optimize performance or to compensate the inability of the environment to support the conceptual architecture. Most often, though, they are signs of architectural erosion resulting from the developers' lack of understanding of the architectural commitments of the system.

Most reverse engineering methods recover their information directly from implementation [HRY95, Kel99, WTM95]. In this way it is possible to get a richer and more up-to-date picture of the system than what is available in documentation. On the other hand, concentrating only on the architecture-as-implemented aspect ignoring the conceptual architecture may provide too fragmented a view of the system [DiM01]. Without an explicit analysis and comparison of the conceptual architecture and the implemented architecture it is not possible to estimate how well the system's implementation conforms to its original architectural requirements.

4.2.2 Source Code Analysis

Recovery of design information from implementation requires *source code analysis*. It can be divided into *lexical analysis*, *parsing*, and *semantic analysis* [ASU86]. Source code analysis corresponds to the functionality of a compiler's or an interpreter's front end. Its purpose is to transform the source code into an internal representation. The internal representation is usually a set of sentences in an *intermediate language*, or an *abstract syntax tree* (AST). An AST-based implementation is generally less efficient, but more flexible. Since many reverse engineering tools use an AST as an internal representation, we will concentrate here on those aspects of source code analysis that are relevant for the AST-based approach.

ASTs support program understanding activities in many ways. They are needed in the analysis of *data dependencies* [ASU86] and *control dependencies* [FOW87], as well as in the generation of *program slices* [Wei84]. Also *dead code* and *function clone detection* is based on ASTs and *call graphs* [MNL96] built from them [TAF00].

In an AST the program structures are represented as a tree or as a directed acyclic graph that explicitly represents the source program's structure with subtrees that correspond to the commands, expressions, and declarations and with leaf nodes corresponding to the identifiers and literals. An AST includes the information from the parse tree in a condensed form: only those nodes that have semantic significance are kept in the tree. On the other hand, the AST also contains the language's predefined entities and structures, which might not have a counterpart in the source code. For a detailed explanation of defining and implementing ASTs in Java refer to [WaB00].

An AST produced from a system may be *partial* or *complete*. A complete AST means that all information available in the source code is presented in its entirety, including all branches of all statements and expressions. A partial AST, in turn, may contain just class and method signatures, for instance. If very large systems are to be analyzed, it may be practical to first conduct partial analysis for the whole system, and then to examine closer only those parts of the system that need special attention [TAF00].

The graph representation of a complex system's source code (e.g. an AST) may become so complex that it is difficult to see the big picture through the myriad of details. To simplify ASTs and to make analyses more efficient, many kinds of optimizations can be used. For example, a set of call relationships between methods of class *A* and another set of methods of class *B* can be simply represented as a call from class *A* to class *B* [SeG98]. The same also applies to object instantiations.

Ciupke has surveyed many possibilities to arrange entities into larger groups and *collapse* such groups (i.e. replace entities in the group by one higher order entity) in order to raise the abstraction level of graph representations [Ciu99]. These *groupings* include collapsing classes to packages and packages to higher-level packages in order to form a package hierarchy. Also, a class hierarchy can be collapsed to its root class, and dynamic method calls can be grouped either with respect to calling and called objects or with respect to given time intervals. As a specific example of grouping, Dósa and Koskimies have designed and implemented a set of algorithms for compressing (parts of) UML class diagrams [DóK99]. Their tool shows complex, automatically extracted diagrams on a higher level of abstraction thus enabling easier navigation and more efficient analyses.

Sometimes more detailed information is needed. For example, detailed information about the receiver (not just its type) of a method can be stored (e.g. whether it is the same object as the caller, or its ancestor, a field, a local variable, a parameter, a method result, or perhaps a call was to a static class method). This information enables more exact recovery of association semantics. In such a situation, an *association* between classes *A* and *B* can be defined to exist only if *A* calls *B* and references *B* through a field [SeG98]. Similarly, *A* can be defined to be in an *aggregation relationship* with *B* only if there is an association between *A* and *B*, and *A* also creates *B*. It should be noted, however, that in some other contexts more relaxed definitions might be more useful.

4.2.3 Template Libraries and Matching Algorithms

Many program understanding methods try to extract design information at various levels of abstraction by matching predefined structural models, such as patterns [Bro96, FGM01], architectural styles [HRY95], abstract design components [Kel99], or plans [Qui94] to the available source code. The templates of structural models are usually stored in a generally applicable or domain-specific library [Qui95].

Regardless of the exact nature of the entities to be found, a generic matching algorithm is required to search the candidate entities and to select the appropriate ones as results. A common way to do this is to formulate an expression or a query that can be evaluated for each candidate. Those entities for which the expression yields a positive value or which satisfy the invariants expressed in the query are considered as matches.

For instance, in [HRY95] a library of formalized descriptions of architectural styles is used to guide the design recovery process. In this context, an architectural style is a description of an expected arrangement of *entities* (*components*) and *relations* (*connectors*) found in software. A component would be a set of procedures (a fairly large code fragment) that act together, for instance, as a layer or an abstract data type. Connectors, in turn, are very specific code fragments or single procedure calls (e.g. system calls) that express how components are linked together. An example of a connector would be a system call to spawn a new (sub)process.

The architectural styles in the library include, *pipe and filter*, *interacting objects*, *abstract data type*, *implicit invocation*, and *layering*. Each style is associated with an appropriate set of queries expressed in a code level vocabulary. More precisely, the queries are written in a language that enables to write routines that access and analyze the AST parsed from the target system's source code. The queries return AST nodes satisfying the invariants expressed in the query.

Entities are defined hierarchically to enable the analyst to expand queries to more general entities or relations. It is also possible to refine queries to more specific ones if needed. The queries can be augmented with simple free-form text searches to enable the extraction of useful information from identifier names and comments.

Once an entity has been recognized as an instance of a template (e.g. a design pattern or an architectural style), the mapping from the template to its instance in the source actually describes a part of the implemented architecture of the system. When all these partial views to the system are put together we get the overall architecture of the system that can be compared to the intended conceptual architecture found in the documentation.

Comparison of the implemented architecture and the intended conceptual architecture can be automated to some extent. An example of a tool that assists architectural compliance checking is Pattern-Lint [SSC96b] (see also chapter 4.1.1). It incorporates static analysis and dynamic visualization to express design properties at a variety of levels of abstraction, to check them, and to express system information selectively and in a problem-specific manner. Even though the tool is able to gather positive evidence of conformance as well as identify violations of design rules, the method requires a human analyst to interpret the results and to decide the possible corrective actions. At the lowest levels of abstraction (e.g. idiom checks) the automatic analysis is hindered by the amount of detail that obscures the design intentions. On the higher levels of abstraction (e.g. design pattern detection) the problem is that there are almost always multiple implementation variations for each abstract design model.

4.2.4 Completeness, Scalability, and Other Restrictions

In practice, the library-based approach to design recovery cannot guarantee complete understanding of the target system, because every non-trivial system always contains some code that is idiosyncratic [Qui95, ToA99]. Thus, the success of any such system depends on how much of the code that is being examined is stereotypical and can be found by matching against predefined patterns.

It is impossible for the library to contain all possible patterns. So, even if the target system did contain many instances of general patterns, it is still up to the analyst to select or define an appropriate library to match against the system under consideration [SeG98]. As a solution to this *library selection problem*, Harris *et al.* propose browsing documentation and source code to get initial knowledge of the system [HRY95].

A more practical problem is that the *scalability* of most traditional program understanding algorithms is weak. In general, there are two dimensions of scaling in pattern-matching algorithms: (1) the size of the program that is being analyzed and (2) the size of the library [Qui95]. In order to make matching feasible, some optimizations and heuristics must be used. To overcome the problems related to the library size, the search library can be augmented with explicit *search control information* or *indexing*. Another optimization possibility is to try to first efficiently determine which kinds of high-level program elements are likely to be present and then concentrate on verifying the presence of only those relevant patterns. Reducing search space by first using automatic modularization and only after that program understanding algorithms to extract information about connections between components, on the other hand, can help in compensating the growth in system size.

One problem with the query-based design recovery approach is that connections between the found design components are usually not adequately described, which means that they may seem isolated from each other. It is also very difficult to match queries expressed in terms of concepts familiar to the application developer to the descriptions of framework entities because there usually is a gap between architectural level and source code level concepts [GaM95]. Harris *et al.* argue that their approach of defining concepts in hierarchical fashion (components, connectors, and source entities each in their own hierarchy) and cross-referencing related concepts between hierarchies bridges this gap [HRY95].

Traditional reverse engineering usually yields only static models of systems. The use of *reflection* in object-oriented languages and other advanced highly dynamic programming techniques makes program source code analysis impossible or at least imprecise with compiler techniques. Similar problems to design recovery are caused by the usage of generic system calls, which may result in component dependencies that are almost impossible to recognize automatically [HRY95]. Detection of these kinds of connections is also difficult to achieve in a language and environment independent way. Dynamic analyses can be used to circumvent these problems. On the other hand, a dynamic analysis deals with one instance or execution of the system at a time, so it cannot directly reveal universal facts about the system's design.

Very dynamic component-based systems, whose architecture may radically change at run-time, pose even harder challenges to architectural modeling. Perrochon and Mann propose *inferred designs* as a means to produce architectural documentation for such systems [PeM99]. Their method is based on *instrumentation* and monitoring of running systems. Perrochon and Mann perform *event mining* from the produced events to infer the current architectural status of the system. The method can use the produced information to check predefined architectural constraints or even to dynamically change the system configuration if needed.

4.2.5 Human Assistance in Design Recovery

It seems impossible to automatically extract complete architectural specifications from source code. Instead, a combination of automatic and user-assisted program understanding processes can result in at least partial specifications. Most authors regard *semi-automatic design recovery* (a combination of automatic and manual design recovery) as the most viable way to acquire program understanding

[Bro96, Kel99, Mar95]. A revised definition of successful reverse engineering could thus be stated as [Qui95]: "*the automated or assisted process of deriving a knowledge base describing a legacy system from its source code, where this knowledge base lessens the effort required to forward engineer a new implementation of that system*."

As an example, Quilici presents a program understanding tool that has a single knowledge base for both automatically extracted and user given design information [Qui95]. This knowledge base is represented formally in order to support forward engineering, but it also has a visual graph representation that can be manipulated by the user to allow her to supplement and adjust it manually.

4.3 Automatic Design Pattern and Hot Spot Recovery

Pattern-based approach to reverse engineering is natural, since it allows one to express those patterns of thought that comprise the rationale behind a system [Kel99, Sne98]. In order to understand a system it is necessary to recover those patterns, either automatically or manually.

All design patterns are not formal enough to allow direct tool support. Instead, the pattern's implementation (e.g. class structure) might be detectable and lead to the identification of the actual pattern. In general, "*a pattern is detectable if its template solution is both distinctive and unambiguous*" [Bro96]. In other words, the structure of the pattern should never be used in other contexts and it should not have any alternative structures. If a design pattern relies heavily on informal semantic descriptions instead of structure it may be impossible to detect it automatically (e.g. the *Interpreter* pattern [GHJ95]). On the other end of spectrum is the *Template Method* pattern whose definition is not based on any semantic meaning. Most design patterns reside somewhere in between.

Besides informal textual descriptions, patterns incorporate also various kinds of diagrams and example implementations describing the structure of the solution. Class diagrams express the main participants (i.e. roles) of the pattern together with their interfaces (method signatures) and relationships (inheritance, aggregation, association). Direction and cardinality of aggregations and use relations are usually obvious, but other associations, such as parameter passing or object creation have varying representations, and the different semantics of inheritance (type compatibility versus implementation inheritance) might not be clearly identified. Call graphs and the actual ordering of calls, in turn, can be deduced from scenario diagrams.

In addition to precise definitions of the structural aspects of the patterns to be detected, automatic pattern recovery requires source code analysis and an algorithm that matches pattern descriptions against the analyzed code (recall chapters 4.2.2 and 4.2.3). Usually both static and dynamic analyses are needed. Source code analysis can be easy or hard, depending on the implementation language. For instance, in Smalltalk the class hierarchy is obtained easily from the *metaclasses*, but the use relations' types are hard to discover, because there is no static typing. Keller *et al.* suggest a list of source code information that is needed for pattern-based recovery of design elements from C++ source code [Kel99]. We have adapted a similar list for systems implemented in Java (see table 4.2).

Program element	**Stored information**
package	*name*
file	*name*, *directory*
type (interface, class, inner class, anonymous class, or primitive type)	*name*, *file*, *owner* (another type, if any), *package*, *modifiers*
generalization relationship	*supertype*, *subtype* (includes information on classes that implement interfaces or extend other classes, as well as interfaces that extend other interfaces)
field or *variable*	*name*, *type*, *owner* (a type or an operation), *modifiers*
operation (method or constructor)	*name*, *owner*, *modifiers*, *kind*, *return type* (if any)
parameter	*name*, *type*, *modifiers*
operation call	*operation*, *sender*, *receiver* (sender and receiver are the static types of the corresponding objects)
object instantiation	*operation*, *sender*, *receiver*
field, *parameter*, or *variable reference*	*referrer*, *target* (referrer is an operation, target is either field, parameter, or variable)

Table 4.2: Source code information required for design pattern recovery in Java systems

4.3.1 Pattern Detection with a Template Library

In pattern recovery systems patterns are typically stored as templates in a template library. A pattern template contains information needed to find its instances in the source code (recall table 4.2). In [Bro96] Brown presents an automatic reverse engineering tool for detecting design patterns in commercial-size Smalltalk programs. It uses structural aspects of patterns to detect instances of them in source code. The following short example illustrates his approach.

An instance of the *Composite* design pattern (recall figure 3.3 in chapter 3.4.1) can involve an abstract interface that has two kinds of implementations: leaves and compound nodes. Compound objects have a one-to-many reference to objects implementing the abstract interface. The essence of the *Composite* pattern is its ability to generate recursive object structures, such as trees.

The *Composite* pattern can be detected by looking for cycles in a combined inheritance and association graph as illustrated in a simplified class diagram of an imaginary GUI toolkit represented in figure 4.3. When a cycle with a one-to-many association is found, it can be strongly suspected that a *Composite* has been used. See [SeG98] for an example of how this same idea can be expressed more formally.

Pattern recovery can be used also to calculate design metrics of the system under analysis. For instance, the number of instances of a pattern indicates the *frequency of reuse* of the identified class structure [ToA99]. The number of relations in a detected pattern, on the other hand, determines the *complexity* of the identified structure.

In the architectural analysis and design metrics toolset Maisa [FGM01, GPN02, PKG00], design pattern recovery is defined as a *constraint satisfaction problem* [Mac92]. Such a problem is given as a set of *variables* and a set of *constraints* restricting the values that can be assigned to those variables. In the context of design pattern recovery, it is natural that the variables represent the roles of a pattern.

Unary constraints represent conditions for a single role (e.g. “the element in role *X* must be an abstract class”), whereas *binary constraints* represent relationships between two roles (e.g. “the class in role *X* must be a subclass of the class in role *Y*”).

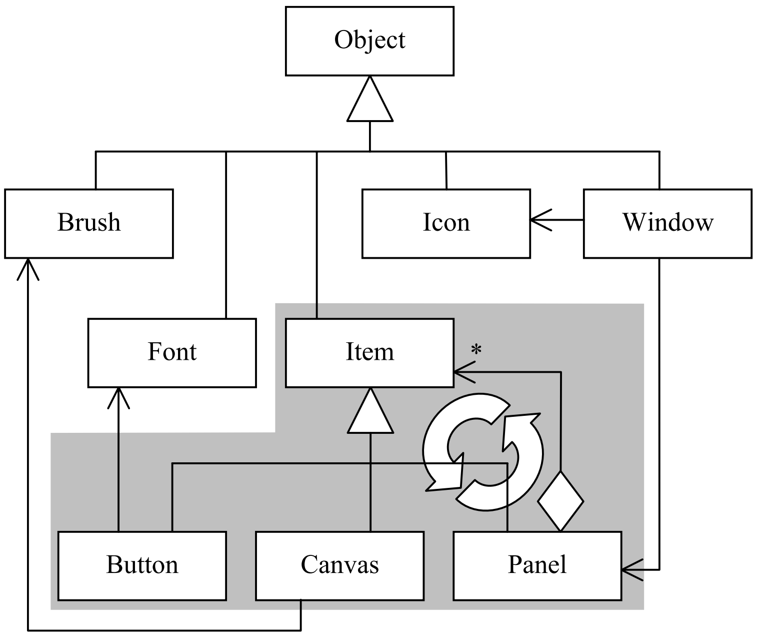


Figure 4.3: Detecting the Composite design pattern in a software system

In Maisa, typically a set of UML class diagrams is analyzed to detect instances of design patterns, but also source code can be analyzed, since Maisa has been integrated with a C++ analyzer. Pattern descriptions as well as source code information are stored as PROLOG facts, and the variable domains are initialized to contain the identifiers from the diagrams that are being analyzed. Efficient search has been achieved by implementing *pruning*, i.e. selection of relevant information and removal of impossible solution candidates with the *AC-3 algorithm* [Mac77].

4.3.2 Concept Analysis: Pattern Inference without a Template Library

It is not always possible to use a pattern library as a basis for pattern extraction. Constructing such libraries requires a lot of effort. If one decides to use an existing library, one is faced with a problem of choosing an appropriate library for the current application. It might be difficult to know which patterns to look for. It may also be the case that the system consists mainly of structures that are not typical to any other application. If no design patterns from the available predefined libraries were adopted for the application, then it is not possible to look for pattern instances in the application by the pattern matching approach. The (perhaps unconsciously) applied patterns can only be *inferred* from the code or the design.

Tonella and Antoniol demonstrate that instances of design patterns that are not known to exist can be found using *concept analysis* [ToA99]. Other work using concept analysis for inferring high-level information about software systems include extraction of *code configurations* [KrS94, Sne96] and analyzing the relation between procedures and global variables [LiS97]. In [SiR97] concept analysis is used to identify modules. In this thesis we use concept analysis to produce role-based annotations for defining the hot spots of a framework. According to our experiences many of the hot spots found in the reuse interfaces of frameworks are structurally framework-specific and not necessarily direct or even variant instances of any general patterns. That is why a library-based solution does not work in a general case for extracting framework annotations. Since concept analysis is not dependent of any predefined library it offers a potentially better solution.

Concept analysis provides a way to discover sensible groupings of *objects* that have common *attributes* in a certain *context* [SiR97]. Formally a context is a triple $C = (O, A, R)$, where O and A are finite sets of objects and attributes, respectively, and R is a binary relation between O and A. As an example, objects could be different kinds of sports and attributes could be characteristics of these sports. R could then be expressed as a table showing which characteristics each sport has (see table 4.4).

R		*attributes*			
		athletics	**ballgame**	**team sport**	**Olympic sport**
objects	**bowling**		√		
	cricket		√	√	
	javelin	√			√
	long jump	√			√
	tennis		√		√
	volleyball		√	√	√

Table 4.4: A simple context of sports and their characteristics

In a context $C = (O, A, R)$ a *concept* is a maximal collection of objects sharing common attributes, i.e. it is a grouping of all the objects that share a set of attributes. Formally: let $X \subseteq O$ and $Y \subseteq A$. Let us also define mappings $\sigma(X) = \{a \in A \mid \forall o \in X\text{: } (o, a) \in R\}$ (the common attributes of X) and $\tau(Y) = \{o \in O \mid \forall a \in Y\text{: } (o, a) \in R\}$ (the common objects of Y). Using these definitions a concept can be defined as a pair of sets (X, Y) such that $Y = \sigma(X)$ and $X = \tau(Y)$, where X is called the concept's *extent* and Y is called the concept's *intent*.

A concept (X_0, Y_0) is a *subconcept* of (X_1, Y_1) if $X_0 \subseteq X_1$ (or equivalently $Y_1 \subseteq Y_0$). The subconcept relation (denoted by $\leq$ in the text and by edges in the figures) forms a complete partial order called *concept lattice* over the set of concepts (see figure 4.5 for an example based on table 4.4). The structure of the lattice is governed by the *basic theorem for concept lattices*:

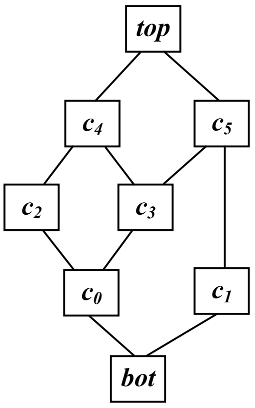
$$\sup_{i \in I}(X_i, Y_i) = (\tau(\sigma(\bigcup_{i \in I} X_i)), \bigcap_{i \in I} Y_i)\,.$$

The theorem says that the *least common superconcept* (i.e. *supremum* denoted by *sup*) of a set of concepts can be computed by intersecting their intents, and by finding the common objects of the resulting intersection. Based on the basic theorem, a simple bottom-up algorithm for computing the concept lattice for a given context can be defined as follows [SiR97]:

(1) Start with the *bottom element of the concept lattice* (*bot* in figure 4.5), i.e. the concept consisting of objects that have all the attributes. The extent of this concept is often empty as in the example.

(2) Compute the *atomic concepts*, i.e. the smallest concepts with the extent containing each of the objects treated as a singleton set. In other words, consider each object $o \in O$ in turn, and identify the attributes of o. Those attributes become the intent of a potential new atomic concept c. The extent of c is formed from o and all other objects that have the attributes enumerated in the intent of c. Finally, accept c as an atomic concept if its extent is smaller than the extents of all those other concepts whose extents also contain o. In our example these rules yield the atomic concepts c_0, c_1, c_2, c_3, and c_4. The concept c_5, for instance, cannot be an atomic concept because all the objects in its extent belong to other concepts (e.g. c_1 or c_3) that are its subconcepts. On the other hand, c_4 is an atomic concept because there is no smaller concept whose extent would contain *bowling*.

(3) Form a work list W containing all pairs of atomic concepts (c', c) such that $c \not\leq c'$ and $c' \not\leq c$. While W is not empty, remove a pair (c_a, c_b) from W. Then compute the supremum of c_a and c_b and assign it to c''. If c'' is a concept yet to be discovered then add all pairs of concepts (c'', c) such that $c \not\leq c''$ and $c'' \not\leq c$ to W. The process is repeated until W is empty. In the example in figure 4.5, c_5 can be constructed, e.g., from c_1 and c_3 (*Olympic sport* is the only common attribute for them). Then, *top* can be constructed, e.g., from c_4 and c_5 (the intersection of their intents is empty). After that, intersecting the intents of the other concepts does not produce any new concepts.



top	({*bowling*, *cricket*, *javelin*, *long jump*, *tennis*, *volleyball*}, ∅)
c_5	({*tennis*, *volleyball*, *javelin*, *long jump*}, {*Olympic sport*})
c_4	({*bowling*, *cricket*, *tennis*, *volleyball*}, {*ballgame*})
c_3	({*tennis*, *volleyball*}, {*ballgame*, *Olympic sport*})
c_2	({*cricket*, *volleyball*}, {*ballgame*, *team sport*})
c_1	({*javelin*, *long jump*}, {*athletics*, *Olympic sport*})
c_0	({*volleyball*}, {*ballgame*, *team sport*, *Olympic sport*})
bot	(∅, {*athletics*, *ballgame*, *team sport*, *Olympic sport*})

Figure 4.5: A concept lattice (and accompanying key) of a context of sports

Tonella and Antoniol extract structural design patterns from code using an adaptation of the concept analysis algorithm given above [ToA99]. They concentrate on structural relations between classes, and identify sets of classes as concept analysis objects and class members or inter-class relations as attributes. In this context concepts are defined as maximal collections of class sets having the same patterns of relations between them. They are therefore good candidates to represent design patterns.

With concept analysis one can even recover patterns that are not known to exist. It can expose structures that manifest, e.g., application-specific solutions or anti-patterns. In that sense concept analysis is a more powerful method than library-based matching. On the other hand, the results of concept analysis are less clear-cut. They require careful interpretation. Furthermore, it is difficult to use concept analysis to automatically detect architectural conflicts if no definition of the desired architecture exists.

In chapter 5 we will show how concept analysis can be used to produce role-based annotations that define the reuse interface of a framework. We will discuss ways to select relevant source code elements and their properties from both the framework itself and its available specializations to be represented as concept analysis objects and attributes. We will also show how the results of concept analysis can be translated to roles and constraints that describe the requirements for the applications specializing the framework.

4.3.3 Concept Partitions

In order to be able to translate the concepts resulting from an analysis of a context to the roles of a framework annotation, we must define an unambiguous mapping from a set of concepts to a set of roles. The mapping will be based on the extents of the concepts. The objects belonging to the extent of a concept are precisely those implementation elements that are intended to be playing the role corresponding to the concept.

In the translation process we are looking for a set of roles where each program element (i.e. each concept analysis object) plays exactly one role (i.e. belongs to exactly one extent). Unfortunately in a general case, an object may belong to a number of extents in a concept lattice (for instance, *volleyball* belongs to the extents of c_2, c_3, c_4, c_5, and *top* in figure 4.5). However, we can form a *concept partition*, i.e. a set of concepts where each object takes part in exactly one concept, from any concept lattice [SiR98]. In general, $P = \{(X_0, Y_0), \ldots, (X_{k-1}, Y_{k-1})\}$ is a concept partition if and only if the extents of the concepts cover the object set (i.e. $\bigcup_{i \in I} X_i = O$) and are pair-wise disjoint (i.e. $X_i \cap X_j = \varnothing$ for all $i \neq j$ and $X_i, X_j \in P$).

It is clear that all contexts have at least the *trivial partition*, which consists of only one concept — the top of the lattice whose extent includes all objects. An *atomic partition*, on the other hand, is a concept partition consisting of exactly the atomic concepts (recall chapter 4.3.2). A concept lattice need not have an atomic partition. For example, the lattice in figure 4.5 does not have an atomic partition: the atomic concepts c_0, c_2, c_3, and c_4 overlap (*volleyball* belongs to the extents of them all).

As we will shortly see, atomic partitions can be used as a starting point when generating all partitions for a given context. That is why it is useful to be able to guarantee the existence of atomic partitions. Contexts that result in atomic concepts forming an atomic partition are said to be *well-formed*. Formally, a context $C = (O, A, R)$ is well-formed if and only if, for every pair of objects $x, y \in O$, $\sigma(\{x\}) \subseteq \sigma(\{y\})$ implies $\sigma(\{x\}) = \sigma(\{y\})$.

Every context can be transformed to a well-formed context by extending it with additional attributes. One way of doing this is to augment the context with *negative information* (see [SiR98] for other options). The idea is to first identify those pairs of objects (x, y) whose extents are in a genuine subset relation with each other (i.e. $\sigma(\{x\}) \subset \sigma(\{y\})$ and $\sigma(\{x\}) \neq \sigma(\{y\})$. Then, for attributes $a \in A$ such that

$a \notin \sigma(\{x\})$ and $a \in \sigma(\{y\})$, a new *complementary attribute* $\overline{a}$ such that $\tau(\{\overline{a}\}) = \{x \in O \mid (x, a) \notin R\}$ (i.e. $\overline{a}$ is an attribute of exactly those objects that do not have attribute a) is formed and added to the context. This is continued until there are no offending pairs (x, y) left, i.e. until the context becomes well-formed. This procedure is defined more precisely in the following algorithm:

```
$A' \leftarrow A$
$R' \leftarrow R$
while $(O, A', R')$ is not well-formed do
   let $x, y \in O$ such that $\sigma(\{x\}) \subset \sigma(\{y\})$
   let $a \in A'$ such that $a \notin \sigma(\{x\})$ and $a \in \sigma(\{y\})$
   $A' \leftarrow A' \cup \{\overline{a}\}$, where $\overline{a}$ is a new attribute
   $R' \leftarrow R' \cup \{(x, \overline{a}) \mid (x, a) \notin R'\}$
endwhile
```

Table 4.6 represents the context of table 4.4 after it has been augmented with negative information (i.e. attributes *non-Olympic sport* and *non-team sport*) in order to make it well-formed. The resulting concept lattice is depicted in figure 4.7. We can see that each object belongs to exactly one atomic concept, so the atomic concepts (c_0, c_1, c_2, c_3, and c_4) form a partition, and the lattice as well as the corresponding context are, indeed, well-formed.

R		*attributes*					
		athletics	**ballgame**	**team sport**	**Olympic sport**	**non-team sport**	**non-Olympic sport**
objects	**bowling**		√			√	√
	cricket		√	√			√
	javelin	√			√	√	
	long jump	√			√	√	
	tennis		√		√	√	
	volleyball		√	√	√		

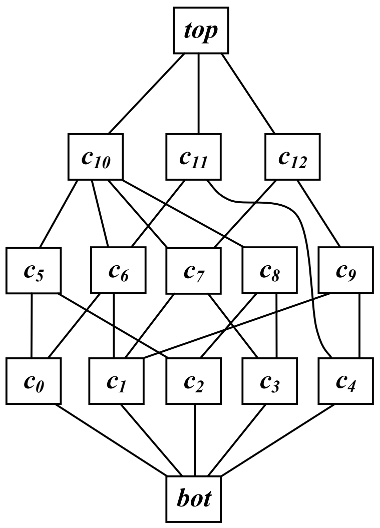
Table 4.6: A well-formed context

Given a well-formed concept lattice, we can define the following relations on its elements: a concept *d* *covers* concept *c* if $c < d$ and there is no other concept *e* such that $c < e < d$. The set of all concepts *d* such that *d covers c* is denoted by *covs*(*c*). The set of elements *subordinate* to *d*, denoted by *subs*(*d*), is the set of all concepts *c* such that $c < d$.

Using these definitions an algorithm can be given that generates all partitions of a well-formed context by starting from the atomic partition, i.e. *covs*(*bot*), and by moving on up to the trivial partition:

```
$A' \leftarrow$ covs($bot$)                          // the atomic partition
$P \leftarrow \{A'\}$
$W \leftarrow \{A'\}$
while $W \neq \varnothing$ do
   remove some partition $p$ from $W$
   for each concept $c \in p$ do
      for each $c' \in$ covs($c$) do
         $p' \leftarrow p -$ subs($c'$)
         if $(\bigcup p') \cap c' = \varnothing$ then       // p' and c' are disjoint
            $p'' \leftarrow p' \cup \{c'\}$
            if $p'' \notin P$ then
               $P \leftarrow P \cup \{p''\}$
               $W \leftarrow W \cup \{p''\}$
            endif
         endif
      endfor
   endfor
endwhile
```

A new partition *p'* is formed from a lower level partition *p* by starting with a copy of *p*. Then a concept *c* of *p* is selected, and a cover *c'* of *c* is chosen. Finally, *c'* is added to the new partition *p'* and any overlapping concepts are removed from *p'*. The algorithm uses a variable *P* to store the formed partitions and a work list *W* for partitions that still need to be considered. Both *P* and *W* will be initialized to contain the atomic partition *A*. The algorithm continues until *W* becomes empty.



top	({*bowling*, *cricket*, *javelin*, *long jump*, *tennis*, *volleyball*}, ∅)
c_{12}	({*tennis*, *javelin*, *long jump*, *bowling*}, {*non-team sport*})
c_{11}	({*tennis*, *volleyball*, *javelin*, *long jump*}, {*Olympic sport*})
c_{10}	({*bowling*, *cricket*, *tennis*, *volleyball*}, {*ballgame*})
c_9	({*tennis*, *javelin*, *long jump*}, {*Olympic sport*, *non-team sport*})
c_8	({*cricket*, *bowling*}, {*ballgame*, *non-Olympic sport*})
c_7	({*tennis*, *bowling*}, {*ballgame*, *non-team sport*})
c_6	({*volleyball*, *tennis*}, {*ballgame*, *Olympic sport*})
c_5	({*volleyball*, *cricket*}, {*ballgame*, *team sport*})
c_4	({*javelin*, *long jump*}, {*athletics*, *Olympic sport*, *non-team sport*})
c_3	({*bowling*}, {*ballgame*, *non-Olympic sport*, *non-team sport*})
c_2	({*cricket*}, {*ballgame*, *team sport*, *non-Olympic sport*})
c_1	({*tennis*}, {*ballgame*, *Olympic sport*, *non-team sport*})
c_0	({*volleyball*}, {*ballgame*, *team sport*, *Olympic sport*})
bot	(∅, {*athletics*, *ballgame*, *team sport*, *Olympic sport*, *non-team sport*, *non-Olympic sport*})

Figure 4.7: A concept lattice analyzed from the context in table 4.6

The concept lattice in figure 4.7 has 16 partitions. All partitions, including the trivial partition P_{top} = {*top*} and the atomic partition $P_0 = \{c_0, c_1, c_2, c_3, c_4\}$, are listed in table 4.8.

P_0	$\{c_0, c_1, c_2, c_3, c_4\}$	P_1	$\{c_1, c_3, c_4, c_5\}$	P_2	$\{c_2, c_3, c_4, c_6\}$	P_3	$\{c_0, c_2, c_4, c_7\}$
P_4	$\{c_0, c_2, c_3, c_9\}$	P_5	$\{c_0, c_1, c_4, c_8\}$	P_6	$\{c_2, c_3, c_{11}\}$	P_7	$\{c_4, c_{10}\}$
P_8	$\{c_4, c_5, c_7\}$	P_9	$\{c_3, c_5, c_9\}$	P_{10}	$\{c_4, c_6, c_8\}$	P_{11}	$\{c_0, c_2, c_{12}\}$
P_{12}	$\{c_0, c_8, c_9\}$	P_{13}	$\{c_8, c_{11}\}$	P_{14}	$\{c_5, c_{12}\}$	P_{top}	$\{top\}$

Table 4.8: The partitions of the concept lattice given in figure 4.7

4.3.4 Framework Hot Spot Recovery

Framework hot spot recovery can be viewed as a special case of (design) pattern recovery. Pattern instances usually reside on the borderline between the framework and the applications specialized from it, and patterns, in general, are used to increase flexibility in reusable architectures.

The use of template methods and associated overriding of hook methods is characteristic to the white-box frameworks, and the knowledge of the existence, location, and rationale of template methods is essential for the framework users [FaS97, Kel99, SRM99]. We follow Pree's definition of hot spots as points in a framework that must be augmented or modified when deriving a new application [Pre95, Pre99]. Hook methods are the main implementation mechanism for hot spots. Thus, template and hook methods are obvious candidates when trying to locate the hot spots of a framework.

Most of the hot spots can often be found by analyzing overridden methods, because polymorphism needed in hook methods is most commonly implemented using method overriding. Demeyer proposes finding hook methods by searching for methods that override another method [Dem98]. Template methods can then be identified as those methods that call the candidate hooks. The exact nature and variability properties of the found hot spots can deduced by examining the invocation relationship between the template and the hook. The call can be made through a self or super reference, via an instance variable or field, via a self or super method returning an instance of the hook class, via a parameter of the template method that is an instance of the hook class, or via a global variable.

Demeyer's idea is implemented in the FAMOOS project as a part of the MOOSE re-engineering environment [FAM02]. The problem with this method is that it produces a lot of *noise*, i.e. template methods that are not part of a hot spot. This is obvious since there are usually hundreds of overriding relationships between methods in any non-trivial application framework, and only a fraction of them is actually related to a hot spot.

One way of dealing with proliferation of hot spot candidates is to reduce the search space. We argue that the main concepts of a mature framework usually map fairly consistently with the top-level interfaces in the framework implementation, and therefore it is efficient to start the search from them. We base our argument on the common observation that a good framework design has only few abstract classes defining important sites of customization [GaB99, SLB00].

Every interface does not necessarily imply a hot spot, however. There are interfaces that are used only internally in the framework and which thus are irrelevant to the framework adapter. They must be identified and left out based on domain experience and the documentation available for the framework.

Despite these ambiguities we feel that it is best to start the analysis from the top-level interfaces of the framework in order to find its most important hot spots as early as possible. In the next chapter we will represent the details of our method for annotating frameworks and for extracting parts of those annotations directly from source code.

5. Annotating Frameworks with Specialization Patterns

In this chapter we will show how a framework's reuse interface can be annotated with role-based specifications. We will also present a role-based pattern model and a pattern extraction method that partly automates the framework annotation process. The method uses information about the framework's structure and intended usage as its input. Its purpose is to translate the input (e.g. the source code of the framework and its available example adaptations) to the language defined by the concepts of the pattern model.

First, in chapter 5.1 we look at examples of hot spots in a real framework. Next, in chapter 5.2 we discuss our pattern model in detail, and show how the model can be used in practice to describe hot spots. Finally, in chapter 5.3 we present the main contribution of this thesis: a method to automatically produce framework annotations from source code.

5.1 Annotating Framework Hot Spots

There are a number of useful heuristics that help in identifying and specifying a framework's hot spots. The heuristics we discuss here are most relevant to white-box frameworks, which use inheritance as the main specialization technique. We assume that the framework has a layered structure (see chapter 2.1) and that its basic concepts are implemented on the highest layer as abstract interfaces. In addition, we assume that we are annotating a fairly mature framework and that we have enough information about the framework's structure and its intended use.

5.1.1 The JUnit Testing Framework

As an example of a mature framework we use JUnit [GaB99, JUn02], a framework for implementing systematic unit testing of Java programs. It was chosen as an example framework because it is commonly known, mature, simple, well designed, implemented in Java, and freely available. The JUnit distribution consists of about 50 classes of which less than 20 belong to the core framework. The rest of the classes are UI components, extensions, and samples.

The design philosophy of JUnit is based on the premise that every program feature must have an associated test. Immediately after a new function has been added to a program, a set of tests must be added to ensure that the function was implemented correctly. Thus, when encountering a failure, one can concentrate on debugging in a limited area of code added after the previous successful tests.

The tight integration of implementation and testing implies that the developers themselves are obliged to write tests for the classes they produce. In order to avoid burdening programmers with too much extra work, testing should be as easy as possible. That is why all tests should be made automatic, and they should check their own results. This will enable developers to test as often as they compile.

A testing framework must be provided for the programmers so that any duplication of effort associated with testing (and regression testing later on) can be avoided. The framework must support writing tests that retain their value over time. Modules must be testable after integration and even in other contexts than originally designed for. Also other people than the original writer must be able to run tests as well as interpret the results, and it must be possible to combine tests from various authors. Finally, the framework should support creating new tests from existing ones and reusing *test fixtures* (i.e. context objects used as test material) to run different tests.

The JUnit testing framework tries to meet all the goals mentioned above. Figure 5.1 shows the core abstractions of JUnit annotated with arrows expressing the applied design patterns. The design of the framework is very simple, yet expressive. There are only a handful of classes, but they have rich interactions between them. There is a high pattern density around the key abstractions of the framework implying a mature framework.

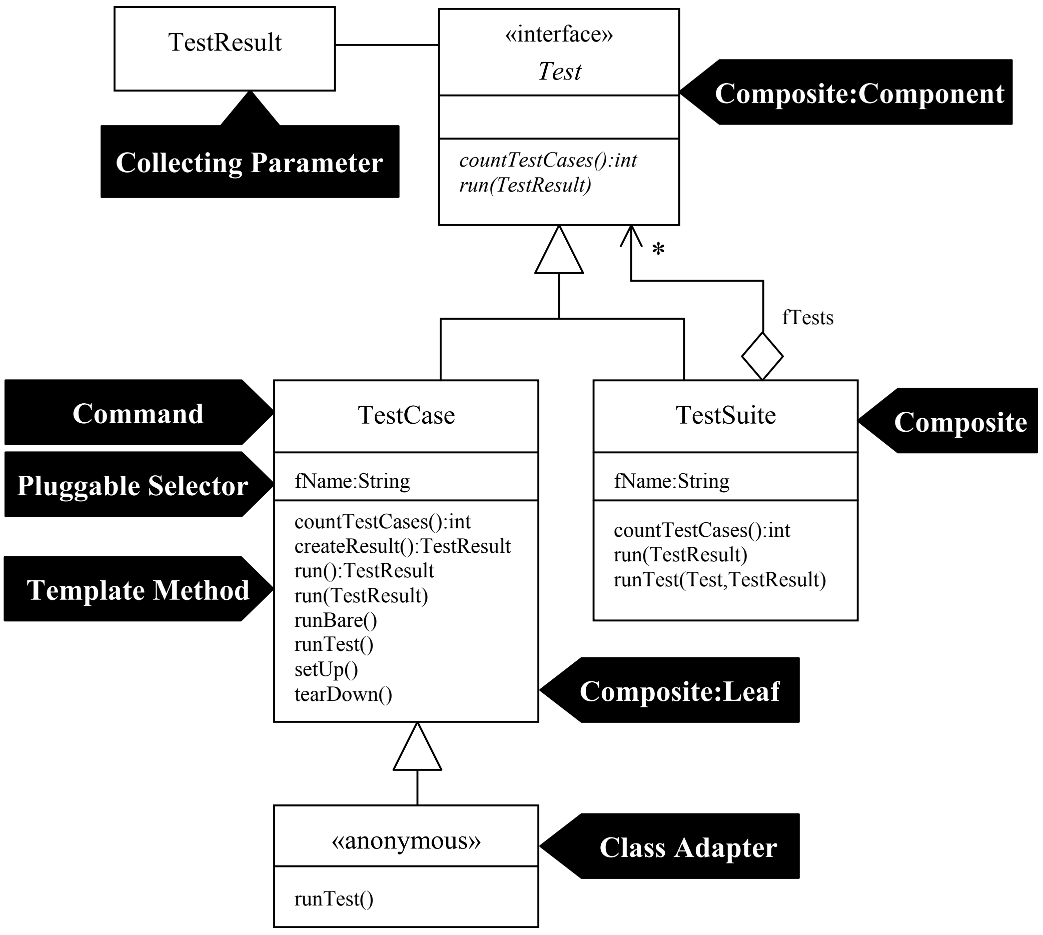


Figure 5.1: Design patterns in JUnit [GaB99]

The centerpiece of the design is the *TestCase* class. It defines the basic organizational building blocks of tests. It adheres to the *Command* design pattern [GHJ95] because its fundamental purpose is to encapsulate (requests to run) tests as objects so that they can be organized and manipulated in various ways. Each *TestCase* object has a name (*fName*) that can be used, for example, in reporting test results. The name is given as a constructor parameter when creating new *TestCase* objects (constructors are left out from the figure for simplicity).

The method that executes tests is called *run*. It is a *Template Method* [GHJ95]. It defines the skeletal algorithm common for all test scripts: set up a fixture for the test (*setUp*), run some code against it (*runTest*), and finally clean up (*tearDown*). This means that each test is run against a fresh fixture to minimize dependencies between tests and to maximize their reusability.

JUnit uses the *Collecting Parameter* design pattern [Bec97] to enable collecting results from several tests. The idea is to pass a *TestResult* object to the *run* methods and store the results into that parameter object. It is thus possible to condense the results, for example, to a cumulative number of successes and a list of failures and their descriptions.

As a basic organizational unit of tests the *TestCase* class is a logical place to define the actual test script methods. *TestCase* defines a uniform interface for executing all tests by calling *run*. It is in the subclasses of *TestCase*, however, where we actually override the *runTest* method and provide the testing actions. Suppose we want to define a test method for each method *m* in a class *C*. Should we have a separate subclass for each method of each class we want to test? We would end up having more test classes than we have actual production classes. It would be hard to even come up with names for all test classes.

Fortunately, JUnit provides ways to avoid such unnecessary proliferation of classes. The default implementation of *runTest* defined in *TestCase* uses the *Pluggable Selector* design pattern [Bec97] implemented by means of Java Reflection API to enable automatic execution of a test method that has the same name as the *TestCase* object. This means that the user can choose a test to be run just by creating a new instance of her test case class with the same name as the chosen test script method. Another way is to use Java's *anonymous inner classes* to define sublasses of *TestCase* where only the *runTest* method is overridden differently. This solution is an instance of the *Class Adapter* design pattern [GHJ95].

To enable hierarchical arrangement of test cases and uniform execution of test suites consisting of many tests, JUnit applies the *Composite* design pattern [GHJ95] to the *TestCase* class. An additional interface (*Test*) is introduced to play the *Component* role in the pattern. The method whose invocations need to be unified is, of course, *run*. *TestCase* implements the *Test* interface as a *Leaf* node. It does not have any subtests. On the other hand, *TestSuite* plays the *Composite* role in the pattern. It has a vector for storing references to its children (*fTests*) and associated accessor methods (left out from the figure for simplicity) to manage the vector. *TestSuite* also implements the *run* method from the *Test* interface by delegating the call to its children.

5.1.2 JUnit Hot Spots

In the previous chapter we identified the main concepts of JUnit and discussed JUnit's internal design. In annotating the hot spots of the framework, we must emphasize the application developer's point of view. Since any framework can be annotated in numerous different ways, the framework annotator must decide what kind of assistance she wants to give to the framework users. Adding constraints will give the user better guidance. However, at the same time she will loose some of her freedom. The formalization of a framework's reuse interface always reveals only a subset of the possible implementation variations. We argue that it is better to first annotate quite a narrow reuse interface, and later extend it to enable more advanced ways of using the framework.

Figure 5.2 illustrates the most important activities involved in any use case of JUnit: defining the test scripts (*DefiningTests*), defining and initializing common test material (*SettingUpFixture*), selecting and grouping tests to be run (*SelectingAndGroupingTests*), and actually running the test scripts with a given test runner (*RunningTests*). To point out where these activities take place (i.e. where the related hot spots are in the framework and what kind of restrictions there are associated with them), the figure shows a partial class diagram of JUnit (the upper part of the diagram) together with some application-level classes (the lower part of the diagram) that exemplify how the constructs of the framework can be extended and used.

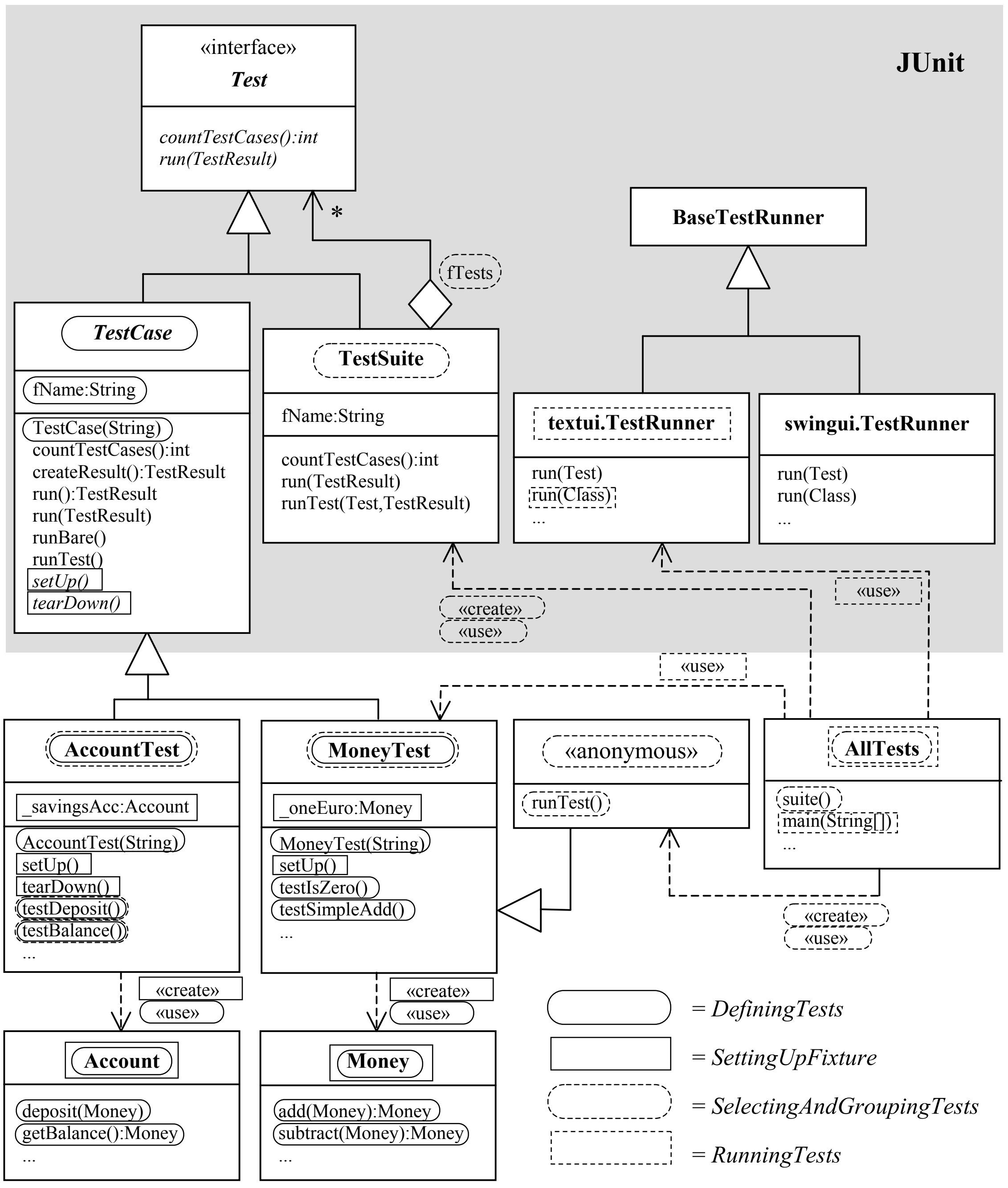


Figure 5.2: Hot spots in the JUnit framework

The *DefiningTests* hot spot involves subclassing the *TestCase* class from the framework. The subclasses (e.g. *MoneyTest* and *AccountTest*) must implement the initialization of the *fName* field in *TestCase*. This can be done most conveniently by defining a constructor that takes a name as an argument and then passes it over to the superclass via calling the constructor of the superclass. The test classes also define a number of test scripts (e.g. *testIsZero*, *testSimpleAdd*, and *testDeposit*) that call methods of the classes to be tested (e.g. *add* and *subtract* in *Money* or *deposit* and *getBalance* in *Account*). Each test script interacts with the objects to be tested and finally verifies the expected results with assertions. For example:

```
public class MoneyTest extends TestCase {
   public MoneyTest(String name) {
      super(name);
   }
   protected void testSimpleAdd() {
      Money oneEuro = new Money(1, "euro");
      Money twoEuros = new Money(2, "euro");
      assert(oneEuro.add(oneEuro).equals(twoEuros));
   }
}
```

Note that various kinds of assertion methods are defined in the *Assert* utility class that has been left out from the diagrams for simplicity. *TestCase* inherits *Assert* so the assertion methods are available to all subclasses of *TestCase*.

The creation of the test material objects can be done in the test scripts themselves, but in order to be able to share these fixture definitions among scripts, the developer must use the *SettingUpFixture* hot spot to define the initialization (e.g. *MoneyTest.setUp*) and optionally also resource deallocation (e.g. *AccountTest.tearDown*) for fixture attributes (e.g. *MoneyTest._oneEuro*, *AccountTest._savingsAcc*) by overriding the corresponding methods declared in *TestCase*. Setting up a test fixture most often involves creating one or more instances of the classes to be tested (e.g. *Money*, *Account*). Here is an example of setting up a fixture and using it in a test script:

```
public class MoneyTest extends TestCase {
   …
   private Money _oneEuro;
   private Money _twoEuros;
   protected void setUp() {
      _oneEuro = new Money(1, "euro");
      _twoEuros = new Money(2, "euro");
   }
   protected void testSimpleAdd() {
      assert(_oneEuro.add(_oneEuro).equals(_twoEuros));
   }
}
```

The *TestSuite* class offers a possibility to combine test cases into tree hierarchies through its *fTests* attribute. From the application developer's point of view this functionality is useful for selecting and grouping together those tests that should be run together. The *SelectingAndGroupingTests* hot spot guides the developer to do just that.

In figure 5.2 the *AllTests* class defines a static *suite* method where it creates a *TestSuite* instance and adds test cases to it. There are four ways to do this, two of which are shown in the figure. First, an instance of an anonymous subclass of *MoneyTest* has been created to explicitly override the *runTest* method to identify a test script (*testSimpleAdd* in this case) that will be added to the test suite. This is the most type safe way of selecting tests to be run. Second, all the test scripts in *AccountTest* (e.g.

testDeposit and *testBalance*) have been added to the suite by giving the test case class as an argument to a framework method that adds the scripts to the suite using reflection. In order to be added to the suite, a test method must fulfill two requirements: its name must start with the prefix "*test*" and it must not take any arguments.

It would be also possible to add only selected scripts from a test case class one at the time by creating an instance of the class with the same name as the selected script for each script to be selected. Finally, it would be possible to add a whole test suite by calling a *suite* method in another test case and by adding the resulting suite as a subsuite. Below is an example of a suite method that creates a *TestSuite* object, adds tests to it using all the methods described above, and finally returns the suite:

```
public static Test suite() {
   TestSuite s = new TestSuite("SomeMoneyTests");
   s.addTest(new MoneyTest("add") {
     public void runTest() {
       testSimpleAdd();
     }
   };);
   s.addTest(AccountTest.class);
   s.addTest(new MoneyTest("testIsZero"));
   s.addTest(MoreAdvancedMoneyTests.suite());
   return s;
}
```

JUnit provides different test runners, which can run a test suite and collect the results. A test runner either expects a static *suite* method as the entry point to get a test to run or it will extract the suite automatically. *RunningTests*, the fourth hot spot depicted in figure 5.2, is meant for selecting the tests to be run and an appropriate test runner class (e.g. *textui.TestRunner*) to execute them. This can be accomplished by defining a *main* method (e.g. *AllTests.main*) where the runner's static *run* method is called with a test case class as an argument. All test scripts defined in the given test case (or in its *suite* method if it has one) will be looked up and executed through reflection. Here is an example:

```
public class AllTests {
   public static void main(String[] args) {
     junit.textui.TestRunner.run(MoneyTest.class);
   }
}
```

Note that figure 5.2 represents only an example of how to use some of the framework's hot spots. A number of more advanced hot spots have been left out completely. Every alternative and detail of utilizing the given hot spots have not been described either. In chapter 5.2.3 we will use *SettingUpFixture* as an example of how to specify a hot spot with our pattern notation (see appendices A and B for more hot spot annotations).

5.2 Specialization Patterns

In the following we will use *specialization patterns* for annotating the reuse interface of a framework [HHK01b]. They are an attempt to combine the intuitive task-driven framework assistance provided by active cookbooks, the concreteness of example applications, and the precise, declarative nature of role-based pattern formalisms.

A specialization pattern is a specification of a program structure, which can be instantiated in several contexts to get different kinds of concrete structures. Specialization patterns are different from design patterns in that they do not have to represent proven and generally applicable solutions to commonly recurring design problems. They are more like *protopatterns* [Kel99, Kot96] because they may describe *ad-hoc* or application-specific solutions expressed in pattern format.

Although specialization patterns often express framework-specific solutions, it is still a worthwhile activity to catalog and communicate them in a systematic way. In fact, whenever a specialization pattern describes a framework hot spot, it is by definition applicable in many contexts, namely in all applications specializing that hot spot.

A specialization pattern is given in terms of roles to be played by structural elements of a program. We call the commitment of a program element to play a particular role a *contract*. A role may stand for a single element, or a set of elements. Thus, a role can have multiple contracts, and a program element can play many roles through a number of contracts. *Cardinality* of a role bounds the number of its contracts.

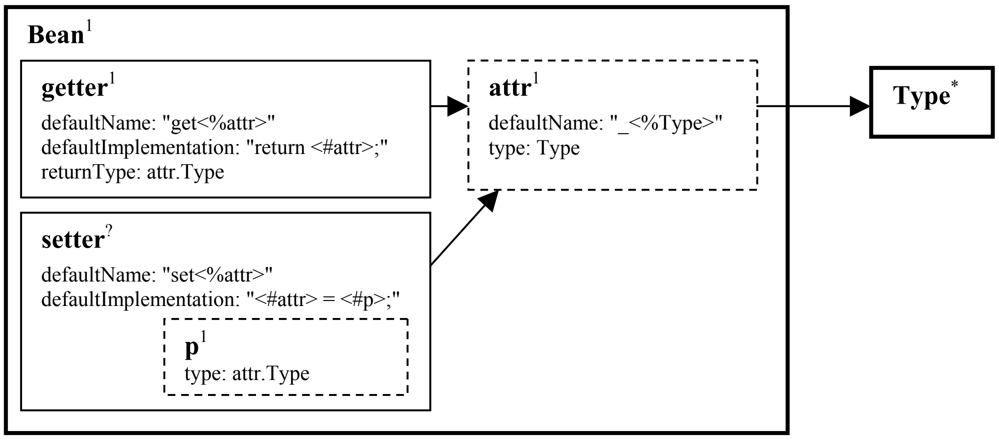
A role is always played by a particular kind of a program element. Consequently, we can speak of *class roles*, *method roles*, *field roles* and so on. For each kind of a role, there is a set of *properties* that can be associated with the role. For instance, for a class role there is an *inheritance property* specifying the required inheritance relationship of each class associated with that role. Properties like this, specifying requirements for the static structure of the concrete program elements playing the role, are called *constraints*.

Unlike constraints, some properties are only meant to affect code generation or dynamic user guidance. For instance, most role kinds support a *default name* property for specifying the name of the program element used when, e.g., a tool generates a default implementation for the element. Those properties are called *templates*. They are used when textual information needs to be generated, but they are not checked afterwards.

5.2.1 Pattern Diagrams

A specialization pattern can be expressed as a *pattern diagram*. Figure 5.3 shows the definition of a pattern representing a reusable structure for a class (*Bean*) having a number of typed (*Type*) attributes (*attr*) with a getter and optionally also a setter method for each of them. The rectangles in the diagram represent roles. The class roles (*Bean*, *Type*) are denoted with thick borders, the method roles (*getter*, *setter*) with thin borders, and the field (*attr*) and parameter (*p*) roles with dashed borders. The relative placement of roles within other roles reflects the declaration hierarchy of roles. Figure 5.3 shows that *getter*, *setter*, and *attr* are all declared within *Bean*, which means that also the program elements playing those roles must be declared within a class that plays the *Bean* role.

Each role has a name written in bold followed by a cardinality constraint given as superscript after the name. Cardinality can be either *exactly one* (1), *from zero to one* (?), *from one to infinity* (+), or *from zero to infinity* (*) with respect to the other roles the role depends on. The *dependencies* are denoted with arrows between roles (e.g. from *attr* to *Type*). (Actually each dependency has also a name, which can be referred to in constraints and templates as will be described in chapter 6.2 and exemplified in appendix B. However, we omit dependency names from pattern diagrams for simplicity.)



Bean¹
getter¹
defaultName: "get<%attr>"
defaultImplementation: "return <#attr>;"
returnType: attr.Type
setter?
defaultName: "set<%attr>"
defaultImplementation: "<#attr> = <#p>;"
p¹
type: attr.Type
attr¹
defaultName: "_<%Type>"
type: Type
Type*

Figure 5.3: A pattern diagram

The main properties of a role can be optionally listed below the role name. For instance, in figure 5.3 there is a *returnType* constraint declared for the *getter* role that states that the program elements playing the *getter* role must have the same return type as is the type of the field playing the *attr* role. Examples of templates include *defaultName* and *defaultImplementation* declared for *getter*. They have a text fragment given within double quotes that can be used for, e.g., code generation. The notation *<#R>* means that the name of an element playing the role *R* will be expanded to that place in the fragment during the evaluation of the template in a tool, e.g., when generating code. *<%R>* means the same except that the first letter of the name of the element will be changed to lower case if necessary to achieve Java style identifier naming.

For a complete syntax and description of the pattern diagram notation refer to [Hau02]. There is also a textual notation for the specialization pattern specifications (see appendix C for the syntax). It is usually used when detailed and complete specifications are required (see appendix B for examples).

5.2.2 Casting

Applying a specialization pattern is called *casting*, and the resulting structure is called a *cast*. Casting means incremental and interactive binding of suitable program elements to the unbound roles of the pattern. The casting process is represented to the developer as a dynamically changing sequence of tasks that guide her in adapting the generic solution proposed by the pattern. *Production tasks* instruct the developer to instantiate and bind a role. *Repairing tasks* assist the user in modifying a program element bound to a role to adhere to the constraints imposed by the role.

Since a cast is an instance of a specialization pattern it can be presented as a similar diagram. Figure 5.4 shows a *cast diagram* based on the pattern given in figure 5.3. The rectangles in the cast diagram represent contracts. Each contract is a manifestation of some role in the corresponding pattern diagram. The names of contracts are of form R_i where R is the corresponding role and subscript i, as a positive integer, identifies the contract amongst all contracts of role R. The arrows between contracts are called *dependency instances*. Each dependency instance between two contracts manifests a dependency defined between the two corresponding roles.

Unbound contracts are visible as production tasks within a tool environment. That is why each contract has a *state*. As a task, a contract can be *done* or *undone*. Furthermore, an undone task may be considered as *mandatory* or *optional*, depending on the cardinality of the associated role and the number of its instances (i.e. contracts). The state of each contract is written in superscript after its name.

The *casting algorithm* generates the tasks. The algorithm assumes a pattern specification and a cast, which it augments with new contracts whenever possible. The newly created contracts imply mandatory or optional production tasks to be carried out by the developer. As she completes a task, the state of the corresponding contract is changed to *done*, and the algorithm is re-evaluated to determine whether it is possible to create new contracts.

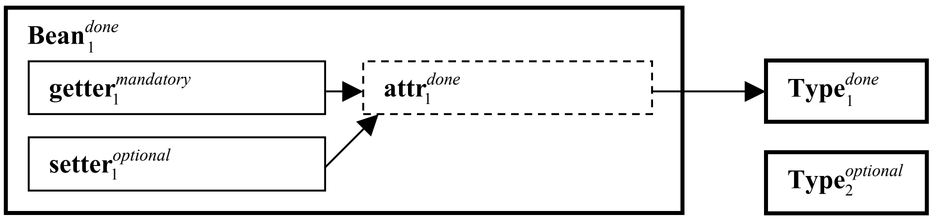

$\mathbf{Bean}_1^{done}$
$\mathbf{getter}_1^{mandatory}$
$\mathbf{attr}_1^{done}$
$\mathbf{setter}_1^{optional}$
$\mathbf{Type}_1^{done}$
$\mathbf{Type}_2^{optional}$

Figure 5.4: A cast diagram

The algorithm processes each role and decides whether it is necessary to create new contracts for that role. This is determined by first constructing all possible combinations of the contracts of the roles that the current role depends on. Then the algorithm checks if a correct amount of contracts exists for each of these combinations. If not, a new contract is created, with its state set to optional or mandatory, depending on whether the lower bound denoted by the cardinality constraint has been exceeded or not.

Figure 5.4 above portrays a *partial* cast, i.e. a pattern instance that is in the middle of instantiation. Some tasks have been done, resulting in a set of contracts. The interpretation of the diagram can be based on the semantic outline sketched for the diagram in figure 5.3. The developer has created a class and a field inside it. An unbound contract $getter_1$ is shown to the user as a task to provide a getter method for that field. As this is a mandatory task, the cast is not yet considered *complete*. The developer has also a choice of continuing with optional tasks to provide a setter for the field or to select a new type for another field. These tasks may, in turn, lead to new tasks.

Figure 5.5 gives an overview of the whole situation at this point. The dashed arrows show how the contracts of the current cast are related to the roles in the pattern diagram above the cast, and how the bound (done) contracts, on the other hand, are also associated to the pieces of source code (i.e. implementation) below the cast.

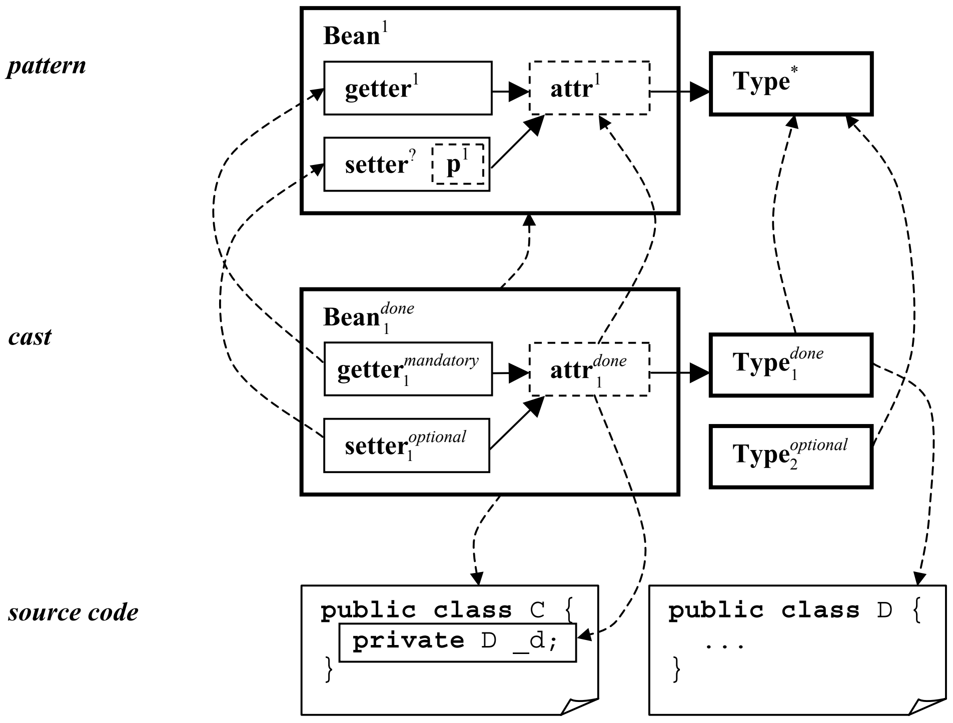


Figure 5.5: Pieces of code cast to roles of a pattern

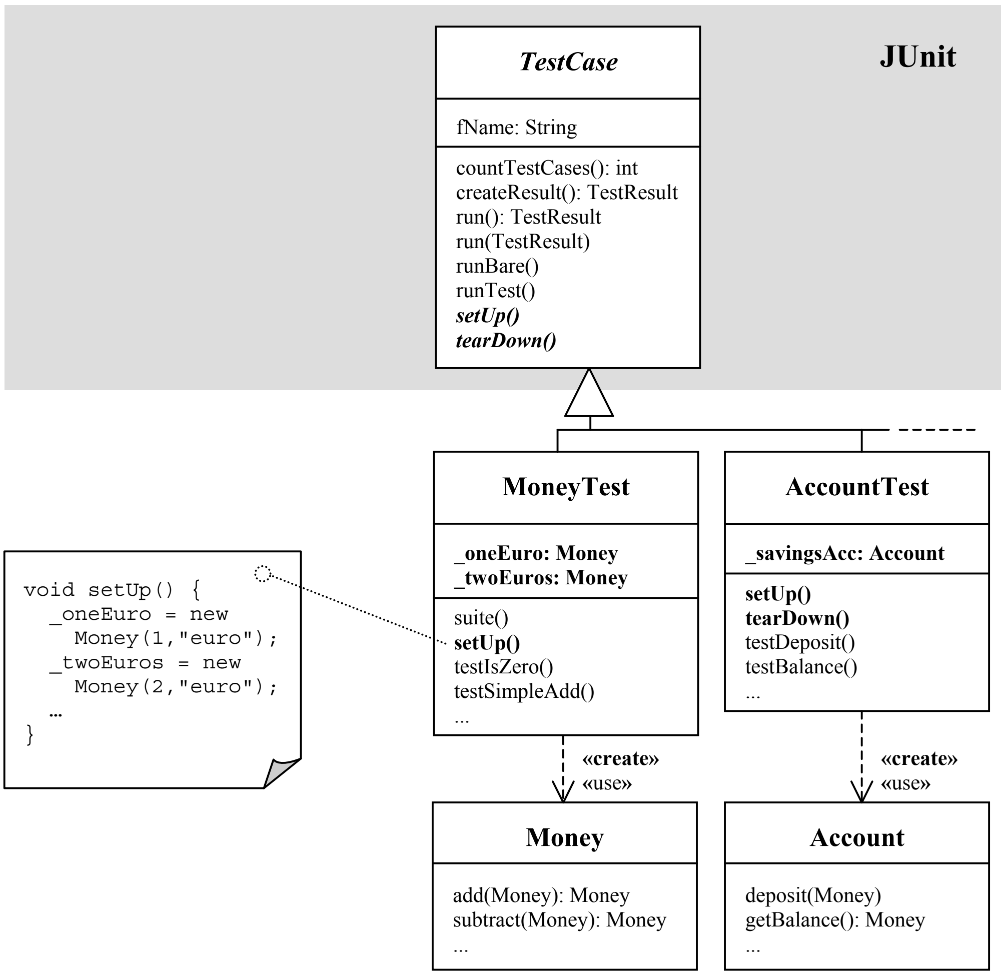
5.2.3 Specifying a Pattern for Setting Up Test Fixtures in JUnit

As an example of how a hot spot that can be annotated with a specialization pattern, we look at *SettingUpFixture* already briefly introduced in chapter 5.1.2. The purpose of this chapter is only to give a concise overview of the usage of our pattern model. For a full account on annotating a framework with specialization patterns, see [Vil01] or [Hau02].

Before explaining the details of the *SettingUpFixture* specialization pattern, it is important to distinguish between framework roles and application roles. A *framework role* is a role that will be bound to a framework source code element by the framework developer. Framework roles can be deduced from the source code directly. For example, there usually exists a one-to-one mapping between a framework interface representing a certain framework's concept and a framework role in a pattern describing ways to implement that interface.

An *application role*, on the other hand, is a role that will be bound to an application source code element later on. Application roles typically depend on the framework roles and contain constraints that guide the framework adapter as she derives her application from the framework. The structure and constraints of application roles should condense the available information on the expected framework adaptations. This information can be gathered from the ready-made default components incorporated in the framework itself as well as from the existing applications already utilizing the framework.

Figure 5.6 represents the part of the diagram from figure 5.2 that contains the *SettingUpFixture* hot spot. The features, which are relevant to that hot spot, are written in bold. There is also a source code fragment showing how the *setUp* method has been implemented in *MoneyTest*.



```
void setUp() {
  _oneEuro = new
    Money(1,"euro");
  _twoEuros = new
    Money(2,"euro");
  …
}
```

Figure 5.6: A hot spot for setting up test fixtures

Figure 5.7 shows a pattern diagram describing the same hot spot. It specifies that if the developer wishes to have a common test fixture for all test scripts defined in a test case, she must bind her *TestCase* subclass to the *UserTestCase* role and then provide or generate a method, which overrides the *setUp* method declared in *TestCase*. She may also optionally override the *tearDown* method to release any resources required by the fixture. The default implementation of *setUp* consists of one initializing assignment statement (*fixtureCreation*) for each fixture attribute (*fix*) defined within the test case. There can be one or more fixture attributes per each fixture class (*Fixture*) selected by the developer.

There are three class roles in this pattern: a framework role and two application roles. Each framework role is typically named after the framework class it describes (e.g. *TestCase* in figure 5.7). The corresponding application role (e.g. *UserTestCase*) represents the set of possible application-specific subclasses to be derived from the framework class. Thus, it has a dependency to the framework role and an associated inheritance constraint. There might be also application roles that do not depend on any framework role (e.g. *Fixture*). The properties that need to be specified for such roles depend on the circumstances. For example, the *Fixture* role does not have any dependencies or constraints.

There are four method roles in figure 5.7. Those method roles that denote methods declared within the framework classes may have constraints restricting some of their properties (like the types of their return values), but it is not necessary, except for documentation purposes. The application method roles, on the other hand, typically have overriding constraints referring to the corresponding framework method roles (like *setUp* and *tearDown* in this pattern).

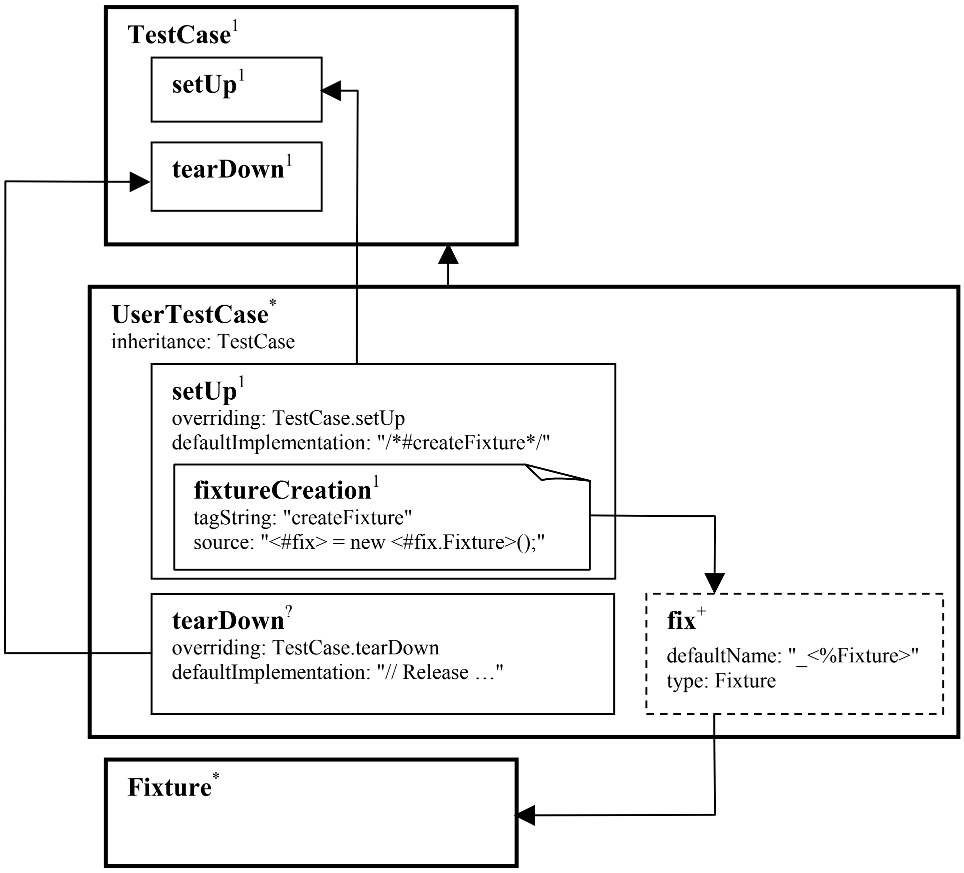

TestCase1
setUp1
tearDown1
UserTestCase*
inheritance: TestCase
setUp1
overriding: TestCase.setUp
defaultImplementation: "/*#createFixture*/"
fixtureCreation1
tagString: "createFixture"
source: "<#fix> = new <#fix.Fixture>();"
tearDown?
overriding: TestCase.tearDown
defaultImplementation: "// Release …"
fix+
defaultName: "_<%Fixture>"
type: Fixture
Fixture*

Figure 5.7: A hot spot specified as a specialization pattern

The method bodies and field initialization clauses can be described with *code snippet roles*. Figure 5.7 shows an example of a code snippet (*fixtureCreation*), which has a dependency to the *fix* field role. Field roles are typically defined as needed, depending on what kind of functionality is described in method bodies. Here the snippet describes creating fixture objects and assigning them to the attributes of the test case class. It is therefore natural to have a field role to represent the attributes and make the method body be dependent of the field role.

The framework developer should specify snippets as generalizations of the most representative examples among the existing implementations. Sometimes it is useful to declare many code snippets under one method role, e.g. to describe algorithmic options. In such a situation, selecting the implementation strategy (defined by a particular code snippet) for the method usually fixes the variation possibilities for the related fields and constructors, too. In such a case, it may be more natural to make the field roles dependent of the code snippet (and not vice versa as in figure 5.7), so that only those fields (and their constructors) that are referred to in that particular method body variation become activated when the corresponding code snippet role is bound.

5.3 Using Concept Analysis to Extract Patterns from Implementation

When we want to annotate reuse interfaces of frameworks with roles we have two objectives that must be brought into balance. First, we try to manage with as few roles as possible. This means that the size of the set of program elements represented by any particular role should be maximized. On the other hand, the cohesion of the set should be maximized also. This problem resembles the general modularity problem of software engineering where the goal is to divide a software system into modules with loose coupling between modules and strong cohesion within them.

Siff and Reps have successfully applied concept analysis for identifying modules in software systems [SiR97, SiR98]. We argue that concept analysis can also be used to produce specialization patterns from various kinds of framework descriptions or source code. Our goal is to annotate frameworks to aid deriving applications from them. To help the annotation process we propose a method to automatically extract specialization patterns, each of which defines one hot spot in the framework together with the requirements for the application-specific increment that uses the hot spot.

The main phases of the pattern extraction process are depicted in figure 5.8. The process uses as input a description of the framework itself and all its available specializations. To ensure the effectiveness of the method, the input should include a representative selection of all possible specializations. The input may be given either as source code or as some other, higher-level representation (e.g. as an AST or as a set of detailed class diagrams) defining the static structure of the framework and its specializations.

In principal, the input can be given at any level of precision. It should be noted, however, that the amount of details present in the input dictates the precision of the analysis and the accuracy of the extracted patterns. For example, to produce patterns that include default implementations for method bodies, the input must contain information about the method implementations found in example applications.

The extraction process starts with *context selection* where a formal context suitable for concept analysis is built by selecting the relevant elements from the input as objects and their relevant properties as attributes (1). The user is responsible for defining the relevancy criteria that distinguish the program elements that are interesting from the point of view of the pattern that is to be extracted. When the context has been formed, the concept analysis algorithm is used to produce a concept lattice from the context (2). After that, a suitable set of concepts is chosen as a blueprint for the pattern to be generated (3). Finally, the extent of each concept is translated to a role and the intent of the concept is translated to a set of constraints for that role (4).

The pattern extraction method is iterative in the sense that it first produces roles that will be declared right under the root node (i.e. the pattern itself) in the role declaration hierarchy. Then, in the next iteration, it continues to extract and declare roles under each role acquired in the first iteration. The number of iterations depends on the level of detail present in the input. A typical extraction process would require at least three iterations, namely the extraction of class roles, the extraction of method roles to be declared under the class roles, and the extraction of code snippet roles to represent default bodies for the methods. The last iteration may optionally include also the creation of class, field, and methods roles to represent the elements that are referred to in the method bodies as well as defining the constructor roles required to describe the initialization of the data fields used in the methods.

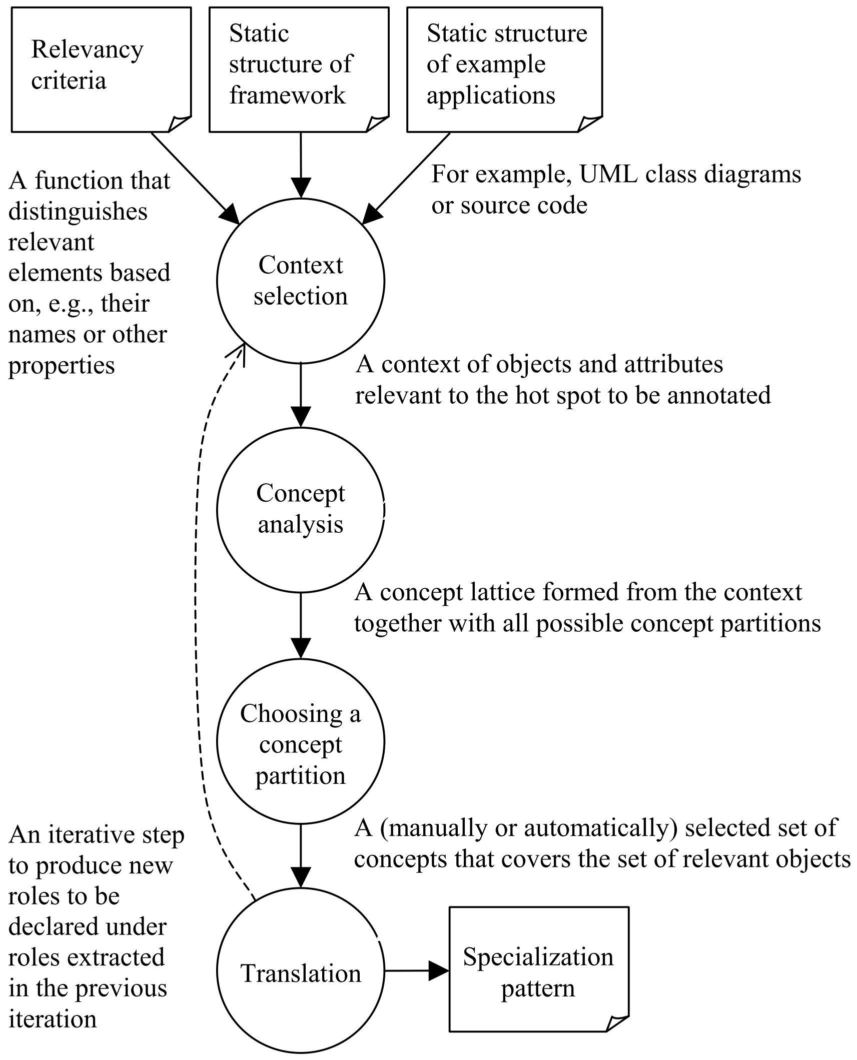

Relevancy criteria
Static structure of framework
Static structure of example applications
A function that distinguishes relevant elements based on, e.g., their names or other properties
For example, UML class diagrams or source code
Context selection
A context of objects and attributes relevant to the hot spot to be annotated
Concept analysis
A concept lattice formed from the context together with all possible concept partitions
Choosing a concept partition
A (manually or automatically) selected set of concepts that covers the set of relevant objects
An iterative step to produce new roles to be declared under roles extracted in the previous iteration
Translation
Specialization pattern

Figure 5.8: Extracting a specialization pattern from implementation

5.3.1 Forming a Context

The concept analysis algorithm is applied to a different context in each iteration. The kind of roles that are to be produced in the current iteration dictates the context (i.e. objects and attributes) that is selected from the input. For example, when extracting class roles, the classes and interfaces present in the input will be selected as objects and their features (e.g. inheritance relationships, declared methods, and data fields) will be selected as attributes. In general, for each role kind the method assumes a set of program elements of which the objects will be selected and a set of *property functions* which dictate the attributes. Attributes are defined by applying each property function to each element and by collecting all distinct results as the set of attributes. This implies that each property function should be applicable for all elements selected as concept analysis objects.

To be more precise, let us define a program element e as a pair (t, n), where $n \in N$ is a name that belongs to the set of unique element names (N) identifying the element among the set of all elements (E) and $t \in T$ is a type (or element kind) that belongs to the set of all possible element types (T). For instance, for a Java system E_{Java}, the set of types $T_{Java} = \{class, method, field, \ldots\}$. Similarly, the set of names N_{Java} could include the path names of all elements (classes, methods, …) in E_{Java}. We usually refer to an element by its name and omit its type if there is no danger of confusion.

A property function $f: E \rightarrow \mathcal{P}(E)$ is a mapping from a program element to a set of (other) program elements. Let $E_t \subset E$ be a set of all program elements of type t. The program element type (e.g. *class* if $t \in T_{Java}$) determines a set of property functions F_t that are applicable to the elements of that type, i.e. $F_t = \{f \mid f: E_t \rightarrow \mathcal{P}(E)\}$.

Given an element $e_t \in E_t$, each property function $f \in F_t$ yields a *property value* $d = f(e_t)$. The property value d is often a set that contains only one element (e.g. if f is a mapping from a method to its return type), but it can also contain multiple elements. For example, for Java classes the set of property functions could be defined as $F_{class} = \{inherits, declares\}$, where *inherits* maps a class to the set of classes and interfaces it extends or implements and *declares* maps a class to the set of methods and fields it declares. Finally, let us define a *property* p_e of an element $e_t \in E_t$ as a pair $(f, d = f(e_t))$, where $f \in F_t$ and $d \in \mathcal{P}(E)$.

The properties that the property functions determine when they are applied to each program element at a time will be represented as attributes in the context that will be built. For instance, consider the following source code fragment as input for extracting class roles:

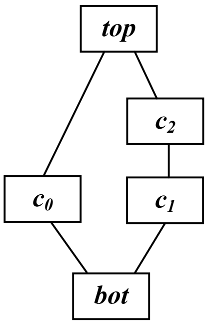
```
class TestCase {
    void setUp() {}
    void tearDown() {}
}

class MoneyTest extends TestCase {
    void setUp() { … }
}

class AccountTest extends TestCase {
    void setUp() { … }
    void tearDown() { … }
}
```

Suppose that the property functions to be considered when producing class roles would include *inherits* and *overrides*, and that they would return the inherited base class of a class and the methods that are overridden in the class, respectively. In addition, let us assume that a special attribute *name* is introduced for those classes that do not have any other attributes determined by the general property functions (in this example *TestCase* does not inherit any classes or override any methods). With these rules for producing attributes with property functions we can form a context for determining the class roles based on the input given above (see table 5.9).

R		*attributes*			
		name TestCase	**inherits TestCase**	**overrides setUp**	**overrides tearDown**
objects	**TestCase**	√			
	MoneyTest		√	√	
	AccountTest		√	√	√

Table 5.9: A context for determining class roles

The analysis of the context given in table 5.9 yields the concept lattice depicted in figure 5.10. Besides the bottom and top concepts, the lattice contains three atomic concepts: c_0, c_1, and c_2. Of these concepts, c_0 represents the base class (called *TestCase*). The other two (c_1 and c_2), on the other hand, reflect the fact that subclasses of *TestCase* should override the *setUp* method and optionally also the *tearDown* method.



top	({*TestCase*, *AccountTest*, *MoneyTest*}, ∅)
c_2	({*MoneyTest*, *AccountTest*}, {*inherits TestCase*, *overrides setUp*})
c_1	({*AccountTest*}, {*inherits TestCase*, *overrides setUp*, *overrides tearDown*})
c_0	({*TestCase*}, {*name TestCase*})
bot	(∅, {*name TestCase*, *inherits TestCase*, *overrides setUp*, *overrides tearDown*})

Figure 5.10: A concept lattice of classes relevant to fixture setup

5.3.2 Relevancy of Program Elements

The example given in table 5.9 and figure 5.10 is unrealistic because the input for analysis was deliberately narrowed down to contain only those program elements that are relevant for setting up fixtures in JUnit. In reality, the input for the pattern extraction process can be as large as the whole implementation code of a framework and a set of example applications. It is clear that the input always contains lots of details that are not relevant to the application developers. Those details must be filtered out from the pattern representation of the framework's reuse interface. We need to pick only relevant elements from the mass of all source code structures or diagram symbols. Furthermore, we must decide which properties of the selected elements should be presented to the framework's users and how.

Without any further modifications the basic outline of the method given above produces one huge pattern that will contain all information in the input source code. In practice this is not a workable solution. To avoid being forced to apply concept analysis to an excessively large context and to enable more precise interpretation of the results, a way to filter out irrelevant input before analysis must be introduced.

The solution is to select only those elements and their properties that are relevant to the hot spot h at hand. No other elements and properties in the input will be considered. In general, the selection criteria are the following:

1. Select as objects only those program elements that have at least one *relevant property*.
2. Select all relevant properties of all selected elements as attributes.
3. If a property value is a set consisting of multiple elements, then introduce an attribute for each member of the set.

In the above definition a relevant property is a property that corresponds to a property value v for which a *relevancy function* $r_h(v) = 1$. In principle, r_h can have any suitable definition. In the following, we will assume that it is a mapping r_h: $E_{Java} \rightarrow \{0, 1\}$ defined by giving a set of program elements $E_h = \{e_1, e_2, ..., e_k\} \subset E_{Java}$ so that

$r_h(v) = 1$, if $v \in E_h$,

$r_h(v) = 0$, otherwise.

The purpose of selection is to produce a relevant context (with respect to a hot spot h) for the concept analysis. This can be done mechanically based on the given relevancy set. To actually get the result described in table 5.9 and figure 5.10 the relevancy set $E_{fixture}$ = {*junit.framework.TestCase*, *junit.framework.TestCase.setUp*, *junit.framework.TestCase.tearDown*} could be used. See appendix B for more examples.

5.3.3 Concept Partitions as Blueprints for Patterns

In our method, the intent of the concept analysis is to produce a pattern that describes a set of program elements so that each relevant element plays exactly one role in the pattern. That is to say, we are looking for a pattern that is *unambiguous* (i.e. each element plays at most one role) and *comprehensive* (i.e. each element plays at least one role).

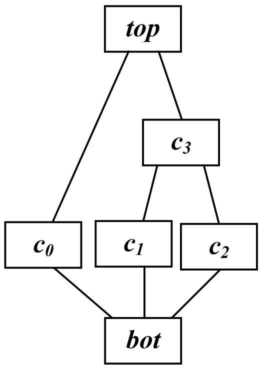
A concept partition is a set of concepts whose extents form a partition of all objects in the given context (recall chapter 4.3.3). In other words, in a concept partition each object takes part in exactly one concept. Thus, a concept partition can be directly translated to an unambiguous and comprehensive pattern by defining a role for each concept.

All partitions of a given context can be formed from its atomic partition. A context must be well-formed in order to have an atomic partition. A non-well-formed context can be transformed to a well-formed context by adding negative information as described in chapter 4.3.3. The context in table 5.9 is not well-formed because two of its atomic concepts (c_1 and c_2) overlap (*AccountTest* belongs to both of them). Table 5.11 represents a new version of the context given in table 5.9, this time with a new attribute *not overrides tearDown*. The resulting concept lattice is depicted in figure 5.12.

R		*attributes*				
		name TestCase	**inherits TestCase**	**overrides setUp**	**overrides tearDown**	**not overrides tearDown**
objects	**TestCase**	√				√
	MoneyTest		√	√		√
	AccountTest		√	√	√	

Table 5.11: A (well-formed) context with negative information

The lattice given in figure 5.12 contains an atomic partition $P_0 = \{c_0, c_1, c_2\}$. It consists of the *TestCase* class itself (c_0) together with the subclasses of *TestCase* with (c_1) and without (c_2) the overriding implementation of the *tearDown* method. In addition to the atomic partition P_0, the lattice has the trivial partition containing only the top element of the lattice and $P_1 = \{c_0, c_3\}$. P_1 represents a view to the input, which omits the optional overriding of *tearDown*: it only differentiates between *TestCase* and its subclasses that override *setUp*.



top	({*TestCase*, *AccountTest*, *MoneyTest*}, ∅)
c_3	({*MoneyTest*, *AccountTest*}, {*inherits TestCase*, *overrides setUp*})
c_2	({*MoneyTest*}, {*inherits TestCase*, *overrides setUp*, *not overrides tearDown*})
c_1	({*AccountTest*}, {*inherits TestCase*, *overrides setUp*, *overrides tearDown*})
c_0	({*TestCase*}, {*name TestCase*, *not overrides tearDown* })
bot	(∅, {*name TestCase*, *inherits TestCase*, *overrides setUp*, *overrides tearDown*, *not overrides tearDown*})

Figure 5.12: A well-formed concept lattice of classes relevant to fixture setup

In general, when there are other (non-trivial) partitions than the atomic partition available, the user must select the partition that most accurately reflects the precision required from the pattern that is to be generated. To avoid user intervention the atomic partition can be automatically chosen by default, although in some case it may contain too many details.

5.3.4 Translating Concepts into Roles

Once a concept partition that describes the relevant portion of the input at the right level of abstraction has been selected, we can translate the concepts in the partition into a pattern. The translation starts with the creation of the roles of the pattern. A new role will be created for each concept in the selected partition.

The translation of a concept C into a role R_C is quite straightforward. The role kind is determined by the element type of the objects included in the extent E_C of C. In other words, we create a class role if the concept describes classes, a method role for methods, and so on.

The role's name and its location in the declaration hierarchy follow directly from the corresponding properties of the originating program elements. If E_C contains multiple elements, it means that R_C will be a generalization of all those elements. In such a case the name of R_C can be coined as a product of the names of the elements in E_C. An alternative strategy is to look for a common prefix or postfix from the names. If such a common part is found, R_C will have that common part as its name. Yet another approach is to derive names for application roles from the names of the corresponding framework roles (e.g., an application role for describing subclasses of the *TestCase* class could be named *UserTestCase* like in figure 5.7).

The declaration location for R_C will be under a role that stands for the declaring elements of the elements in E_C. If R_C stands for multiple elements, they all must be declared inside a Java element that is represented by the same role *R* as the declarers of the other elements are represented by. R_C will then be declared inside *R*.

5.3.5 Decorating Roles with Properties

In chapter 5.3.2 we defined the relevancy of a program element with regards to the hot spot the pattern that is currently being extracted is describing. Next, we will see how the relevant properties of the elements in the extent of a concept are mapped to the corresponding role's constraints and templates.

The constraints and templates for each role R_C are defined based on the intent I_C of the corresponding concept *C*. More precisely, for each attribute *a* in I_C a *property lookup* is performed in R_C. If R_C can contain a constraint (or a template) c_a corresponding to *a* then c_a will be declared for R_C. Both the type (e.g. an *inheritance* constraint) and value (e.g. (*class*, *Base*)) of c_a are determined by the property (e.g. (*inherits*, (*class*, *Base*))) corresponding to *a*. If the value is a reference to an element that corresponds to another role R_D (i.e. belongs to the extent of the associated concept *D*) then a dependency *d* whose target will be R_D is declared in R_C and used as the value for c_a.

The property lookup may fail for some attributes. These attributes can just be ignored. Typically they would include those attributes that will be realized as roles on the next level in the declaration hierarchy. For example, an attribute (*declares*, (*method*, *m*)) denoting that a class defines a method *m* will not be present as a constraint in the class role, but it may imply that a method role will be declared within the class role later on. Note that attributes for which the property lookup fails are not redundant: they serve to classify the objects into separate concepts (extents) just as the other attributes.

The cardinality of R_C is based on the extent E_C. By default the cardinality is exactly one (1). If there is at least one element corresponding to the declaring role *R'* of R_C that does not have a child element in R_C then the cardinality will be from zero to one (?). This is because the input confirms that it is not mandatory for the elements corresponding to *R'* to have a child element corresponding to R_C. For instance, consider a class role *Sub* that describes a set of subclasses of some base class. Some of the subclasses may override a method *m* originally defined in the base class while others retain the default implementation. In such a case, an optional method role to override *m* should be declared for *Sub*.

If there are multiple elements in E_C (either without any declaring elements or with at least some of them declared by the same declaring element) *and* R_C does not refer (either directly or indirectly through its dependencies) to any other role whose cardinality is from zero or one to infinity (+ or *), then the cardinality is set to zero to infinity (*). This rule is based on the observation that, in general, having multiple elements in E_C implies that R_C is a role that describes potentially many implementation structures (e.g. various subclasses of a base class). On the other hand, if it is dependent of another role R_D (based on concept D's extent E_D) that may have multiple elements associated with it, it is possible that the many elements in E_C can be interpreted to be in a "one-for-each" relationship with the elements in E_D. In such a case the correct cardinality is the default, i.e. exactly one (1).

5.3.6 An Example: Extracting a Pattern for Fixture Setup

Using the atomic partition P_0 of the concept lattice given in figure 5.12 (recall chapter 5.3.3) and the translation method described in the previous chapters yields the class roles of the pattern in figure 5.13. There is one class role for each concept in P_0: *TestCase* corresponds to c_0, *AdvancedUserTestCase* to c_1, and *SimpleUserTestCase* to c_2, respectively. The inheritance constraints of *SimpleUserTestCase* and *AdvancedUserTestCase* (and their dependencies to *TestCase*, which describes the base class) have been declared according to the intents of the corresponding concepts c_2 and c_1. The inheritance constraints are the only constraints in the class roles because there are no corresponding constraints for the other attributes (i.e. the property lookup fails for *name TestCase*, *overrides setUp*, *overrides tearDown*, and *not overrides tearDown*). Note that the names of the roles that describe subclasses have been changed from the original ones (*MoneyTest*, *AccountTest*) to better reflect the intent of the roles. Also the cardinalities have been changed from exactly one to multiple; had we used a more realistic input with several subclasses the method would have been able to generate the cardinalities correctly, too.

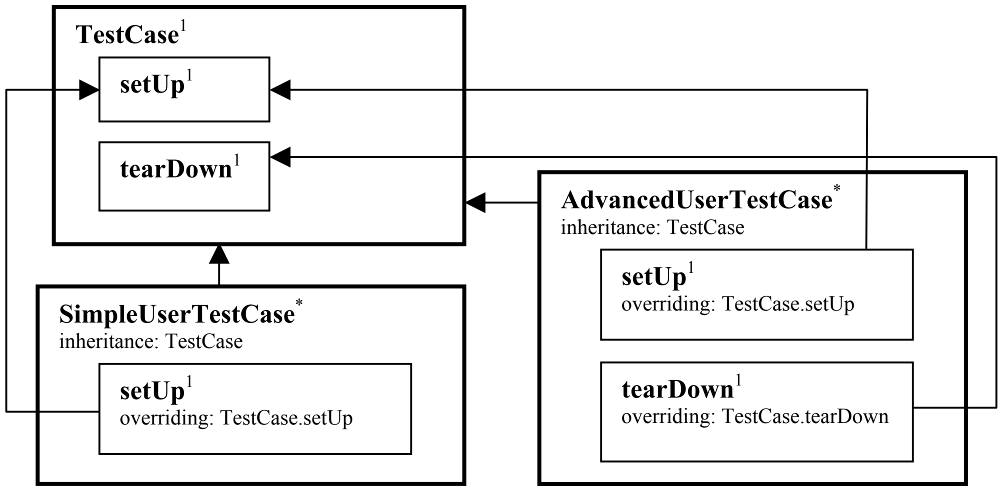
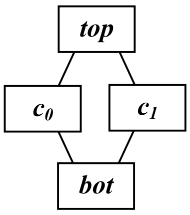


Figure 5.13: An extracted pattern (a simplified pattern for setting up fixtures)

After the class roles have been declared, each of them is considered in turn for determining the method roles within them. The methods declared in the classes in the extent of the corresponding concept of each class role are used to form a new context. For example, the methods declared in *AccountTest* include *setUp* and *tearDown*. There are also other methods in *AccountTest*, but they are not relevant to this pattern. The relevant methods of *AccountTest* together with their properties (*overrides TestCase.setUp* and *overrides TestCase.tearDown*) form a context that yields a simple concept lattice given in figure 5.14. The lattice can be used to determine the method roles to be declared under *AdvancedUserTestCase*.

The contexts and concept lattices for the methods of *TestCase* and *MoneyTest* are similar to the one given in figure 5.14. Also the corresponding method roles can be generated analogously. If the input would contain also information about the actual implementation of the method bodies, we could continue the extraction process by declaring code snippet roles for the method bodies and by declaring field roles for the elements referred to in the bodies. The resulting pattern would then resemble the one given in figure 5.7.



top	({*AccountTest.setUp*, *AccountTest.tearDown*}, ∅)
c_1	({*AccountTest.tearDown*}, {*overrides TestCase.tearDown*})
c_0	({*AccountTest.setUp*}, {*overrides TestCase.setUp*})
bot	(∅, {*overrides TestCase.setUp*, *overrides TestCase.tearDown* })

Figure 5.14: A concept lattice for determining method roles

There is a slight difference in the structure of the patterns (figure 5.7 contains only one class role to represent the subclasses, whereas figure 5.13 has two). The difference stems from the fact that the actual implementation of the pattern extraction method does not consider overriding relationships between methods as attributes while producing class roles. In reality, there are actually three atomic concepts in the concept lattice for determining the class roles for the *SettingUpFixture* pattern: one that describes the *TestCase* class, another for the subclasses, and a third one for the type of the fixture attributes (see appendices A and B for details).

6. Case Study: Annotating JUnit with Fred's Pattern Extractor

In this chapter we will illustrate how to implement the pattern-based framework annotation method described in chapter 5. First, in chapter 6.1 we briefly describe the programming environment into which our pattern extractor is integrated. In chapter 6.2 we give an account of the pattern extractor itself. A case study where we automatically produce specialization patterns for the main hot spots of the JUnit testing framework is represented in chapter 6.3. Finally, in chapter 6.4 we compare the results of automatic pattern extraction to a manually prepared annotation of the same framework and discuss the limitations of the pattern extraction method.

6.1 Fred Framework Engineering Environment

Fred (FRamework EDitor) is a programming environment providing task-driven assistance for framework usage [FRE02, HHK01a, HHK01b]. Fred implements, among other things, the specialization pattern model discussed in chapter 5. Our original motivation was to support specialization of Java frameworks, but it has later turned out that the approach can be used to guide programming according to various kinds of other architectural or coding conventions, too. As an example, we have modeled parts of the JavaBeans architecture as patterns, obtaining thus an environment for JavaBeans programming.

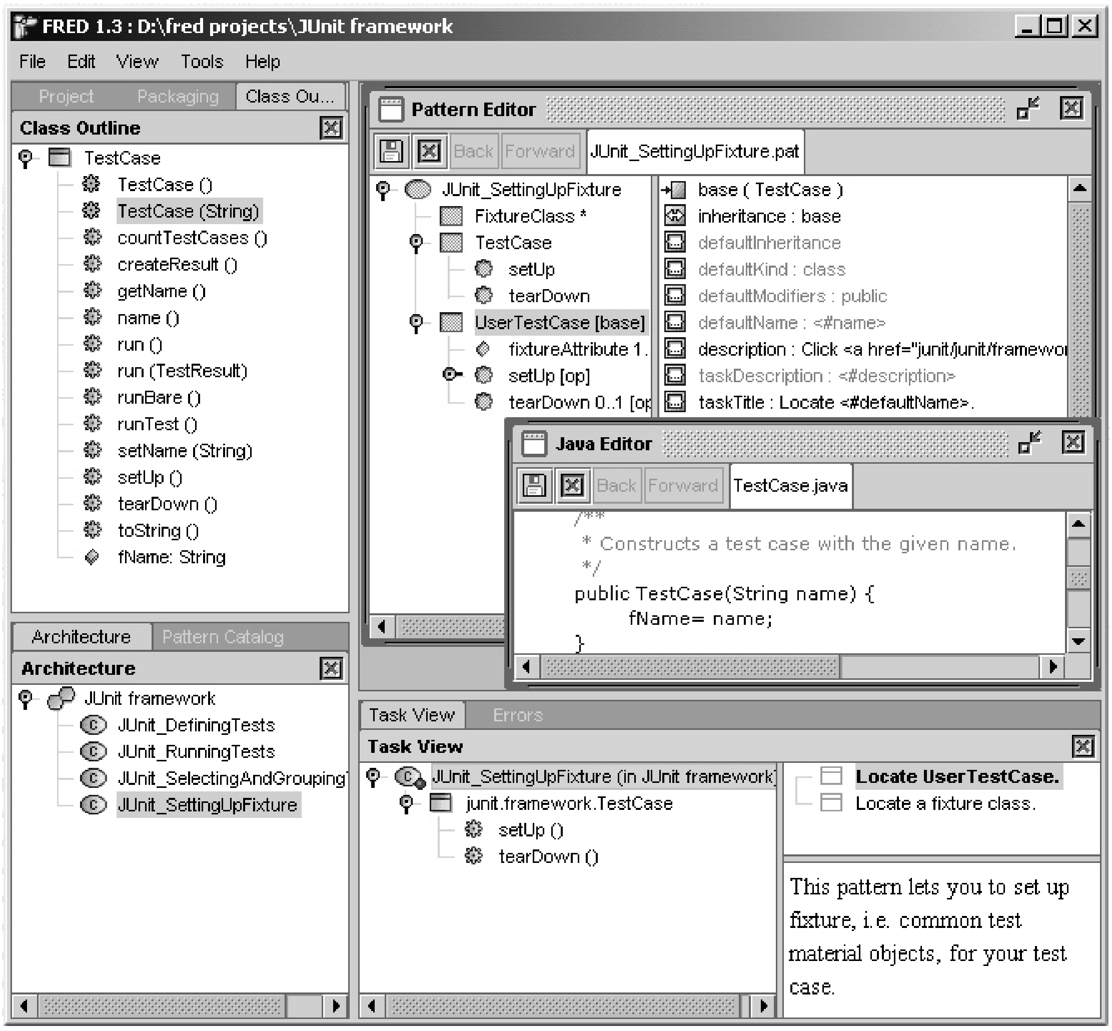
Fred supports both a framework developer in creating the specialization patterns for a framework and an application developer in specializing the framework by following a task list generated from the patterns. It guides the application developer through the task list, dynamically adjusts the list and the accompanying documentation according to the choices made by the developer, and verifies that the syntactic and semantic constraints of the framework are not violated. Fred provides thus an interactive programming environment in which specialization tasks can be executed incrementally in small pieces, allowing the application programmer to generate code, browse and edit the source code, and cancel the tasks if needed.

The user interface of Fred is shown in figure 6.1. It contains a number of views and tools to manage specialization patterns and Java programming projects. The user can organize her workplace freely by selecting the tools she needs and by placing them to tabbed panes or floating on the desktop.

The framework developer uses *Pattern Editor* (in the top right corner in figure 6.1) to create and manage patterns she needs to specify the reuse interface of the framework. Working with *Pattern Editor* involves defining roles and dependencies, associating constraints with them, giving default names and code fragments to be used in code generation, as well as giving help texts and descriptions. In figure 6.1, the user is developing the *SettingUpFixture* specialization pattern. You can see the role declaration hierarchy on the left and the properties of the selected role (*UserTestCase*) on the right (e.g. the dependency to the *TestCase* role and the associated *inheritance* constraint).

In figure 6.1, the framework annotator is almost done with the four patterns (*DefiningTests*, *RunningTests*, *SelectingAndGroupingTests*, and *SettingUpFixture*) she is working on. She has already created a project (*JUnit framework*) where she has instantiated the patterns to test them on JUnit.

Architecture View on the bottom left displays the instantiated patterns in the current project. To see how the *SettingUpFixture* pattern works, the user has selected its instance in the architecture. *Task View* (on bottom right) shows the details of the selected pattern instance, i.e. the tasks together with the cast representing the already bound roles. A description of the selected pattern is shown below the task list.



```
/**
 * Constructs a test case with the given name.
 */
public TestCase(String name) {
     fName= name;
}
```

This pattern lets you to set up fixture, i.e. common test material objects, for your test case.

Figure 6.1: The user interface of Fred

The process of binding framework roles to the corresponding framework elements (while leaving the rest of the roles for the application developer to bind) is called pattern *initialization*. In this case the framework annotator has already bound all the framework roles (the *TestCase* class role with its two method roles *setUp* and *tearDown*). On the other hand, the contracts for the application-specific subclasses are not bound yet, so they are represented as tasks for the user to provide the missing classes ("*Locate UserTestCase*" and "*Locate a fixture class*").

After the framework developer has initialized all her patterns, the framework annotation is ready to be used. An application developer can open the saved framework project as a part of her application project. Fred will then provide a task list for systematically deriving an application from the framework. In general, the tasks can either guide the user in providing program elements to be bound to roles or instruct on how to fix possible constraint violations the already bound elements cause. Some of the tasks are mandatory, while some of them are optional. Also, some tasks are mutually ordered and must be solved in a certain sequence.

Source code for realizing the tasks can be provided in several ways: by coding from scratch, by introducing a binding to a suitable class or method that already exists, or by tailoring a default code snippet generated by Fred. For these, Fred provides a dedicated *Java Editor* (visible in the middle in figure 6.1) that allows the user to edit her source files. *Class Outline* (in the top left corner), in turn, provides an overview of the selected implementation element and a way to navigate in the source code. In addition, the *Packaging* and *Project* views (not visible in the figure) allow the user to browse classes and source files associated with the project.

Java Editor parses the source incrementally enabling interactive constraint checks and accurate insertion of generated code. Changes in the source code are monitored as the user types it in, and possible violations of constraints immediately result in new repairing tasks. Hence, the proper use of the framework is constantly validated and supervised by the system. Besides the standard tools, like *Java Editor* and *Class Outline*, also more high-level (framework-specific) tools can be provided by specializing Fred's general tool framework.

6.2 Pattern Extractor

Pattern Extractor is an implementation of the concept analysis method described in chapter 5.3. Its purpose is to produce Fred specialization patterns from Java source code. It can be used to generate an initial version of a pattern annotation for a framework when given the framework implementation together with a representative set of example applications as input. The user can then refine the patterns to get a complete and fully functional annotation.

The UML class diagram in figure 6.2 illustrates the structure of *Pattern Extractor* and shows how the different modules are connected to the rest of the Fred environment. The modules identified in figure 6.2 implement the elementary phases of pattern extraction depicted in the data flow diagram in figure 5.8.

The *PatternExtractor* class represents the whole user interface of the tool. When the user has entered the input packages and relevancy criteria she wishes to use in the extraction, the *PatternExtractor* class creates a new *Translator* object and passes that information to it.

Translator acts as the coordinator of the pattern extraction process. It uses *ContextBuilder* to select relevant Java source code elements from the input (*AST*) and to produce a concept analysis context from them (phase 1 in figure 5.8). The context is handed over to the concept analysis subsystem that implements the general concept analysis algorithms, such as making the given context well-formed, building a concept lattice for it, and calculating the concept partitions (phase 2 in figure 5.8). These algorithms are not dependent on Java or Fred's pattern model.

When the context has been analyzed and an atomic concept partition has been formed, *Translator* uses the pattern model to create roles that correspond to the concepts in the partition (phase 4 in figure 5.8).

It also decorates the roles with properties that reflect the intents of the concepts. The current implementation always uses the atomic partition as the basis for role generation, so there is no need for explicit partition selection (phase 3 in figure 5.8).

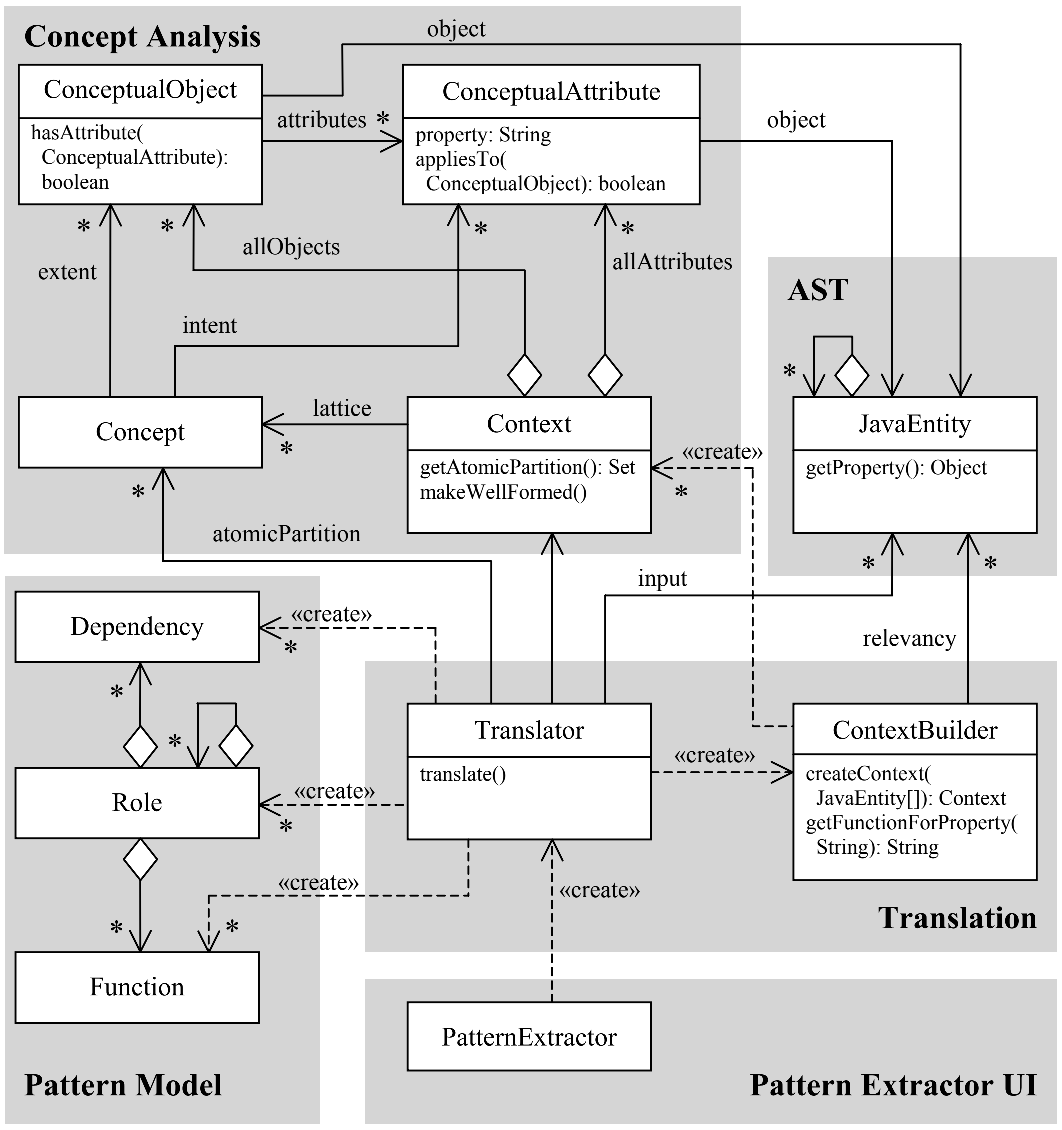


Figure 6.2: Pattern Extractor design as a UML class diagram

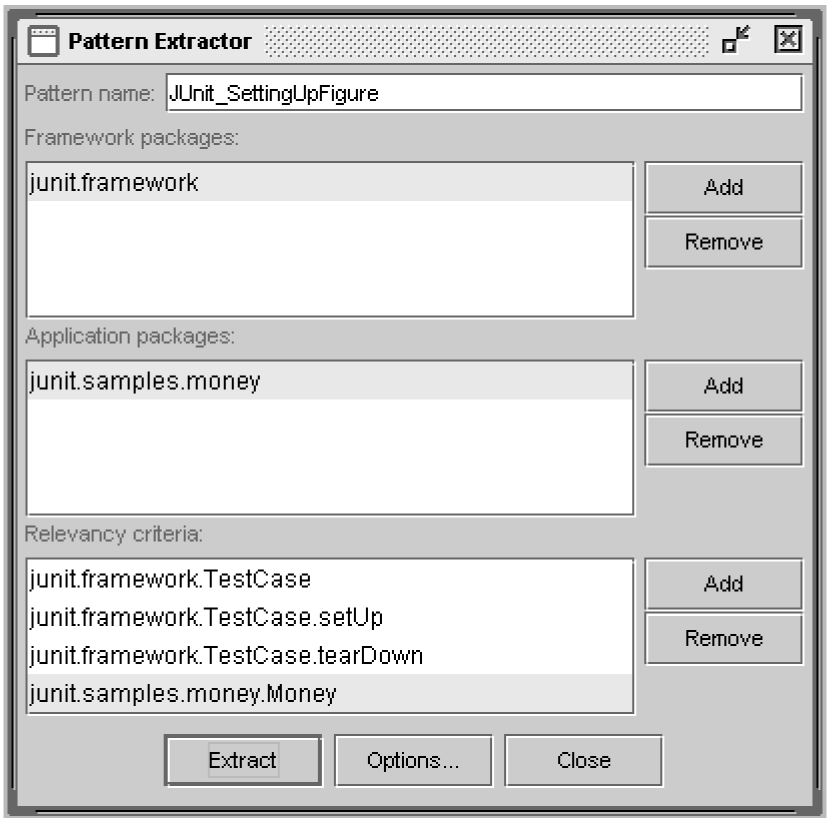
In the following we take a look at *Pattern Extractor*'s user interface and give an overview of how roles are extracted from the input. Then we continue with more detailed discussions on Fred's pattern model, context building, and translation of concepts to roles, respectively.

6.2.1 Pattern Extractor User Interface

The user interface of *Pattern Extractor* is depicted in figure 6.3. It consists of a text field for the name that is given for the pattern to be extracted and three lists for giving the input and the relevancy criteria for the extraction process (recall chapter 5.3). There is also an option dialog for giving settings that will customize the extraction process.

The input is given simply by naming Java packages (collections of classes) that will be treated either as parts of the framework that the pattern is supposed to annotate or as belonging to the set of example applications. The contents of the given input packages are searched from the source code and class files of the currently open project. Also the current class path will be searched.

The input is filtered according to the given relevancy criteria, which are defined in the form of a list of program elements. The list should contain the full names of those elements that are considered to be relevant for the pattern to be generated. If a name ends with an asterisk, then all elements whose names start with the prefix preceding the asterisk will be considered relevant.



Pattern name: JUnit_SettingUpFigure

Framework packages:
junit.framework

Application packages:
junit.samples.money

Relevancy criteria:
junit.framework.TestCase
junit.framework.TestCase.setUp
junit.framework.TestCase.tearDown
junit.samples.money.Money

Figure 6.3: Pattern Extractor user interface

On the whole, the user interface of *Pattern Extractor* is very simple and intuitive. It is quite straightforward to identify the correct input. Getting the relevancy settings right for each pattern may require some experimentation, though. It is crucial to find a balance between giving too small or too large relevancy sets. The risk with the former extreme is that while a small relevancy set usually results in a simple pattern, it may be too general to provide enough assistance for the application developer. On the other hand, giving a large relevancy set yields patterns with more roles and constraints, which means more detailed guidance for the application developer. However, at the same time the risk of including irrelevant and confusing information to the extracted pattern increases.

Restricting and selecting the input for the concept analysis affects directly the quality of the results of the analysis. In principle, the information needed to come up with suitable relevancy criteria should be available in documentation or at least in the implementation code of the framework. Our experiences with the JUnit framework suggest that, in practice, the specification of the relevancy criteria does not stipulate in-depth knowledge of the framework source code and thus is not a serious obstacle in using

Pattern Extractor. However, it is clear that further research is needed to find systematic methods for selecting the right input for each pattern to be extracted.

6.2.2 An Overview of Extracting Roles from Input

Figure 6.4 provides a generic sequence diagram representing the main phases of the iterative role extraction process. *PatternExtractor* is responsible for creating an instance of *Translator* to translate the given input (an AST consisting of *JavaEntity* objects) to a pattern using the given relevancy criteria (a subset of input). The actual extraction process is invoked when *PatternExtractor* calls *translate* on *Translator*. *Translator* then calls *extractRoles* on itself by giving the target role (i.e. the pattern itself) and the source (i.e. classes in the input) for the extraction.

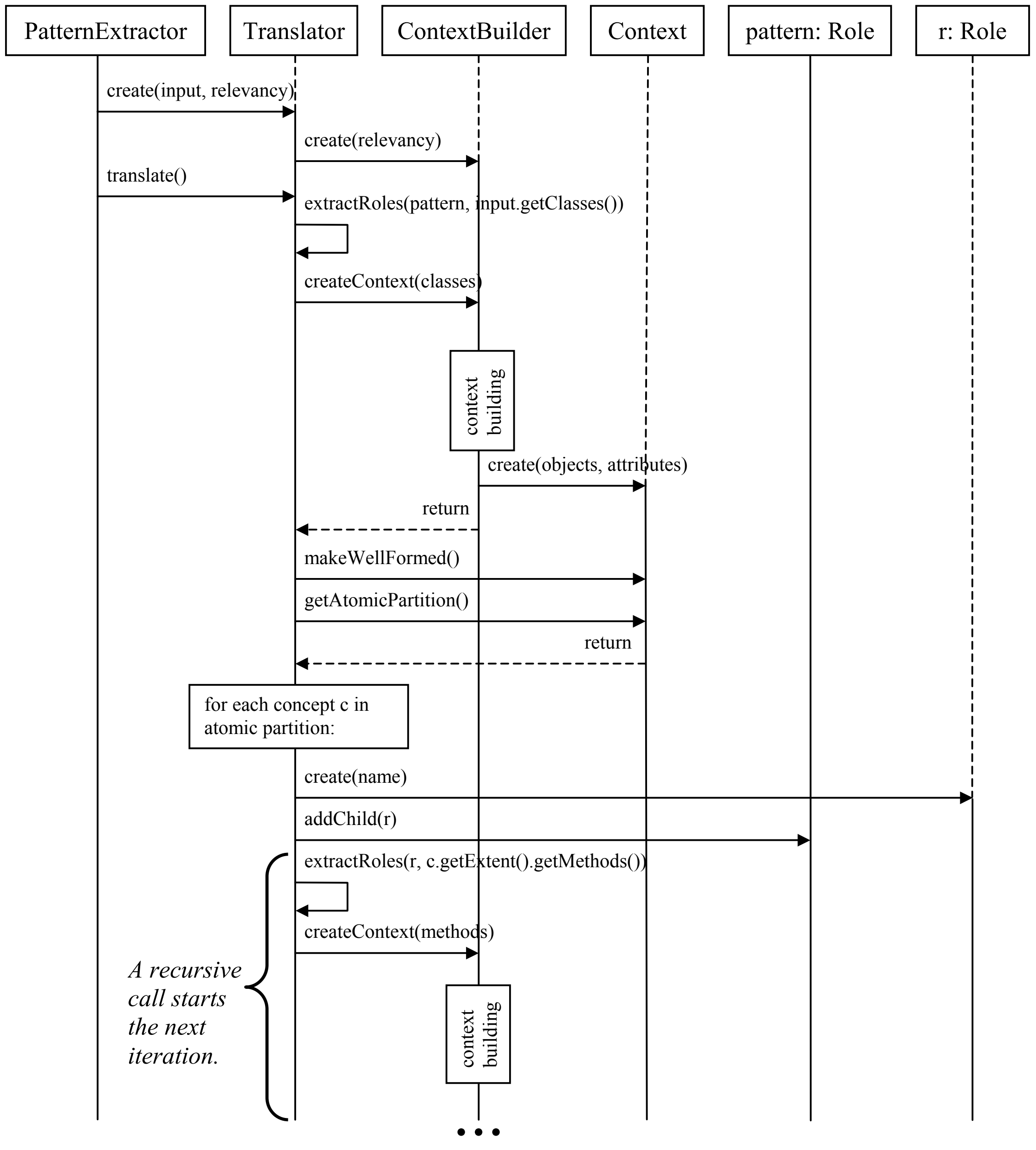


Figure 6.4: Role extraction process

Translator delegates the building of concept analysis context to *ContextBuilder*, which determines the objects and attributes from the given classes and their properties. Then *ContextBuilder* creates the context object and returns it to the *Translator*. *Translator* makes sure that the context is well-formed and then asks for the atomic partition of the context lattice.

The roles are generated from the concepts in the atomic partition. When all concepts have been processed, *Translator* calls recursively *extractRoles* with each already created role as the target, in turn. The program elements are gathered for each *extractRoles* call from the extent of the corresponding concept. These recursive calls start the next iteration in which method roles will be declared under the already declared class roles. The recursion continues similarly when code snippet roles are declared under the method roles to represent the method bodies in the third iteration.

6.2.3 Fred's Pattern Model

Pattern Extractor uses Fred's pattern model to actually create pattern structures that are the output of the extraction process. There are three main classes in Fred's pattern model implementation: *Role*, *Dependency*, and *Function*. Roles can have other roles declared within them, and all patterns consist hierarchically of roles. Each role has a set of functions. These functions represent the various properties that can be associated with the role (see table 6.5).

Constraint/template	Role kind	Purpose
inheritance	Class	The enforced inheritance relationship.
overriding	Method	The enforced overriding relationship.
returnType	Method	The enforced return type for the method.
type	Parameter, exception, field	The enforced type.
defaultName	All role kinds	The name used for the language structure when generating code from the role description.
description	All role kinds	The description shown for the task related to the role.
taskTitle	All role kinds	The title shown for the task related to the role.
taskDescription	All role kinds	The description shown for the task related to the role.
defaultInheritance	Class	The inherited (or extended) class (or interface) used when generating code from the role description.
defaultKind	Class	When creating code from the role, determines whether the generated structure will be a class or an interface.
defaultModifiers	Class, method, constructor, parameter, field	The Java modifiers used when generating code.
default-Implementation	Method, constructor	The operation body used when generating code.
defaultReturnType	Method	The return type used when generating code.
defaultInitializer	Field role	The initializer used for a field when generating code from the role.
defaultType	Parameter, field	The type used when generating code from the role.
parameterNumber	Parameter	The position of the parameter in the parameter list.
source	Code snippet	The text fragment used when generating code.
tagString	Code snippet	The tag that determines the location where to insert the code.

Table 6.5: Properties for each role kind [Vil01]

Table 6.5 lists all available properties together with the role kinds they apply to and their purpose. The first four of them (*inheritance*, *overriding*, *returnType*, *type*) are constraints, the rest are templates. Constraints are checked whenever the associated implementation elements change. A violation of a constraint results in a notification in Fred UI. Templates, on the other hand, are used for generating code and dynamic user documentation.

In addition to the properties described in table 6.5, a role always has a cardinality constraint associated with it. If the role has no dependencies then the cardinality expresses the number of program elements that must be bound to the role for each element that is bound to the declaring role.

Dependencies can be declared for roles to model relationships between implementation elements that are bound to them. The target of a dependency is either a role or another dependency visible in the name scope of the dependency's declaring role. Also, a path notation (like in Java) can be used to refer inward in the name scope hierarchy in the target expressions.

Dependencies affect the interpretation of the cardinality constraint. If a role contains dependencies, the cardinality constraint specifies the number of implementation elements there must be bound to the role for each combination of elements playing the target roles of the dependencies. Dependencies are also used for referring to other roles in constraints and templates. For example, a class role *Sub* might have dependency *b* to another class role *Base*. Then an inheritance constraint referring to *b* would mean that any class bound to *Sub* must inherit a specific class playing the role *Base*.

6.2.4 Building Contexts from Java Source Code Elements

The *ContextBuilder* class is responsible for turning the given input (an AST representation of a set of Java program elements given as an argument to the *createContext* method) into a context consisting of a set of context analysis objects (instances of *ConceptualObject*) and attributes (instances of *ConceptualAttribute*) in the light of the given relevancy criteria (initialized when constructing the context builder). Typically, the relevancy criteria would be a subset of input consisting of one or more classes (or methods) that are the key concepts in the hot spot that the pattern to be extracted is intended to describe.

All Java program elements are represented in figure 6.2 by their base class *JavaEntity*. Similarly, the generic *getProperty* method stands for all possible ways of querying various properties of different elements. In practice, of course, there are dozens of accessor methods for those properties. *ContextBuilder* implements the mapping (i.e. the property lookup) between those properties and the constraints and templates of the roles (represented by the *Function* objects) in the *getFunctionForProperty* method. This mapping is used for determining relevant objects and their attributes as well as for declaring correct constraints and templates based on the intents of the concepts. See table 6.6 for details on what elements and properties are checked for each kind of role.

ContextBuilder creates a concept analysis object from each relevant element in the input. An element is relevant if either it is itself included in the given relevancy criteria set or it has a property whose value is included in the relevancy set, i.e. it has at least one relevant property. All properties of all elements in the input are considered when making the attribute set for the context. If an element has a property that has a relevant value, then a concept analysis attribute is created for that property. In some cases a property function may yield a value that is actually a set (e.g. the implemented interfaces of a class). In such a case an attribute is created for each element in the set.

Role kind	**Elements to select as concept analysis objects**	**Property functions determining the attributes**	**Constraints and templates to extract**
Class role	classes and interfaces	*getSuperClass*	*inheritance*
		getInterfaces	*inheritance*
		getDeclaredMethods	*none*
Method role	methods of classes and interfaces corresponding to the role under which the method role will be declared	*getOverriddenMethods*	*overriding*
		getReturnType	*returnType*
Code snippet role	methods bodies corresponding to the role under which the code snippet role will be declared	*getUsedTypes*	*source*
Parameter role	formal parameters of methods corresponding to the method role under which the parameter role will be declared	*getType*	*type*
Exception role	exception declarations of methods corresponding to the method role under which the exception role will be declared	*getType*	*type*
Field role	fields of classes corresponding to the role under which the field role will be declared	*getType*	*type*
Constructor role	constructors of classes corresponding to the role under which the constructor role will be declared	*getParameterTypes*	*none*

Table 6.6: Selecting objects and attributes

Note that some properties are only queried for making distinctions in the concept analysis, but they do not affect role property creation. For example, the declared methods of a class make a difference when forming concepts of classes, but they do not contribute to any class role property. Instead, they are used when forming new input for the next iteration, which will eventually produce method roles from concepts whose extents contain methods.

6.2.5 Creating and Decorating Roles

When a context has been built, it is analyzed to produce a corresponding concept lattice. If the lattice does not have an atomic partition, then negative information needs to be added until the context becomes well-formed, and an atomic partition can be formed (recall chapter 4.3.3).

The atomic partition (a set of *Concept* objects) is the basis for the role creation: a new role is created for each concept in the partition. *Translator* stores references to all created roles and their corresponding concepts so that they can be used later on when decorating roles with properties.

When creating a new role, the role kind is determined based on the types (or kinds) of the program elements in the concept's extent (actually the first object in the extent determines the kind, since all objects in the extent are assumed to be of the same type). The name of the role is based on the names of all objects in the extent. If there is a common leading or trailing part for all elements in the extent, then that is used as the role name, otherwise the role name is simply the catenation of the names of the objects in the extent.

The role creation process is repeated recursively for each created new role such that first a new input is formed from the Java elements declared under the elements in the extent of the already created role. A new context is created using those elements as input. The roles resulting from the following concept analysis and translation are added under the role created already in the previous iteration. Note that the root role is the pattern itself, which is created before the whole extraction process begins.

After all roles have been created they will be decorated with properties (constraints and templates). All properties are represented with *Function* objects as described above. The decoration process basically enumerates through each role *r*, finds the concept *c* and the intent corresponding to the role, and then considers each attribute *a* in the intent one at a time (see figure 6.7).

If there exists a non-empty mapping from the element property determined by the attribute to a role property (i.e. if *getFunctionForProperty* returns a valid function name) then a new corresponding *Function* object *f* is created. More precisely, a function (*sF*) is fetched from the superclass of *r* (i.e. *super*) using the function name returned by *getFunctionForProperty*. Then that function is overridden in *r*, which will result in the actual creation of the correct function *f*.

The value for the function *f* is set by calling *setExpression* on it. The value is based on the value component of the attribute. If the value is a plain string then it is used as it is. If, on the other hand, the value refers to a Java element for which a role has been created, then a dependency is declared for the role currently under decoration, and that dependency is used as a value for the function. The role corresponding to the value element will be the target of the dependency. The name for the dependency is defined as an abbreviation of the target role's name.

If the role that is being decorated corresponds to a concept with extent *E*, then the cardinality for the role will be either:

- *from zero to infinity* (*) if there exist at least two elements with the same declaring class (or without a declaring class) in *E*, *and* if the role does not have a dependency that either refers to a role whose cardinality is multiple (at most value is > 1) or refers to a role that (recursively) has such dependencies;
- *from zero to one* (?) if there are less elements in *E* than in the extent of the concept corresponding to the declaring role, which implies that there is at least one element corresponding to the declaring role that does not have a child element corresponding to the given role; or
- *exactly one* (1) in all other cases.

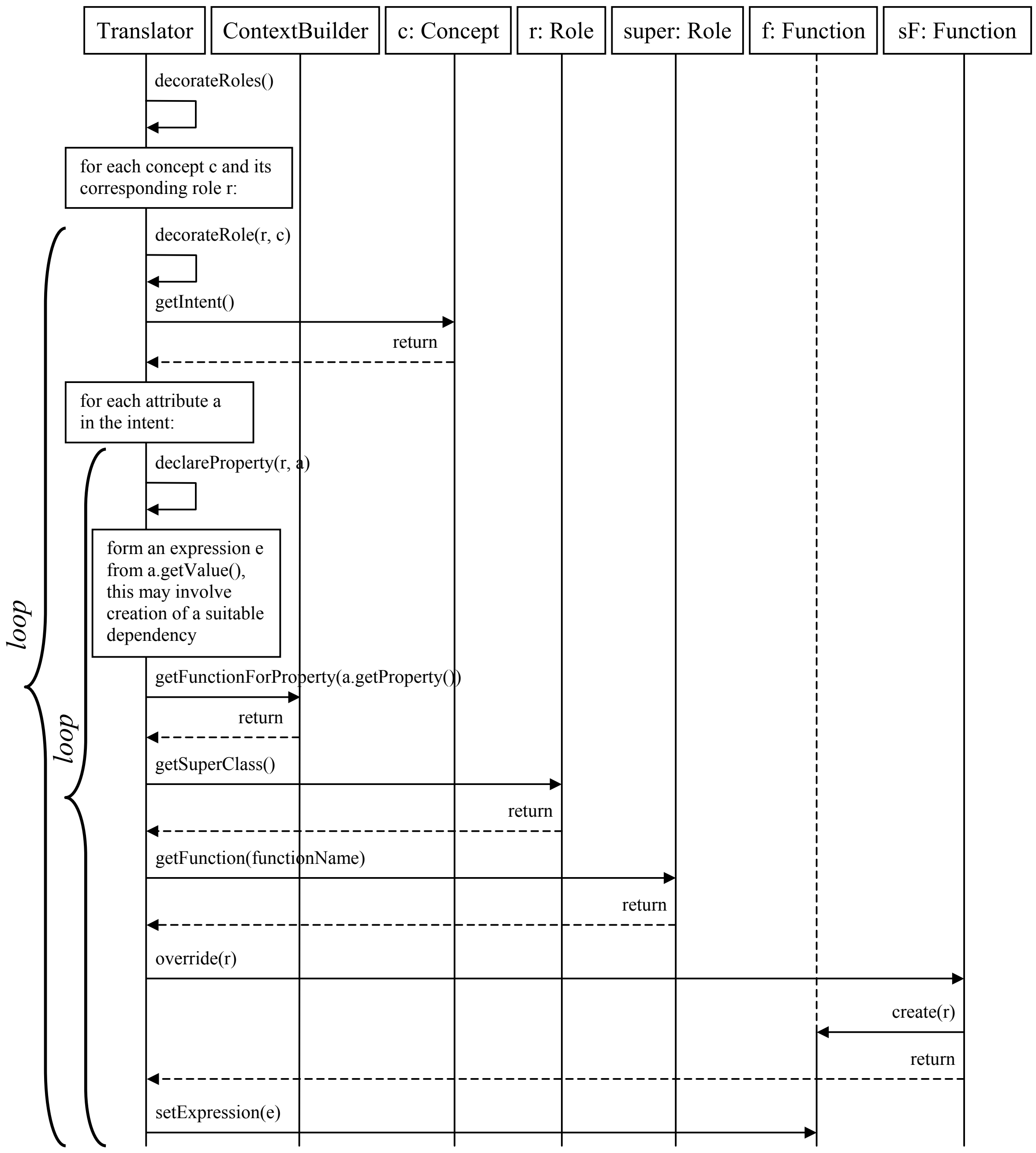

Translator
ContextBuilder
c: Concept
r: Role
super: Role
f: Function
sF: Function
decorateRoles()
for each concept c and its corresponding role r:
decorateRole(r, c)
getIntent()
return
for each attribute a in the intent:
declareProperty(r, a)
form an expression e from a.getValue(), this may involve creation of a suitable dependency
getFunctionForProperty(a.getProperty())
return
getSuperClass()
return
getFunction(functionName)
return
override(r)
create(r)
return
setExpression(e)
loop
loop

Figure 6.7: Decorating roles with Function objects (i.e. constraints and templates)

6.3 Extracting Specialization Patterns for JUnit

The *Pattern Extractor* tool has been developed and evaluated in the context of a case study where we have annotated the JUnit framework [GaB99, JUn02] for Fred. In total, eight specialization patterns were extracted to annotate the reuse interface of JUnit. Four of those patterns can be characterized as basic patterns, i.e. patterns that are involved in any specialization of JUnit. These patterns correspond to the hot spots introduced in chapter 5.1.2. Other patterns describe hot spots that are relevant to more advanced or infrequent ways of using the framework.

The input packages given for all the extractions were *junit.framework* (framework) and *junit.samples.money* (application). In addition, *junit.extensions* was used as a framework input package for the advanced patterns and *junit.runner* and *junit.textui* were used as framework and application input for the *RunningTests* pattern. See appendix B for used relevancy criteria for each pattern. The patterns themselves are given as simplified pattern diagrams in appendix A and as complete textual specifications in appendix B. In the textual representations, those parts of the specifications that were added or modified after the automatic extraction phase are highlighted so as to make it easy to assess the effectiveness of the automation.

The extracted and adjusted annotation was used to produce a test set for some of the example classes provided with JUnit in the *junit.samples.money* package. The source code of the test set is given in appendix D. Over 70 percent of the source code was automatically generated. The overall usability of the pattern annotation after adjustments was comparable with the hand-written annotations made for various frameworks within the Fred project (see e.g. [Vil01]). The assistance that the extracted annotation provided for specializing JUnit was as extensive, as detailed, and as controlled as with the manually prepared annotations.

Table 6.8 summarizes the results of the case study. It shows the characteristics of each produced specialization pattern including the kind of the pattern (basic or advanced) and the number of non-empty lines in the textual pattern specification (a measure for the pattern size). The quality of the extraction process is measured by giving the number of non-empty lines containing parts that were removed from the pattern when adjusting it, the number of lines containing parts that were added or modified, and the number of those adjusted lines that contained actual functional modifications (as opposed to mere renaming of roles or adjustments to the descriptions, task titles, or comments). A manually prepared annotation for JUnit that was designed according to the recommendations and guidelines given in [Vil01] was used as a reference.

Specialization pattern	**Kind**	**Produced lines**	**Lines to remove**	**Added or modified lines**	**Lines with functional changes**	**Strict extraction %**	**Loose extraction %**
DefiningTests	Basic	114	3	86	47	22	58
SettingUpFixture	Basic	63	7	20	4	57	93
SelectingAnd GroupingTests	Basic	122	21	75	35	21	65
RunningTests	Basic	54	14	20	6	37	85
DecoratingTests	Adv.	103	3	54	29	45	71
Running RepeatedTests	Adv.	47	3	21	8	49	82
RunningTests InThreads	Adv.	61	15	28	12	30	74
Testing Exceptions	Adv.	69	5	42	21	32	67
Total	-	633	71	346	162	-	-
Average		79	9	43	20	37	74

Table 6.8: Characteristics of the extracted JUnit patterns

The measures in table 6.8 were used to calculate two ratios of automatically produced specification lines to the whole line count in the textual representation of the pattern. The *strict extraction percentage* is the ratio of the automatically produced lines (minus removed, added, and modified lines) to the total number of lines. The *loose extraction percentage* is the ratio of the automatically produced lines minus removed lines and lines containing functionally relevant modifications to the total number of lines minus removed lines.

The strict extraction percentage is a measure that considers all removals, additions, and modifications to be equally burdening for the framework annotator. The loose extraction percentage, on the other hand, omits the removals altogether and considers only non-trivial modifications as burdens for the framework annotator. The strict and loose extraction percentages define thus a range in which the real effective net benefit of the automatic extraction probably lies. Since the average extraction percentage is between 37 and 74 for the extracted patterns, it can be concluded that roughly half of the annotations can be automatically produced.

Usually the parts of the extracted annotations that need most adjustments are code snippets. Even though the framework annotator is forced to modify almost all code snippets by hand, the automatically extracted versions give her generally a good starting point. For example, in the *SettingUpFixture* pattern the produced *fixtureCreation* code snippet initially contained all fixture object initialization statements extracted from the *setUp* method of *MoneyTest* with the names of the created object's classes replaced with a reference to the corresponding class role (*FixtureClass*) and with field names replaced with references to the corresponding field role (*fix*). So the only modification actually needed in the *source* property of the snippet was to remove all initialization statements except for the one that was kept as a representative example.

In some cases also the dependencies of the produced snippet needed to be changed (or new ones had to be added). In *fixtureCreation*, for instance, the separate dependencies to the fixture attribute and to its type were replaced with a single dependency to the fixture attribute because the type of the attribute can be reached through that same dependency. Yet another example of a straightforward modification to the extracted code snippets would be the definition of multiple snippets to represent alternative implementation solutions for a whole method body or for a part of it. In general, the framework annotator can be expected to find many such repeating patterns in code snippets that lead her to make appropriate modifications to them quite easily.

6.4 Discussion and Future Work

In chapter 4.2.4 we listed problems related to design recovery. A recovery method that is based on matching design model templates to the source code representation in order to find instances of such models will inevitably suffer from the library selection problem. In other words it is difficult to prepare a library of model templates that would be applicable in all cases and it is equally laborious to define a custom library for each system to be analyzed.

Our solution is based on a concept analysis algorithm that does not need a model template library. On the other hand, our method requires the user to define relevancy criteria, which focus the analysis on the interesting parts of the system. The difference is that the relevant program elements can be easily listed based on the high-level documentation of the system (or by quickly browsing the source code to find the roots of the inheritance chains and other candidates for the main concepts of the system if documentation is unavailable), whereas preparing a custom model template library for a system requires detailed knowledge of the implementation of the system. Indeed, it can be argued that to guarantee successful recovery of all instances of all (even application-specific) design models one has to know all those instances in the implementation beforehand.

Another obstacle in design recovery is the scalability of the algorithms being used. The worst-case running time of the concept lattice construction algorithm used in this work is exponential [Sne96]. In practice, however, the running time is polynomial and thus feasible even for rather large contexts because there typically is $O(n)$ or $O(n^2)$ concepts in a lattice with n objects and n attributes as opposed to the maximal $O(2^n)$ concepts.

Also Tonelli and Antoniol use concept analysis to recover (design) patterns from source code [ToA99]. They define all possible *combinations* of classes as objects. This means that Tonella and Antoniol are forced to limit their analysis to class combinations with fewer than three or four classes when using any non-trivial system as input. In our approach we define one entity (a class, a method, and so on) as an object. This means that that the size of the patterns that can be extracted and the size of the systems used as input are not limited. Furthermore, focusing the analysis only on relevant elements and doing the analysis iteratively ensure that our method scales well to meet all practical requirements.

In chapter 4.2.4 we noted that when instances of patterns or other design models are recovered from source code one after another, it might be difficult to see the relationships between the recovered patterns. This isolation of recovered patterns remains a problem also in our method. This is mostly due to the fact that Fred's pattern model does not currently support inter-pattern references. The only way of expressing dependencies between patterns is to repeat common roles in patterns and to use descriptions and documentation to highlight the connections.

Even though Fred is mainly targeted at describing the static structure of systems and enforcing constraints that can be statically checked, it can also be used to provide user guidance for frameworks that rely heavily on dynamic mechanisms, such as reflection. On the source code level, various coding conventions can be defined as code snippets. For example, the *SelectingAndGroupingTests* hot spot of JUnit (recall chapter 5.2.1) uses reflection for identifying the test scripts to be executed. With our method we were able to extract code snippets that accurately capture the requirements for the successful application of that hot spot.

Automatic extraction of patterns that annotate framework hot spots reduces the routine work associated with documenting frameworks. It becomes economically viable to annotate also frameworks that are not widely used. Furthermore, it may become affordable to annotate even frameworks that are still evolving substantially. If framework annotation can also happen during framework development, it can provide valuable insight by making the hot spots and the whole framework usage process explicit. This may uncover hidden deficiencies in framework architecture early in the development. Of course, maintaining pattern annotations for an evolving framework development means additional effort, but if annotation is cheap (because of the automation) then this problem could be overcome.

The results of our case study show that about half of the specialization pattern code for annotating the reuse interface of a framework can be automatically extracted from source code. Of the other half that needs to be manually given or at least modified to suit the context, most consists of code snippets. This results from the fact that analyzing general patterns from method bodies (expressions and statements) is generally a hard task. In addition, Fred currently does not support method implementations (bodies) in its AST representation of the source code. That is why the extraction of the code snippets is based directly on the tokenized source text of the method bodies, which allows only very primitive analyses.

Our results show that automatic extraction of specialization patterns yields quite accurate overall picture of the structures of the patterns, and that it is indeed only the details that need further modifications in order to make the patterns usable. In addition to the code snippets, majority of the additions and modifications were related to textual templates (descriptions, default names, task titles) and renaming of roles (to give more general and more descriptive names for the application roles). We think that it would be possible to reduce the need for that kind of modifications by increasing the configurability of the extraction tool. Examples of such enhancements would probably involve at least better utilization of formal comments in the input to produce more verbose role descriptions, introducing parameterized algorithms for generating appropriate names for roles, and full source code analysis of method bodies to enable more sophisticated extraction of code snippets.

Structuring a software system reflects design decisions that are inherently subjective [SiR98]. Similarly, there is always a creative element in preparing documentation or an annotation for a reusable system. This means that substantial user interaction will be required in any approach that tries to assist, e.g., annotating a framework reuse interface. In *Pattern Extractor*, the user is responsible for structuring the annotation into separate specialization patterns by specifying appropriate relevancy criteria. The tool selects relevant program elements according to the criteria and uses the selected elements as input for the concept analysis. The analysis, in turn, produces a concept lattice that is translated to a pattern describing the input.

Even though it is unlikely that annotating framework reuse interfaces can ever be fully automated, there is still a lot of room for improvements in *Pattern Extractor*. Further research is needed to build a precise model of how the selection of program elements and their properties (input and relevancy criteria) affect the produced specialization patterns. Based on a detailed model it would be easier to make justified decisions on which properties of the program elements should be used as concept analysis attributes and how the input should be divided into separate contexts to produce structured annotations.

The pattern extraction method described in this thesis must be applied to a variety of frameworks in order to truly validate its potential and especially to enhance the tool implementation. As our experiences accumulate we expect to be able to formulate practical and generally applicable guidelines for structuring the input and defining the relevancy criteria to easily produce useful pattern annotations. In our view, finding the main concepts of a framework from its documentation is the key to understanding the framework's reuse interface. At the same time it is also an essential precondition of using *Pattern Extractor* to successfully produce specialization patterns to annotate the reuse interface. If no documentation is available the same information must be gathered from the system's source code. Systematic analysis of the framework's class hierarchy as described in [Vil01] and automatic pattern detection (see, e.g., [Bro96] or [FGM01]) are good ways to get started. However, there clearly is a need for further research on automatic program analysis methods, especially for methods that concentrate on analysis of reusable assets, such as object-oriented application frameworks and reusable software components.

7. Conclusion

In this thesis we have presented a new tool-supported approach for annotating reuse interfaces of Java frameworks. Our original motivation was to investigate advanced and automatic ways to import existing code into the Fred framework engineering environment for systematic management. We soon realized that the most laborious task involved in preparing a framework for Fred is the construction of a suitable pattern set. With the pattern extraction tool presented in this thesis many trivial details of the framework annotation process can be automated. Using the tool, most roles, dependencies, constraints, and default values can be accurately deduced from the framework source code and existing example applications.

We began the introduction to this thesis by arguing that reuse, especially the reuse of high-level design artifacts and architectures, is the key to success in software engineering. The benefits of reuse include, e.g., increased productivity and product quality as well as decreased time-to-market.

Currently, object-oriented application frameworks represent the state of the art in reusing both functionality and architecture of software systems. The basic characteristics of frameworks were introduced in chapter 2. We saw how frameworks can be designed by stepwise generalization from a set of representative example applications. We also discussed white-box reuse and black-box reuse in the context of framework evolution. As an example of a typical way to organize a framework's implementation we represented a hierarchy of three layers: interface layer, core implementation layer, and default component layer. Then we looked at framework specialization. We gave an overview of the framework-based software development process, and showed an example of how the application developer can use the constructs of the framework in practice.

From the application developer's point of view the framework's reuse interface is its most important aspect. The problems and solutions related to framework specialization were discussed in detail in chapter 3. We used the example-based and hierarchy-based approaches to teaching frameworks as a starting point for our account on the framework usability. Then we went on to represent framework cookbooks, hooks, and patterns as especially suitable methods to document frameworks. Finally, we discussed formal ways to annotate the reuse interface of a framework in order to provide tool support for framework usage.

The absence or low quality of documentation is a major obstacle for a software engineer that tries to reuse existing resources. The documentation is especially critical for an application developer that tries to specialize a framework. This is because frameworks are generally large, complex, and abstract software systems. In chapter 4 we discussed framework design recovery as a means to complement or substitute outdated framework documentation. We started by motivating traditional reverse engineering with architectural erosion, which is the reason why the implemented architecture of a software system is seldom consistent with its documented architecture. We also gave examples of ways to represent the properties of the recovered architecture, including software metrics and visualization. Then we discussed the general properties of program comprehension tools. Special attention was paid to the source code analysis for capturing instances of design structures from implementation. In particular, concept analysis was introduced as a method that allows detecting recurring structures in source code without requiring a predefined template library that would be matched against the code.

In chapter 5 we introduced our pattern-based formalism for annotating the reuse interface of frameworks. We also showed how a framework annotation could be extracted from the source code of the framework and a set of example applications using an adaptation of the concept analysis method. The main phases of the method are the construction of a concept analysis context from the given input (in the light of the given relevancy criteria), the analysis of the context to produce a concept lattice, and the translation of the concepts in the lattice to a pattern where a role corresponds to a concept and constraints and templates of the role correspond to the intent of the concept.

In chapter 6 we gave an account of the Fred framework engineering environment that implements our pattern model, and the pattern extraction tool that implements the pattern extraction method. The pattern extraction tool was applied to the JUnit testing framework in a case study. We managed to extract a significant portion of the reuse interface annotation directly from the source code of the framework and the samples that were delivered with the framework.

Defining the input for *Pattern Extractor* is quite uncomplicated, even based on limited documentation. For example, the extraction of the specialization patterns for JUnit required relevancy criteria that were easy to gather from the brief overview given in [GaB99] and summarized in chapter 5.1.1. The extraction process produced precise and detailed information about the roles, restrictions, and dependencies related to the hot spots of the framework. This information was not explicit in the documentation or even in the comments of the implementation. The resulting patterns needed some modifications before they could be taken into use. However, even the parts of the patterns that had to be adjusted contained useful information, and the adjustments were mostly straightforward to implement.

Based on our experiences we can say that automatic extraction of specialization patterns makes framework annotation faster and allows for making annotations even for frameworks that are still under development. Automatic extraction can also increase quality of the produced annotations, because it may reveal shortcomings, omissions, or inconsistencies in manually prepared annotations made for the same framework.

References

ACL96 Alencar P., Cowan C., Lucena C., A Formal Approach to Architectural Design Patterns. In: Proc. 3rd *International Symposium of Formal Methods Europe (FME'96)*, Oxford, England, March 1996, 576-594.

Ale79 Alexander C., *The Timeless Way of Building*. Oxford University Press, 1979.

Arn92 Arnold R., *Software Reengineering*. IEEE Press, 1992.

ASU86 Aho A., Sethi R., Ullman J., *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, 1986.

Bau99 Bauer M., Metrics. In: Ducasse S. (ed.), The FAMOOS Object-Oriented Reengineering Handbook. FAMOOS report, ESPRIT project no. 21975, 1999, 22-30.

BBM96 Basili V., Briand L., Melo W., How Reuse Influences Productivity in Object-Oriented Systems. *Communications of the ACM* 39, 10, 1996, 104-116.

Bec00 Beck K., *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

Bec88 Beck K., Using a Pattern Language for Programming (A Part of the Summary of Discussions from OOPSLA'87's Methodologies & OOP Workshop). In: Addendum to the Proc. OOPSLA'87, *ACM SIGPLAN Notices* 23, 5, 1988, 16.

Bec97 Beck K., *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

BeJ94 Beck K., Johnson R., Patterns Generate Architectures. In: Proc. *European Conference on Object-Oriented Programming (ECOOP'94)*, Bologna, Italy, July 1994, Springer LNCS 821, 139-149.

BeP99 Beckenkamp F., Pree W., Neural Network Components. In: *Implementing Application Frameworks — Object-Oriented Frameworks at Work* (Fayad M., Schmidt D., Johnson R. eds.), Wiley, 1999, 95-112.

BHB99 Bowman I., Holt R., Brewster N., Linux as a Case Study: Its Extracted Architecture. In: Proc. *21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, California, USA, May 1999, IEEE Computer Society Press, 555-563.

BMM98 Brown W., Malveau R., McCormick H., Mowbray T., *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.

BMM99 Bosch J., Molin P., Mattsson M., Bengtsson P., Fayad M., Framework Problems and Experiences. In: *Building Application Frameworks* (Fayad M., Schmidt D., Johnson R. eds.), Wiley, 1999, 55-82.

BMR96 Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., *A System of Patterns — Pattern-Oriented Software Architecture*. Wiley, 1996.

Boo96 Booch G., *Object Solutions: Managing the Object-Oriented Project*. Addison-Wesley, 1996.

BoR96 Booch G., Rumbaugh J., Unified Method for Object-Oriented Development. Technical Report, Rational Software Corporation, 1996.

Bro96 Brown K., Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk. Technical Report TR-96-07, University of Illinois at Urbana-Champaign, 1996.

Bus96 Buschmann F., Reflection. In: Vlissides J., Coplien J., Kerth N. (eds.), *Pattern Languages of Program Design 2*, Addison-Wesley, 1996, 271-294.

CaL01 Cardone R., Lin C., Comparing Frameworks and Layered Refinement. In: Proc. *23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Canada, May 2001, IEEE Computer Society Press, 285-294.

CDS97 Codenie W., De Hondt K., Steyaert P., Vercammen A., From Custom Applications to Domain-Specific Frameworks. *Communications of the ACM* 40, 10, 1997, 71-77.

ChC96 Chikofsky E., Cross II J., Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software* 7, 1, 1996, 13-17.

CiN99 Cinnéide O., Nixon P., A Methodology for the Automated Introduction of Design Patterns. In: Proc. *International Conference on Software Maintenance (ICSM'99)*, Oxford, England, August-September 1999.

CIR93 Campbell R., Islam N., Raila D., Madany P., Designing and Implementing Choices: An Object-Oriented System In C++. *Communications of the ACM* 36, 9, 1993, 117-126.

Ciu99 Ciupke O., Grouping. In: Ducasse S. (ed.), The FAMOOS Object-Oriented Reengineering Handbook. FAMOOS report, ESPRIT project no. 21975, 1999, 81-89.

Cli96 Cline M., The Pros and Cons of Adopting and Applying Design Patterns in the Real World. *Communications of the ACM* 39, 10, 1996, 47-59.

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

CoP95 Cotter S., Potel M., *Inside Taligent Technology*. Addison-Wesley, 1995.

CoS95 Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*. Addison-Wesley, 1995.

Cun95 Cunningham W., The CHECKS Pattern Language of Information Integrity. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 293-320.

Dem98 Demeyer S., Analysis of Overridden Methods to Infer Hot Spots. In: Proc. *European Conference on Object-Oriented Programming (ECOOP'98) Workshops, Demos, and Posters (Workshop Reader)*, Brussels, Belgium, July 1998, Springer LNCS 1543, 66-67.

Deu89 Deutsch L., Design Reuse and Frameworks in the Smalltalk-80 System. In: Biggerstaff T., Perlis A. (eds.), *Software Reusability Vol. II*, ACM Press 1989, 57-71.

DiM01 Ding L., Medvidovic N., A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution. In: Proc. *Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, Amsterdam, Netherlands, August 2001, 191-200.

DóK99 Dósa Rácz F., Koskimies K., Tool-Supported Compression of UML Class Diagrams. In: Proc. *«UML»'99 — The Unified Modeling Language: Beyond the Standard (Second International Conference)*, Fort Collins, CO, USA, October 1999, Springer LNCS 1723, 172-187.

Duc99 Ducasse S. (ed.), The FAMOOS Object-Oriented Reengineering Handbook. FAMOOS report, ESPRIT project no. 21975, 1999.

DuJ96 Durham A., Johnson R., A Framework for Run-time Systems and Its Visual Programming Language. In: Proc. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, San Jose, California, USA, October 1996, ACM SIGPLAN Notices 31, 10, 1996, 406-420.

EgG92 Eggenschwiler T., Gamma E., The ET++ Swaps Manager: Using Object Technology in the Financial Engineering Domain. In: Proc. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'92)*, Vancouver, British Columbia, Canada, October 1992, ACM SIGPLAN Notices 27, 10, 1992, 166-178.

EGK01 Eick S., Graves T., Karr A., Marron J., Mockus A., Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Transactions on Software Engineering* 27, 1, 2001, 1-12.

EHL99 Eden A., Hirshfeld Y., Lundqvist K., LePUS — Symbolic Logic Modeling of Object Oriented Architectures: A Case Study. In: Proc. *Second Nordic Workshop on Software Architecture (NOSA'99)*, University of Karlskrona/ Ronneby, Ronneby, Sweden, 1999.

FaJ00 Fayad M., Johnson R. (eds.), *Domain-Specific Application Frameworks — Frameworks Experience by Industry*. Wiley, 2000.

FAM02 The FAMOOS homepage. http://iamwww.unibe.ch/~famoos/, 2002.

FaS97 Fayad M, Schmidt D., Object-Oriented Application Frameworks. *Communications of the ACM* 40, 10, 1997, 32-38.

FGM01 Ferenc R., Gustafsson J., Müller L., Paakki J., Recognizing Design Patterns in C++ Programs with the Integration of Columbus and MAISA. In: *Proc. 7th Symposium on Programming Languages and Software Tools (SPLST 2001)*, Szeged, Hungary, June 2001, 58-70.

FHL97 Froehlich G., Hoover H., Liu L., Sorenson P., Hooking into Object-Oriented Application Frameworks. In: Proc. *19th International Conference on Software Engineering* (*ICSE'97)*, Boston, Massachusetts, USA, May 1997, IEEE Computer Society Press, 491-501.

FHL99 Froehlich G., Hoover J., Liu L., Sorenson P., Reusing Hooks. In: *Building Application Frameworks* (Fayad M., Schmidt D., Johnson R. eds.), Wiley, 1999, 219-236.

FMW97 Florijn G., Meijers M., van Winsen P., Tool Support for Object-Oriented Patterns. In: Proc. *European Conference on Object-Oriented Programming (ECOOP'97)*, Jyväskylä, Finland, 1997, Springer LNCS 1241, 472-496.

FoO95 Foote B., Opdyke W., Lifecycle and Refactoring Patterns That Support Evolution and Reuse. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 239-258.

FOW87 Ferrant J., Ottenstein K., Warren J., The Program Dependence Graph and Its Use in Optimization. *ACM Transaction on Programming Languages and Systems* 3, 9, 1987, 319-349.

Fow99 Fowler M. et al., *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

FPR00 Fontoura M., Pree W., Rumpe B., UML-F: A Modeling Language for Object-Oriented Frameworks. In: Proc. *European Conference on Object-Oriented Programming (ECOOP 2000)*, Sophia Antipolis and Cannes, France, June 2000, Springer LNCS 1850, 63-83.

FRE02 Fred Site. http://practise.cs.tut.fi/fred, 2002.

FSJ99a Fayad M., Schmidt D., Johnson R. (eds.), *Building Application Frameworks — Object-Oriented Foundations of Framework Design*. Wiley, 1999.

FSJ99b Fayad M., Schmidt D., Johnson R. (eds.), *Implementing Application Frameworks — Object-Oriented Frameworks at Work*. Wiley, 1999.

GaB99 Gamma E., Beck K., JUnit: A Cook's Tour. *Java Report* 4, 5, 1999, 27-38.

GaM95 Gangopadhyay D., Mitra S., Understanding Frameworks by Exploration of Exemplars. In: Proc. *Seventh International Workshop on CASE (CASE'95)*, Toronto, Canada, July 1995, 90-99.

GHJ95 Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

GoR89 Goldberg A., Robson D., *Smalltalk-80: The Language*. Addison-Wesley, 1989.

GPN02 Gustafsson J., Paakki J., Nenonen L., Verkamo I., Architecture-Centric Software Evolution by Software Metrics and Design Patterns. In: Proc. *6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, Budapest, Hungary, March 2002, 108-115.

GuB01 van Gurp J., Bosch J., Design, Implementation, and Evolution of Object Oriented Frameworks: Concepts and Guidelines. *Software — Practice & Experience* 31, 3, 2001, 277-300.

HaN90 Harandi M., Ning J., Knowledge-Based Program Analysis. *IEEE Software* 7, 1, 1990, 74-81.

Hau02 Hautamäki J., Task-Driven Framework Specialization — Goal-Oriented Approach. Licentiate thesis, Department of Computer and Information Sciences, University of Tampere, 2002.

HaY85 Harandi M., Young F., Template Based Specification and Design. In: Proc. *3rd International Workshop on Software Specifications and Design*, London 1985, 94-97.

HFR00 Harrison N., Foote B., Rohnert H., *Pattern Languages of Program Design 4*. Addison-Wesley, 2000.

HHG90 Helm R., Holland I., Gangopadhyay D., Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. *ACM SIGPLAN Notices* 25, 10, 1990, 169-180.

HHK01a Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J., Generating Application Development Environments for Java Frameworks. In: Proc. *3rd International Conference on Generative and Component-Based Software Engineering (GCSE 2001)*, Erfurt, Germany, September 2001, Springer LNCS 2186, 163-176.

HHK01b Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J., Annotating Reusable Software Architectures with Specialization Patterns. In: Proc. *Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, Amsterdam, Netherlands, August 2001, 171-180.

HJE95 Hüni H., Johnson R., Engel R., A Framework for Network Protocol Software. In: Proc. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'95)*, Austin, Texas, USA, October 1995, ACM SIGPLAN Notices 30, 10, 1995, 358-369.

Hol93 Holland I., The Design and Representation of Object-Oriented Components. Ph.D. thesis, Northeastern University, 1993.

HRY95 Harris D., Reubenstein H., Yeh A., Reverse Engineering to the Architectural Level. In: Proc. *17th International Conference on Software Engineering (ICSE'95)*, Seattle, Washington, USA, April 1995, IEEE Computer Society Press, 186-195.

JCJ92 Jacobson I., Christerson M., Jonsson P., Övergaard G., *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

JKN95 Järnvall E., Koskimies K., Niittymäki M., Object-Oriented Language Engineering with TaLE. *Object-Oriented Systems* 2, 1995, 77-98.

JoF88 Johnson R., Foote B., Designing Reusable Classes. *Journal of Object-Oriented Programming (JOOP)* 1, 2, 1988, 22-35.

Joh92 Johnson R., Documenting Frameworks Using Patterns. In: Proc. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'92)*, Vancouver, British Columbia, Canada, October 1992, ACM SIGPLAN Notices 27, 10, 1992, 63-76.

Jol99 Jolin A., Usability and Framework Design. In: *Building Application Frameworks* (Fayad M., Schmidt D., Johnson R. eds.), Wiley, 1999, 153-162.

JUn02 JUnit: Testing Resources for Extreme Programming (JUnit Home Page). http://www.junit.org, 2002.

Kel99 Keller R., Schauer R., Robitaille S., Pagé P., Pattern-Based Reverse-Engineering of Design Components. In: Proc. *21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, California, USA, May 1999, IEEE Computer Society Press, 226-235.

Ker95 Kerth N., Caterpillar's Fate: A Pattern Language for the Transformation from Analysis to Design. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 293-320.

KiB96 Kim J., Benner K., An Experience Using Design Patterns: Lessons Learned and Tool Support. *Theory and Practice of Object Systems (TAPOS)* 2, 1, 1996, 61-74.

KNS92 Kozaczynski W., Ning J., Sarver T., Program Concept Recognition. In: Proc. *Seventh Knowledge-Based Software Engineering Conference (KBSE'92)*, McLean, Virginia, USA, September 1992, 216-225.

KoM95 Koskimies K., Mössenböck H., Designing a Framework by Stepwise Generalization. In: Proc. *5th European Software Engineering Conference (ESEC'95)*, Sitges, Spain, September 1995, Springer LNCS 989, 479-498.

KoM96 Koskimies K., Mössenböck H., Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. In: Proc. *18th International Conference on Software Engineering (ICSE'96)*, Berlin, Germany, March 1996, IEEE Computer Society Press 1996, 366-375.

Kot96 Kotula J., Discovering Patterns: An Industry Report. *Software — Practice and Experience* 26, 11, 1996, 1261-1276.

KrP88 Krasner G., Pope S., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming (JOOP)* 1, 3, 1988, 26-49.

KrS94 Krone M., Snelting G., On the Inference of Configuration Structures from Source Code. In: Proc. *16th International Conference on Software Engineering (ICSE'94)*, Sorrento, Italy, May 1994, IEEE Computer Society Press, 49-57.

Kru95 Kruchten P., The 4+1 View Model of Architecture. *IEEE Software* 12, 6, 1995, 42-50.

LaK94 Lajoi R., Keller R., Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert. ftp://ftp.iro.umontreal.ca/pub/gelo/Publications/Papers/concert.acfas94.ps.Z, 1994.

LaN95 Lange D., Nakamura Y., Interactive Visualization of Design Patterns Can Help in Framework Understanding. In: Proc. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'95)*, Austin, Texas, USA, October 1995, ACM SIGPLAN Notices 30, 10, 1995, 342-357.

Lea94 Lea D., Christopher Alexander: An Introduction for Object-Oriented Designers. *ACM SIGSOFT Software Engineering Notes* 19, 1, 1994, 39-46.

Lea96 Lea D., *Concurrent Programming in Java — Design Principles and Patterns*. Addison-Wesley, 1996.

LiS97 Lindig C., Snelting G., Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. In: Proc. *19th International Conference on Software Engineering (ICSE'97)*, Boston, Massachusetts, USA, May 1997, IEEE Computer Society Press, 349-359.

Mac77 Mackworth A., Consistency in Network of Relations. *Artificial Intelligence* 8, 1, 1977, 99-118.

Mac92 Mackworth A., The Logic of Constraint Satisfaction. *Artificial Intelligence* 58, 1-3, 1992, 3-20.

Mar95 Martin R., Discovering Patterns in Existing Applications. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 365-394.

MaS99 Mamrak S., Sinha S., A Case Study: Productivity and Quality Gains Using an Object-Oriented Framework. *Software — Practice & Experience* 29, 6, 1999, 501-518.

MBF99 Mattson M., Bosch J., Fayad M., Framework Integration Problems, Causes, Solutions. *Communications of the ACM*, 42, 10, 1999, 81-87.

MDE97 Meijler T., Demeyer S., Engel R., Making Design Patterns Explicit in FACE — A Framework Adaptive Composition Environment. In: Proc. *European Software Engineering Conference (ESEC/FSE'97)*, Zurich, Switzerland, 1997, Springer LNCS 1301, 94-110.

Men97 Menzies T., Object-Oriented Patterns: Lessons from Expert Systems. *Software — Practice & Experience* 27, 12, 1997, 1457-1478.

Mey92 Meyer B., *Eiffel: The Language*. Prentice Hall, 1992.

Mik98 Mikkonen T., Formalizing Design Patterns. In: Proc. *20th International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998, IEEE Computer Society Press, 115-124.

MNL96 Murphy G., Notkin D., Lan E., An Empirical Study of Static Call Graph Extractors. In Proc. *18th International Conference on Software Engineering (ICSE'96)*, Berlin, Germany, March 1996, IEEE Computer Society Press, 90-99.

MoN96 Moser S., Nierstrasz O., The Effect of Object-Oriented Frameworks on Developer Productivity. *IEEE Computer* 29, 9, 1996, 45-51.

MRB97 Martin R., Riehle D., Buschmann F. (eds.), *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

MRT98 McLellan S., Roesler A., Tempest J., Spinuzzi C., Building More Usable APIs. *IEEE Software* 15, 3, 1998, 78-86.

Nem97 Nemirovsky A., Building Object-Oriented Frameworks. A Taligent white paper, http://www.ibm.com/developerworks/library/oobuilding/index.html, 1997.

NiP99 Nikander P., Pärssinen J., A JavaBeans Framework for Cryptographic Protocols. In: *Implementing Application Frameworks — Object-Oriented Frameworks at Work* (Fayad M., Schmidt D., Johnson R. eds.), Wiley, 1999, 555-587.

OCS00 Ortigosa A., Campo M., Salomon R., Towards Agent-Oriented Assistance for Framework Instantiation. In: Proc. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*, Minneapolis, Minnesota, USA, October 2000, ACM SIGPLAN Notices 35, 10, 2000, 253-263.

Opd92 Opdyke W., Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign, ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z, 1992.

OrC99 Ortigosa A., Campo M., SmartBooks: A Step Beyond Active-Cookbooks to Aid in Framework Instantiation. In: Proc. *Technology of Object-Oriented Languages and Systems (TOOLS'99 Europe)*, Nancy, France, May 1999, IEEE Press, 131-140.

PeM99 Perrochon L., Mann W., Inferred Designs. *IEEE Software* 16, 5, 1999, 46-51.

PKG00 Paakki J., Karhinen A., Gustafsson J., Nenonen L., Verkamo I., Software Metrics by Architectural Pattern Mining. In: *Proc. International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, Beijing, China, August 2000, 325-332.

PPS95 Pree W., Pomberger G., Schappert A., Sommerlad P., Active Guidance of Framework Development. *Software — Concepts and Tools* 16, 3, 1995, 136-145.

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

Pre99 Pree W., Hot-Spot-Driven Development. In: *Building Application Frameworks* (Fayad M., Schmidt D., Johnson R. eds.), Wiley, 1999.

PSK96 Paakki J., Salminen A., Koskinen J., Automated Hypertext Support for Software Maintenance. *The Computer Journal* 39, 7, 1996, 577-597.

PVR99 Potok T., Vouk M., Rindos A., Productivity Analysis of Object-Oriented Software Development in Commercial Environment. *Software — Practice & Experience* 29, 10, 1999, 833-847.

Qui94 Quilici A., A Memory-Based Approach to Recognizing Programming Plans. Communications of the ACM 37, 5, 1994, 84--93.

Qui95 Quilici A., Reverse Engineering of Legacy Systems: A Path Toward Success. In: Proc. *17th International Conference on Software Engineering (ICSE'95)*, Seattle, Washington, USA, April 1995, IEEE Computer Society Press, 333-336.

RaL89 Raj R., Levy H., A Compositional Model for Software Reuse. In: Proc. *European Conference on Object-Oriented Programming (ECOOP'89)*, Nottingham, UK, July 1989, Cambridge University Press, 1989, 3-25.

Ran96 Ran A., MOODS: Models for Object-Oriented Design of State. In: Vlissides J., Coplien J., Kerth N. (eds.), *Pattern Languages of Program Design 2*, Addison-Wesley, 1996, 119-142.

RBJ97 Roberts D., Brant J., Johnson R., A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)* 3, 4, 1997, 253-263.

RBP91 Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

Rie00 Riehle R., Framework Design — A Role Modeling Approach. Ph.D. thesis, ETH Zürich, Institute of Computer Systems, 2000.

RiG98 Riehle D., Gross T., Role Model Based Framework Design and Integration. In: Proc. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, Vancouver, British Columbia, Canada, October 1998, ACM SIGPLAN Notices 33, 10, 1998, 117-133.

RiN00 Rine D., Nada N., Three Empirical Studies of a Software Reuse Reference Model. *Software — Practice and Experience* 30, 6, 2000, 685-722.

RiW88 Rich C., Waters R., The Programmer's Apprentice: a Research Overview. *IEEE Computer* 21, 11, 1988, 11-24.

RiZ95 Riehle D., Züllighoven H., A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 9-42.

RJB99 Rumbaugh J., Jacobson I., Booch G., *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

SaN01 Santelices R., Nussbaum, M., A Framework for the Development of Videogames. *Software — Practice & Experience* 31, 11, 2001, 1091-1107.

SBF96 Sparks S., Benner K., Faris C., Managing Object-Oriented Framework Reuse. *IEEE Computer* 29, 4, 1996, 52-62.

Sch95 Schmidt H., Creating the Architecture of a Manufacturing Framework by Design Patterns. In: Proc. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'95)*, Austin, Texas, USA, October 1995, ACM SIGPLAN Notices 30, 10, 1995, 370-384.

SeG98 Seemann J., von Gudenberg J., Pattern-Based Design Recovery of Java Software. In Proc. *International Symposium on Foundations of Software Engineering (FSE'98)*, Florida, USA, November 1998, 10-16.

Sha96 Shaw M., Some Patterns for Software Architectures. In: Vlissides J., Coplien J., Kerth N. (eds.), *Pattern Languages of Program Design 2*, Addison-Wesley, 1996, 255-269.

SiR97 Siff M., Reps T., Identifying Modules via Concept Analysis. In: Proc. *International Conference on Software Maintenance (ICSM'97)*, Bari, Italy, October 1997, 170-178.

SiR98 Siff M., Reps T., Identifying Modules via Concept Analysis. TR-1337, Computer Sciences Department, University of Wisconsin, Madison, WI, 1998.

SJF96 Schmidt D., Johnson R., Fayad M. (eds.), Special Issue on Patterns and Pattern Languages. *Communications of the ACM* 39, 10, 1996.

SLB00 Shull F., Lanubile F., Basili V., Investigating Reading Techniques for Object-Oriented Framework Learning. *IEEE Transactions on Software Engineering* 26, 11, 2000, 1101-1118.

Sne96 Snelting G., Reengineering of Configurations Based on Mathematical Concept Analysis. *ACM Transactions on Software Engineering and Methodology* 5, 2, 1996, 146-189.

Sne98 Sneed H., Human Cognition of Complex Thought Patterns — How Much Is Our Perception of the Present Determined by Our Experience of the Past. In: Proc. *6th International Workshop on Program Comprehension (IWPC'98)*, Ischia, Italy, June,1998.

SRM99 Schauer R., Robitaille S., Martel F., Keller R., Hot Spot Recovery in Object-Oriented Software with Inheritance and Composition Template Methods. In: Proc. *International Conference on Software Maintenance 1999 (ICSM'99)*, Oxford, England, August-September 1999, IEEE Press, 220-229.

SSC96a Sefika M., Sane A., Campbell R., Architecture-Oriented Visualization. In: Proc. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, San Jose, California, USA, October 1996, ACM SIGPLAN Notices 31, 10, 1996, 389-405.

SSC96b Sefika M., Sane A., Campbell R., Monitoring Compliance of a Software System with Its High-Level Design Models. In: Proc. *18th International Conference on Software Engineering (ICSE'96)*, Berlin, Germany, March 1996, IEEE Computer Society Press, 387-396.

Sun02a Sun Microsystems, Inc., The Source for Java™ Technology. http://java.sun.com, 2002.

Sun02b Sun Microsystems, Inc., Java Foundation Classes. http://java.sun.com/products/jfc/, 2002.

Sun02c Sun Microsystems, Inc., JavaBeans™. http://java.sun.com/products/javabeans/, 2002.

Sun02d Sun Microsystems, Inc., Javadoc Tool Homepage. http://java.sun.com/j2se/javadoc/, 2002.

Szy97 Szyperski C., *Component Software — Beyond Object Oriented Programming*. Addison-Wesley, 1997.

TAF00 Tonella P., Antoniol G., Fiutem R., Calzolari F., Reverse Engineering 4.7 Million Lines of Code. *Software — Practice & Experience* 30, 2, 2000, 129-150.

ToA99 Tonella P., Antoniol G., Object Oriented Design Pattern Inference. In: Proc. *International Conference on Software Maintenance (ICSM'99)*, Oxford, England, August-September 1999, IEEE Computer Society Press, 1999, 230-239.

VCK96 Vlissides J., Coplien J., Kerth N. (eds.), *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

Vil01 Viljamaa A., Pattern-Based Framework Annotation and Adaptation — A Systematic Approach. Licentiate thesis, Report C-2001-52, University of Helsinki, Department of Computer Science, 2001.

Vil95 Viljamaa P., The Patterns Business: Impressions from PLoP-94. *ACM SIGSOFT Software Engineering Notes* 20, 1, 1995, 74-78.

Vli96 Vlissides J., Protection, Part I: The Hollywood Principle. *C++ Report* 8, 2, 1996, 14-19.

VlL90 Vlissides J., Linton M., Unidraw: A Framework for Building Domain-Specific Graphical Editors. *ACM Transactions on Information Systems* 8, 3, 1990, 237-268.

WaB00 Watt D., Brown D., *Programming Language Processors in Java — Compilers and Interpreters*. Prentice Hall, Pearson Education Limited, 2000.

WaC94 Waters R., Chikofsky E. (eds.), Special Section on Reverse Engineering. *Communications of the ACM* 37, 5, 1994, 22-93.

Wei84 Weiser M., Program Slicing. *IEEE Transactions on Software Engineering* 10, 4, 1984, 352-357.

WoL95 Wolf K., Liu C., New Clients with Old Servers: A Pattern Language for Client/Server Frameworks. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 51-64.

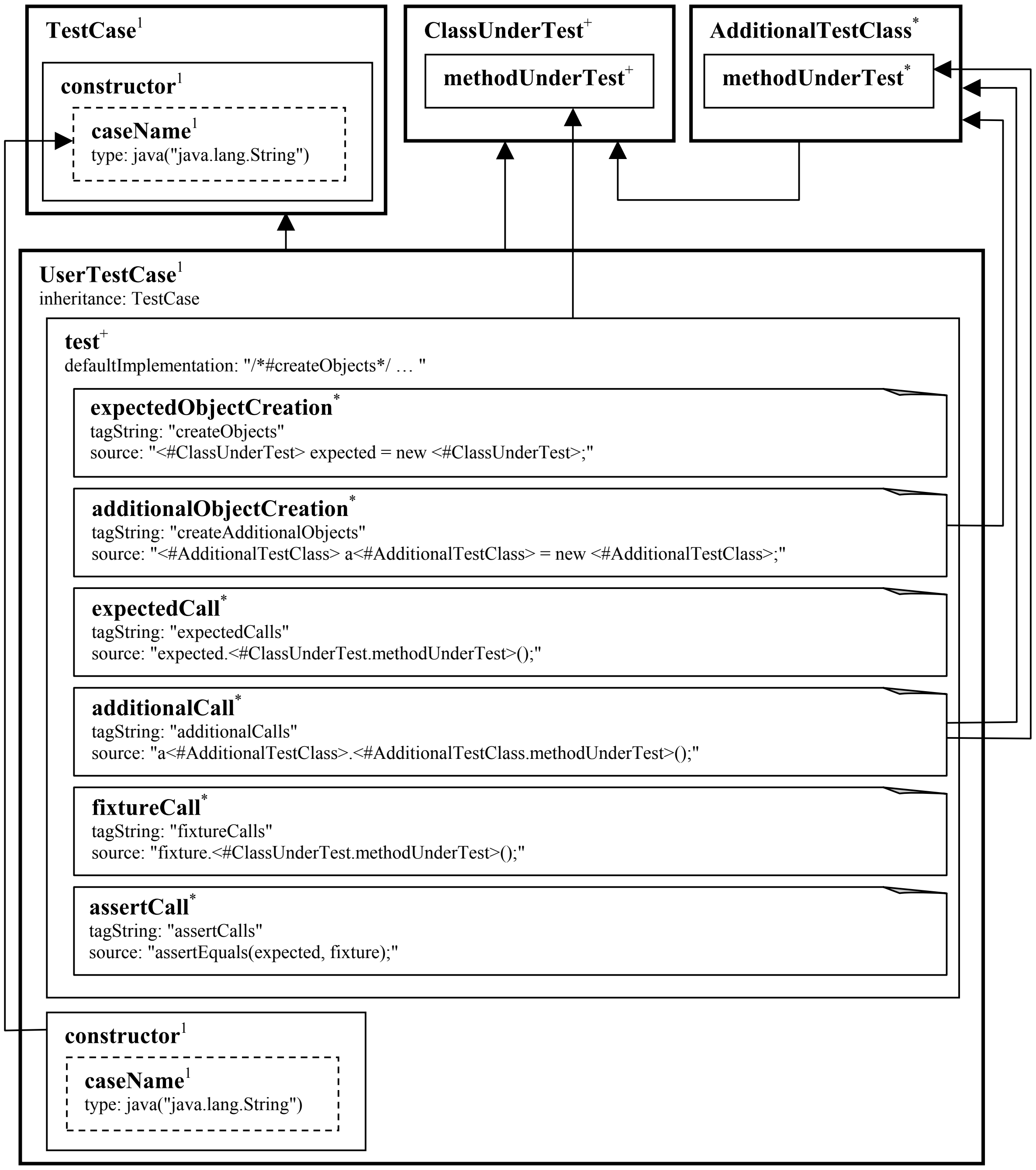
WTM95 Wong K., Tilley S., Muller H., Storey M., Structural Redocumentation: A Case Study. *IEEE Software* 12, 1, 46-54.

Yel96 Yelland P., Creating Host Compliance in a Portable Framework: A Study in the Reuse of Design Patterns. In: Proc. *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, San Jose, California, USA, October 1996, ACM SIGPLAN Notices 31, 10, 1996, 18-29.

Zim95 Zimmer W., Relationships Between Design Patterns. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 345-364.

Appendix A: Pattern Diagrams for Annotating the JUnit Framework

This appendix introduces eight specialization patterns that annotate the reuse interface of the JUnit testing framework. The patterns are depicted here as simplified pattern diagrams. For the complete specifications in textual format refer to appendix B. The first four of the patterns (*DefiningTests*, *SettingUpFixture*, *SelectingAndGroupingTests*, *RunningTests*) are basic patterns, which are needed in every specialization of JUnit. The rest of the patterns (*DecoratingTests*, *RunningRepeatedTests*, *RunningTestsInThreads*, *TestingExceptions*) describe more advanced ways of using JUnit.



TestCase1
constructor1
caseName1
type: java("java.lang.String")
ClassUnderTest+
methodUnderTest+
AdditionalTestClass*
methodUnderTest*
UserTestCase1
inheritance: TestCase
test+
defaultImplementation: "/*#createObjects*/ … "
expectedObjectCreation*
tagString: "createObjects"
source: "<#ClassUnderTest> expected = new <#ClassUnderTest>;"
additionalObjectCreation*
tagString: "createAdditionalObjects"
source: "<#AdditionalTestClass> a<#AdditionalTestClass> = new <#AdditionalTestClass>;"
expectedCall*
tagString: "expectedCalls"
source: "expected.<#ClassUnderTest.methodUnderTest>();"
additionalCall*
tagString: "additionalCalls"
source: "a<#AdditionalTestClass>.<#AdditionalTestClass.methodUnderTest>();"
fixtureCall*
tagString: "fixtureCalls"
source: "fixture.<#ClassUnderTest.methodUnderTest>();"
assertCall*
tagString: "assertCalls"
source: "assertEquals(expected, fixture);"
constructor1
caseName1
type: java("java.lang.String")

Figure A.1: The DefiningTests specialization pattern as a pattern diagram

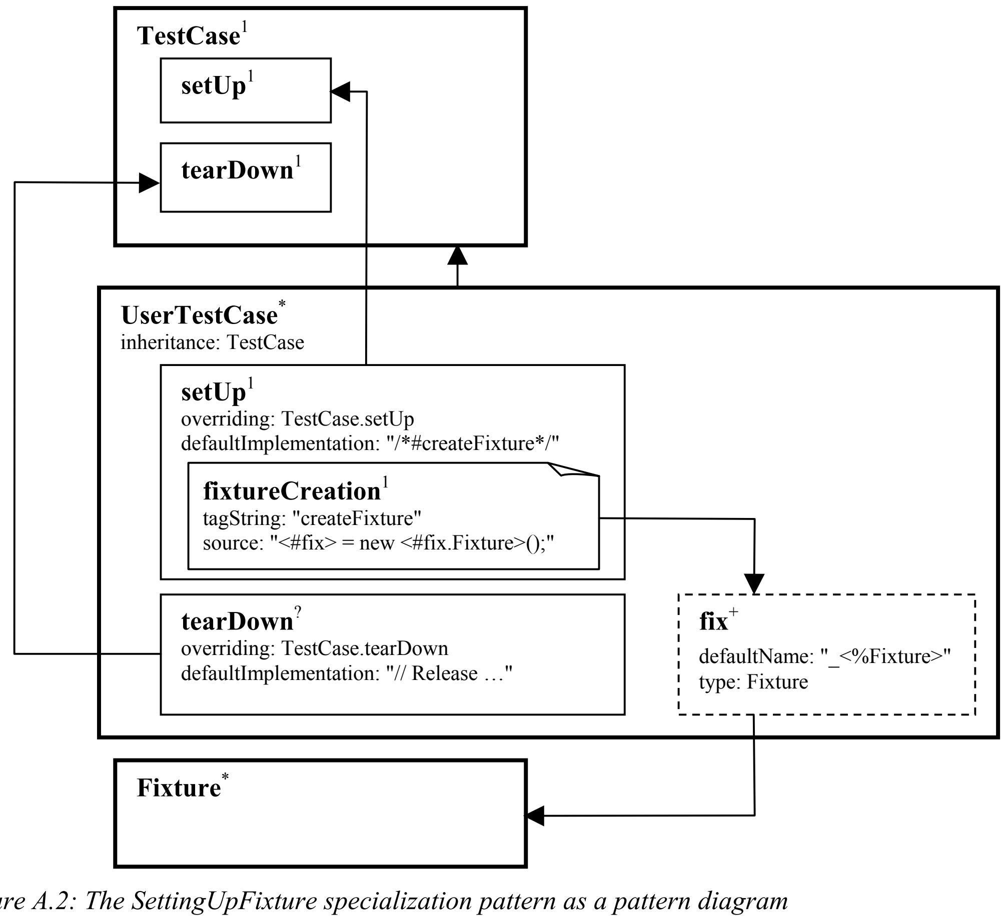

TestCase1
setUp1
tearDown1
UserTestCase*
inheritance: TestCase
setUp1
overriding: TestCase.setUp
defaultImplementation: "/*#createFixture*/"
fixtureCreation1
tagString: "createFixture"
source: "<#fix> = new <#fix.Fixture>();"
tearDown?
overriding: TestCase.tearDown
defaultImplementation: "// Release …"
fix+
defaultName: "_<%Fixture>"
type: Fixture
Fixture*

Figure A.2: The SettingUpFixture specialization pattern as a pattern diagram

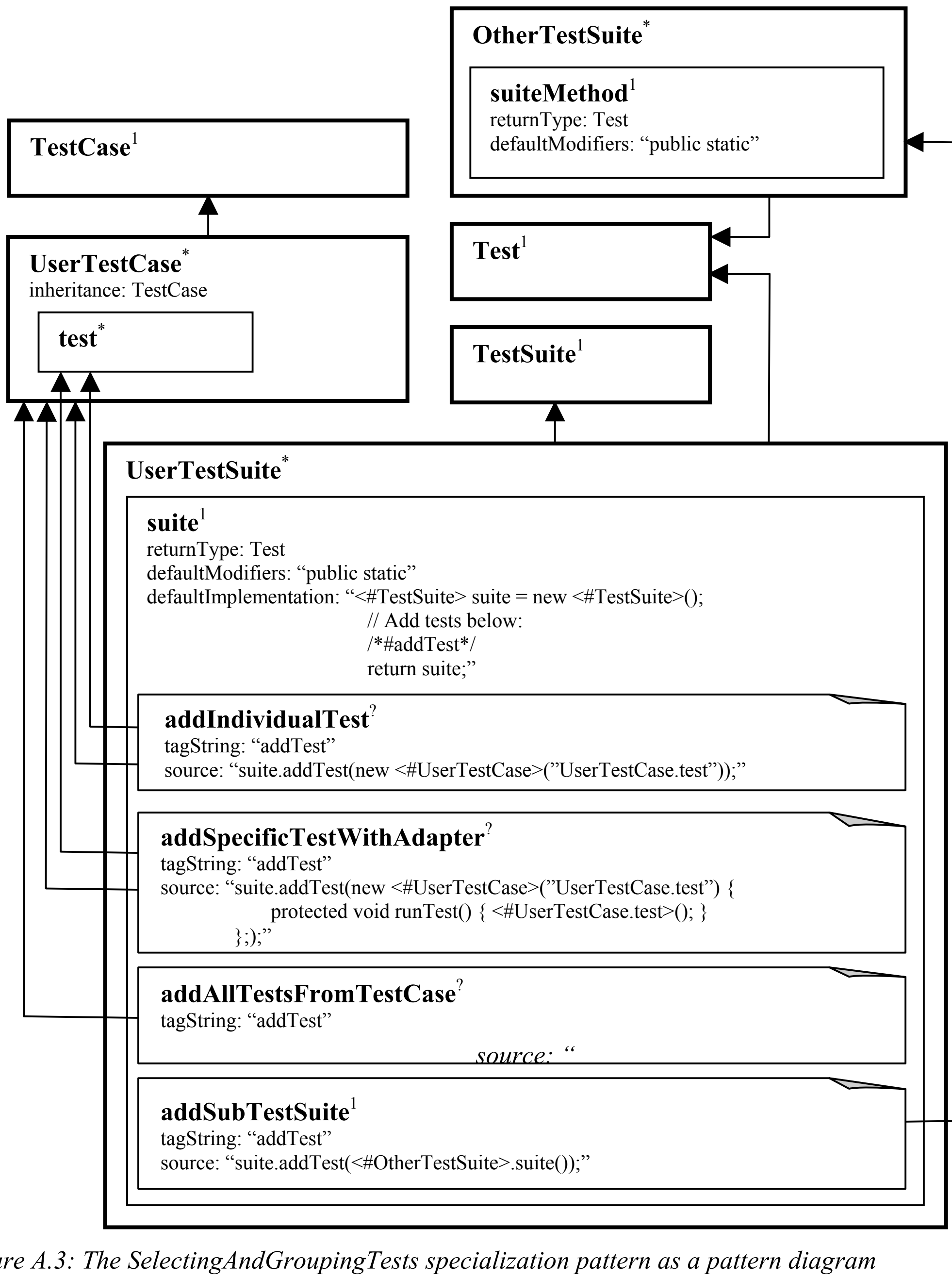

OtherTestSuite*
suiteMethod1
returnType: Test
defaultModifiers: “public static”
TestCase1
Test1
UserTestCase*
inheritance: TestCase
test*
TestSuite1
UserTestSuite*
suite1
returnType: Test
defaultModifiers: “public static”
defaultImplementation: “<#TestSuite> suite = new <#TestSuite>();
// Add tests below:
/*#addTest*/
return suite;”
addIndividualTest?
tagString: “addTest”
source: “suite.addTest(new <#UserTestCase>(”UserTestCase.test”));”
addSpecificTestWithAdapter?
tagString: “addTest”
source: “suite.addTest(new <#UserTestCase>(”UserTestCase.test”) {
protected void runTest() { <#UserTestCase.test>(); }
};);”
addAllTestsFromTestCase?
tagString: “addTest”
source: “
addSubTestSuite1
tagString: “addTest”
source: “suite.addTest(<#OtherTestSuite>.suite());”

Figure A.3: The SelectingAndGroupingTests specialization pattern as a pattern diagram

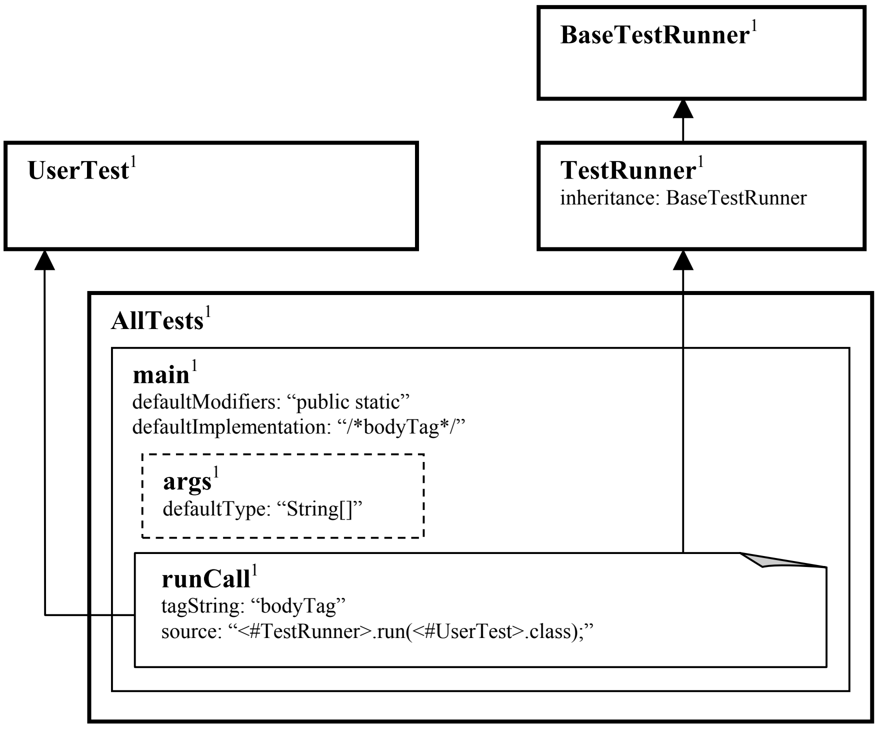

BaseTestRunner1
UserTest1
TestRunner1
inheritance: BaseTestRunner
AllTests1
main1
defaultModifiers: “public static”
defaultImplementation: “/*bodyTag*/”
args1
defaultType: “String[]”
runCall1
tagString: “bodyTag”
source: “<#TestRunner>.run(<#UserTest>.class);”

Figure A.4: The RunningTests specialization pattern as a pattern diagram

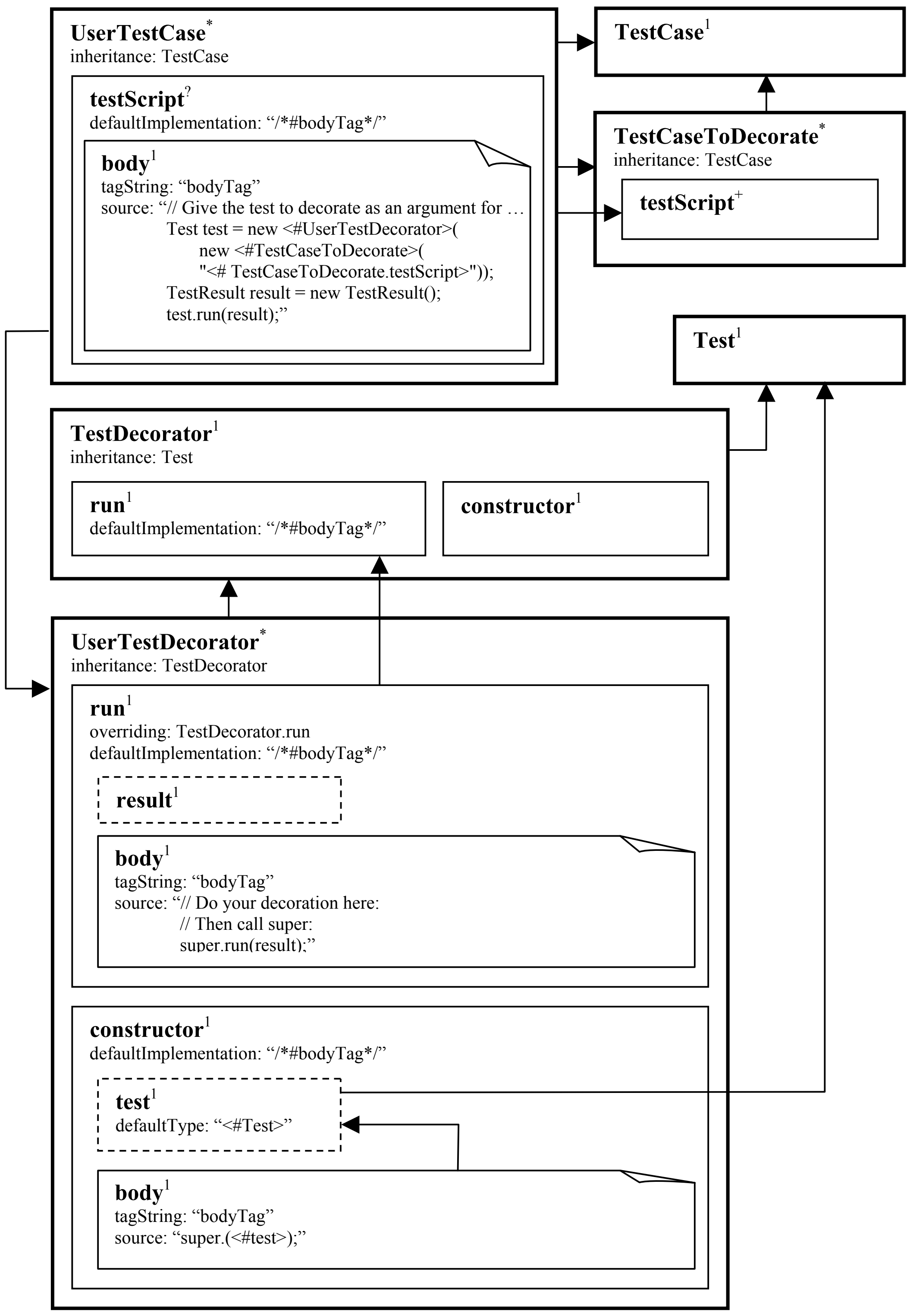
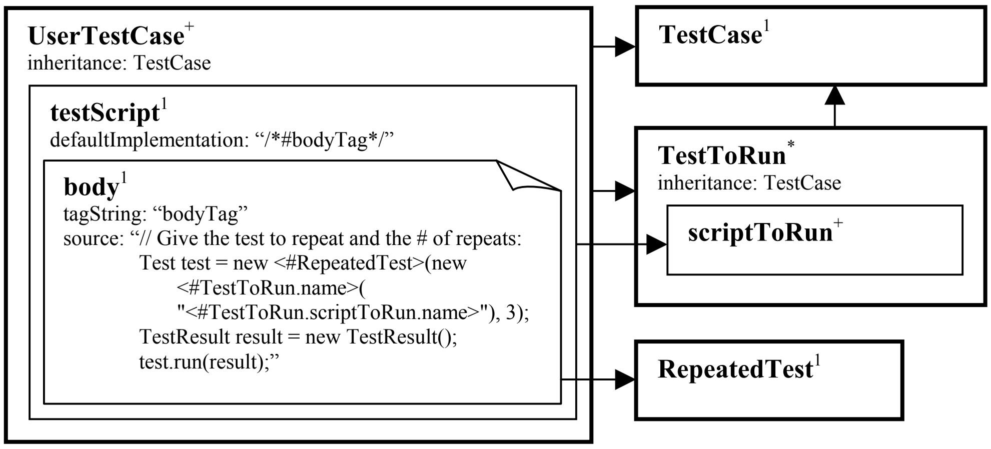
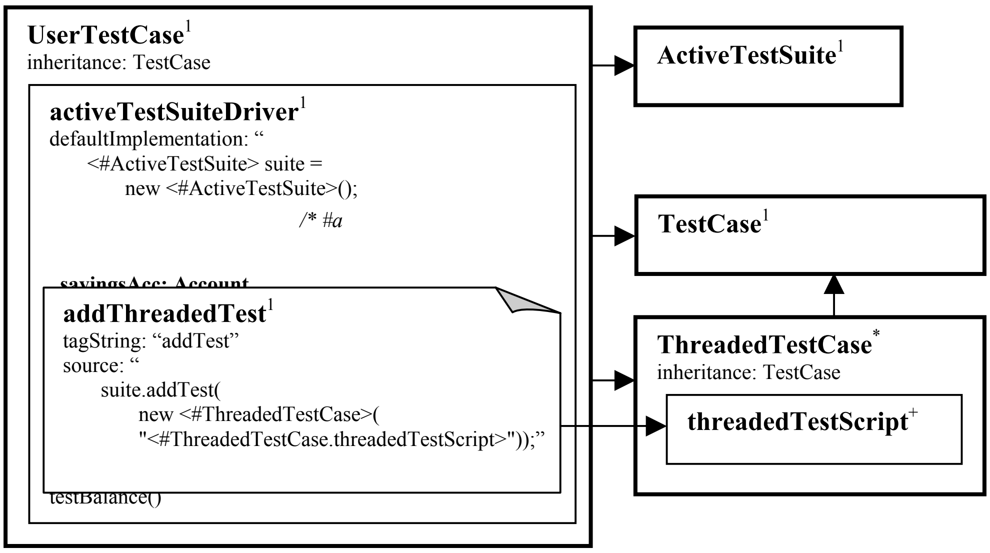

UserTestCase*
inheritance: TestCase
testScript?
defaultImplementation: "/*#bodyTag*/"
body1
tagString: "bodyTag"
source: "// Give the test to decorate as an argument for …
Test test = new <#UserTestDecorator>(
new <#TestCaseToDecorate>(
"<# TestCaseToDecorate.testScript>"));
TestResult result = new TestResult();
test.run(result);"
TestCase1
TestCaseToDecorate*
inheritance: TestCase
testScript+
Test1
TestDecorator1
inheritance: Test
run1
defaultImplementation: "/*#bodyTag*/"
constructor1
UserTestDecorator*
inheritance: TestDecorator
run1
overriding: TestDecorator.run
defaultImplementation: "/*#bodyTag*/"
result1
body1
tagString: "bodyTag"
source: "// Do your decoration here:
// Then call super:
super.run(result);"
constructor1
defaultImplementation: "/*#bodyTag*/"
test1
defaultType: "<#Test>"
body1
tagString: "bodyTag"
source: "super.(<#test>);"

Figure A.5: The DecoratingTests specialization pattern as a pattern diagram



```
source: “// Give the test to repeat and the # of repeats:
            Test test = new <#RepeatedTest>(new
                  <#TestToRun.name>(
                  "<#TestToRun.scriptToRun.name>"), 3);
            TestResult result = new TestResult();
            test.run(result);”
```

Figure A.6: The RunningRepeatedTests specialization pattern as a pattern diagram



```
defaultImplementation: “
      <#ActiveTestSuite> suite =
            new <#ActiveTestSuite>();
                                        /* #a
```

```
source: “
      suite.addTest(
            new <#ThreadedTestCase>(
            "<#ThreadedTestCase.threadedTestScript>"));”
```

Figure A.7: The RunningTestsInThreads specialization pattern as a pattern diagram

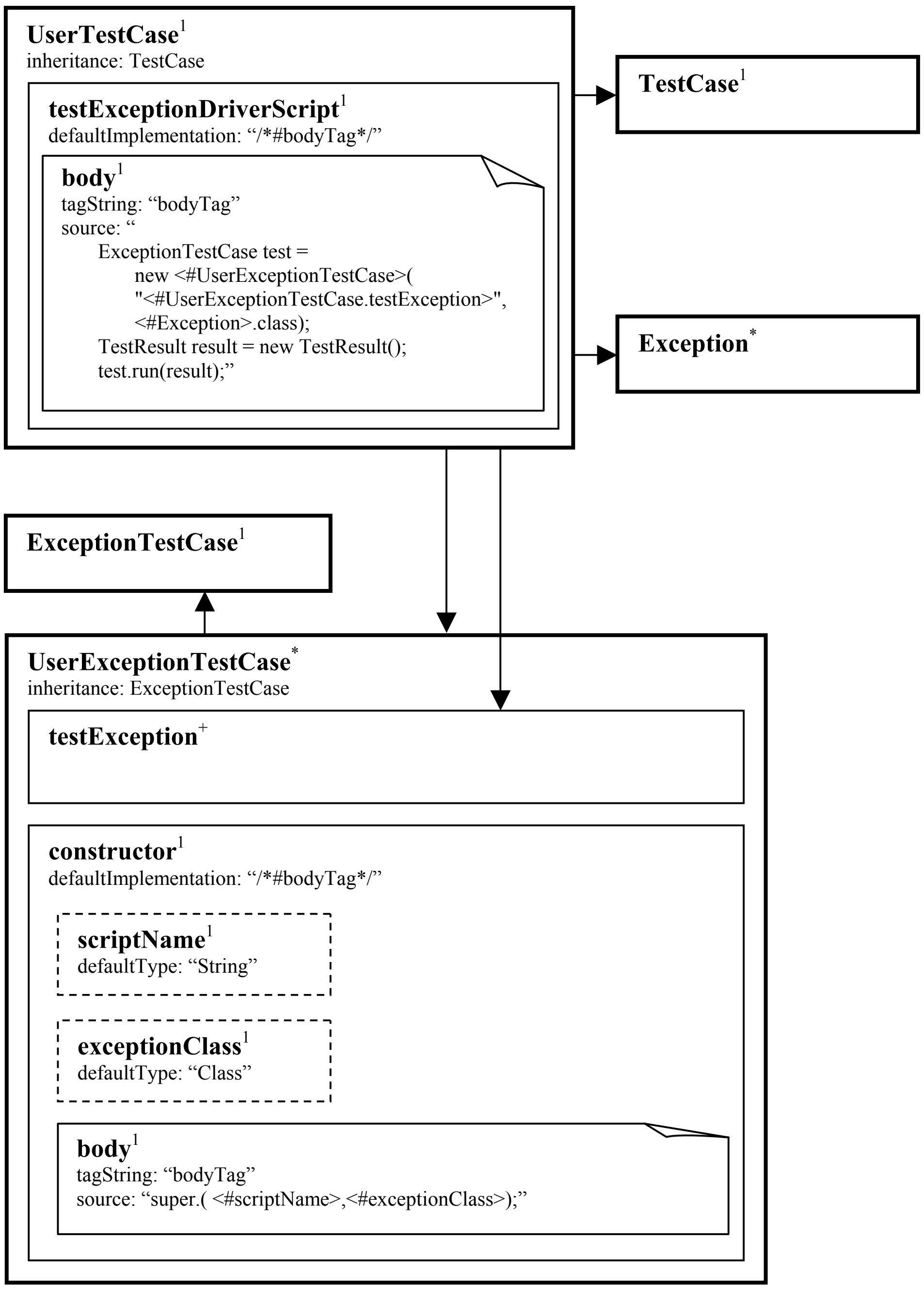


Figure A.8: The TestingExceptions specialization pattern as a pattern diagram

Appendix B: JUnit Specialization Patterns in a Textual Notation

This appendix describes the specialization patterns prepared to annotate the reuse interface of the JUnit framework. The syntax for the notation is given in appendix C. Each pattern description is preceded by a listing of source code packages that were used as input for the pattern and a listing of the elements that constituted the relevancy criteria for the automatic pattern extraction process.

The results of the pattern extraction process are summarized after the specification itself. Adjustments made to the generated pattern in order to make it more usable are illustrated within the specification source text. Those parts that were removed from the final pattern are written in *italics*, and those parts that were added or changed are written in **bold**.

Note that since the results are based on counting lines of pattern specification code the formatting of the pattern source text has an effect on them. That is why the results should be taken as being suggestive rather than final or exact.

The DefiningTests specialization pattern:

Framework packages:	junit.framework
Application packages:	junit.samples.money
Relevancy:	junit.framework.TestCase, junit.framework.TestCase.TestCase, junit.samples.money.MoneyTest.test*, junit.samples.money.MoneyTest.MoneyTest, junit.samples.money.IMoney

```
pattern DefiningTests;

class role ClassUnderTest 1..* {
   description: "You have to specify at least one class to be tested.";
   taskTitle: "Locate <#defaultName>.";

   method role methodUnderTest 1..* {
      description: "This is a role for the methods to be tested.";
      taskTitle: "Locate <#defaultName>.";
   }
}

class role TestCase {
   description: "Click <a href="junit/junit/framework/<#name>.html">here</a> to
      learn more about <#name>s.";
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";

   constructor role constructor 0..* {
      description: "Click <a href="junit/junit/framework/TestCase.html#
         <#name>">here</a> to learn more about <#name>s.";
      taskTitle: "F r a m e w o r k - Locate <#defaultName>.";

      parameter role caseName {
         type: java("java.lang.String");
         defaultType: "type";
      }
   }
}
```

```
class role UserTestCase 0..* [cut: ClassUnderTest, tc: TestCase] {
   inheritance: tc;
   description: "Click <a href="junit/junit/framework/<#tc.name>.html">
      here</a> to learn more about <#tc.name>s.";
   taskTitle: "Subclass <#tc.name>
      to define a test case for testing <#cut.name>.";
   defaultName: "<#cut>Test";

   method role test 1..* [mut: cut.methodUnderTest] {
      description: "A test script is a method of a test case object that performs
         tests. The names of test script methods usually start with "test".";
      taskTitle: "Define a  test script that tests <#mut.name>.";
      defaultImplementation: "// Add creation of test objects below:
         /* #createObjects */
         // Add creation of additional objects below:
         /* #createAdditionalObjects */
         // Add calls to additional objects below:
         /* #additionalCalls */
         // Add calls to expected objects below:
         /* #expectedCalls */
         // Add calls to fixture objects below:
         /* #fixtureCalls */
         // Add calls to assert methods below:
         /* #assertCalls */";

      code snippet role expectedObjectCreation 0..* {
         tagString: "createObjects";
         source: "<#cut.name> expected = new <#cut.name>();";
         description: "This code snippet creates …";
         taskTitle: "Provide code that creates …";
         defaultName: "createA<#cut.name>ToBeUsedInTest";
      }

      code snippet role fixtureCall 0..* {
         tagString: "fixtureCalls";
         source: "fixture.<#mut.name>();";
         description: "This code snippet calls …";
         taskTitle: "Provide code that calls …";
         defaultName: "call<%mut.name>";
      }

      code snippet role assertCall 0..* {
         tagString: "assertCalls";
         source: "assertEquals(expected, fixture);";
         description: "This code snippet compares …";
         taskTitle: "Provide code that compares …";
         defaultName: "callAssert";
      }

      code snippet role additionalCall 0..* [addMut:
         AdditionalTestClass.methodUnderTest, add: AdditionalTestClass] {
         description: "This code snippet calls …";
         taskTitle: "Provide code that calls …";
         tagString: "additionalCalls";
         source: "a<#add.name>.<#addMut.name>();";
      }

      code snippet role additionalObjectCreation 0..* [add: AdditionalTestClass] {
         description: "This code snippet creates …";
         defaultName: "createA<#add.name>ToBeUsedInTest";
         source: "<#add.name> a<#add.name> = new <#add.name>();";
         tagString: "createAdditionalObjects";
         taskTitle: "Provide code that creates …";
      }

      code snippet role expectedCall 0..* {
         defaultName: "call<%mut.name>";
         description: "This code snippet calls …";
         source: "expected.<#mut.name>();";
         tagString: "expectedCalls";
         taskTitle: "Provide code that calls …";
      }
   }

   constructor role constructor 0..1 {
      description: "Click <a href="junit/junit/framework/<#name>.html#
         <#name>">here</a> to learn more about <#name>s.";
      taskTitle: "Provide a constructor that calls super with the given test case
         name as an argument.";
      defaultImplementation: "super(name);";

      parameter role caseName {
         defaultName: "name";
         type: java("java.lang.String");
         defaultType: "type";
      }
   }
}

class role AdditionalTestClass 0..* [cut: ClassUnderTest] {
   inheritance: im;
   description: "This is a role for additional classes needed in testing
      <#cut.name>.";
   taskTitle: "Locate an additional class needed to test <#cut.name>.";

   method role methodUnderTest 0..* {
      description: "This is a role for the methods that are needed in testing
         <#cut.name>.";
      taskTitle: "Locate <#defaultName>.";
   }
}
```

Results for *DefiningTests*	
Non-empty lines:	114
Lines to remove:	3
Lines to add or modify:	86
Description or comment modifications:	39
Modifications purely related to role renaming:	0
Functional changes:	88 – 39 – 0 = 47
Extracted part of the pattern counting all additions, removals, and modifications:	(114 – 3 – 86) / 114 ≈ 22 %
Extracted part of the pattern disregarding removals and modifications due to better descriptions and naming:	(111 – 47) / 111 ≈ 58 %

The SettingUpFixture specialization pattern:

Framework packages:	junit.framework
Application packages:	junit.samples.money
Relevancy:	junit.framework.TestCase, junit.framework.TestCase.setUp, junit.framework.TestCase.tearDown, junit.samples.money.Money

```
pattern JUnit_SettingUpFixture;

class role Fixture 0..* {
   description: "A fixture class is a class, which is needed as test material.";
   taskTitle: "Locate <#defaultName>.";
}

class role TestCase {
   description: "Click <a href="junit/junit/framework/<#name>.html">
      here</a> to learn more about <#name>s.";
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";

   method role setUp {
      description: "Click <a href="junit/junit/framework/TestCase.html#
         <#name>">here</a> to learn more about <#name>s.";
      taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
   }

   method role tearDown {
      description: "Click <a href="junit/junit/framework/TestCase.html#
         <#name>">here</a> to learn more about <#name>s.";
      taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
   }
}

class role UserTestCase 0..* [tc: TestCase] {
   inheritance: tc;
   description: "Click <a href="junit/junit/framework/<#tc.name>.html">
      here</a> to learn more about <#tc.name>s.";
   taskTitle: "Subclass <#tc.name>.";

   method role setUp [su: tc.setUp] {
      overriding: su;
      description: "Click <a href="junit/junit/framework/<#tc.name>.html#
         <#name>">here</a> to learn more about <#name>.";
      taskTitle: "Override <#defaultName>.";
      defaultImplementation: "// Add creation of fixture objects below:
         /*#bodyTag*/";

      code snippet role fixtureCreation [attr: fix, m: Money] {
         description: "This code snippet initializes <#attr.name> with an instance of
            the fixture class <#attr.typeName> to be used in the test scripts.";
         taskTitle: "Provide code that assigns an instance of
            <#attr.typeName> to <#attr.name>.";
         source: "// Add creation of fixture objects below:
            /*#bodyTag*/
            <#attr.name> = new <#attr.typeName>(12, "CHF");
            <#attr.name> = new <#m.name>(14, "CHF");
            <#attr.name> = new <#m.name>(21, "USD");
            <#attr.name> = new <#m.name>(7, "USD");
            fMB1 = new <#m.name>Bag(<#attr.name>, <#attr.name>);
            fMB2 = new <#m.name>Bag(<#attr.name>, <#attr.name>);
         tagString: "bodyTag";
      }
   }

   method role tearDown 0..1 [td: tc.tearDown] {
      overriding: td;
      description: "Click <a href="junit/junit/framework/<#tc.name>.html#
```

```
         <name>">here</a> to learn more about <#name>.";
      taskTitle: "Override <#defaultName>.";
      defaultImplementation: "// Release the resources here.";
   }

   field role fix 1..* [typeName: Fixture] {
      type: typeName;
      description: "All fixture objects (including <#typeName.name> objects)
         are stored as fields in a test case.";
      taskTitle: "Provide a <#typeName.name> object as test fixture.";
      defaultName: "a<#typeName>";
   }
}
```

Results for *SettingUpFixture*	
Non-empty lines:	63
Lines to remove:	7
Lines to add or modify:	20
Description or comment modifications:	15
Modifications purely related to role renaming:	1
Functional changes:	20 – 15 – 1 = 4
Extracted part of the pattern counting all additions, removals, and modifications:	(63 – 7 – 20) / 63 ≈ 57 %
Extracted part of the pattern disregarding removals and modifications due to better descriptions and naming:	(56 – 4) / 56 ≈ 93 %

The SelectingAndGroupingTests specialization pattern:

Framework packages:	junit.framework
Application packages:	junit.samples.money
Relevancy:	junit.framework.TestCase, junit.framework.TestSuite, junit.samples.money.MoneyTest.test*, junit.samples.money.AccountTest, junit.samples.money.AccountTest.test*, junit.samples.money.AllTests, junit.samples.money.AllTests.suite

```
pattern JUnit_SelectingAndGroupingTests;

class role TestCase {
   description: "Click <a href="junit/junit/framework/<#name>.html">
      here</a> to learn more about <#name>s.";
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
}

class role TestSuite {
   description: "Click <a href="junit/junit/framework/<#name>.html">
      here</a> to learn more about <#name>s.";
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
}

class role UserTestCase 0..* [tc: TestCase] {
   inheritance: tc;
   description: "Click <a href="junit/junit/framework/<#tc.name>.html">
      here</a> to learn more about <#tc.name>s.";
   taskTitle: "Locate a test case whose tests are to be selected as part of a suite.";

   method role test 0..* {
      description: "A test script is a method of a test case object that
         performs tests. The names of test script methods usually start with "test".";
      taskTitle: "Locate a test script that is to be added as a part of
         a test suite.";
      defaultImplementation: "/*#bodyTag*/";

      code snippet role testAccountTestInThreads#body 0..1 {
         ...
      }

      code snippet role test 0..* {
         ...
      }

      code snippet role testTestWithdrawMultipleTimes#body 0..1 {
         ...
      }

      code snippet role testDepositNullExceptionDriverScript#body 0..1 {
         ...
      }

      code snippet role testTestDepositWithDelayedTestCase#body 0..1 {
         ...
      }
   }
}

class role OtherTestSuite 0..* [userSuite: UserTestSuite] {
   taskTitle: "Locate another test suite that is to be added as a subsuite for
      the <#userSuite.name>.";
   description: "This role is for other test suites that are to be added as subsuites
      for the <#userSuite.name>. These classes must implement a public static suite()
      method.";
   method role suiteMethod [test: Test] {
      defaultModifiers: "public static";
      returnType: test;
      description: "This is a role for the method that returns
         the actual test suite.";
      taskTitle: "Locate <#defaultName>.";
      defaultName: "suite";
   }
}

class role UserTestSuite 0..* [test: Test, ts: TestSuite] {
   description: "Click <a href="junit/junit/framework/<#ts.name>.html">
      here</a> to learn more about <#ts.name>s.";
   taskTitle: "Provide a class that defines the test suite that groups selected
      tests and other test suites together in its public static suite() method.";

   method role suite {
      defaultModifiers: "public static";
      returnType: test;
      defaultReturnType: "returnType";
      description: "This role is for the <#signature> method that creates a
         test suite, adds selected tests to it, and then returns it.";
      taskTitle: "Provide a <#signature> method that creates a test suite,
         adds selected tests to it, and then returns the suite.";
      defaultImplementation: "<#ts.name> suite = new <#ts.name>();
         // Add tests below:
         /*#bodyTag*/
         return suite;";

      code snippet role addIndividualTest 0..1 [script: UserTestCase.test,
         utc: UserTestCase] {
         description: "This code snippet is used for generating code that
            adds a call to <#script.name> to the test suite.";
         taskTitle: "Locate <#defaultName>.";
         source: "<#ts.name> <#s.name> = new <#ts.name>();
            // Add tests below:
            /*#bodyTag*/
            <#s.name>.add<#ts.name>(<#utc.name>.class);
            suite.addTest(new <#utc.name>("<#script.name>"));
            return <#s.name>;";
         tagString: "bodyTag";
      }

      code snippet role addSpecificTestWithAdapter 0..1 [case: UserTestCase,
         script: UserTestCase.test] {
         tagString: "bodyTag";
         source: "suite.addTest(new <#case.name>("<#script.name>") {
               protected void runTest() {
                  <#script.name>();
               }
            };);";
         description: "This code snippet is used for generating code that adds a call
            to <#script.name> to the test suite. This method uses an anonymous inner
            class as an adapter.";
         taskTitle: "Provide code that adds a call to <#script.name> to the test suite
            using an anonymous inner class.";
      }

      code snippet role addAllTestsFromTestCase 0..1 [case: UserTestCase] {
         tagString: "bodyTag";
         source: "suite.addTestSuite(<#case.name>.class);";
         description: "This code snippet is used for generating code that
            adds a call to all test methods in <#case.name> to the test suite.";
         taskTitle: "Provide code that adds a call to all test methods
            in <#case.name> to the test suite.";
      }

      code snippet role addSubTestSuite [other: OtherTestSuite] {
         tagString: "bodyTag";
         source: "suite.addTest(<#other.name>.suite());";
         description: "This code snippet is used for generating code that
            adds the whole suite defined in <#other.name> as a subsuite of this test
            suite.";
         taskTitle: "Provide code that adds the whole suite defined in
            <#other.name> as a subsuite of this test suite.";
      }
   }
```

```
}

class role Test {
   description: "Click <a href="junit/junit/framework/<#name>.html">
      here</a> to learn more about <#name>s.";
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
}
```

Results for *SelectingAndGroupingTests*	
Non-empty lines:	122
Lines to remove:	21
Lines to add or modify:	75
Description or comment modifications:	40
Modifications purely related to role renaming:	0
Functional changes:	75 – 40 – 0 = 35
Extracted part of the pattern counting all additions, removals, and modifications:	(122 – 21 – 75) / 122 ≈ 21 %
Extracted part of the pattern disregarding removals and modifications due to better descriptions and naming:	(101 – 35) / 101 ≈ 65 %

The RunningTests specialization pattern:

Framework packages: junit.framework, junit.runner

Application packages: junit.samples.money, junit.textui

Relevancy: junit.runner.BaseTestRunner,
junit.textui.TestRunner,
junit.textui.TestRunner.run,
junit.samples.money.AllTests,
junit.samples.money.AllTests.main

```
pattern JUnit_RunningTests;

class role UserTest {
   description: "Use the SelectingAndGroupingTests pattern to define a set of
      tests to be run.";
   taskTitle: "Locate a test suite or a test case to be run.";
}

class role BaseTestRunner {
   description: "Click <a href="junit/junit/runner/<#name>.html">
      here</a> to learn more about <#name>s.";
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
}

class role TestRunner [btr: BaseTestRunner] {
   inheritance: btr;
   description: "Click <a href="junit/junit/runner/<#btr.name>.html">
      here</a> to learn more about <#btr.name>s.";
   taskTitle: "Locate a <#defaultName>. Use junit.textui.TestRunner,
      junit.awtui.TestRunner, junit.swingui.TestRunner, or your own subclass
      of <btr.name>.";

   method role run 0..* {
      description: "Click ...";
      taskTitle: "Locate <#defaultName>.";
      defaultImplementation: "/*#bodyTag*/";

      code snippet role run#body 0..1 [ubtr: UserBaseTestRunner] {
         description: "...";
         taskTitle: "Locate <#defaultName>.";
         source: "...";
         tagString: "bodyTag";
      }
   }
}

class role AllTests {
   description: "This is the class that executes the tests through its main method.";
   taskTitle: "Provide the main class whose main method will execute the tests.";

   method role main {
      defaultModifiers: "public static";
      defaultReturnType: "void";
      description: "Click ...";
      taskTitle: "Locate <#defaultName>.";
      defaultImplementation: "/*#bodyTag*/";

      parameter role args {
         defaultType: "String[]";
      }

      code snippet role runCall [runner: TestRunner, userTest: UserTest] {
         description: "This code snippet is used for generating code that
            calls run on the <#runner.name> by giving the <#userTest.name> as an
            argument. The test suite is automatically generated for all test methods
            defined in <#userTest.name>.";
         taskTitle: "Provide code that runs the tests defined in <#userTest.name>.";
         source: "/*#bodyTag*/
```

```
            <#runner.name>.run(<#userTest.name>.class);";
         tagString: "bodyTag";
      }
   }
}
```

Results for *RunningTests*	
Non-empty lines:	54
Lines to remove:	14
Lines to add or modify:	20
Description or comment modifications:	13
Modifications purely related to role renaming:	1
Functional changes:	20 – 13 – 1 = 6
Extracted part of the pattern counting all additions, removals, and modifications:	(54 – 14 – 20) / 54 ≈ 37 %
Extracted part of the pattern disregarding removals and modifications due to better descriptions and naming:	(40 – 6) / 40 ≈ 85 %

JUnit_DecoratingTests:

Framework packages:	junit.extensions, junit.framework
Application packages:	junit.extensions, junit.samples.money
Relevancy criteria:	junit.extensions.TestDecorator, junit.extensions.TestDecorator.TestDecorator, junit.extensions.TestDecorator.run, junit.samples.money.AccountTest, junit.samples.money.AccountTest.testTestDepositWithDelayedTestCase, junit.framework.TestCase

```
pattern JUnit_DecoratingTests;

class role UserTestDecorator 0..* [td: TestDecorator] {
   inheritance: td;
   description: "Click <a href="junit/junit/extensions/<#td.name>.html">
               here</a> to learn more about <#td.name>s.";
   taskTitle: "Subclass <#td.name> to define your own test decorator.";

   method role run [r: td.run] {
      overriding: r;
      description: "Click <a href="junit/junit/extensions/<td.name>.html#
         <#name>">here</a> to learn more about <#name>s.";
      taskTitle: "Override <#defaultName>.";
      defaultImplementation: "/*#bodyTag*/";

      code snippet role body {
         tagString: "bodyTag";
         source: "// Do your decoration here:
            // Then call super:
            super.run(result);";
         taskTitle: "Provide code that does the actual decoration and
            then calls super with the given test result as an argument.";
      }

      parameter role result {
      }
   }

   constructor role constructor [t: Test] {
      defaultImplementation: "/* #bodyTag */";
      taskTitle: "Provide a constructor for your test decorator.";

      parameter role test {
         defaultType: "<#t.name>";
         defaultName: "test";
      }

      code snippet role body [t: test] {
         tagString: "bodyTag";
         source: "super(<#t.name>);";
         taskTitle: "Provide code that calls super with the given test as
            an argument.";
      }
   }
}

class role TestDecorator {
   description: "Click <a href="junit/junit/extensions/<#name>.html">
      here</a> to learn more about <#name>s.";
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";

   method role run {
      description: "Click <a href="junit/junit/extensions/TestDecorator.html#
```

```
         <#name>">here</a> to learn more about <#name>s.";
      taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
      defaultImplementation: "/*#bodyTag*/";
   }

   constructor role constructor {
      description: "Click <a href="junit/junit/extensions/TestDecorator.html#
         <#name>">here</a> to learn more about <#name>s.";
      taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
   }
}

class role Test {
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
   description: "Click <a href="junit/junit/framework/<#name>.html">
      here</a> to learn more about <#name>s.";
}

class role UserTestCase 0..* [tc: TestCase,
   tctd: TestCaseToDecorate, tstd: TestCaseToDecorate.testScript,
   utd: UserTestDecorator] {
   inheritance: tc;
   description: "Click <a href="junit/junit/framework/<#tc.name>.html">
      here</a> to learn more about <#tc.name>s.";
   taskTitle: "Locate a test case that uses <#utd.name> to
      decorate <#tstd.name>.";

   method role testScript 0..1 {
      description: "A test script is a method of a test case object that
         performs tests. The names of test script methods usually start with "test".";
      taskTitle: "Provide a test script for testing <#tstd.name> with <#utd.name>.";
      defaultName: "test<%tstd.name>With<%utd.name>";
      defaultImplementation: "/*#bodyTag*/";

      code snippet role body [utd: UserTestDecorator, utc: UserTestCase] {
         description: "Click ...";
         taskTitle: "Define a method body that gives the test to decorate
            as an argument for the decorator constructor.";
         source: "/* #bodyTag */
            // Give the test to decorate as an argument ...
            Test test = new <#utd.name>(
               new <#tctd.name>("<#tstd.name>"), 5000);
            TestResult result = new TestResult();
            test.run(result);";
         tagString: "bodyTag";
      }
   }
}

class role TestCase {
   description: "Click <a href="junit/junit/framework/<#name>.html">
      here</a> to learn more about <#name>s.";
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
}

class role TestCaseToDecorate 0..* [tc: TestCase] {
   inheritance: tc;
   taskTitle: "Locate a test case to decorate.";
   description: "Click <a href="junit/junit/framework/<#tc.name>.html">
      here</a> to learn more about <#tc.name>s.";

   method role testScript 1..* {
      description: "A test script is a method of a test case object that
         performs tests. The names of test script methods usually start with "test".";
      taskTitle: "Locate a test script to decorate.";
   }
}
```

Results for *DecoratingTests*	
Non-empty lines:	103
Lines to remove:	3
Lines to add or modify:	54
Description or comment modifications:	25
Modifications purely related to role renaming:	0
Functional changes:	54 – 25 – 0 = 29
Extracted part of the pattern counting all additions, removals, and modifications:	(103 – 3 – 54) / 103 ≈ 45 %
Extracted part of the pattern disregarding removals and modifications due to better descriptions and naming:	(100 – 29) / 100 ≈ 71 %

JUnit_RunningRepeatedTests:

Framework packages:	junit.extensions, junit.framework
Application packages:	junit.samples.money, junit.tests
Relevancy criteria:	junit.extensions.RepeatedTest, junit.samples.money.AccountTest, junit.samples.money.AccountTest.testTestWithdrawMultipleTimes, junit.framework.TestCase

```
pattern JUnit_RunningRepeatedTests;

class role RepeatedTest {
   description: "Click <a href="junit/junit/extensions/<#name>.html">
      here</a> to learn more about <#name>s.";
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
}

class role UserTestCase 1..* [tc: TestCase, ttr: TestToRun] {
   inheritance: tc;
   description: "Click <a href="junit/junit/framework/<#tc.name>.html">
      here</a> to learn more about <#tc.name>s.";
   taskTitle: "Locate or provide a test case that will run the selected tests
      in <#ttr.name> multiple times.";

   method role testScript 0..1 [ts: ttr.scriptToRun] {
      description: "A test script is a method of a test case object that performs
         tests. The names of test script methods usually start with "test".";
      taskTitle: "Provide a test script that runs <#ts.name> multpile times.";
      defaultImplementation: "/*#bodyTag*/";
      defaultName: "test<%ts.name>MultipleTimes";

      code snippet role body [rt: RepeatedTest] {
         description: "Click ...";
         taskTitle: "Add code that uses <#rt.name> to run <#ts.name> multiple times.";
         source: "/*#bodyTag*/
            // Give the test to repeat and the number of repeats:
            Test test = new <#rt.name>(new ttr.name>("<#ts.name>"), 3);
            TestResult result = new TestResult();
            test.run(result);
            assertEquals(3, result.runCount());";
         tagString: "bodyTag";
      }
   }
}

class role TestToRun 0..* [tc: TestCase] {
   inheritance: tc;
   taskTitle: "Locate a class that contains a test to be run multiple times.";
   description: "Click <a href="junit/junit/framework/<#tc.name>.html">
      here</a> to learn more about <#tc.name>s.";

   method role scriptToRun 1..* {
      description: "A test script is a method of a test case object that performs
         tests. The names of test script methods usually start with "test".";
      taskTitle: "Locate a test script to be run multiple times.";
   }
}

class role TestCase {
   description: "Click <a href="junit/junit/framework/<#name>.html">
      here</a> to learn more about <#name>s.";
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
}
```

Results for *RunningRepeatedTests*	
Non-empty lines:	47
Lines to remove:	3
Lines to add or modify:	21
Description or comment modifications:	12
Modifications purely related to role renaming:	1
Functional changes:	21 – 12 – 1 = 8
Extracted part of the pattern counting all additions, removals, and modifications:	(47 – 3 – 21) / 47 ≈ 49 %
Extracted part of the pattern disregarding removals and modifications due to better descriptions and naming:	(44 – 8) / 44 ≈ 82 %

JUnit_RunningTestsInThreads:

Framework packages:	junit.extensions, junit.framework
Application packages:	junit.samples.money, junit.tests
Relevancy criteria:	junit.extensions.ActiveTestSuite, junit.samples.money.AccountTest, junit.samples.money.AccountTest.testAccountTestInThreads, junit.framework.TestCase

```
pattern JUnit_RunningTestsInThreads;

class role TestCase {
   description: "Click <a href="junit/junit/framework/<#name>.html">
      here</a> to learn more about <#name>s.";
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
}

class role UserTestCase 0..* [tc: TestCase,
   ttc: ThreadedTestCase, ats: ActiveTestSuite] {
   inheritance: tc;
   description: " Click <a href="junit/junit/framework/<#tc.name>.html">
      here</a> to learn more about <#tc.name>s.";
   taskTitle: "Locate or provide a test case that will act as a driver for running
      test script in separate threads.";

   method role activeTestSuiteDriver 0..1 {
      description: "Click <a href=junit/junit/extensions/<#ats.name>.html#
         <#name>">here</a> to learn more about <#ats.name>s.";
      taskTitle: "Provide <#defaultName> for running selected tests each in
         a separate thread.";
      defaultImplementation: "<#ats.name> suite = new <#ats.name>();
         /* #addTest */
         TestResult result = new TestResult();
         suite.run(result);";

      code snippet role addThreadedTest [tts: ThreadedTestCase.threadedTestScript,
         utc: UserTestCase, ats: ActiveTestSuite] {
         description: "Click ...";
         taskTitle: "Provide <#defaultName> for running selected tests each in
            a separate thread.";
         source: "<#ats.name> suite= new <#ats.name>();
            /* #addTest */
            suite.addTest(new <#ttc.name>("<#tts.name>"));
            suite.addTest(new <#utc.name>("testGetBalance"));
            suite.addTest(new <#utc.name>("testDeposit"));
            TestResult result = new TestResult();
            suite.run(result);";
         tagString: "bodyTag";
      }
   }

   method role createActiveTestSuite 0..1 {
      description: "Click <a href="junit/junit/tests/ActiveTestTest.html#
         <#name>">here</a> to learn more about <#name>s.";
      taskTitle: "Locate <#defaultName>.";
      defaultImplementation: "/*#bodyTag*/";
   }
}

class role ActiveTestSuite {
   description: "Click <a href="junit/junit/extensions/<#name>.html">
      here</a> to learn more about <#name>s.";
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
}

class role ThreadedTestCase 0..* [tc: TestCase] {
   inheritance: tc;
   taskTitle: "Locate a test case whose test scripts you would like to run in
      separate threads.";
   description: "Click <a href="junit/junit/framework/<#tc.name>.html">
      here</a> to learn more about <#tc.name>s.";

   method role threadedTestScript 1..* {
      description: "A test script is a method of a test case object that performs
         tests. The names of test script methods usually start with "test".";
      taskTitle: "Locate a test script you would like to run in a separate thread.";
   }
}
```

Results for *RunningTestsInThreads*	
Non-empty lines:	61
Lines to remove:	15
Lines to add or modify:	28
Description or comment modifications:	16
Modifications purely related to role renaming:	0
Functional changes:	28 – 16 – 0 = 12
Extracted part of the pattern counting all additions, removals, and modifications:	(61 – 15 – 28) / 61 ≈ 30 %
Extracted part of the pattern disregarding removals and modifications due to better descriptions and naming:	(46 – 12) / 46 ≈ 74 %

JUnit_TestingExceptions:

Framework packages:	junit.extensions, junit.framework
Application packages:	junit.samples.money, java.lang
Relevancy criteria:	java.lang.Exception, junit.extensions.ExceptionTestCase, junit.samples.money.AccountTest, junit.samples.money.AccountTest.testDepositNullExceptionDriverScript, junit.samples.money.AccountExceptionTestCase, junit.samples.money.AccountExceptionTestCase. AccountExceptionTestCase, junit.samples.money.AccountExceptionTestCase.testDepositNullException

```
pattern JUnit_TestingExceptions;

class role ExceptionTestCase {
   description: "Click <a href="junit/junit/extensions/<#name>.html">
      here</a> to learn more about <#name>s.";
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
}

class role UserExceptionTestCase 0..* [etc: ExceptionTestCase] {
   inheritance: etc;
   description: "Click <a href="junit/junit/extensions/<#etc.name>.html">
      here</a> to learn more about <#etc.name>s.";
   taskTitle: "Subclass <#etc.name> to define a test case that can test exceptions
      thrown by the methods to be tested.";

   constructor role constructor {
      description: "Click ...";
      taskTitle: "Locate <#defaultName>.";

      parameter role scriptName {
         defaultType: "String";
      }

      parameter role exceptionClass {
         parameterNumber: "2";
         defaultType: "Class";
      }

      code snippet role body {
         source: "super(scriptName, exceptionClass);";
         tagString: "bodyTag";
         taskTitle: "Provide code that calls super with given test script name and
            exception class as arguments.";
      }
   }

   method role testException 1..* {
      description: "A test script is a method of a test case object that performs
         tests. The names of test script methods usually start with "test".";
      taskTitle: "Provide a test script that should cause an exception to be thrown.";
      defaultImplementation: "/*#bodyTag*/";
   }
}

class role UserTestCase [tc: TestCase, uetc: UserExceptionTestCase,
   uetcs: UserExceptionTestCase.testException, e: Exception] {
   inheritance: tc;
   description: "Click <a href="junit/junit/framework/<#tc.name>.html">
      here</a> to learn more about <#tc.name>s.";
   taskTitle: "Locate or provide a test case that will call <#uetcs.name>.";

   method role testExceptionDriverScript {
      description: "A test script is a method of a test case object that performs
         tests. The names of test script methods usually start with "test".";
      taskTitle: "Provide a test script that calls <#uetcs.name>.";
      defaultImplementation: "/*#bodyTag*/";

      code snippet role body [etc: ExceptionTestCase, uetc: UserExceptionTestCase] {
         description: "Click ...";
         taskTitle: "Provide code that calls <#uetcs.name>.";
         source: "/* #bodyTag */
            <#etc.name> test = new <#uetc.name>(
               "<#uetc.name>", <#e.name>.class);
            TestResult result = test.run();
            assertEquals(0, result.failureCount());";
         tagString: "bodyTag";
      }
   }
}

class role TestCase {
   taskTitle: "F r a m e w o r k - Locate <#defaultName>.";
   description: "Click <a href="junit/junit/framework/<#name>.html">
      here</a> to learn more about <#name>s.";
}

class role Exception 0..* {
   description: "Locate an exception class.";
   taskTitle: "The task here is to locate an exception that is supposed to be thrown
      by a method to be tested.";
}
```

Results for *TestingExceptions*	
Non-empty lines:	69
Lines to remove:	5
Lines to add or modify:	42
Description or comment modifications:	20
Modifications purely related to role renaming:	1
Functional changes:	42 – 20 – 1 = 21
Extracted part of the pattern counting all additions, removals, and modifications:	(69 – 5 – 42) / 69 ≈ 32 %
Extracted part of the pattern disregarding removals and modifications due to better descriptions and naming:	(64 – 21) / 64 ≈ 67 %

Appendix C: Syntax of the Textual Notation for Specialization Patterns

This appendix is an edited version of appendix A in [Vil01]. It describes the syntax of the textual notation for specialization patterns. The syntax is given in EBNF (Extended Backus-Naur Form). The terminal symbols in the EBNF description below are written in bold. Symbol λ denotes an empty production. For the semantics of this notation please refer to [Vil01].

PATTERN →	pattern PATTERN_NAME ; { ROLE }*
ROLE →	CLASS | METHOD | CONSTR | FIELD | PARAM | ISSUE | CODE_SN
CLASS →	**class role** ROLE_HEADER **{** CLASS_ATTR { ROLE }* **}**
METHOD →	**method role** ROLE_HEADER **{** METHOD_ATTR { ROLE }* **}**
CONSTR →	**constructor role** ROLE_HEADER **{** CONSTR_ATTR { ROLE }* **}**
FIELD →	**field role** ROLE_HEADER **{** FIELD_ATTR { ROLE }* **}**
PARAM →	**parameter role** ROLE_HEADER **{** PARAM_ATTR { ROLE }* **}**
ISSUE →	**issue role** ROLE_HEADER **{** ISSUE_ATTR { ROLE }* **}**
CODE_SN →	**code snippet role** ROLE_HEADER **{** CODE_SN_ATTR { ROLE }* **}**
ROLE_HEADER →	ROLE_NAME CARDINALITY DEPENDENCIES
ROLE NAME →	VARIABLE_DECL
CARDINALITY →	**0..1** | **0..*** | **1..*** | λ
DEPENDENCIES →	**[** DEP_NAME**:** DEP_TYPE { **,** DEP_NAME**:** DEP_TYPE } * **]**
DEP_NAME →	VARIABLE_DECL
DEP_TYPE →	REFERENCE_EXPR
VARIABLE_DECL →	*character string*
CLASS_ATTR →	COMMON_ATTR INHERITANCE DEF_INHERITANCE DEF_KIND DEF_MODIFIERS
METHOD_ATTR →	COMMON_ATTR OVERRIDING RETURN_TYPE DEF_IMPL DEF_MODIFIERS DEF_RETURN_TYPE
FIELD_ATTR →	COMMON_ATTR TYPE DEF_INITIALIZER DEF_MODIFIERS DEF_TYPE
PARAM_ATTR →	COMMON_ATTR TYPE DEF_MODIFIERS DEF_TYPE PARAM_NUMBER
ISSUE_ATTR →	COMMON_ATTR
CONSTR_ATTR →	COMMON_ATTR DEF_IMPL DEF_MODIFIERS
CODE_SN_ATTR →	COMMON_ATTR SOURCE TAG_STRING
COMMON_ATTR →	DEF_NAME DESCRIPTION TASK_ DESCRIPTION TASK_TITLE
DEF_IMPL →	**defaultImplementation : ”** TEXTUAL_EXPR **” ;** | λ
DEF_INHERITANCE →	**defaultInheritance : ”** TEXTUAL_EXPR **” ;** | λ
DEF_INITIALIZER →	**defaultInitializer : ”** TEXTUAL_EXPR **” ;** | λ
DEF_KIND →	**defaultKind : ”** TEXTUAL_EXPR **” ;** | λ
DEF_MODIFIERS →	**defaultModifiers : ”** TEXTUAL_EXPR **” ;** | λ
DEF_NAME →	**defaultName : ”** TEXTUAL_EXPR **” ;** | λ
DEF_RETURN_TYPE →	**defaultReturnType : ”** TEXTUAL_EXPR **” ;** | λ
DEF_TYPE →	**defaultType : ”** TEXTUAL_EXPR **” ;** | λ
DESCRIPTION →	**description : ”***character string* **” ;** | λ
INHERITANCE →	**inheritance :** REFERENCE_EXPR **;** | λ
OVERRIDING →	**overriding :** REFERENCE_EXPR **;** | λ
PARAM_NUMBER →	**parameterNumber :** *character string* | λ
RETURN_TYPE →	**returnType :** REFERENCE_EXPR **;** | λ
SOURCE →	**source : ”** TEXTUAL_EXPR **” ;** | λ
TAG_STRING →	**tagString : ”***character string* **” ;** | λ
TASK_DESCRIPTION →	**taskDescription : ”** *HTML string* **”** | λ
TASK_TITLE →	**taskTitle : ”** *character string* **”** | λ
TYPE →	**type :** REFERENCE_EXPR **;** | λ
TEXTUAL_EXPR →	*character string*
REFERENCE_EXPR →	*character string*

Appendix D: An Example Result of Applying JUnit Specialization Patterns

The classes in the files listed below make up a testing application, which is a specialization of the JUnit framework. The application tests the *junit.samples.Money* class that comes with the JUnit distribution as well as the *junit.samples.Account* that uses a *Money* object to store its balance. (*Account* was written by the author to provide more test material and input for the pattern extraction method described in this thesis.)

The source files listed below are: *AccountExceptionTestCase.java*, *AccountTest.java*, *AllTests.java*, *DelayedTestCase.java*, *MoneyTest.java*, and *MoneyTest.java*. They were all produced by using the automatically extracted and manually modified JUnit annotation described in appendices A and B. Those parts of the code that were added or modified after the automatic code generation are written in bold. Of the total of 119 non-empty and non-commented lines of code there are only 26 (about 24 %) lines that were added or modified. Therefore we can conclude that the code generation provided by Fred and the JUnit annotation was quite effective.

AccountExceptionTestCase.java:

```
package junit.samples.money;

import junit.extensions.*;

public class AccountExceptionTestCase extends ExceptionTestCase {
   public AccountExceptionTestCase (String scriptName, Class exceptionClass) {
      /* #bodyTag */
      super(scriptName, exceptionClass);
   }

   public void testDepositNullException () {
      (new Account("euro")).deposit(null);
   }
}
```

AccountTest.java:

```
package junit.samples.money;

import junit.framework.*;
import junit.extensions.*;

public class AccountTest extends TestCase {
   public AccountTest (String name) {
      super(name);
   }

   public void testDeposit () {
      // Add creation of test objects below:
      /* #createObjects */
      // Add creation of additional objects below:
      /* #createAdditionalObjects */
      // Add calls to additional objects below:
      /* #additionalCalls */
      // Add calls to expected objects below:
      /* #expectedCalls */
      // Add calls to fixture objects below:
```

```
      /* #fixtureCalls */
      aAccount.deposit(aMoney);
      // Add calls to assert methods below:
      /* #assertCalls */
      assertEquals(aMoney, aAccount.getBalance());
   }

   public void testWithdraw () {
      // Add creation of test objects below:
      /* #createObjects */
      // Add creation of additional objects below:
      /* #createAdditionalObjects */
      IMoney balance = aAccount.getBalance();
      Money aMoney = new Money(2, "euro");
      // Add calls to additional objects below:
      /* #additionalCalls */
      // Add calls to expected objects below:
      /* #expectedCalls */
      // Add calls to fixture objects below:
      /* #fixtureCalls */
      aAccount.withdraw(aMoney);
      // Add calls to assert methods below:
      /* #assertCalls */
      assertEquals(balance.subtract(aMoney), aAccount.getBalance());
   }

   private Account aAccount;

   protected void setUp () {
      // Add creation of fixture objects below:
      /*#bodyTag*/
      aMoney = new Money(10, "euro");
      aAccount = new Account("euro");
   }

   private Money aMoney;

   public void testTestWithdrawMultipleTimes () {
      /*#bodyTag*/
      // Give the test to repeat and the number of repeats:
      Test test = new RepeatedTest(new AccountTest("testWithdraw"), 3);
      TestResult result = new TestResult();
      test.run(result);
      assertEquals(3, result.runCount());
   }

   public void testDepositNullExceptionDriverScript () {
      /* #bodyTag */
      ExceptionTestCase test =
                      new AccountExceptionTestCase("testDepositNullException",
                      IllegalArgumentException.class);
      TestResult result = new TestResult();
      test.run(result);
      assertEquals(0, result.failureCount());
   }

   public void testAccountTestInThreads () {
      /*#bodyTag*/
      ActiveTestSuite suite = new ActiveTestSuite();
      /* #addTest */
      suite.addTest(new AccountTest("testGetBalance"));
      suite.addTest(new AccountTest("testWithdraw"));
      suite.addTest(new AccountTest("testDeposit"));
      TestResult result = new TestResult();
      suite.run(result);
   }

   public void testTestDepositWithDelayedTestCase () {
      /* #bodyTag */
      // Give the test to decorate as an argument for the decorator constructor.
      Test test = new DelayedTestCase(new AccountTest("testDeposit"), 5000);
      TestResult result = new TestResult();
      test.run(result);
   }

}
```

AllTests.java:

```
package junit.samples.money;

import junit.framework.*;

public class AllTests {
   public static Test suite () {
      TestSuite suite = new TestSuite();
      // Add tests below:
      /*#bodyTag*/
      suite.addTestSuite(AccountTest.class);
      suite.addTestSuite(MoneyTest.class);
      return suite;
   }

   public static void main (String[] args) {
      /*#bodyTag*/
      junit.awtui.TestRunner.run(AllTests.class);
   }
}
```

DelayedTestCase.java:

```
package junit.samples.money;

import junit.extensions.*;
import junit.framework.*;

public class DelayedTestCase extends TestDecorator {
   private long fMilliseconds;

   public DelayedTestCase (Test test, long ms) {
      /* #bodyTag */
      super(test);
      fMilliseconds = ms;
   }

   public void run (TestResult result) {
      /*#bodyTag*/
      // Do your decoration here:
      try {
         Thread.sleep(fMilliseconds);
      } catch (InterruptedException ex) { ex.printStackTrace(); }
      // Then call super:
      super.run(result);
   }
}
```

MoneyTest.java:

```
package junit.samples.money;

import junit.framework.*;

public class MoneyTest extends TestCase {
   public MoneyTest (String name) {
      super(name);
   }

   public void testSimpleAdd () {
      // Add creation of test objects below:
      /* #createObjects */
      Money expected = new Money(26, "CHF");
      // Add creation of additional objects below:
      /* #createAdditionalObjects */
      // Add calls to additional objects below:
      /* #additionalCalls */
      // Add calls to expected objects below:
```

```
      /* #expectedCalls */
      // Add calls to fixture objects below:
      /* #fixtureCalls */

      // Add calls to assert methods below:
      /* #assertCalls */
      assertEquals(expected, f12CHF.add(f14CHF));
   }

   public void testSimpleSubtract () {
      // Add creation of test objects below:
      /* #createObjects */
      Money expected = new Money(2, "CHF");
      // Add creation of additional objects below:
      /* #createAdditionalObjects */
      // Add calls to additional objects below:
      /* #additionalCalls */
      // Add calls to expected objects below:
      /* #expectedCalls */
      // Add calls to fixture objects below:
      /* #fixtureCalls */
      // Add calls to assert methods below:
      /* #assertCalls */
      assertEquals(expected, f14CHF.subtract(f12CHF));
   }

   protected void setUp () {
      // Add creation of fixture objects below:
      /*#bodyTag*/
      f7USD = new Money(7, "USD");
      f21USD = new Money(21, "USD");
      f14CHF = new Money(14, "CHF");
      f12CHF = new Money(12, "CHF");

      fMB1 = new MoneyBag(f12CHF, f7USD);
      fMB2 = new MoneyBag(f14CHF, f21USD);
   }

   private Money f12CHF;
   private Money f14CHF;
   private Money f7USD;
   private Money f21USD;
   private MoneyBag fMB1;
   private MoneyBag fMB2;
}
```

Helsinki 2002
Yliopistopaino
Pikapaino