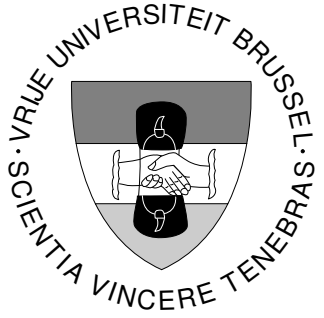


Vrije Universiteit Brussel
Programming Technology Laboratory
Faculteit Wetenschappen - Departement Informatica

Automated Support For Framework-Based Software Evolution

Tom Tourwé

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Promotor: Prof. Dr. Theo D'Hondt
Co-Promotor: Dr. Tom Mens

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my promotor, Prof. Dr. Theo D'Hondt, for his unconditional support and guidance during the last five years, for granting me the opportunity to obtain a PhD, for allowing me to freely investigate whatever topic I deemed interesting, and for wanting to promote my work.

I am also greatly indebted to Dr. Tom Mens. Without him, this dissertation would probably not have seen the light. He provided me with a topic to investigate, was always ready to discuss important issues, read probably each and every draft of this text, and still provided valuable comments each time. I couldn't have wished for a better co-promotor. Thanks, Tom!

Johan Brichau, Gerrit Cornelis, Sofie Goderis, Kris Gybels, Carine Lucas, Kim Mens and Werner Van Belle all read various drafts of this dissertation, and provided valuable comments and suggestions. If it was not for them, this text would probably be even less readable, so thank you very much!

Thanks also to all my other colleagues at the Programming Technology Lab, for making it a fun and inspiring place to work. I hope they will forgive me for neglecting many of my organizational tasks, and I promise that there will be a constant supply of candy bars in the future.

I also want to thank my friends, Birgit, Gerrit, Stef, Arn, Koen, Katleen, Tine and Piet, for their support during these last few months, for being there, for making me laugh and allowing me to relax, and for constantly asking when my dissertation would finally be finished. Withouth such extreme pressure, I would probably still be writing.

And, of course, I would like to thank my family, for giving me the opportunity to study, for allowing me to pursue whatever degree I wanted, and for supporting me in various ways, too much to mention on one page.

Chapter 1

Introduction

The topic of this dissertation is the instantiation and evolution of object-oriented frameworks. We propose to document important information about the framework's design explicitly and formally, to allow tools to use this information in an active way to support developers while performing their task. Such support includes providing high-level automated transformations to instantiate and evolve a framework correctly, detecting and avoiding design drift within the framework's implementation and assessing the impact of evolution on the framework and its existing instances.

1.1 Thesis Statement

Object-oriented frameworks are often touted for leveraging the promise of large-scale reuse [Fis87]. Traditional class libraries and toolkits only allow reuse of individual classes and methods. Frameworks, on the other hand, allow reuse of the design of a framework and enable developers to construct applications by providing application-specific code at predefined places. As such, applications can be developed much faster and with minimal effort [FS97]. Moreover, they are less prone to errors and look more consistent to the end user, since they are based on the same "core" implementation [MB99, MN96].

While it is widely acknowledged that such *framework-based* development has some important advantages, it also suffers from a number of important problems:

- building a concrete application, and thereby reusing the design of the framework, is a difficult task. The developer responsible should have a good understanding of the inner workings of the framework and of its design to construct an application with the desired behavior that fits into this design. Due to missing, inappropriate or outdated framework documentation, such information is not always available [BD99, Ret91, BGK98].

- frameworks are subject to evolution, just like any other software system, due to new or changing user requirements and bug fixes [DH98, RJ96, JR91]. Such evolution may change the design of the framework, upon which applications depend. As a result, these applications may not adhere to the new design, and may no longer exhibit the desired behavior.

- frameworks are developed, maintained and instantiated by a team of developers. Developers may thus make changes to the framework independently of one another. Naturally, all such changes should be merged into one single version. However, changes made by one developer may rely on assumptions that are broken by changes made by another developer. When merging such conflicting changes, the resulting version of the framework will be inconsistent and may contain errors [Men02].

In this dissertation, we will tackle the above problems and seek to alleviate them, by providing automated support for framework-based development. This is expressed in our thesis statement:

Thesis. *Elaborate automated tool support for framework-based software development can be provided by explicitly documenting a framework's design in a formal way.*

Framework-based software development includes many different activities, the most important of which are implementing, evolving and instantiating the framework. In this dissertation, we mainly focus on the evolution and instantiation activities. We do acknowledge the fact that implementing a framework is an inherently difficult task. However, implementing a high-quality framework is an iterative process, which inevitably requires a number of successive evolutions. Therefore, evolving a framework necessarily forms part of implementing it. On the other hand, instantiation and evolution of a framework is far from trivial as well, and the problems associated with such activities are specific to the framework-based software development process. Moreover, the amount of time spent instantiating and evolving a framework is far greater than the time spent implementing it [Som96]. These are thus the activities that would benefit the most from automated support.

The support for framework-based development we envision includes (semi-)automatic instantiation and evolution of the framework, where a developer is guided interactively by a tool while he performs the necessary changes. The guidance we offer for these activities ranges from pointing out which changes a developer needs to make, automatically updating parts of the documentation, notifying him when he violates the intended design of the framework or its applications, assessing the impact of a particular change and detecting possible conflicting changes.

The design information that we rely upon for our approach should be used in an active way. A tool should be able to use this information to provide specialized guidance and should have access to the source code of the framework to check the information, reason about and manipulate it, as appropriate. An absolute prerequisite for enabling such tool support is that the provided information is specified in a formal way and allows us to define exactly and precisely what the effect of a particular change upon the implementation is. This ensures the information is specified in an unambiguous, accurate and concise manner. Furthermore, the information should be specified explicitly, since it is often impossible to extract it automatically or it is simply not available in the implementation.

As is stated explicitly in our thesis statement, our approach to support framework-based development relies on information about the design of the framework. Such information includes the responsibilities, relationships and collaborations between class hierarchies, classes and methods, as well as information about the specific ways in which the design can be reused, extended and changed.

1.2 Scope of the Dissertation

Because our thesis statement is quite general, it would be impossible to prove it entirely in this dissertation. Therefore, we will incorporate some restrictions in order to reduce the scope of the dissertation.

First of all, frameworks are subject to anticipated as well as unanticipated evolution. Therefore, any approach that supports framework-based software development should support both kinds of evolution. In the first part of this dissertation, we will focus on supporting anticipated evolution only. In later chapters, we will suggest how support for a particular kind of unanticipated evolution can be supplied as well, by incorporating refactorings into our approach. Studying how full-fledged support for unanticipated evolution can be provided based on our approach is considered future work, however.

Second, we will first analyze how we can exploit design information for supporting framework-based development. Other sources of information, such as architectural and domain knowledge, could be useful as well for this purpose, but will not be considered initially. This is motivated by the fact that using architectural knowledge to support evolution has already been studied elsewhere [Men00, Flo00, Min97, MNS95], and that we believe that domain-specific knowledge

can only be used to complement generally applicable knowledge, and not instead of it.

Last, we restrict ourselves to the domain of object-oriented application frameworks at first. The techniques and tools presented in this dissertation should however be applicable in other domains as well, such as product line architectures [Bos00] and component-oriented programming [Szy97], for example. An in-depth discussion and assessment regarding this issue is outside the scope of this dissertation however, and can be considered future work.

1.3 Approach and Contributions of the Dissertation

1.3.1 Approach

The approach we will take to prove our thesis is the following.

We will construct an environment that allows a developer to document a framework's design by means of the design patterns it uses. While we could employ other high-level descriptions of the framework, we selected design patterns for several reasons: they are generally applicable, well-documented, well-understood and commonly used. The environment will use design pattern information in an active way, to

- check the conformance of the framework's implementation to its intended design
- provide high-level transformations that perform changes to the framework (semi-)automatically and guide a developer when instantiating or evolving the framework
- assess the impact of these transformations on the framework itself and its derived applications

Furthermore, the environment will make sure that this documentation is kept up to date automatically. To achieve all this, the environment will be integrated into a standard development environment and use the technique of *declarative meta programming* [DV98, Men00, Wuy01]. As such, it has access to the framework's source code and can reason about and manipulate this code and its documentation as necessary.

As an underlying basis for this environment, we will use a formal model that allows us to describe design patterns in an unambiguous, accurate and concise manner. The model will be based upon an advanced abstraction of design patterns, called *metapatterns* [Pre95, Pre97, Pre94], to ensure its scalability and manageability. Transformations will be defined upon this model, that perform changes to the implementation of these design patterns, and thus to the framework's design, while at the same time, they ensure the appropriate design constraints remain satisfied and the documentation is up to date. These transformations can be combined to form higher-level transformations that correspond to typical evolution and instantiation transformations applied to a framework. Based on this formal model and associated transformations, we will formally define change propagation and software merging algorithms. These enable us to assess the impact of the transformations on the framework and its existing instances, detect possible merge conflicts and suggest ways in which these can be resolved.

We want to stress that the main focus of our dissertation lies in the area of *language engineering* as opposed to the *software engineering* area. To support various aspects of the software engineering process, a suitable environment with a solid formal basis is an absolute prerequisite. This is exactly what we will focus on first. Only when we have such an environment at our disposal will we be able to focus fully on the software engineering aspects. Note that by no means our focus on *language* engineering implies that we will introduce a new programming language or add new language features to an existing programming language. First of all, introducing a new programming language would prohibit the use of our approach on already existing frameworks. Second, introducing new language features into an existing programming language would restrict the kind of information we can use to that exposed by those features, and would thus sacrifice flexibility.

1.3.2 Contributions

The main contributions of this dissertation can be divided into two major groups.

The *scientific/fundamental* contributions are the following:

- we provide a formal framework for the definition of *fundamental metapatterns*, that allows us to define all of Pree's metapatterns [Pre95], as well as a number of other useful metapatterns in a formal way. As metapatterns are an abstraction of design patterns, this formal definition at the same time serves as a basis to define design patterns formally.
- for each of these fundamental metapatterns, we explicitly define their associated constraints, which allow us to check whether the actual framework implementation conforms to its intended design, and vice versa.
- we identify a small set of primitive transformations that can be applied to these fundamental metapatterns. Much like refactorings, these primitive transformations can be combined to form higher-level transformations, that can be used to instantiate and evolve a framework.
- we define change propagation and software merging algorithms, that are based on the formal metapattern model and its associated transformations, and that can be used to assess the impact of a particular change upon the framework and its existing instances.

Additionally, we also provide a number of *practical* contributions:

- we present an environment in which this formal model is implemented, that provides elaborate support for framework-based development based on this model and that is integrated into a standard development environment.
- we show how a declarative meta-programming environment can be used to implement sophisticated tools and techniques that support various software development activities.
- we identify previously undocumented design problems in HotDraw, a popular and well-known framework that we used to perform our experiments.

1.4 Research Context

The research reported upon in this dissertation forms part of a larger research effort conducted at the Programming Technology Lab. The main goal of this research is to build sophisticated tools and techniques that support the software development process in all of its aspects.

Several people involved in this research use the emerging technique of *declarative meta programming* to build state-of-the-art programming environments:

- In his dissertation on "Type-Oriented Logic Meta Programming" [DV98], Kris De Volder proposes to use a logic meta-programming language (called *TyRuBa*) to extend the expressiveness of current type systems. He advocates the active usage of types, to enable conditional implementation of methods and automatic positioning of abstract code, for example. His approach is based on representing a program in terms of high-level descriptions and low-level pieces of Java source code. An intelligent code generator assembles all descriptions and generates appropriate Java code automatically. As it turned out, the approach was general enough to describe *aspect-oriented* programming as well [DV99].
- Kim Mens uses the declarative meta-programming approach in his dissertation [Men00] to verify the conformance of an implementation to its intended architecture automatically. He achieves this by taking advantage of the powerful features of a logic programming language, to define an expressive architectural language for declaring architectures and their mapping onto implementation artifacts. Based on this high-level architecture language, he defines a conformance checking algorithm and provides a prototype implementation for it.

- In Roel Wuyts' dissertation [Wuy01], declarative meta programming is used to support the co-evolution of design and implementation. He defines design as an abstraction of the implementation and expresses design as a logic program. By integrating the logic meta-programming language into a standard development environment, he is able to implement a synchronization framework that keeps the design and the implementation of a system synchronized automatically.

- Johan Brichau builds upon the work of Kris De Volder, and uses declarative meta-programming to tackle the problem of aspect combination in aspect-oriented programming [Bri00]. Aspects are described as a logic program, together with a specification of how different aspects can be combined. Code generators are then defined that make use of such specifications to effectively weave different aspects into the source code of a system.

The technique of declarative meta programming has also been used to reason about the (static) structure of object-oriented programs [Wuy98], and to define architectural transformations that allow significant optimization of object-oriented programs [TDM99]. As we will show in this dissertation, declarative meta programming can also be used to implement an environment that supports framework-based development. The technique of declarative meta programming, and the specific reason why it is very well suited for our purposes, will be discussed in more detail in Chapter 6.

Other people have exploited other interesting techniques:

- In [Luc97], Carine Lucas introduced *reuse contracts* as structured documentation to support the evolution of reusable components. Reuse contracts document structural dependencies between components in an object-oriented system. This provides reusers of these components with crucial information about their operational behavior. Furthermore, reuse contracts can only be composed or adapted by means of certain predefined *reuse operators*. These enable reusers to explicitly document the assumptions they make about the components they reuse. As a result, applications are more robust to change, since the documentation allows us to verify whether these assumptions are broken when changes are made. Similarly, explicitly documenting where the general design of an application is not respected helps in assessing where co-operation with this part might cause problems.

- Koen De Hondt's dissertation [DH98] introduced the notion of *software classification* as an approach to architectural recovery in evolving object-oriented systems. He defines a classification as a grouping of related source code artifacts (although classifications can also contain other classifications). A classification may be assembled manually by a developer, or may have been computed automatically (based on method tagging, for example). Once such classifications become explicit in the software development environment, they can be exploited in subsequent software development activities. Applications of software classification are expressing multiple views on software, recovery of collaboration and reuse contracts and management of changes, for example.

- In his dissertation [Men99], Tom Mens introduced a domain-independent definition of the reuse contract approach, and illustrated how it can be used to provide support for software merging. He proposes to represent software artifacts by graphs, and evolution by means of graph rewriting. By relying on the formal properties of graph rewriting, such as the notion of parallel dependence, he can formally define which pairs of modification operations (formally represented as graph productions) yield an evolution conflict.

1.5 Outline of the Dissertation

In Chapter 2 we introduce the notions of object-oriented frameworks, design patterns and meta-patterns. Readers familiar with these topics can safely skip the corresponding sections. We also provide an overview of related work on the topic of the dissertation.

In Chapter 3, we will study a small-scale framework that we implemented and that serves as a proof of concept throughout the dissertation. We will discuss the design patterns that this framework uses and show that they expose important and valuable information. Moreover, we will illustrate that evolving even such a small framework is far from straightforward, and explain the various problems that a developer is faced with. At the same time, we will discuss how information about design patterns can facilitate this task.

In Chapter 4, we summarize the problems associated with framework-based development we want to tackle, based on the discussion in Chapter 3. We propose an approach to overcome these problems, that uses design patterns as an explicit documentation technique and metapatterns as a basis to ensure the scalability and extensibility of our approach.

In Chapter 5, we provide a formal definition for metapatterns, that allows us to describe accurately and precisely a metapattern's participants, its relations and collaborations, as well as the constraints it imposes upon the implementation. Furthermore, we formally define transformations that automatically make changes to these metapatterns, while at the same time ensuring the metapattern constraints remain satisfied. Together, the metapattern descriptions and the transformations allow us to define how changes propagate through the framework.

In Chapter 6, we show how this formal model can be integrated into an environment that supports framework-based development and how it can be used in practice. We will illustrate how the example Scheme framework can be documented explicitly by means of the design pattern instances it uses, and how such information allows us to provide support for instantiating as well as evolving the framework.

In Chapter 7, we illustrate how support for software merging can be provided by means of the metapattern transformations and refactorings. We will formally define the conditions that give rise to merge conflicts when the transformations and/or the refactorings are applied in parallel. This not only allows us to detect such conflicts, but also enables us to state how such conflicts can possibly be resolved.

In Chapter 8, we perform some initial experiments on a real-world framework in order to validate our approach. We will document the HotDraw framework, and show how support is provided for its instantiation. We will also consider how the HotDraw framework evolved and discuss how this impacted its derived applications. Our choice for HotDraw is motivated by the fact that we need a controlled setting in which to perform some preliminary analysis of the strengths and weaknesses of our approach. Only in a later stadium will we be able to validate on industrial case studies. Nonetheless, despite the fact that HotDraw is a widely used, well studied and extensively documented framework, we will show that the developers responsible for its evolution failed to notice various design conflicts and inconsistencies, that can be identified by our approach.

In Chapter 9, we summarize the contributions and conclusions of this dissertation and discuss some interesting issues that remain to be investigated.

Chapter 2

Preliminaries

In this chapter, we provide some background information on the topic of this dissertation and discuss some work that is related to it. We first introduce the notion of (object-oriented) frameworks and design patterns. Then, we present an overview of related work in the area of documentation, instantiation and evolution of object-oriented frameworks and we discuss metapatterns.

2.1 Framework Terminology

This section introduces the terminology that will be used throughout the rest of the dissertation. We will clarify what we mean by a framework, its design, instantiation and evolution and distinguish between the different developers that work with a framework. Readers that are already familiar with those concepts can safely skip this section.

2.1.1 Definition of a Framework

Frameworks have been gaining widespread acceptance in the last decade and are still increasing in popularity. To understand why this is the case, we should first define what a framework is. Many definitions of the concept of a framework have been proposed in the literature [JF88, JR91, Tal94, Szy98, Bra98], and each one is equally valid. For the purpose of this dissertation, we use the definition found in [JF88]:

Definition 1 (Framework) *A framework is a set of classes that embodies an abstract and reusable design for solutions to a family of related problems in a particular domain.*

This definition clearly states that frameworks are specific to a particular domain. Over the years, many different frameworks have been developed for a myriad of domains, including graphical user interface frameworks [WGM89, WRS90, Gol00], graph-based editors [Bra95] and network servers [Mic96], to mention only a few. A good framework captures important knowledge about the domain with the specific intent of reusing this knowledge across different applications. This reuse is achieved by carefully analyzing the domain and identifying the parts that are common to all applications and the parts that differ. The common parts are referred to as the *commonalities* of the applications, while the differing parts are called the *variabilities*. A framework captures these commonalities and thus allows applications to reuse the common parts. Additionally, the framework specifically provides support for the variabilities, by defining *hooks* that allow applications to provide their own specific behavior.

Furthermore, the definition states that a framework offers a particular *design* that is *abstract* and that can be *reused* by many different applications. This is an important consideration, as it means that frameworks allow reuse beyond the implementation level. As a result, we can develop an application without the need to design it from scratch, which is a significant advantage: applications can be developed much faster, are less prone to errors and look more consistent to

the end user, since they are all based on the same "core" implementation. This is exactly what makes framework-based development so attractive.

2.1.2 Definition of Design

The *design* of a framework is defined in the following way [GHJV94]:

Definition 2 (Framework Design) *The design of a framework contains its partitioning into classes and objects, the key responsibilities of these classes, their interfaces, the structural and behavioral relationships between them and the overall thread of control.*

In fact, the design translates the commonalities and variabilities of the domain into so-called *frozen spots* and *hot spots* respectively [Pre95]. The frozen spots implement those parts of the framework that are fixed and need not be changed by specific applications. The hot spots on the other hand, define those parts where particular applications can specify their own implementation in order to tailor the framework to their specific needs. Typically, hot spots are implemented by abstract classes and abstract methods. An abstract class captures an important concept of the domain and defines the appropriate interface for it. It defers the implementation of application-specific behavior to the appropriate subclasses by providing the necessary abstract methods but may also implement default behavior if this is possible.

The design also predefines the structural and behavioral relationships between the classes present in the framework. The former are concerned with the mutual relation between specific classes. They specify for example whether inheritance, aggregation or a combination of both should be used. The behavioral relationships are more concerned with the runtime behavior of the application. They focus on the way objects of specific classes should be composed, for example.

Furthermore, the design captures the overall thread of control of an application by specifying how the (abstract) classes interact and collaborate based on the interfaces they provide. Applications can provide their own specific behavior by deriving concrete classes from the abstract classes and overriding the key (abstract) methods, thereby filling in the hot spots. These application-specific methods will eventually get called by the framework when the application is running. Observe the pronounced contrast with applications built using a class library. The methods that are provided by the class library are called by the application code, whereas in this case, the methods of the application code are called by the framework. Frameworks thus achieve what is called *inversion of control*, which is often referred to as the *Hollywood Principle*: "Don't call us, we'll call you" [PC95].

2.1.3 Instantiation of a Framework

A framework only offers an abstract design for an application. In order to build a concrete application, the design (and the corresponding implementation) needs to be completed first. This activity is called *instantiating* the framework:

Definition 3 (Framework Instantiation) *Deriving an application from an existing framework is called* instantiating *(or* specializing*) that framework. A concrete application derived from a given framework is called an* instance*, an* instantiation *or a* specialization *of that framework.*

A framework is instantiated by providing it with the required application-specific behavior. Typically, this behavior is added to the framework by filling in the appropriate hot spots. Most of the time, this boils down to deriving concrete classes from the appropriate abstract classes and overriding their abstract methods. The structural and behavioral relationships defined by the framework and the overall thread of control can thus be reused. As such, building a concrete application through a framework requires a minimum of effort, which is one of the important advantages of framework-based development.

Note that the design of an application is dictated by the framework from which the application is derived, since it is the design of the framework that is actually reused. When instantiating

a framework, care should thus be taken that the application does not break the design of the framework, since this can possibly result in code duplication, inconsistencies or unwanted behavior. Thus the framework actually constrains the design of the application.

2.1.4 Evolution of a Framework

As a framework is reused to build concrete applications, certain shortcomings in its design may be discovered. For example, the design may not be suited to allow certain extensions easily, or it may not incorporate the appropriate abstractions needed by most applications in the domain. Moreover, new requirements may need to be incorporated into the framework and it may turn out that the design of the framework obstructs a smooth integration. To cope with these situations, the design of the frameworks should be adapted. This activity is called *evolving* the framework:

Definition 4 (Framework Evolution) *Each change that is made to the framework's design is considered an* evolution *of that framework.*

Evolving a framework is a difficult task. Care should be taken that important relationships and collaborations between classes and methods that should be preserved, remain intact. Furthermore, applications derived from the framework make assumptions about the framework's design, since they build on the abstract classes and methods of the framework. If the design is changed, these assumptions may no longer be valid. Evolving a framework will thus have an impact on all applications derived from it. This impact has to be assessed and the necessary changes to the applications need to be applied as well. Observe also that changes to the design of the framework inevitable require changes to the implementation as well, since design is merely an abstraction of the implementation [Wuy01]. Not every change to the implementation is a change to the design however. For example, changing the name of a temporary variable can hardly be called a change to the design of a framework, and is thus not considered as an evolution of it.

We can distinguish between two different types of evolution: *anticipated* and *unanticipated* evolution. As its name suggests, anticipated evolution occurs when the original developer of a framework has anticipated the changes a framework may be subject to, and has made special provisions so that these changes can be integrated smoothly into the existing design. Conversely, unanticipated evolution occurs whenever the changes that should be applied to a framework were not foreseen by its original developers. Such changes are thus hard to integrate into the existing design, and often require preparatory changes first.

A developer can evolve a framework by hand or with the help of a tool that guides him while performing his task [RBJ97, FMvW97, O'C01]. We will refer to the former approach as *manual evolution*, while we will consider the latter approach *supported evolution*. When evolving a framework manually, it is the developer's responsibility to make sure the appropriate design constraints are still adhered to after the changes. If this is not the case, the framework's design may deteriorate. After a couple of such erroneous changes, the framework's implementation drifts away further from the intended design. This phenomenon is referred to as *design drift* [vGB01]. Moreover, the developer should also be able to assess the impact of his changes on the existing framework instances. With supported evolution, the developer uses a tool that helps him to make the appropriate changes. The tool may supervise the developer to make sure he does not violate particular design constraints, can provide parts of the necessary implementation automatically and can point out where framework instances may be impacted, for example. In this way, the risk of design drift is alleviated to a large extent.

An important thing to observe is that our definition of evolution allows instantiation to be seen as a special kind of evolution. When instantiating a framework, its design is changed because classes and methods are added to its implementation. Furthermore, both instantiation and evolution should ensure that the appropriate design constraints are preserved. The main difference between evolution and instantiation is that the latter operation does not change any of the framework's design constraints, whereas the former does.

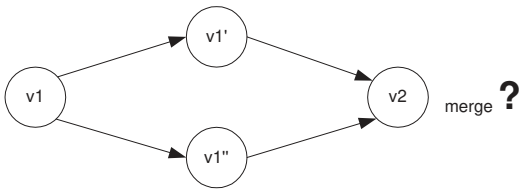


Figure 2.1: Merging parallel changes

2.1.5 Software Merging

A framework is a large and complex software artifact, which is often developed by a whole team of developers. This can give rise to a situation where two (or more) developers evolve the framework at the same time, independently of one another.The result is that two versions of the framework exist, each one incorporating only one particular evolution. Such a situation should be avoided, as it leads to a proliferation of different versions of the framework that can quickly become unmanageable [CDHSV97]. To overcome this problem, the different versions of the framework should be merged into one single version (see Figure 2.1) [SLMD96, Men99, Men02].

Software merging is a time-consuming, complicated and error-prone process, because many interconnected elements are involved and because merging depends on both the syntax and the semantics of these elements. Moreover, *merge conflicts* may occur, due to the fact that parallel changes may involve the same artifact.

Definition 5 (Merge Conflict) *A* merge conflict *occurs when two (or more) changes are applied in parallel, and one change breaks the assumptions made by other changes*

To help a developer in merging different versions of the software and detecting possible merge conflicts, a number of different merge algorithms exist [Men02]:

Textual merging With textual merging, software artifacts that should be merged are considered as mere text and are compared to one another. The most common approach is to use *line-based* merging, which detects common text lines in parallel modifications, as well as text lines that have been inserted, deleted, modified or moved.

Syntactic merging also takes the syntax of software artifacts into account. A merge tool that uses syntactic merging will only report conflicts when the merged result is not syntactically correct with respect to the programming language's syntax. Such conflicts are normally detected by an ordinary compiler

Semantic merging approaches detect conflicts when the merged result is syntactically correct, but behaves in unforeseen ways. Such behavioral conflicts can only be detected by taking into account the run-time semantics of the code.

Structural merging arise when one of the changes to a software artifact is a restructuring (or a refactoring) and the merge algorithm cannot decide in which way the merged result should be structured.

Moreover, we can distinguish between state-based and change-based merge techniques that can be used to detect syntactic, semantic of structural merge conflicts. With state-based merging, only the information in the original version and/or its revisions is considered during the merge. In contrast, change-based merging additionally uses information about the precise changes that were performed during evolution of the software. Operation-based merging is a particular flavor of change-based merging, that models changes as explicit operations or transformations. These evolution operations can be arbitrarily complex,. With operation-based merging, we do not need to compare the parallel versions of a software artifact entirely, since it suffices to compare only the

evolution operations that have been applied to obtain each of the versions. For certain combinations of operations, conflicts can be reported. Operation-based merging is general in the sense that it can be used for detecting syntactic, structural and even some semantic conflicts. In Chapter 7, we will elaborate on the issue of software merging in more detail.

2.1.6 Framework Developers

Framework development typically includes two separate activities: developing/evolving the framework and instantiating it. Although both activities can be performed by the same developers, we prefer to distinguish between two kinds of developers: *framework developers* and *application developers*:

Definition 6 (Framework Developers) *The* framework developer *designs and implements the framework and creates the skeleton for the applications that are built upon the framework. The* application developer *instantiates the framework to build concrete applications.*

The framework developer is responsible for designing and implementing the framework. Therefor, the framework developer is often an expert in the domain for which the framework is built, since he needs to be able to identify the appropriate commonalities and variabilities. Furthermore, a framework developer should possess excellent programming skills, to ensure a flexible, extensible and reusable design. Framework developers are often also responsible for documenting the framework and explaining how it should be instantiated. They have intimate knowledge of the framework and its internal workings, and should communicate this information to the application developers. Due to this fact, they are also in charge of evolving the framework, when required.

An application developer builds concrete applications by instantiating the framework, by filling in the appropriate hot spots with the application-specific behavior. Application developers typically don't need to be experts in the domain, as they can just reuse the framework and the knowledge incorporated in it. Furthermore, instantiating the framework only requires information about the specific hot spots, so an application developer typically has little knowledge of the internal workings of the framework. As a consequence, they are normally not responsible for evolving it. However, they do need to be able to evolve their applications appropriately in order to resolve conflicts due to the evolution of the framework itself. In practice, however, application developers often also evolve the framework, since they know the functionality that is missing and may want to incorporate it themselves. However, since they lack the appropriate knowledge, the risk of introducing errors and inconsistencies in the framework design and implementation is very high.

2.2 Introduction to Design Patterns

One of the major contributions of a framework is its abstract and reusable design. Formulating such a design is a very hard task. The framework developer needs to identify the classes that need to be defined, their individual responsibilities, the mutual relationships between them, the interface each of them needs to implement and the way their instances collaborate. Furthermore, in order to achieve the required level of reusability, the developer also needs to recognize which parts of the framework could possibly be reused across different applications and needs to implement these parts as flexible as possible. It is clear that this is not at all straightforward.

Successful designs do exist, however, and much can be learned from them. The particular problems the design addresses and the specific solutions it offers should be studied so as to build a library of tricks and techniques for solving related problems. In fact, this is what distinguishes experienced software developers from inexperienced ones: due to the knowledge they acquired over time, experienced software developers are often able to implement a good solution for a particular problem by adapting a solution already applied for a similar problem.

In recent years, much attention has been paid to studying good designs with the particular aim of building a library of flexible and reusable solutions for a range of problems that frequently

occur when designing software. This resulted in *design pattern catalogs* [GHJV94, BMR+96, CS95, MM97, Lea96, ABW98], that are put together by experienced developers with the aim of communicating successful and flexible designs for problems that occur over and over again. In the remainder of this section, we first present a brief overview of design patterns in general (for a more detailed discussion, we refer to [GHJV94]). In the remainder of the dissertation, we will make use of design patterns to document the design of an existing framework, and show how the information they convey can be used to provide support for framework-based development.

2.2.1 Defining and Describing Design Patterns

A pattern, in the broadest sense of the word, describes a successful and reusable solution for a problem in a particular context that appears over and over again [Ale79]. A *design pattern*, which is a particular flavor of a pattern, then forms a description of communicating objects and classes that are customized to solve a general design problem in a particular context [GHJV94]. It focuses on the relationships between classes and the way their objects interact in order to achieve a certain behavior in a reusable and flexible way. A design pattern *instance* is a particular occurrence of a design pattern in a software system, just as an object is a particular instance of a class.

A design pattern generally consists of four essential elements:

- The *pattern name* is used to identify the design pattern. Naming a design pattern allows us to talk about a design at a higher level of abstraction and enables us to define a common design vocabulary. As such, it makes it easier to communicate the design and its particular benefits and trade-offs to the developers. This is important with respect to the documentation, for example.
- The *problem* describes the specific circumstances under which the design pattern should be applied. It explains the problem and the context in which it occurs in detail. This can be done by providing simple examples as an illustration or by including a list of conditions that should be met before the design pattern can be applied.
- The *solution* describes the solution the design pattern proposes to solve the given problem. It does this in terms of classes and methods, their mutual relationships, responsibilities and collaborations. The solution does not describe a concrete design, let alone an implementation, since it must be applicable in many different circumstances. It only describes a template for the specific design and implementation.
- The *consequences* are the particular benefits and trade-offs that result when applying the design pattern. This is an important factor to consider when looking for a particular design pattern and considering various design alternatives. Often, the consequences will involve the space and time trade-offs of the design pattern, as well as specific language and implementation issues. The consequences also include the impact of the design pattern on the flexibility, reusability and extensibility of the design.

Design patterns are documented in design pattern catalogs [GHJV94, BMR+96, CS95, MM97, Lea96, ABW98], of which [GHJV94] is probably the most well-known. These catalogs each describe the design patterns in a consistent format, which includes various sections that provide detailed information on the essential elements presented above. An important property of this format is that it does not only focus on the problem the design pattern tries to solve. This would only capture the end result of the design process, as a description of the classes and their relationships. To be able to fully understand, communicate and reuse the design, it must also describe the various decisions, alternatives and trade-offs that led to it. The format therefore consists of the following sections:

Pattern Name and Classification The name of the design pattern under consideration.

Also Known As This section lists the other names by which this design pattern also goes, if there are any such names.

Intent The intent briefly describes the specific purpose of the design pattern and provides an answer to the question which design problem the design pattern addresses.

Motivation The motivation section often contains a concrete example of the design problem and shows how the class and object structures in the design pattern can solve it.

Applicability The applicability section lists the circumstances in which the design pattern can be applied and provides hints on how to recognize these situations.

Structure This section consists of a graphical representation of the participants of the pattern and the relationships between them.

Participants A list of the classes, objects and methods that participate in the design pattern, together with a description of their primary responsibilities.

Collaborations A description of how the participants of the design pattern collaborate to carry out their responsibilities and achieve a certain behavior.

Implementation The implementation section explains which techniques and language features can be used to implement the design pattern's solution. If there are any language-specific issues, they are also addressed.

Sample Code This section contains sample code fragments that show how a design pattern's solution could be implemented in a particular programming language.

Consequences This section discusses the various trade-offs and results of using the design pattern and explains how the design pattern makes the design more flexible and reusable.

Known Uses This section contains examples of the use of the design pattern in real-world systems.

Related Patterns This section discusses other design patterns that can possibly also be used to solve the given problem and explains the primary differences between them. It also mentions how this design pattern can be combined with other design patterns.

As can be seen, a lot of useful and important information is conveyed within this template. It describes under which circumstances the design pattern can be used to solve a particular problem and it mentions the particular advantages and disadvantages of using the design pattern, in order to allow estimating its costs and benefits. The solution is described in terms of classes and their objects, their specific roles, relationships and collaborations and the distribution of responsibilities amongst them. Some of this information is expressed in natural language (the *Intent, Motivation* and *Applicability* sections, for example), while other information is presented in a more formal way (such as in the *Structure, Participants* and *Sample Code* sections).

2.2.2 Design Pattern Benefits

Design patterns can help in solving many of the problems developers face when designing software. In the following paragraphs, we will shed a light on how they achieve this.

- *Design patterns define a common vocabulary among developers.* Naming a design pattern allows developers to communicate the design of a framework at a higher level of abstraction. Developers can talk about a design with other developers and in the documentation, without bothering about the specific implementation details. Having a common vocabulary makes it easier to think about designs and to communicate them and their benefits and trade-offs to others.

- *Design patterns help in identifying abstractions and the objects that capture them.* In order for a framework to be successful, it has to provide the appropriate abstractions for its domain, so that they can be reused by applications. Identifying these abstractions, defining the classes that represent them, together with their respective responsibilities is thus an important task of a framework developer. This is not at all trivial, since many factors come into play: the flexibility, efficiency, reusability, etc of the classes has to be determined. Often, an object-oriented framework is modelled after the real world, and objects found during the analysis phase are translated into design. Not all objects do have a counterpart in the real world, however. Design patterns help in identifying such less obvious abstractions and the objects that can capture them.

- *Design patterns help to determine the right granularity for objects.* Objects can vary greatly in size and number. Low-level entities, such as arrays, as well as high-level entities, such as complete applications, can be represented by an object. It is the developer's task to determine the appropriate granularity. Design patterns can help here, by describing how to represent complete subsystems as objects, how to support huge numbers of objects at the finest granularities, how to decompose objects into smaller ones and how to distribute responsibilities appropriately.

- *Design patterns help in defining appropriate interfaces.* The framework developer is responsible for defining the appropriate interface for an object. A good interface is the key to flexibility and reuse, since late binding depends upon it: objects with the same interface can be substituted for one another. Design patterns help defining interfaces by identifying key elements and the kinds of data that get sent across an interface. They also specify relationships between different interfaces, require classes to have the same interface or place constraints on the interfaces of some classes.

- *Design patterns help in specifying actual object implementations.* They explain the circumstances under which class inheritance should be used instead of interface inheritance, for example. They promote programming to an interface instead of an actual implementation, which makes a framework independent of the concrete objects it uses and reduces implementation dependencies. Furthermore, they show how important object-oriented features, such as inheritance, late binding and polymorphism, and powerful tricks and techniques, such as object composition and delegation, can be used to implement flexible and reusable frameworks.

- *Design patterns help to implement frameworks that are designed for change.* Each design pattern lets some aspect of a framework vary independently of other aspects, which makes the framework more robust towards a particular change. In other words, design patterns help in implementing the hot spots of a framework in a flexible and extensible way. This helps in avoiding major redesigns in the future by ensuring that a framework can change in specific and predefined ways.

Besides these benefits, the information conveyed within a design pattern description can also be used to document the design of a framework and communicate it to other developers [BJ94, Joh92, LK94, MDE97]. This will be shown in Chapter 3.

2.2.3 Design Pattern Pitfalls

Obviously, design patterns not only offer advantages, but also exhibit some important disadvantages

- *Design pattern can complicate the design.* Design patterns achieve flexibility by introducing extra abstractions and indirections. These may make the resulting design harder to understand at first glance, since these abstractions and indirections may not be obvious from the start. Carefully documenting the design patterns that are used in a framework is thus crucial

for understanding its design, especially for novice developers that may not yet be familiar with the concept of design patterns.

- *Design patterns may induce overdesign.* Exactly because of the popularity of design patterns and their alleged benefits, developers may be tempted to overdesign a framework. They may apply design patterns in places where their flexibility is not strictly needed, thereby contributing to a more complicated design. This is acknowledged in [GHJV94], where it is stated that one should design to be as flexible as needed, not as flexible as possible.

- *Design patterns may influence performance.* Because design patterns introduce extra abstractions and indirections, the implementation they lead to is often less efficient than a more straightforward implementation [TDM99, SLCM00]. Performance is already an issue in object-oriented programming languages, since late binding and polymorphism are responsible for the fact that compilers can not apply traditional optimization techniques (such as inlining) [Cha92]. Adding more flexibility through extra indirections clearly only aggravates this situation.

- [AC98] argues that the common design vocabulary defined by design patterns easily becomes unmanageable given the current increase in the number of design patterns. Furthermore, because so many design patterns keep being discovered in so many domains, it will become harder to identify the design pattern that can or should be used to solve a particular problem. This may dissuade developers from using design patterns.

- *Design patterns suffer from traceability and implementation overhead problems* [Bos98]. On a more technical level, design patterns suffer from what is called the *traceability* problem. The application of a particular design pattern is often obfuscated, since a developer is required to implement the design pattern over multiple classes and methods, because the programming language does not support a corresponding concept. A conceptual entity at the design level is thus scattered over multiple places in the implementation. Furthermore, while design patterns ensure a flexible and reusable implementation, the implementation of these design patterns itself is not at all reusable. Due to a lack of powerful environments that support working with and manipulating design patterns, the developer is forced to implement several classes and methods with trivial behavior over and over again. This problem is known as the *implementation overhead* problem.

- *Design patterns have no formal basis.* The increase in the number of design patterns, and the fact that design patterns have no formal basis at all, makes it hard to provide appropriate tool support, as is acknowledged in [EHY98, EGY96, EGHY99, Ede00, TN01]. Such tool support is needed, however, in order to assist developers in using design patterns and profiting maximally from their undeniable benefits. Tool support for automatic code generation [BFVY96, DV01], retrofitting design patterns in an existing design [TB95, TB99, RBJ97], choosing the appropriate design pattern, and so on, will become indispensable.

2.2.4 Conclusion

From the above discussion, a number of important issues can be concluded. First of all, design patterns expose important information about the design of a framework.They identify the key classes and methods of the framework, and thereby clarify the particular relationships, collaborations and interactions between them. Second, design patterns specify how a framework can be evolved. They implement the hot spots and the frozen spots of the framework, and in this way define how and where the framework can be extended. Third, design patterns provide hints for the actual implementation of classes and methods. The relationships and interactions defined by a design pattern show how classes should be related, which other classes a class should use and how a method should call other methods to implement the appropriate behavior. Fourth, due to a lack of a formal model for design patterns, providing tool support is difficult, as is using design patterns as a documentation aid.

2.3 Related Work

Related work is situated in the areas of framework documentation techniques, tools that support the instantiation and evolution of a framework, and design pattern formalization.

2.3.1 Framework Documentation Techniques

Since documenting a framework in an adequate way is an inherently difficult task, several approaches to framework documentation exist. Most framework-documentation techniques focus on the application developer, since he must implement the concrete applications and should reuse the framework. We can distinguish between those approaches to documentation that merely try to document the framework, and those that try to use the documentation in an active way to support the developer working with the framework. We will discuss approaches of the latter kind in other sections, and will restrict ourselves to passive documentation techniques here. Moreover, we do not aim to give a complete overview of the domain of framework documentation techniques. Rather, we highlight some of the more interesting approaches.

Several authors advocate the use of patterns as a form of framework documentation. In [Joh92], Johnson proposes to use patterns to describe the purpose of a framework, learn application developers to use a framework without knowing the internal details and teach many of the design details embodied in a framework. He validates this approach by defining a number of such patterns for the *HotDraw* framework [Bra95]. [MCK97] builds upon this approach and uses a combination of *catalog patterns*, *application patterns* and *design patterns* to document a framework. Catalog patterns document a framework's application domain, main features and scope, whereas application patterns corresponds to Johnsons notion of patterns to document a framework, and design patterns provide detailed information about the design of the framework. Similarly, [LK94] also uses Johnsons patterns (which are termed *motifs* in their approach) in combination with design patterns and formal contracts [HHG90]. Again, the emphasis of this approach is on documenting how to use a framework. [OQC97] promotes the principle of *documenting by designing* by using design patterns. The idea is that design pattern related activities, such as searching for the appropriate design pattern and fitting it into the source code, produce parts of the design documentation. Such documentation shows the path that was followed from the initial problem to the actual solution, and discusses the advantages and disadvantages of the resulting design.

In [FPR00], Pree presents an extension to the *Unified Modeling Language* (UML) which supports working with object-oriented frameworks. The general goal is to make the intentions of the framework developer more clear by explicitly representing framework variabilities (the hotspots of the framework) and their instantiation restrictions in the various UML diagrams. This is achieved by using stereotypes to introduce tagged values. For example, methods can be marked with a tagged value {*variable*} to indicate that their implementation can vary depending on the framework instantiation. Classes can be marked with a tagged value {*extensible*} which denotes that their interface may be extended during instantiation.

[BGK98] introduces the concept of *reuse cases* to document the way a framework can be instantiated. A reuse case is actually a specialized *use case* [JCJO92]. Such a use case describes a subset of a system functionality in terms of the interactions between the system and a set of users. A reuse case, on the other hand, describes a well-defined way of reusing a framework. A reuse case has a *name* that uniquely identifies it, a *purpose* that captures the effect of executing the reuse case in the context of the framework, the *roles* of the actors in the reuse case (such as framework developer, application developer, etc.) and the *scenario* that presents the actual steps involved in reusing the framework. Additionally, there may be cross-references to further information, including other reuse cases, architectures, design patterns, contracts and source code.

In [GM95], the authors argue that applications should be build in a top-down fashion as opposed to the traditional bottom-up way. Instead of assembling fine-grained components, application developers should try to understand and adapt so-called *exemplars*. An exemplar is an executable visual model consisting of instances of concrete classes together with an explicit representation of their collaborations. For each abstract class in the framework, at least one of its concrete classes

has to be present in the exemplar. Since even large frameworks have only a limited number of abstract classes, a small number of instances already suffice for creating an exemplar. Exemplars are provided by framework developers, and are explored interactively by application developers in order to understand the relationships and responsibilities of classes in the framework. To build an application, the application developer chooses a particular exemplar and modifies it to his specific needs.

In summary, besides the fact that these documentation techniques do not allow the information to be used actively by a supporting tool, they exhibit a number of other disadvantages. First of all, the information is not expressed in a formal way, but only in natural language. This makes it inherently ambiguous since it is up to the reader of the documentation to interpret it. Second, the documentation is not updated automatically when the framework or its applications evolve. Since a framework is subject to constant evolution, this is quite cumbersome and the cause of why documentation is often outdated.

2.3.2 Tool Support for Framework Instantiation

To support the task of an application developer, several tools have been developed that provide active guidance for instantiating a framework. We will discuss a number of these tools in the following subsections.

Specialization Patterns

In [HHK+01a, HHK+01b], the authors show how the hot spots of a framework can be documented by means of *specialization patterns*. A specialization pattern is a specification of a recurring program structure that can be instantiated in several contexts. It consists of *roles* that are played by structural elements of the framework, and various *properties* associated with these roles. Structural elements can be classes, methods or variables, so there are *class roles, method roles* and *field roles*. For each kind of role, there is a set of properties that can be associated with the role. For a class role, for example, there is an *inheritance* property that specifies the required inheritance relationship for each associated class. These properties are called *constraints*, since they specify requirements for the static structure of the program elements. Furthermore, a distinction is made between *framework* roles, that are bound to framework source code entities, and *application* roles, that are bound to application-specific source code entities. When a framework developer has specified the hot spots of the framework as specialization patterns, *pattern initialization* takes place, a process that instantiates the patterns and binds their framework roles to framework source code entities. All other roles are left for the application developer to bind. Based on these unbound roles, a task list will be generated that contains the tasks an application developer has to perform. This task list is inherently dynamic: as a task is completed, new tasks can be added to the list. Changes that are made by the developer performing a task are monitored and their validity is checked against the constraints specified in the patterns. Possible violations result in the generation of *refactoring* tasks. In this way, the proper use of the framework is constantly validated and supervised by the tool.

Currently, specialization patterns are mainly intended to be used as an underlying basis for tools that support framework instantiation. Although they contain important information about the design of a framework, their current incarnation does not easily allow them to be used for documentation purposes. They do not expose information about the design rationale, for example, and it is not clear if information about bound framework and application roles is stored, in which form it is stored, and whether it is available for querying and browsing. Moreover, specialization patterns can not be used to support framework evolution, as they only allow to extend a framework with classes and methods, but do not include actions that remove or change existing elements. A forteriori, specialization patterns can not be used to support software merging. Another major shortcoming of the approach is that it ignores the fact that a framework consists of many specialization patterns, that are related: one element may fulfill a particular role in a multitude of specialization patterns. Such 'relationships' between specialization patterns are not

explicit however, and a developer is thus forced to bind the element to each role separately and explicitly. This leaves room for errors, of course, which might result in incorrect applications being instantiated. An improvement would thus consist of automatically deriving which specialization patterns are related and how this affects the binding of elements to roles.

Smartbooks

In [OC00, OCS99], an environment is introduced to automatically guide an application developer when instantiating a framework, based on a combination of *user-task modeling* and *least commitment planning* methods. The environment is build around a planner that, given the functionality the application developer wants, automatically produces an *instantiation plan*. This plan is a set of tasks that must be executed in order to obtain the desired functionality. The input for the planner is a set of rules that are provided by the framework developer and describe the necessary steps to obtain the desired functionality. Such rules actually consist of a list of preconditions and a postcondition and represent an instantiation task. This task is executed whenever the preconditions hold and results in a satisfaction of the postcondition. The planner thus always tries to satisfy the preconditions of a rule for which the postconditions are goals. The planner is integrated in an environment that allows the framework developer to document the framework by means of instantiation rules (together with traditional design information) that describe the functionality that can be implemented using the framework and how this functionality is related with the framework components (e.g. the source code). The environment then shows all functionalities provided by the framework to the application developer, who can select the appropriate functionality for his specific purposes. This results in the generation of an instantiation plan, containing the various tasks that the application developer should carry out. Because the environment is integrated in a standard development environment, and because the information is provided in a formal and executable medium (resembling a logic programming language), some tasks can be performed automatically, whereas others must be executed manually by the application developer.

Reasoning With Design Knowledge

In [DMDGD01], Demeuter et al introduce a knowledge-based system that incorporates knowledge about the design of the framework and the specific ways in which it can be reused. They show how this system can be used to interactively guide the application developer, thereby enhancing the quality of the applications he derives from the framework. The knowledge-based system is integrated within a standard development environment, and interacts with the developer, similar to a programming wizard. The control over the reuse process is in turn with the system and with the developer. The developer can turn to the system for help, and the system reacts by providing him with a number of possible reuse recipes. The developer then performs the necessary steps to execute the recipe. When the developer has made some changes, the system can again take control and provide him with further instructions. The system remembers the specific steps of the recipe that the developer already performed and those that he did not. It can thus adapt its advice to the concrete situation the developer is in.

The knowledge-based system that is used in this approach is a general system with some specific extensions to integrate it into the development environment. It can be provided with *rule sets* that describe how to extend the framework, e.g. which classes need to be subclassed and which methods need to be overridden. Such rules are specified in an intuitive Scheme-like programming language. Different rule sets have to be provided for different frameworks, but the knowledge-based system can be reused, thanks to its generality.

2.3.3 Tool Support for Framework Evolution

Refactoring

When evolving an object-oriented framework, it is good practice to first improve its existing design and only in a later phase extending it. The main motivation is that this should allow the

desired extensions to be integrated more smoothly. Hence refactorings can be used to support unanticipated evolution. In that case, they are applied as a preparatory step, to change the design of the framework in order to allow for a smoother integration of the other changes. Moreover, splitting the evolution into a redesign and an extension phase allows the developer to validate the new design separate from the forthcoming extensions. If it is guaranteed that the redesign phase preserves the overall behavior of the framework, then errors can only be introduced in the extension phase which is the only portion of the framework that should be validated. Preserving the behavior of a framework while at the same time altering its design can best be achieved by automating the transformations that should be applied. Such automated transformations that preserve the behavior are called *refactorings* [Opd92, Rob99, Fow99, Tic01].

In his doctoral dissertation, Opdyke was the first to coin the term refactoring [Opd92]. He identified a number of low-level refactorings for the C++ programming language (such as adding a new class or removing a formal parameter from a method), and provided a formal elaboration of the conditions under which these refactorings preserve the behavior of the framework. Basically, a refactoring transformation can only be applied if the framework's implementation satisfies a number of preconditions. For example, adding a new class to the framework requires that this class does not already exist. Opdyke also showed how higher-level refactorings can be defined by combining the lower-level ones. Such higher-level refactorings are also guaranteed to preserve the behavior, since they are composed of lower-level refactorings, that preserve the behavior. Examples of high-level refactorings are introducing abstract classes into the framework [OR93] and turning inheritance relationships into aggregation relationships [JO93].

Based on Opdyke's ideas, tool support for refactoring in the Smalltalk programming language was provided by means of the *Refactoring Browser* [RBJ97]. This browser is tightly integrated with the Smalltalk development environment and provides functionality similar to a standard browser. Additionally, it allows a developer to select a particular refactoring transformation, and automatically applies it after checking its preconditions. Besides implementing those refactorings defined by Opdyke that are applicable to the Smalltalk programming language, some extra refactorings are incorporated as well, such as *move method across object boundary* and *extract code as method*.

Fragment Model

[FMvW97, Mei96, vW96, Gru97] introduces a tool that provides extensive support for working with design patterns, both when developing new frameworks and evolving existing ones. The tool incorporates three integrated views on the framework that each operate on a different level of abstraction: the code view (classes, methods, etc. . .), the design view (abstraction of the code plus additional information) and instances of design patterns in the framework. Each view supports operations particular to its level of abstraction. On the design pattern level, a developer can instantiate design patterns, on the design level he can split classes into a hierarchy and on the code level he can provide methods with an implementation, for example. Assistance for working with design patterns is provided in three ways. The environment provides code generation facilities for design pattern instances, it can integrate design pattern instances into already existing code by binding program elements to roles in a design pattern and it can check whether design pattern instances still meet the invariants governing the design patterns and report errors if this is not the case. The tool is thus mainly useful for supporting anticipated evolution, since it relies on information about the design patterns and support predefined operations on them.

To achieve this kind of support, the *fragment model* is used. A fragment represents a particular design element, which can range from single classes and methods to complete design patterns. Fragments have roles, that contain references to other fragments, and can have behavior associated with them, for checking constraints or implementing design pattern specific operations, for example. These design pattern-specific operations are used to evolve design pattern instances in predefined ways at a high-level of abstraction. They automatically clone instances of existing fragments and bind them to the appropriate roles. Operations are provided for adding a concrete factory class to an instance of the *Abstract Factory* design pattern and for adding a new composite

method to an instance of the *Composite* design pattern, for example. The operations themselves are implemented in the Smalltalk programming language. Constraints are also pieces of Smalltalk code that implement boolean checks and that can use a number of predefined inquiry operators to get to the properties of the fragment they are working on. The constraints are validated whenever an editing operation (as provided by the fragment) has modified the fragment, or whenever validation is triggered by another fragment. When inconsistencies are detected, exceptions are raised. The system includes several possible exception handlers (of which only one can be active at any given moment). The exception handler is responsible to implement the action that needs to be taken. Several types are implemented that allow the developer to ignore, discard, warn, repair or choose between different options when differences between the fragment and the implementation are encountered.

Reuse Contracts

Reuse contracts are a methodology for identifying possible evolution conflicts caused by different developers evolving the same software artifact in parallel. Originally, reuse contracts were developed to deal with evolution of classes at the implementation level [CDHSV97, SLMD96]. Later the focus shifted to the design level, by looking at evolution of class collaborations [Luc97]. In both domains, the approach is general enough to support anticipated as well as unanticipated evolution.

The approach consists of documenting the design of the framework by means of reuse contracts, and its evolution by means of reuse operators.. A reuse contract is a description of an interface and documents the protocol between classes. A reuse contract is subject to different reuse operators, that represent the different ways in which the contract, and thus the interface, can evolve. Examples of such operators are *refinement, extension* and *coarsening*. Together, reuse contracts and reuse operators document that part of the design of a framework that is relevant to an application developer. Moreover, they document the assumptions made by application developers about the way the framework is reused. This documentation facilitates the task of these developers, by indicating how much work is necessary to update previously built applications, where and how to test and how to adjust these applications.

When a framework's design and evolution is documented by means of reuse contracts and reuse operators, an operation-based merge conflict detection algorithm can be defined. By storing all possible operations in a *merge matrix*, the operations that have been applied can be mutually compared, and it can be derived whether they lead to an inconsistency. Different kind of such conflicts can be detected, such as *naming conflicts*, that are due to two developers introducing an artifact with the same name, or *method capture conflicts*, that arise due to the fact that a developer extends a superclass' interface with a new method, that is already present in one of the superclass' subclasses. These methods that were already present now accidently override the newly added method, which may not be what was intended.

The reuse contract approach was applied to a number of other domains as well: to document the evolution of UML interaction diagrams [MLS99] and even at the level of requirements analysis [DH98], for example. Clearly, this shows that the approach is applicable to a broad domain, and as such, a domain-independent definition of the reuse contract approach was defined by Mens in [Men99]. In this approach, software artifacts are represented by graphs, and evolution is represented by means of graph rewriting. Type graphs are used to specify domain-specific constraints that have to be satisfied by all graphs in a particular domain. By relying on the formal properties of graph rewriting, such as the notion of parallel dependence, one can formally define which pairs of modification operations (formally represented as graph productions) yield an evolution conflict.

Miscellaneous

Feather uses a variant of the reuse contract approach [Fea89]. Instead of mutually comparing the operations, they are analyzed to determine what changes to specification properties they induce, and this information is used instead. Conflicts can then be detected by comparing all possible

changes to specification properties. This makes it easier to introduce new modification operations. One only needs to determine how the newly introduced operations can be decomposed in terms of changes to specification properties.

As an alternative to merge matrices, as used by the reuse contracts approach and the approach of Feather, conflicts sets are used by Edwards [Edw97] to group together potentially conflicting combinations of operations based on the application-supplied semantics. Depending on the kind of application, the kinds of operations and associated merge conflicts can differ dramatically. Conflicts sets correspond to the types of conflicts that may exist in an application. As such, they are statically defined, in the sense that they remain fixed as long as the application semantics does not change. Operations that belong to the same conflict set may potentially cause conflicts when merged together. Any given operation may simultaneously participate in multiple conflict sets. Conflict sets address scalability, since they restrict the number of operations that we must consider when searching for conflicts.

Horwitz, Prins and Reps [HPR89] were the first to develop a powerful algorithm for merging program versions without semantic conflicts, based on the semantics of a very simple assignment-based programming language. The merge algorithm uses the underlying representation of program dependence graphs, and uses the notion of program slicing to find potential merge conflicts. Despite its power, the algorithm poses a number of limitations, the most important being that it is restricted to a particular programming language, that is extremely simple compared to current-day programming languages.

2.3.4 Formal Models for Design Patterns

LePUS

LePUS [EHY98, EGY96, EGHY99] (LanguagE for Patterns Uniform Specification) is a formal specification language that can be used to specify the static structure of a design pattern accurately, completely and concisely, in a language- and implementation-independent way. The language was conceived out of the observations that current object notations are inadequate for specifying design patterns and that the detailed verbal descriptions that are used are often imprecise, ambiguous and vague. Moreover, formally defining design patterns is a prerequisite for allowing tool support.

A specification of a design pattern in LePUS unambiguously defines its specific constructs and relations and the constraints it imposes upon an implementation. Such specification includes a list of participants (classes and functions) and a list of relations (such as *inherits, invokes*, etc.) that hold between these participants. Sets of classes and functions of any dimension are allowed, and the relations easily extend to participants of higher dimensions as well. Moreover, additional abstractions, such as class hierarchies, function clans (functions with identical signature that are implemented throughout a class hierarchy) and tribes (sets of clans with respect to a common class set), are also provided since they form such important concepts in object-oriented programming languages.

As a concrete example of how LePUS can be used to specify design patterns, consider the graphical specification of the *Strategy* design pattern in Figure 2.2. The corresponding formula is:

$$
\begin{aligned}
&\exists context, client \in \mathcal{C};\\
&operations \in \mathcal{F};\\
&Algorithms, Configure - Context \in 2^{\mathcal{F}};\\
&Strategies \in \mathcal{H} : clan(operation, context) \wedge\\
&clan(Algorithm, Strategies) \wedge\\
&tribe(Configure - Context, client) \wedge\\
&Invocation^{\rightarrow}(operation, Configure - Context) \wedge\\
&Invocation^{\leftrightarrow}(Algorithms, context) \wedge\\
&Argument - 1^{\rightarrow}(Algorithms, context) \wedge\\
&Creation^{\rightarrow \mathcal{H}}(Configure - Context, Strategies) \wedge\\
&Assignment^{\rightarrow \mathcal{H}}(context, Configure - Context, Strategies) \wedge\\
&Reference - to - Single^{\leftrightarrow}(context, Strategies)
\end{aligned}
$$

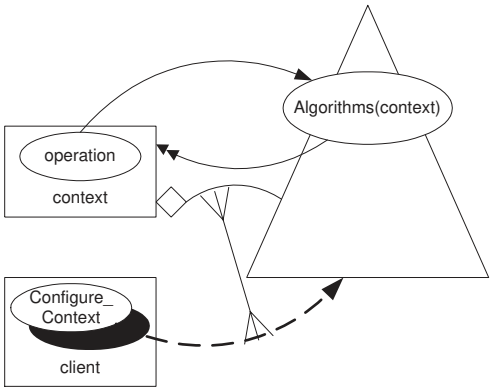


Figure 2.2: Specification of the *Strategy* design pattern in LePUS

where $\mathcal{C}$ is the set of all classes, $\mathcal{F}$ the set of all functions and $\mathcal{H}$ the set of class hierarchies.

LePUS was used to formally define most of the design pattern in [GHJV94]. It was observed that many of the behavioral aspects of design patterns can be expressed, although the focus of the language was on static relations. No difficulties were found in describing behavioral design patterns, for example. Additionally, based upon these definitions, a tools was built that is able to recognize design patterns in a particular software implementation, and that can introduce a design pattern into this implementation. This is difficult to achieve without a precise formal definition.

DPDOC

[CH00b, CH00a] presents a prototype of a tool that makes it easy and safe for developers to use design patterns and automatically maintain up to date documentation of the framework based on these design patterns. The tool supports design pattern visibility and rule checking. Design pattern visibility is supported by modeling the design patterns by a grammar with syntactic and context-sensitive rules, which formally specify the various roles and rules of the design pattern. As an example, in the *Decorator* design pattern, there are *Component, Decorator* and *decorated component* roles, while a rule is used to express the constraint that a decorated component role should occur within a *Decorator* role. By formulating a design pattern by means of a grammar, it is actually reified into an explicit language construct. Furthermore, by using reference attributes, which support the easy specification of non-local dependencies, design pattern instances can be connected to the source code that implements them. It thus becomes possible for a developer to see which design patterns are used in the code and what roles are played by the different source code artifacts. Moreover, the documentation does not become outdated when the program is changed, since references of attributes to source code can even be automatically computed to some extent. Evidently, all this can be used to tackle the traceability problem associated with design patterns, since tools can be built upon this technique that allows easy browsing of all available information. Automatic rule checking is used to check whether a particular design pattern is used correctly. Because the design patterns are modelled using a grammar and references to the appropriate source code artifacts are present, checking that the constraints of a design pattern are abided by thus amounts to perform checks in a way similar to how a compiler performs compile-time checks. The authors also propose a tool, called *DPDOC*, that unifies the techniques presented above with the development environment.

Contracts

Contracts [HHG90, Hol92] are a means to document *behavioral compositions* in an object-oriented framework. A behavioral composition is defined as a group of related objects that work together to

perform a particular task or maintain a certain invariant. Design patterns define such behavioral compositions, and can thus be documented by means of contracts.

A contract formally defines the behavioral composition of a set of communicating participants. To this extent, it formally specifies the following aspect of such a composition:

1. a contract identifies the participants in the composition and their so-called *contractual obligations*. A contractual obligation consists of both *type obligations* and *causal obligations*. The former describe the variables and methods a participant must define, while the latter specify an ordered sequence of actions that a participant must perform, upon the receipt of a particular message. It is thus through these causal obligations that contracts capture the behavioral dependencies between objects and their methods.

2. a contract defines the invariants that participants in the behavioral composition cooperate to maintain and the actions that should be performed to resatisfy this invariant should it become false. An example of such an invariant is that a *View* in the *Model-View-Controller* paradigm [KP88] should always reflect the state of the *Model*.

3. It specifies preconditions on participants to establish the contract and the methods which need to be called at runtime to instantiate the contract.

Two important operations are defined on contracts: *refinement* and *inclusion*. Both can be used to express complex behavior in terms of simpler behavior. The refinement operation allows to specialize the contractual obligations and invariants of a contract. A contract is refined by either specializing the type of a participant or derive a new invariant which implies the old. As such, a refined contract defines a more specialized behavioral composition. The inclusion operation allows contracts to be composed from simpler contracts. Its intent is to describe the behavior of some of the participants in a behavioral composition in terms of simpler compositions. This is achieved by including *subcontracts* in the definition of a contract. A subcontract imposes additional obligations on participants, over and above those defined in the contract.

Contracts are only an abstract specification of a behavioral composition. Concrete classes of a framework should be mapped onto the participants of a contract. This is achieved through *conformance declarations*. Such declarations are actually specifications of how a class supports the role of a participant in a contract, by means of the variables and the methods that it defines. Besides being part of the contract approach, conformance declarations also form an important part of the documentation of a framework. They can be used to factor a large class interface, consisting of a large number of methods, into meaningful related subsets.

Discussion

Most of the tools that were discussed above either focus on the documentation of the framework, its instantiation or its evolution. None of them provides support for all these activities in an integrated fashion. This is largely due to the fact that these tools are not based on an appropriate model of the framework, that incorporates all necessary information and that allows us to use and manipulate this information in an active way. A tool that supports documentation, instantiation as well as evolution of a framework, should use an adequate formal model of the framework, at the appropriate level of granularity. Several such models are proposed in literature, as we have discussed.

2.3.5 Metapatterns

Many design patterns serve a completely different purpose, but share the same underlying structure and only differ in some implementation details, such as the specific method invocations that occur. Consider for example an instance of the *State* design pattern [GHJV94], as depicted in Figure 2.3, and compare it with the instance of the *Strategy* design pattern [GHJV94] in Figure 2.4. As can be seen, the structure of both instances is very similar. Each one defines an association

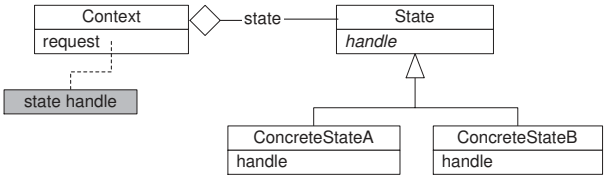


Figure 2.3: Structure of the *State* design pattern

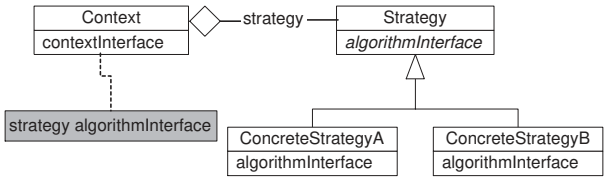


Figure 2.4: Structure of the *Strategy* design pattern

relationship between a class and a class hierarchy. In these particular cases, the `Context` class refers to the `State` hierarchy or to the `Strategy` hierarchy. Moreover, both `Context` classes provide method definitions that use the association relationship to call methods in the hierarchies. For example, the `request` method calls the `handle` method, and the `contextInterface` method calls the `algorithmInterface` method. The methods that are called are defined as abstract methods in the root of the class hierarchy, and are provided with a concrete implementation in the leaf classes of that hierarchy.

The only way in which these two structures differ, is in the specific interactions between the methods. The *Collaborations* section of the *State* design pattern includes a detailed discussion as to which state succeeds another one, under which conditions and how this should be implemented by the methods. Such information is not included in the *Strategy* design pattern, since it serves a completely different purpose.

Many other examples of design patterns that share similar structures can be found. An instance of the *Abstract Factory* design pattern can be considered as a collection of instances of the *Factory Method* design pattern, for example. An abstraction of design patterns, based on their structure, the relationships and the collaborations, is thus possible. Metapatterns provide such an abstraction [Pre95, Pre94, Pre97].

Metapattern Definition

The definition of metapatterns is based on a distinction between *template* and *hook* methods and the corresponding *template* and *hook* classes.

Template methods are concrete methods that define the control flow of an algorithm and provide an implementation in terms of some other methods. These other methods are the hook methods, and can either be abstract methods, regular methods that provide a default implementation or a template method in their turn. The distinction between a template method and a hook method clearly depends upon the point of view. A method `m` is a hook method if it is called by another method, which then acts as the template method. At the same time, the method `m` can be a template method as well, as it may call other methods itself.

The classes that implement template methods and hook method are called template and hook classes, respectively. A template class is parameterized by a hook class, since a template class should contain a reference to a hook class, in order for the template methods to be able to call the appropriate hook methods. Template and hook methods may be defined in the same class as well. In that case, the class acts as a template and a hook class at the same time.

Hook classes are typically abstract, since they contain the hook methods which may be abstract. As such, hook classes need to be subclassed in order to override the hook methods. Template classes on the other hand, typically do not need subclasses, as the template methods need not be overridden in general. A class may contain both template methods and hook methods at the same time.

Metapatterns predefine particular combinations of template classes and hook classes. Two aspects influence these combinations:

1. The cardinality of the association relationship between the template class and the hook class. An object of the template class may refer to exactly one object of the hook class, or it may refer to multiple objects of this class.

2. The hierarchical relationship between the template class and the hook class. The template class and the hook class may be unified into one class, or they may or may not be related via an inheritance relation.

As an example, in the *1:1 Connection* metapattern (see Figure 2.5), the template class `T` and the hook class `H` are not related via inheritance, and an object of the template class refer to exactly one object of the hook class. In the *1:N Recursive Connection* metapattern, on the other hand, the template class `T` is a subclass of the hook class `H`, and objects of the former class refer to a multitude of objects of the latter class. The *Unification, 1:1 Recursive Unification* and *1:N Recursive Unification* metapatterns are examples of metapatterns where the template and the hook class are unified into one single class.

Note that it does not follow immediately from Figure 2.5 that the template and hook class in the *1:1 Connection* and the *1:N Connection* metapatterns are not related via an inheritance link. Implicitly, however, there is no inheritance relation between them. If there was, the resulting structure would look exactly like the structure of the the *1:1 Recursive Connection* and the *1:N Recursive Connection* metapatterns, respectively.

Just as a design pattern instance is a particular occurrence of a design pattern, a metapattern instance is a specific occurrence of a particular metapattern.

Application of Metapatterns

As a concrete example of how metapatterns are an abstraction of design patterns, consider Figure 2.6, which illustrates how the *State* and *Strategy* design patterns can both be mapped onto the *1:1 Connection* metapattern. In both cases, the `Context` class of the design pattern structure plays the role of the template class in the metapattern structure. Similarly, the `State` and `Strategy` classes are mapped onto the hook class. Methods in the design pattern structure are also mapped in a similar way onto methods in the metapattern structure. For example, both the `request` and the `contextInterface` methods are mapped onto the template method of the metapattern structure, whereas the `handle` and the `algorithmInterface` methods are mapped onto the hook method in this structure.

Discussion

Since metapatterns are only an abstraction of design patterns, they do not include each and every detail. In fact, metapatterns only contain the information conveyed within the *Structure* and the *Participants* sections of a design pattern template description. Metapatterns also contain information about the specific collaborations between participants, although not in as much detail as design patterns. They do not contain information about the *Applicability, Trade-offs* or *Consequences* however.

In general, it is difficult to prove that metapatterns do form a *suitable* abstraction of design patterns. It is not known, for example, if all design pattern structures can be described by means of metapatterns. In [Pre95], an existing real-world framework is adequately and extensively documented by means of the metapatterns, which shows the applicability of the approach. In

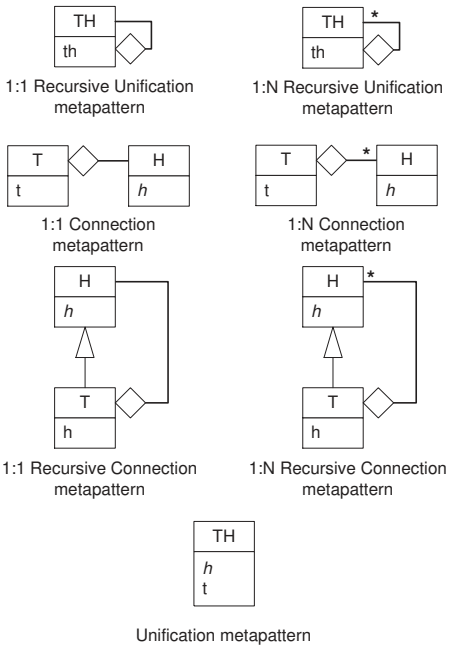


Figure 2.5: The Existing Metapatterns

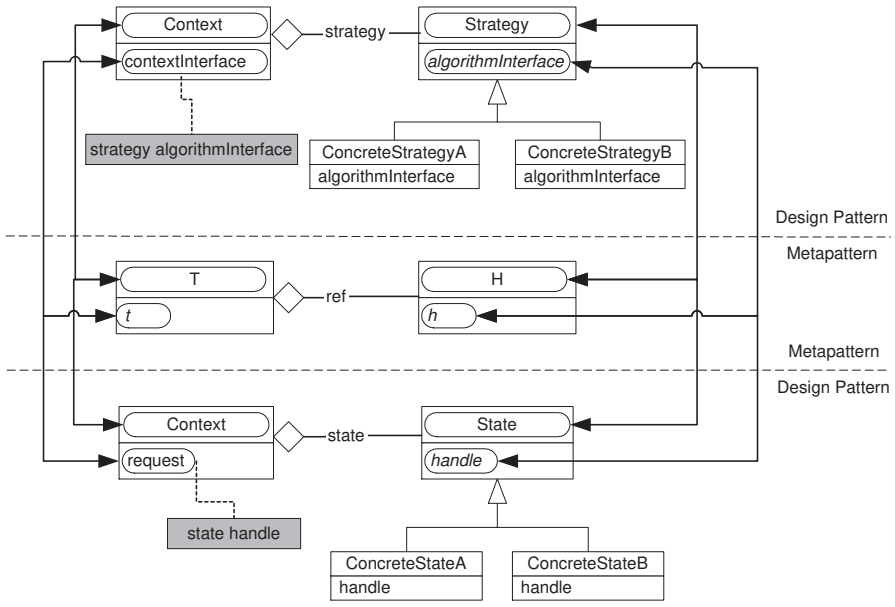


Figure 2.6: The *State* and *Strategy* design patterns are mapped onto the *1:1 Connection* metapattern.

subsequent chapters, we will show how several widely-used design patterns can be mapped onto metapatterns and how metapatterns are useful as a basis for providing tool support for framework-based development.

Chapter 3

The Scheme Framework Example

In this chapter, we will explicitly document the implementation of a small-scale framework for building Scheme interpreters by means of the design patterns that it uses. Moreover, we will present an example of how this framework can be evolved, and how the information exposed by the design patterns can be used for this purpose. All issues addressed in this chapter, will be used in subsequent chapters to present the problem statement of the dissertation and to illustrate our approach to support framework-based development.

3.1 Introduction

The framework we present in this chapter is a framework for building Scheme interpreters [AS85, Dyb96, FMK96]. It was implemented in the context of this dissertation in order to experiment with different flavors of declarative meta-programming languages. The framework allows us to implement and experiment with different Scheme variants. In addition, it is used as a means to clarify the problem statement of the dissertation. To this extent, we present an example of how the framework can evolve, explain the difficulties that arise when doing so and discuss how having appropriate information at our disposal can help to alleviate these problems. Furthermore the framework is used in various other places in the dissertation as an example of how the approach we propose can be used in practice. Note however, that we will validate our approach on another framework, that we did not implement ourselves, in Chapter 8.

The framework is implemented in Squeak [IKM+97], a Smalltalk dialect. It consists of 108 classes and 608 methods in total. Of these 608 methods, there are only 384 methods that actually implement important behavior, all other methods are initialization methods, accessor methods, and so on. As such, the framework is relatively small and is ideally suited as a case study for performing some initial experiments.

The remainder of this chapter is organized as follows. First, we will present a very short introduction to the Scheme programming language for readers that are not familiar with it. Then, we will explain the global architecture of a Scheme interpreter as implemented by the framework, which at the same time explains the implementation of the framework itself to some extent. Following the approach taken by Johnson and Beck [Joh92, BJ94], a detailed description of this implementation follows, by means of an explanation of the different design patterns that are used in the framework. The next part of this chapter provides an example of how the framework can be evolved, and explains the various difficulties that arise.

3.2 Introduction to the Scheme Programming Language

In the following subsections, we will provide a very short introduction to the Scheme language, so that readers not familiar with it are able to understand the discussions in subsequent sections and

chapters. For an in-depth discussion of the language, and all of its variants, we refer the reader to Abelson and Sussman's excellent introduction to the Scheme programming language [AS85].

3.2.1 The Language

Scheme is a simple, yet very expressive and powerful programming language, with very simple syntax and well-defined semantics [IEE95].

A Scheme program consists of a number of expressions that are evaluated by the Scheme interpreter. The language defines a number of *evaluation rules* that define how these expressions should be evaluated. In comparison with other programming languages, Scheme has only a limited number of evaluation rules, which for the most part explains its simplicity. Like any other programming language, Scheme has a number of *primitive expressions*, such as numbers, symbols, strings and so on. These are accompanied by a number of *primitive procedures*, built into the language, that can be used to manipulate the expressions. Examples are the procedures for arithmetic and string manipulation, or the various input-output procedures. Additionally, Scheme defines a number of *special forms*, for defining variables (*define, let, let*, ...*), conditional statements (*if, cond, case, ...*), for defining procedures (*lambda, letrec, ...*), and so on. Each of these special forms has a different format, and consequently has a different evaluation rule as well. Besides primitive procedures and special forms, Scheme allows developers to define their own procedures as well. Such procedures are called *user-defined* procedures. They allow a developer to attach a name to a number of expressions and refer to those expressions as a unit. As such, user-defined procedures form a powerful abstraction technique.

3.2.2 Scheme Variations

Different variations on the standard Scheme interpreter can be considered.

A standard Scheme interpreter uses an *applicative-order* evaluation model. In such a model, a procedure is applied to a number of arguments by first evaluating these arguments, then fetching the procedure's body and evaluating it with each formal parameter replaced by the corresponding argument. The *normal-order* evaluation model is an alternative model that does not evaluate the operands of a compound expression until their values are actually needed. Instead, it first substitutes procedure bodies until it obtains an expression involving only primitive operators and then performs the evaluation.

Ordinary Scheme interpreters are statically scoped, which means that the value of a variable is computed with respect to the environment in which the expression containing the variable was defined. A dynamically-scoped interpreter, on the other hand, will compute the value of an expression based on the environment in which the expression is evaluated.

The framework has specific provisions that allow us to experiment with all these different variants of a Scheme interpreter. We can construct a standard, statically-scoped interpreter that uses applicative-order evaluation, for example, or we can implement a dynamically-scoped interpreter with a normal-order evaluation model. Many other variations are possible, since the framework is quite extensible.

3.3 Architecture of the Scheme Framework

The interpreter implemented by our framework consists of three distinct phases, as depicted in Figure 3.1: the parsing phase, the analysis phase and the evaluation phase. The primary reason why we explicitly separate the analysis phase from the evaluation phase is to optimize the execution time of a program. The analysis and evaluation could be done simultaneously, but an expression that needs to be evaluated many times will then be analyzed just as many times, which is completely unnecessary and decreases performance [AS85].

The job of the parser is to parse a Scheme expression and convert it to an abstract syntax tree. The parser will only detect syntactical errors (such as improper balance of parentheses), and

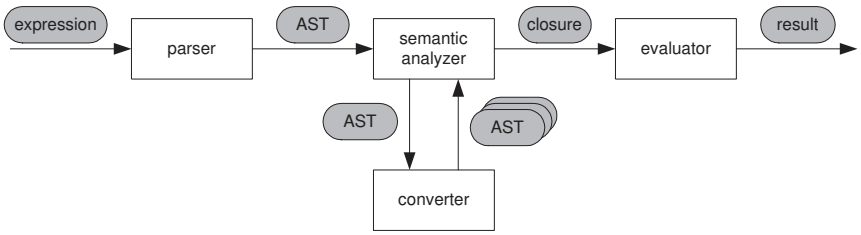


Figure 3.1: Architecture of the Scheme interpreter

does not check the well-formedness of the expression. Some well-formedness checks are needed, however, in order to ensure that special forms are used correctly. To this extent, an analysis phase follows the parsing phase.

The input for the analyzer consists of the abstract syntax tree constructed by the parser, and the output is a *closure* object that can be evaluated. During analysis, it is checked which kind of expression we are dealing with (a special form or a user-defined procedure) and whether this expression has the appropriate form. This is achieved by using a *converter*, which splits the expression in its appropriate subexpressions. For example, an *if* expression is split into a condition, a consequent and an alternate part.

If the expression is well-formed, it is transformed into a closure object and passed on to the evaluator. This evaluator will simply evaluate the closure according to the evaluation rules of the Scheme language and will return the corresponding value. Using such a closure object achieves the required separation of analysis and evaluation. If an expression needs to be evaluated many times, it will be analyzed once and transformed into a closure object which will be executed as many times as needed.

This architecture of a Scheme interpreter is reflected in the implementation as well. The parser, implemented by the `SchemeParser` class, uses classes from the `ScExpression` hierarchy to construct an abstract syntax tree corresponding to a particular expression. These classes provide an `analyze` method, that implements the analysis phase by checking the well formedness of the tree. Checking the well-formedness is implemented by using an appropriate converter object from the `SchemeConverter` hierarchy, which is responsible for breaking down an expression into its constituent parts, if this is possible. After analysis, a closure object corresponding to the expression object is returned. This object is in fact an instance of a specific subclass of the `Closure` class, which implements the evaluation algorithm by defining an `eval:` method.

3.4 Design Patterns in the Scheme Framework

The framework for the Scheme interpreter is implemented by using a number of design patterns so as to ensure its flexibility and reusability. In this section, we will discuss these design patterns, their instances in the framework, and the rationale behind their use. This helps to understand the implementation of the framework and provides extensive and detailed information about its design. Later on, we will use this information when we provide an example of how the framework can be evolved and when we discuss the problem statement of the dissertation and the particular solution we propose.

3.4.1 The *Abstract Factory* design pattern

General Discussion

The *Abstract Factory* design pattern is used to make a framework, or any object-oriented software system in general, independent of the concrete classes that it uses. To this extent, it provides an interface for creating a family of related or dependent objects without specifying their concrete

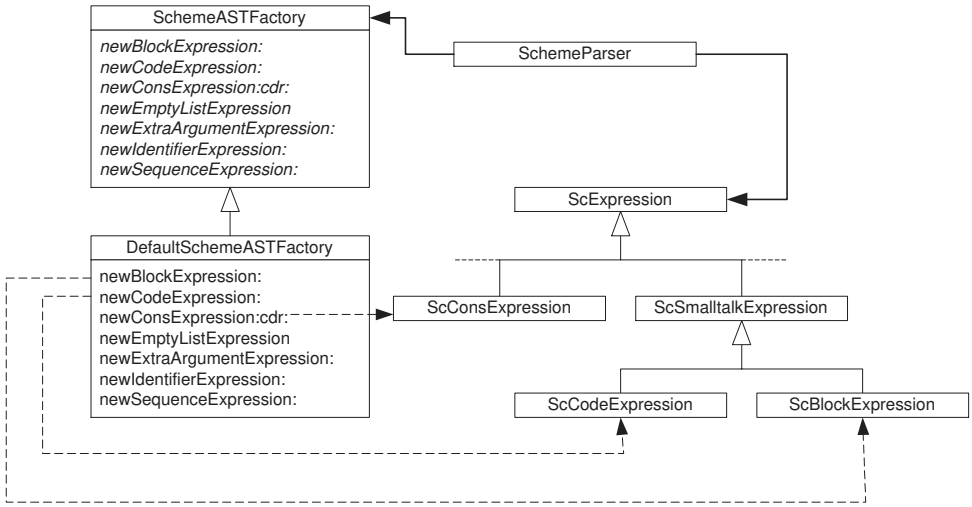


Figure 3.2: *ASTFactory* instance of the *Abstract Factory* design pattern

classes. Objects of these classes can only be created through this interface. Furthermore, this interface is implemented by an abstract class (which we will call the *factory class*), and all concrete subclasses of this class provide a concrete implementation for the interface by specifying the concrete classes that will be instantiated.

Using the *Abstract Factory* design pattern has several benefits. For one, it makes an application independent of the concrete classes that it uses. The application can manipulate objects using their abstract interfaces, but does not need to know the concrete objects that are created. Second, the pattern makes it easy to switch between different sets of objects. The factory class and its subclasses are the one and only spot in the application where objects are created. Defining a new subclass of the factory class and using an object of this new class ensures that another set of objects is used throughout the whole application. Third, the pattern ensures consistency among objects. When an object of a certain family is used, it is important not to mix objects and consistently use the other objects of that family as well. The *Abstract Factory* design pattern enforces this in a straightforward way.

The *Abstract Factory* design pattern in the Scheme Framework

The Scheme framework uses only one instance of the *Abstract Factory* design pattern, which we will call the *ASTFactory* instance and which is depicted in Figure 3.2. It shows how the `SchemeParser` class uses a factory class `SchemeASTFactory` that implements an interface to create `ScExpression` objects. Using the design pattern in this way allows us to switch between different expression objects should this be necessary. The classes in the `ScExpression` hierarchy implement an analysis algorithm. If we would like to experiment with a different algorithm, we can easily do so. We define a new family of `ScExpression` classes and create a new subclass of the `SchemeASTFactory` that instantiates objects of these classes. Then, we tell the application to use this factory instead of the standard `DefaultSchemeASTFactory`.

3.4.2 The *Chain of Responsibility* design pattern

General Discussion

The *Chain of Responsibility* design pattern is used to decouple the sender of a message from its receiver by giving more than one object a chance to handle the request. This is useful when the

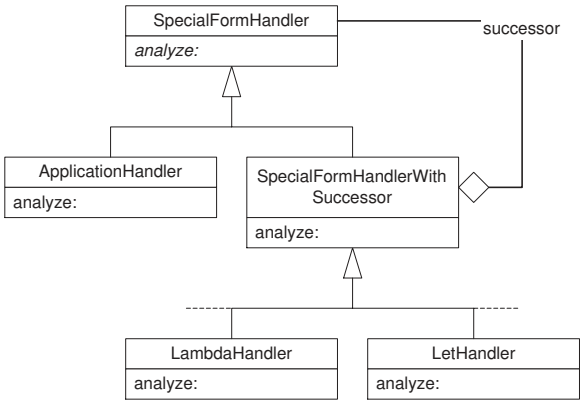


Figure 3.3: *SpecialFormHandler* instance of the *Chain of Responsibility* design pattern

sender knows the set of objects that may be able to handle a request, but does not known which specific object is able to do so. The solution this design pattern offers consists of chaining this set of objects to each other, and passing the request along the chain until an object handles it. The first object in the chain receives the request and decides whether it can handle it. If this is the case, it does so, but if it is not, it passes the request on to the next object in the chain, which acts likewise.

This design pattern offers two particular advantages. First, it reduces the coupling between different classes of the framework. Neither the receiver, nor the sender of the request need to know about each other. As such, objects do not need a reference to all candidate receivers, but only keep a reference to their successor in the chain. Second, it ensures the flexibility because extra handlers can be added easily and without making any changes to the sender of the request.

The *Chain of Responsibility* design pattern in the Scheme Framework

The *Chain of Responsibility* design pattern is used in the Scheme framework for analyzing `ScConsExpression` objects. Such an object may represent either a simple function call or a call to one of the various Scheme special forms. The analyzer needs to convert such an object into a closure object that can be evaluated. In order to do so, it needs to know the kind of object that it is dealing with and, if it is a call to a special form, check its well formedness.

Instead of implementing this kind of behavior with a conditional statement, we use the *SpecialFormHandler* instance of the *Chain of Responsibility* design pattern. When a `ScConsExpression` object receives an `analyze` message, it forwards this message to a predefined chain of handlers. The framework defines a handler for each of the special forms that is used in our Scheme interpreter. We have a separate handler for a `define`, `lambda` and `let` special form, for example.

The `analyze:` method in a specific handler object simply checks whether this object can handle the `ScConsExpression` object passed along. If so, it calls the `handle:` message, which will instantiate the appropriate closure object after having checked the well formedness of the expression. If the object can not handle the `ScConsExpression` object, it simply forwards it to the next handler in the chain by calling the `analyze:` method of that handler.

To avoid falling off the end of the chain, the last handler is the `ApplicationHandler`, which has no successor. This reflects the fact that, if a request is passed along the chain and no handler of a special form can handle it, we simply regard the `ScConsExpression` object as a call to an ordinary function.

3.4.3 The *Factory Method* design pattern

General Discussion

The *Factory Method* design pattern defines an interface for creating an object, but lets subclasses decide which concrete class to instantiate. This is useful because often, two classes need to collaborate and the one class knows *when* to instantiate the other, but it cannot anticipate *which* specific class it needs to instantiate. Using the *Factory Method* design pattern, we define an abstract method that is responsible for creating the appropriate object and consistently call this method when such an object is needed. As such, we create a hook for subclasses that can override this factory method and instantiate an object of the appropriate class.

The specific benefits of the *Factory Method* design pattern are the following. First, it provides a hook for subclasses, which allows an abstract class to implement certain behavior regardless of the concrete objects that are used. Subclasses only need to override the factory method and create a different object in order to change this behavior. Second, the pattern can be used to connect parallel class hierarchies. This is useful when objects delegate some of their responsibilities to other objects, and only some specific combinations of these objects are allowed. This often results in two parallel class hierarchies (as can be seen in Figure 3.4 for example) and the factory method acts as the bridge between the two.

The *Factory Method* and *Abstract Factory* design patterns may seem very similar. Both design patterns deal with the process of object instantiation. In fact, the *Abstract Factory* design pattern is often implemented with factory methods. The major difference between the two design patterns, is that the *Abstract Factory* design pattern can be used to ensure that a family of related objects is created, and that it provides a framework with a single spot in which objects are created, so that it becomes easy to switch between different object families. The *Factory Method* design pattern on the other hand is mostly used to connect two parallel class hierarchies and is often combined with the *Template Method* design pattern to factor out common behavior and provide hooks for subclasses.

The *Factory Method* design pattern in the Scheme Framework

The *Factory Method* design pattern is used four times in the Scheme framework.

First of all, the *ExpressionClosureCreation* instance connects the `ScExpression` and the `Closure` hierarchies (see Figure 3.4). A `Closure` object closely resembles the structure of the abstract syntax tree represented by an `ScExpression` object: for each concrete `ScExpression` class, a corresponding `Closure` class exists. As a consequence, each concrete class in the `ScExpression` hierarchy is associated with a concrete class in the `Closure` hierarchy via a factory method `newClosure`. This method is defined as an abstract method in the `ScExpression` class, and is overridden in all of its concrete subclasses, where it creates the appropriate `Closure` object.

The second and the third instance of the *Factory Method* design pattern are located in the `SpecialFormHandler` hierarchy. Similar to the use in the `ScExpression` hierarchy, the *SpecialFormClosureCreation* instance is used in the `SpecialFormHandler` hierarchy to connect the various handler classes to their corresponding `Closure` classes by means of a `newClosure` method, as depicted in Figure 3.5.

If we are dealing with a Scheme special form, we first need to check the well formedness of the expression before we can transform it into the appropriate closure object. To this extent, we use the classes from the `SchemeConverter` hierarchy. Once again, each specific handler class needs an appropriate `SchemeConverter` class. Thus, we use a *SpecialFormConverterCreation* instance of the *Factory Method* design pattern to connect the two parallel hierarchies, as can be seen in Figure 3.6.

The last occurrence of the *Factory Method* design pattern, the *EnvironmentCreation* instance, is located in the hierarchy that represents the environments used for the evaluation process (see Figure 3.7). The `RootEnvironment` class defines a factory method `newEnvironment`, that returns a `NormalEnvironment` instance. The method is called whenever a procedure call occurs (see [AS85] for a detailed explanation of environments). Note that the implementation in this instance of

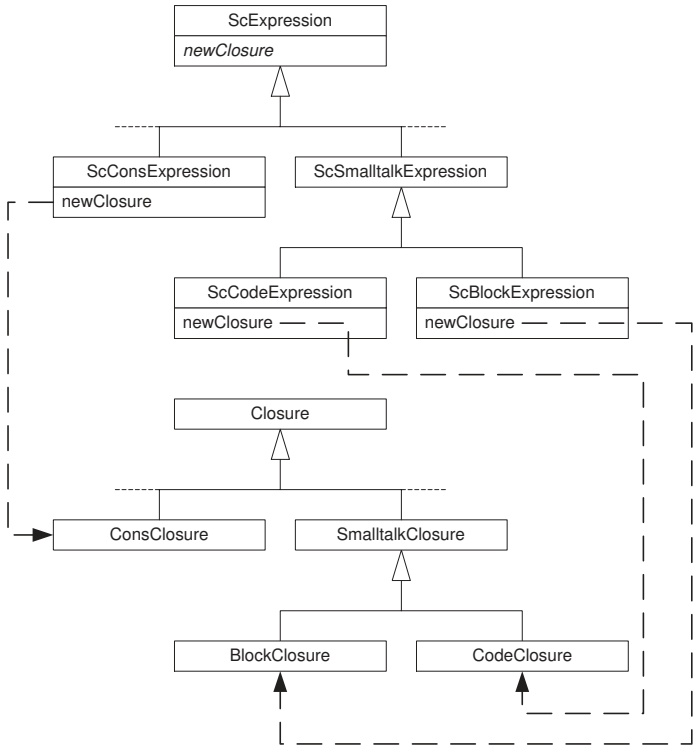


Figure 3.4: *ExpressionClosureCreation* instance of the *Factory Method* design pattern

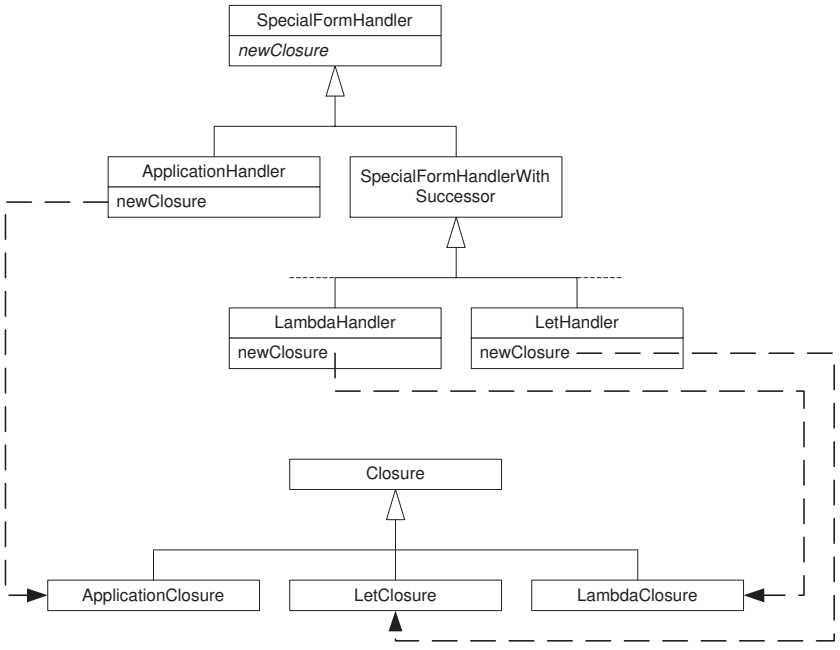


Figure 3.5: *SpecialFormClosureCreation* instance of the *Factory Method* design pattern

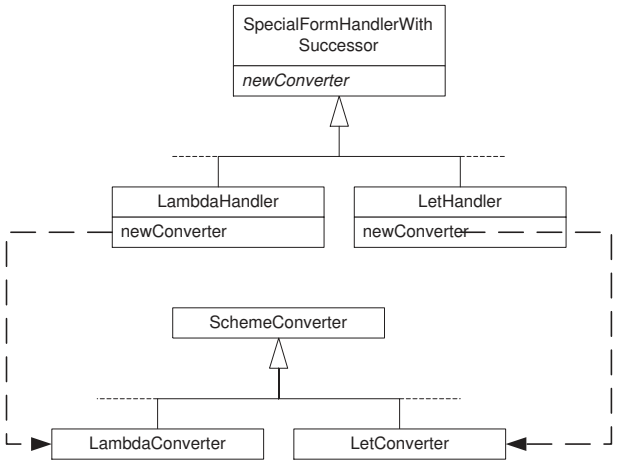


Figure 3.6: *SpecialFormConverterCreation* instance of the *Factory Method* design pattern

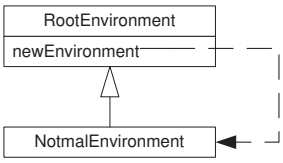


Figure 3.7: *EnvironmentCreation* instance of the *Factory Method* design pattern

the design pattern deviates from the "standard" implementation of the *Factory Method* design pattern. First of all, the factory method returns instances of classes from the same hierarchy in which the method is defined. Second, the factory method in the root of the class hierarchy (the `RootEnvironment` class) provides a default implementation and is never overridden in any of the concrete leaf classes of the hierarchy. This is due to the fact that, every time the method is called, a new instance of the same class, the `NormalEnvironment` class, has to be allocated.

3.4.4 The *Composite* design pattern

General Discussion

The *Composite* design pattern is used to compose objects into tree structures to represent part-whole hierarchies. It allows us to treat individual objects and compositions of objects in a uniform way. This is achieved by defining an abstract class that implements an appropriate interface for all objects in the composition. All subclasses of this abstract class represent the leaf classes of the composition, except one, which represents the composite object and contains a collection of references to other objects. Typically, the leaf classes provide their own concrete implementation for the interface defined by the abstract class, while the composite class implements the interface by iterating over the objects it contains and simply forwarding the message.

Using the *Composite* design pattern offers some important advantages. First of all, it allows us to represent and build compositions of objects that can be manipulated in an easy and straightforward way. Second, it greatly simplifies client code that has to deal with both individual and composite objects. Those objects have the same interface, which allows them to define the action that should be performed in response to a certain message in an appropriate way (e.g. composite objects iterate over their elements and forward the message).

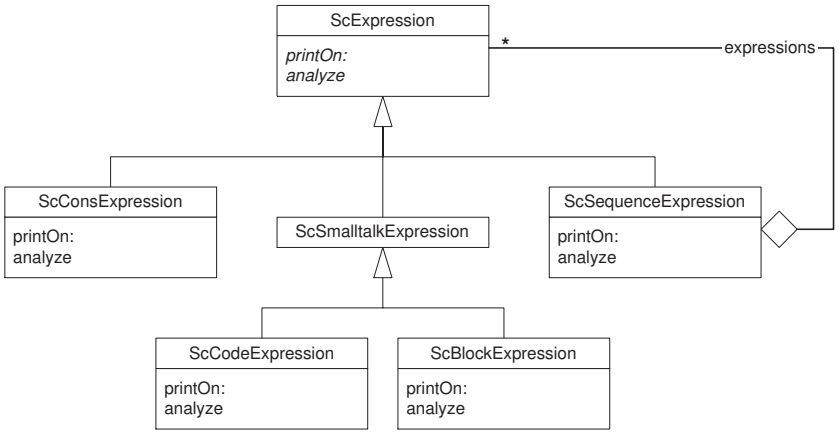


Figure 3.8: *CompositeExpression* Instance of the *Composite* design pattern

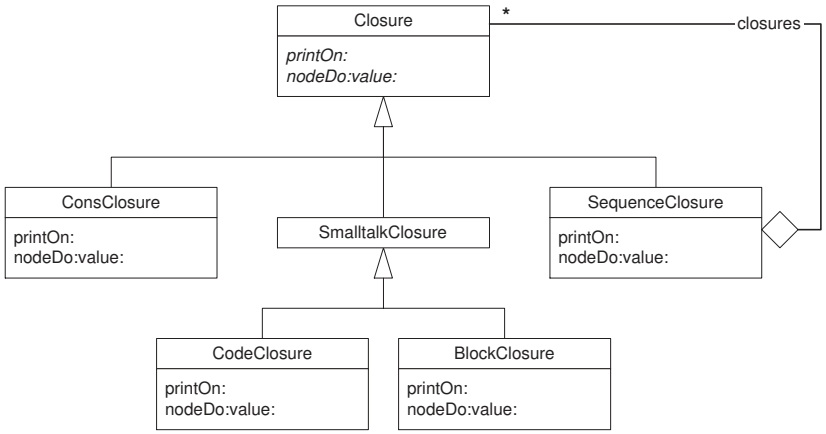


Figure 3.9: *CompositeClosure* Instance of the *Composite* design pattern

The *Composite* design pattern in the Scheme Framework

Two instances of the *Composite* design pattern can be found in the Scheme framework.

The `ScExpression` hierarchy uses a *CompositeExpression* instance, where the `ScSequenceExpression` represents a composite object that is part of an abstract syntax tree (Figure 3.8). Such an object is created by the parser when the body of a user-defined procedure consists of multiple expressions, for example. Two important operations are defined on this instance of the pattern: `printOn:` is used to print a textual representation of the receiver on a given stream, and `analyze` is used to transform an expression object into a closure object. The implementation of these operations in the `ScSequenceExpression` class resembles the standard implementation of a composite method in that it iterates over the objects contained within the composite object and simply forwards the message onto these objects. For example, the `printOn:` method in the `ScSequenceExpression` class looks as follows (in Smalltalk syntax):

```
printOn: aStream
   self expressions do: [:expr | expr printOn: aStream ]
```

As every class in the `ScExpression` hierarchy is associated with a class in the `Closure` hierarchy, the `ScSequenceExpression` class has a `SequenceClosure` counterpart, which represents a composite closure object (see Figure 3.9, which depicts the *CompositeClosure* instance). Such

an object thus consists of other closure objects. The interface defined for this instance consists of only two methods: `printOn:`, which again prints a textual representation of the receiver onto a given stream, and `nodeDo:value:` which is part of the implementation of the *Visitor* design pattern (see Section 3.4.5). The `printOn:` method is once again implemented in the standard way in the `SequenceClosure` class. The `nodeDo:value:` method on the other hand provides its own implementation and does not follow the template implementation dictated by the *Composite* design pattern.

3.4.5 The *Visitor* design pattern

General Discussion

The *Visitor* design pattern allows us to represent an operation that is to be performed on the elements of an object structure and to define new operations without changing the classes of the elements on which they operate. This is useful when many distinct and unrelated operations need to be performed upon objects in a structure and defining all these operations in the classes of these objects would pollute their interface. The underlying idea of the *Visitor* design pattern is to separate the operations from the elements upon which they will be performed. To do so, a separate *visitor* hierarchy is defined for representing the operations, and a specific *visit* method is introduced in the element classes to enable collaboration between the two hierarchies. A concrete element "accepts" a visitor object, sends it a message that is specific for this element and passes itself as an argument. The visitor object will then perform the operation for that specific element as a reaction to the message.

Using the *Visitor* design pattern has some particular advantages over implementing an operation in the element classes. First of all, it makes adding new operations easy. This simply boils down to defining a new subclass in the visitor hierarchy and passing an instance of this subclass to the visit method of the first object of the object structure. There is no need to make any changes to the element classes that make up the object structure. Second, each operation is represented by a concrete visitor, which means that the behavior implemented by that operation is not spread over the different element classes, but that it is localized in one place.

The *Visitor* design pattern in the Scheme Framework

The *Visitor* design pattern is used in the Scheme framework for implementing operations on the `ScExpression` hierarchy. Each concrete class in this hierarchy implements a `nodeDo:` method, that calls the appropriate method of the `AbstractASTEnumerator` class, as is depicted in Figure 3.10. At the moment, only one such operation is actually provided by the framework. The `SchemeToSmalltalkConverter` visitor writes a Smalltalk representation of a Scheme expression onto a stream. This is used to serialize Scheme expressions, so that we can define methods in an ordinary Smalltalk class that consists of Scheme functions. The representation that is used is actually Smalltalk code that builds the abstract syntax tree of the Scheme expression and evaluates it.

We can however easily imagine that other operations need to be defined on the abstract syntax tree. For one, the `printOn:` and `analyze` methods are also candidates to be implemented via the *Visitor* design pattern, because they are implemented by all of the concrete leaf classes of the `ScExpression` hierarchy. In case of `printOn:`, the specific reason we do not implement it by means of a visitor class is because it would seriously degrade performance. Indeed, the `printOn:` method is part of the standard Smalltalk framework, and is used many times, so using a visitor mechanism is very costly. The `analyze` method on the other hand is also not implemented by means of a `Visitor` class, because it is spread over two separate hierarchies, the `ScExpression` and `SpecialFormHandler` hierarchies, and the latter does not use the *Visitor* design pattern. It is thus impossible to centralize the behavior of the analysis algorithm in one class, so it remains spread over the different classes.

Another instance of the *Visitor* design pattern, the *ClosureVisitor* instance, appears in the `Closure` hierarchy (see Figure 3.11). This hierarchy defines a `nodeDo:value:` method, that is

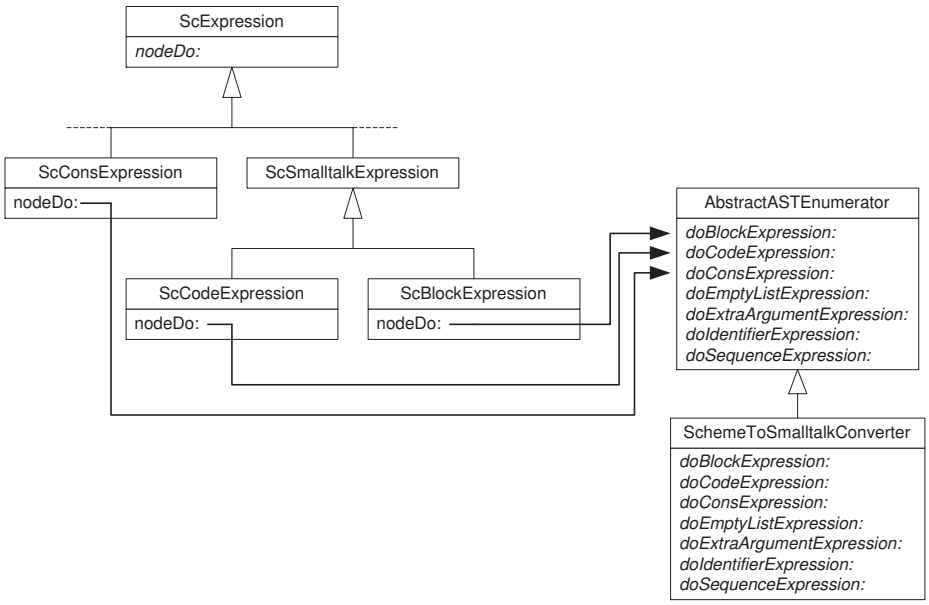


Figure 3.10: *ASTVisitor* instance of the *Visitor* design pattern

overridden in each concrete leaf class and calls the appropriate method of the `ClosureVisitor` hierarchy. This hierarchy contains a number of concrete visitors. The `SimpleEvalClosureVisitor` class implements the evaluation rules of the Scheme language and is thus used to evaluated the various closure objects, for example.

It is interesting to note that the classes in the `Closure` hierarchy and the `nodeDo:value:` method they implement, participate in both *Composite* and *Visitor* design pattern instances. This is a perfect illustration of how design pattern instances can overlap, when they contain the same participants.

3.4.6 The *Strategy* design pattern

General Discussion

The purpose of the *Strategy* design pattern is to define a family of algorithms, encapsulate each one in a separate class and make them interchangeable. This way, clients can be parameterized by a specific strategy object and thus need not be changed when they want to use a different algorithm.

The *Strategy* design pattern offers many benefits. For one, it encapsulates each algorithm in a separate class, thereby making it more explicit. Inheritance can then be used to capture the commonalities between these algorithms in a common superclass and reusing them in the specific subclasses. Furthermore, capturing related algorithms into a hierarchy makes them interchangeable for clients. Also, by encapsulating the algorithm and using inheritance, polymorphism and delegation, the *Strategy* design pattern can often be used to avoid writing conditional statements.

The *Strategy* design pattern in the Scheme Framework

The *Strategy* design pattern is used two times in the Scheme framework. First of all, the *ApplyStrategy* instance represents the various evaluation models that exist for the Scheme language (Section 3.2.2). This particular instance of the design pattern is depicted in Figure 3.12. The `ApplyStrategy` class is an abstract class that defines only one method: `apply:onOperands:inEnvironment:`. This method is overridden in all concrete subclasses of `ApplyStrategy`. In

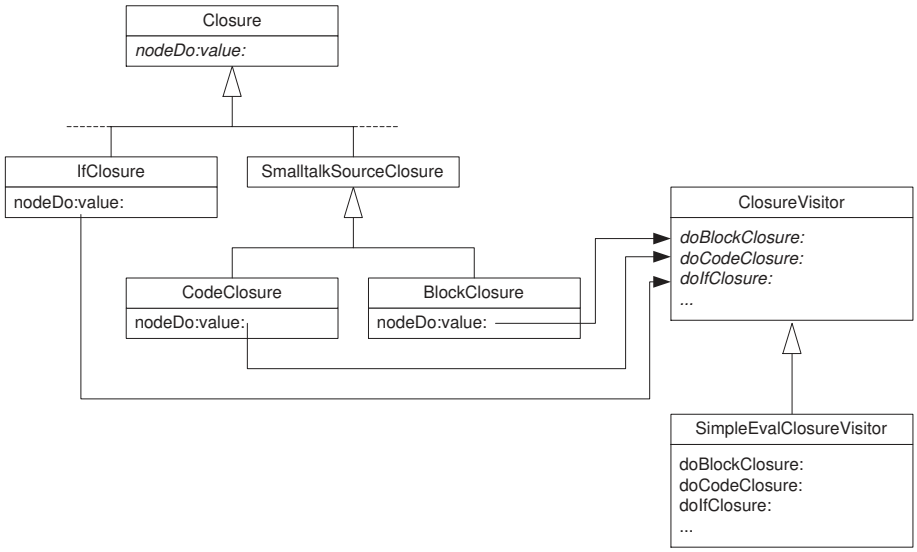


Figure 3.11: *ClosureVisitor* instance of the *Visitor* design pattern

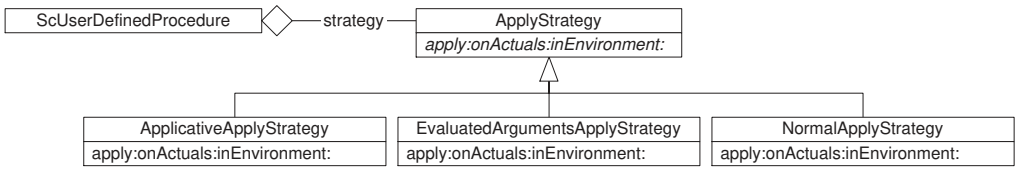


Figure 3.12: *ApplyStrategy* instance of the *Strategy* design pattern

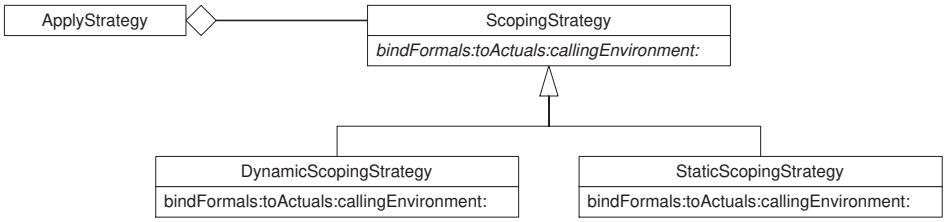


Figure 3.13: *ScopingStrategy* instance of the *Strategy* design pattern

the `ApplicativeApplyStrategy` and `NormalApplyStrategy` classes, this method implements the applicative order and normal order evaluation models respectively, while in the `EvaluatedArgumentsApplyStrategy` it is used to apply a procedure on operands that are already evaluated. This is often needed to implement *meta-procedures*, such as `eval` and `apply`, for example.

The second instance of the *Strategy* design pattern, the *ScopingStrategy* instance, is used for implementing static and dynamic scoping. Figure 3.13 shows how the `ScopingStrategy` class implements an abstract method `bindFormals:toOperands:callingEnvironment:`. This method is overridden in the two leaf classes of the hierarchy, `StaticScopingStrategy` and `DynamicScopingStrategy`, to implement static and dynamic scope algorithms respectively. The method is called by the `apply:onOperands:inEnvironment:` method of the `FunctionApplyStrategy` hierarchy, whenever a procedure call is evaluated.

3.4.7 The *Template Method* design pattern

General Discussion

The *Template Method* design pattern is used to define a skeleton of an algorithm in a method, while some specific steps of the algorithm are deferred to the subclasses. These subclasses can change the specific parts of the behavior of the algorithm without changing its overall structure. This is achieved by defining a specific method, called the template method, that calls certain other (possibly abstract) methods of the same class. Subclasses are allowed to override these methods, while the template method should preferably never be overridden.

The *Template Method* design pattern is often used to avoid code duplication, by factoring out code common to a number of subclasses in an abstract superclass. The commonalities of the methods in the subclasses are captured in the template method, which is defined in an abstract class and which calls some specific abstract methods. These abstract methods represent the variabilities of the method and are overridden in the various subclasses to provide the subclass-specific behavior. Template methods are thus a fundamental technique for code reuse. Furthermore, using the *Template Method* design pattern leads to an inverted control structure, as the parent class calls the methods of a subclass and not the other way around. The *Template Method* design pattern can also be used to control subclass extension. Defining a template method that calls some specific other (abstract) methods at certain points, only permits extensions at exactly these points.

The *Template Method* design pattern in the Scheme Framework

The *Template Method* design pattern appears two times in the Scheme framework.

The first usage of this pattern is in the `SpecialFormHandler` hierarchy. Figure 3.14 depicts the *SpecialFormHandlerTM* instance. The `analyze:` method of the `SpecialFormHandler` class is a template method. Its responsibility is to check whether a certain handler object is capable of analyzing the expression that is passed to it and take appropriate action based on this decision. This behavior is implemented by first sending the message `canHandle:` to the object itself. `canHandle:` is an abstract method in the class `SpecialFormHandler`, as this class can not decide which of its subclasses is able to analyze the expression. Each concrete handler subclass

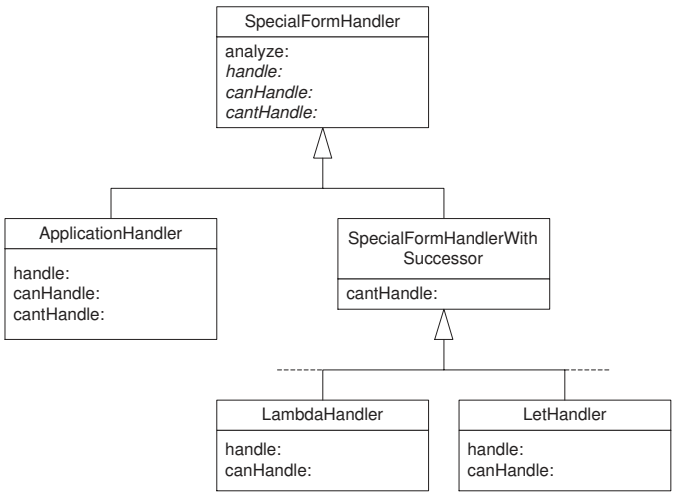


Figure 3.14: *SpecialFormHandlerTM* Instance of the *Template Method* design pattern

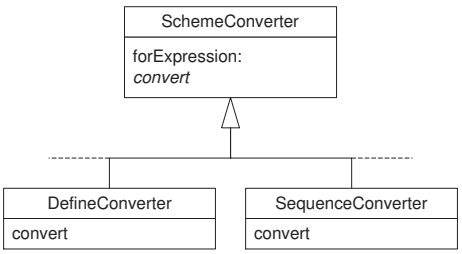


Figure 3.15: *ConverterTM* instance of the *Template Method* design pattern

of `SpecialFormHandler` will thus have to override the `canHandle:` method. If it turns out the handler can analyze the expression, `canHandle:` will return true and `analyze:` will proceed to call the `handle:` method, which will check the well-formedness of the expression and return the appropriate `Closure` object. Once again, the `SpecialFormHandler` class itself can not know how the well formedness of the expression can be checked and which closure object should be returned. This can only be decided in the specific subclasses of this class. Thus, `handle:` is declared abstract in the `SpecialFormHandler` class and is overridden by all its concrete subclasses.

If the handler is not able to analyze the expression, the `canHandle:` method will return false and the `analyze:` method will call the `cantHandle:` method. This method is again declared abstract in the `SpecialFormHandler` class, and is overridden only in the `ApplicationHandler` and `SpecialFormHandlerWithSuccessor` classes. In the latter case, this method simply forwards the `analyze:` method to the next handler in the chain. The implementation of the `cantHandle:` method in the `ApplicationHandler` class simply raises an error, as this is not possible since this handler can analyze any expression.

The second occurrence of the *Template Method* design pattern is in the `SchemeConverter` hierarchy, where the *ConverterTM* instance is used. (Figure 3.15). When a specific `Converter` object is instantiated, it is initialized by calling the `forExpression:` method. This method is a template method which calls the `convert` method on the receiver. The latter is an abstract method that is overridden by all concrete subclasses and that does the actual conversion of the expression, i.e. it breaks the expression into the appropriate subexpressions.

3.4.8 The *Singleton* design pattern

General Discussion

The *Singleton* design pattern is used to ensure that only one instance of a specific class can be used by an application. Furthermore, it provides a unique access point for that instance. This is achieved by making the class itself responsible for keeping track of its instance. The class implements a class method, that is called whenever an instance of the class is needed. The method simply checks whether an instance of the class already exists. It this is not the case, it creates one, and returns it. If an instance already exists, it is simply returned immediately.

The *Singleton* design pattern offers the advantage that it controls how and when clients of a class access its instances. It allows to reduce the space requirements of an application by ensuring that no more than one instance of a class is created. If the need arises, however, the *Singleton* design pattern can easily be adapted to allow for a number of instances to be created. Furthermore, the design pattern provides a unique access point for the instance of a class, but it avoids using a global variable to do so. As such, the name space of a program is not polluted with unnecessary global variables.

The *Singleton* design pattern in the Scheme framework

The *Singleton* design pattern is used three times in the Scheme framework. The `SchemeASTFactory, ScopingStrategy` and `RootEnvironment` classes all need to be instantiated exactly one time. An instance of the *Abstract Factory* design pattern is usually combined with an instance of the *Singleton* design pattern [GHJV94], as an abstract factory does not contain state and does only needs to be instantiated once. Thus, the `SchemeASTFactory` is implemented as a singleton. Since the `ScopingStrategy` class also does not contain state, it uses the *Singleton* design pattern as well. The `RootEnvironment` class, on the other hand, does contain state. Since there can only be one global environment in the Scheme system, however, this class can also be instantiated only once. The `NormalEnvironment` subclass of `RootEnvironment`, however, can and needs to be instantiated many times, since each time a procedure is called, a new environment object has to be created.

3.4.9 Summary

In this section, we explained the purpose of a number of important and widely used design patterns, and documented their occurrences in the Scheme framework. In this way, we explained the design and, up to a certain extent, the implementation of the Scheme framework. In the next section, we will show how the information conveyed within the design patterns can be used to guide a developer when evolving the Scheme framework.

We wish to point out that the documentation of the design and implementation of the framework by means of design patterns proves to be very accurate. 94 % of all classes present in the framework participate in one or more design pattern instance. This is of course due to the fact that all important class hierarchies of the framework participate in at least one design pattern instance. Likewise, 76 % of all methods in the framework play a role in one or more design pattern instances. This last number does not take into account trivial methods, such as initialization or accessor methods, with no really interesting behavior. Documenting the framework by means of design patterns thus covers a large amount of the important classes and methods of the framework.

3.5 Evolving the Scheme Framework: Adding a New Special Form

3.5.1 Introduction

The Scheme interpreter as implemented by the framework only supports a subset of the special forms that are defined by the Scheme language. In this section: we will show how a *cond* special form can be added to the framework. This special form represents a conditional with multiple branches, as opposed to the *if* special form already present in the framework that has only one condition, one consequent and one alternate branch. This example serves to explain the difficulties that arise when evolving a framework in detail, which allows us to clarify the problem statement of this dissertation and elaborate on the specific solution we propose to alleviate it in the next chapter. Note that the original framework developers designed the framework in such a way that new special forms can be added easily. We are thus dealing here with an example of anticipated evolution of the framework.

As we will see, evolving a framework requires being able to identify the part of the framework that needs to be changed, inferring which changes are necessary and assessing the impact of these changes on other parts of the framework. We will show that high-level knowledge about the global architecture of a framework can already help in identifying those parts of the framework that will need changes. However, the kind of information this architectural view offers is far from sufficient to identify the particular changes that should be made to the identified parts. This is due to the fact that the information about the architecture is too high level. In order to be able to evolve the framework correctly, lower-level information about the specific relationships and interactions in the framework and the responsibilities of individual classes and methods is necessary.

3.5.2 Consulting the Global Architecture

By consulting the global architecture of the framework (see Figure 3.1 and Section 3.3), we can already infer some important information. The parser should not be extended. The syntax, and thus the abstract syntax tree, of a *cond* special form is similar to the syntax of other special forms and user-defined procedures. Such expressions can already be parsed and an appropriate abstract syntax tree can be constructed for them. The analyzer, however, will need to be adapted. It is the analyzer's duty to check the well-formedness of an expression and convert it into an appropriate `Closure` object that can be evaluated. For special forms, this behavior is implemented by the various classes in the `SpecialFormHandler` hierarchy. A new `CondHandler` class, corresponding to the *cond* special form, should thus be added to this hierarchy. To check the well-formedness of an expression, the `CondHandler` class makes use of an appropriate converter object. As such, a new class, `CondConverter`, should be added to the `SchemeConverter` hierarchy that is responsible for breaking down a *cond* expression into its constituent parts. Furthermore, a new class, `CondClosure`, that represents a *cond* expression should be added to the `Closure` hierarchy as well. This class is involved in the evaluation process, and the appropriate evaluation rules for a *cond* expression should be implemented on it.

Although it is clear that valuable information can be deduced from a high-level architectural description of the framework, this information is far from sufficient. We only know that a number of classes has to be added to some particular class hierarchies. This reveals nothing however about the specific ways in which these classes interact and collaborate. We do not know which methods need to be implemented by these classes, or which methods they need to override. Furthermore, even if this would be known, we have no information about how these methods should behave. Which other methods do they need to call, for example? Moreover, the architecture reveals little or no information that can be used to assess the impact of a particular change on the rest of the framework. The architecture presents knowledge about how the different parts of the framework are related, but this knowledge is too coarse grained. It only allows us to identify the related parts of the framework, which may need changes simply because they are related, but it remains difficult to identify what these changes are. To overcome all these problems, more detailed information

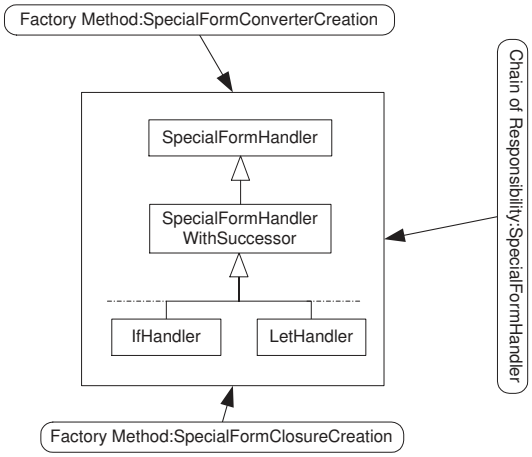


Figure 3.16: Design Patterns in which the `SpecialFormHandler` hierarchy participates

about the framework is needed. In the following sections, we will show how the design patterns that are present in the framework can help in inferring the necessary information.

3.5.3 Consulting the Design Patterns

From the discussion on design patterns presented in the previous section, we know that all special forms are explicitly represented in the Scheme framework as a subclass of the `SpecialFormHandlerWithSuccessor` class. As such, we introduce a new subclass `CondHandler` of this class. We will now discuss which methods this class should define and which other classes it should use to implement its behavior. In concreto, when adding, removing or changing a particular source code entity, we will always consider the design patterns in which this entity participates. We will check whether the proposed evolution results in an implementation that conforms to the rules of these design patterns, and if not, perform additional changes that make sure the implementation again adheres to these rules.

Changes to the `CondHandler` class

When adding a class to a hierarchy, we should first consider its place in that hierarchy, e.g. we should determine which class should be its superclass and which classes should be its subclasses. In this particular case, the `CondHandler` class is a concrete class and should thus be added as a leaf class to the `SpecialFormHandler` hierarchy. Additionally, we should consider which methods the new class should implement. Since the `CondHandler` class is a concrete class, it is clear that it should provide a concrete implementation for all abstract methods defined in the hierarchy. If we do not consider the additional information provided by design patterns, we can not provide an appropriate implementation for these methods automatically. These are thus the only changes that we can identify.

If we do consider the various design pattern instances in which the `CondHandler` class plays a role, we can identify more changes, and even infer the appropriate implementation for a number of methods. The `CondHandler` class forms part of the `SpecialFormHandler` hierarchy, which plays a role in three different patterns (Figure 3.16).

First of all, all concrete subclasses of the `SpecialFormHandlerWithSuccessor` class play the role of concrete handler classes in the *SpecialFormHandler* design pattern instance (Section 3.4.2). As such, these classes should all implement the interface for handling requests. In this particular case, this behavior is implemented by one method, the `analyze:` method, which should be implemented by all concrete handlers, and thus by the `CondHandler` class.

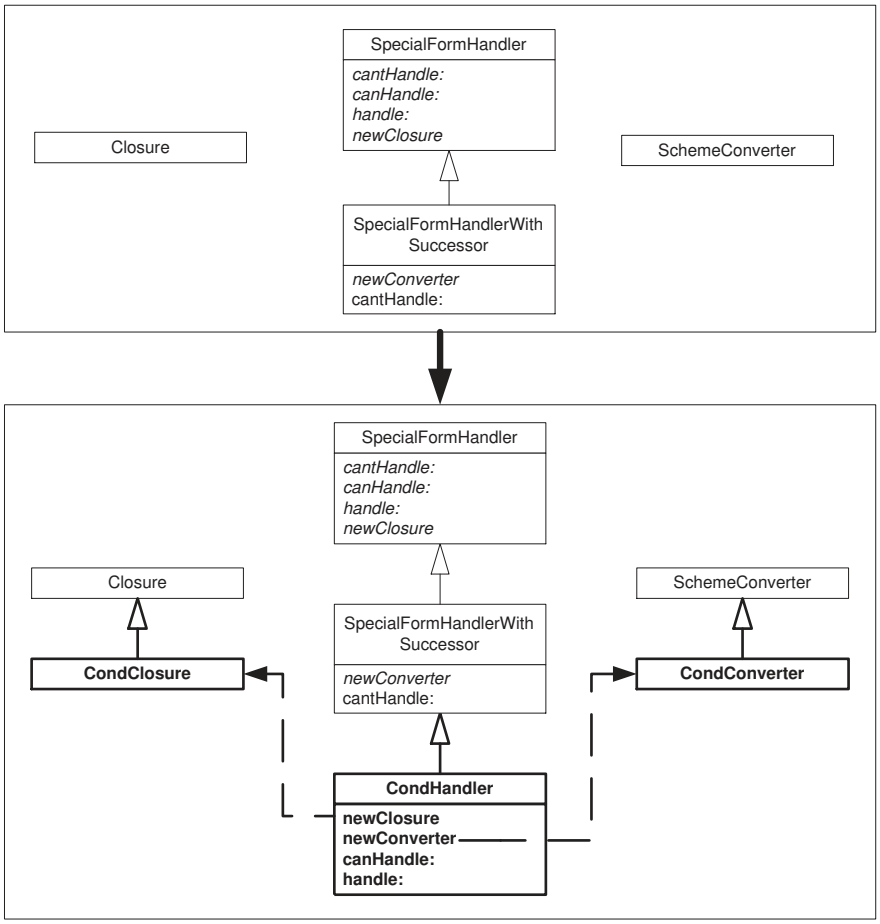


Figure 3.17: Changes to the `CondHandler` class

Second, all concrete subclasses of the `SpecialFormHandlerWithSuccessor` class play a role in the *SpecialFormClosureCreation* design pattern instance (Section 3.4.3). Each concrete handler class is associated with a concrete `Closure` class via the `newClosure` factory method. Thus, the `CondHandler` class should implement this method. In this particular case, we can even provide an implementation for this method automatically: it should simply instantiate a class from the `Closure` hierarchy that can handle a *cond* expression and return the resulting object. Such a class does not yet exist, as the *cond* special form is just introduced into the framework. A new `CondClosure` class thus has to be added to the `Closure` hierarchy.

Furthermore, the `CondHandler` class also plays a role in the *SpecialFormConverterCreation* instance of the *Factory Method* design pattern. Each concrete handler class is associated with a concrete converter class through the `newConverter` factory method. As a consequence, this method should be implemented by the `CondHandler` class. Once again, we can provide an implementation for this method: its responsibility is to return an instance of the appropriate object of the `SchemeConverter` hierarchy. Since no converter class for a *cond* expression is originally defined in the framework, it should be added. The `SchemeConverter` hierarchy is thus extended with a `CondConverter` class.

Third, the `CondHandler` class plays a role in the *Template Method* design pattern (Section 3.4.7). The `analyze:` method defined in the `SpecialFormHandler` abstract class relies on three other abstract methods: `canHandle:`, `handle:` and `cantHandle:`. These three methods need to be defined by all concrete handler classes, which should provide the appropriate implementation. The `CondHandler` class only needs to implement two of these methods: `canHandle:` and `handle:`. The `cantHandle:` method is implemented by the `SpecialFormHandlerWithSuccessor` class and provides the default behavior of forwarding the request to the next handler in the chain.

Figure 3.17 shows the result when all these changes have been applied to the `CondHandler` class. Note how a `CondClosure` and a `CondConverter` class have been added to the `Closure` and `SchemeConverter` class hierarchies. These classes do not yet contain an implementation for the methods that they should define. Determining which methods should be added will be discussed next.

Changes to the `CondClosure` class

As was discussed in the previous section, when implementing the `newClosure` method, a new `CondClosure` class should be added to the `Closure` hierarchy. When adding this class, we should check whether it plays a role in any of the design patterns used by the framework. If so, we should check that it adheres to the constraints imposed by these design patterns in order to ensure its correct behavior.

As it turns out, the `CondClosure` class plays a role in the *CompositeClosure* design pattern instance used in the `Closure` hierarchy (Section 3.4.4). As such, it should all implement the composite methods defined by this design pattern instance. The `CondClosure` class should thus provide a concrete implementation for the `printOn:` and the `nodeDo:value:` methods, as these are the only composite methods present in that design pattern instance.

Furthermore, since the `Closure` hierarchy also participates in the *ClosureVisitor* instance, other changes are mandatory. Each concrete subclass in the `Closure` hierarchy has an associated method in the `ClosureVisitor` hierarchy, and so should the new `CondClosure` class. Thus, a new `doCondClosure:` method is introduced in the `ClosureVisitor` class, and a concrete implementation is provided for all of its subclasses.

Figure 3.18 shows the changes that have been applied to the `Closure` and `ClosureVisitor` hierarchies.

Changes to the `CondConverter` class

Based on a similar reasoning as above, the addition of a `newConverter` method to the `CondHandler` class required adding a new `CondConverter` class to the `SchemeConverter` hierarchy. Once again,

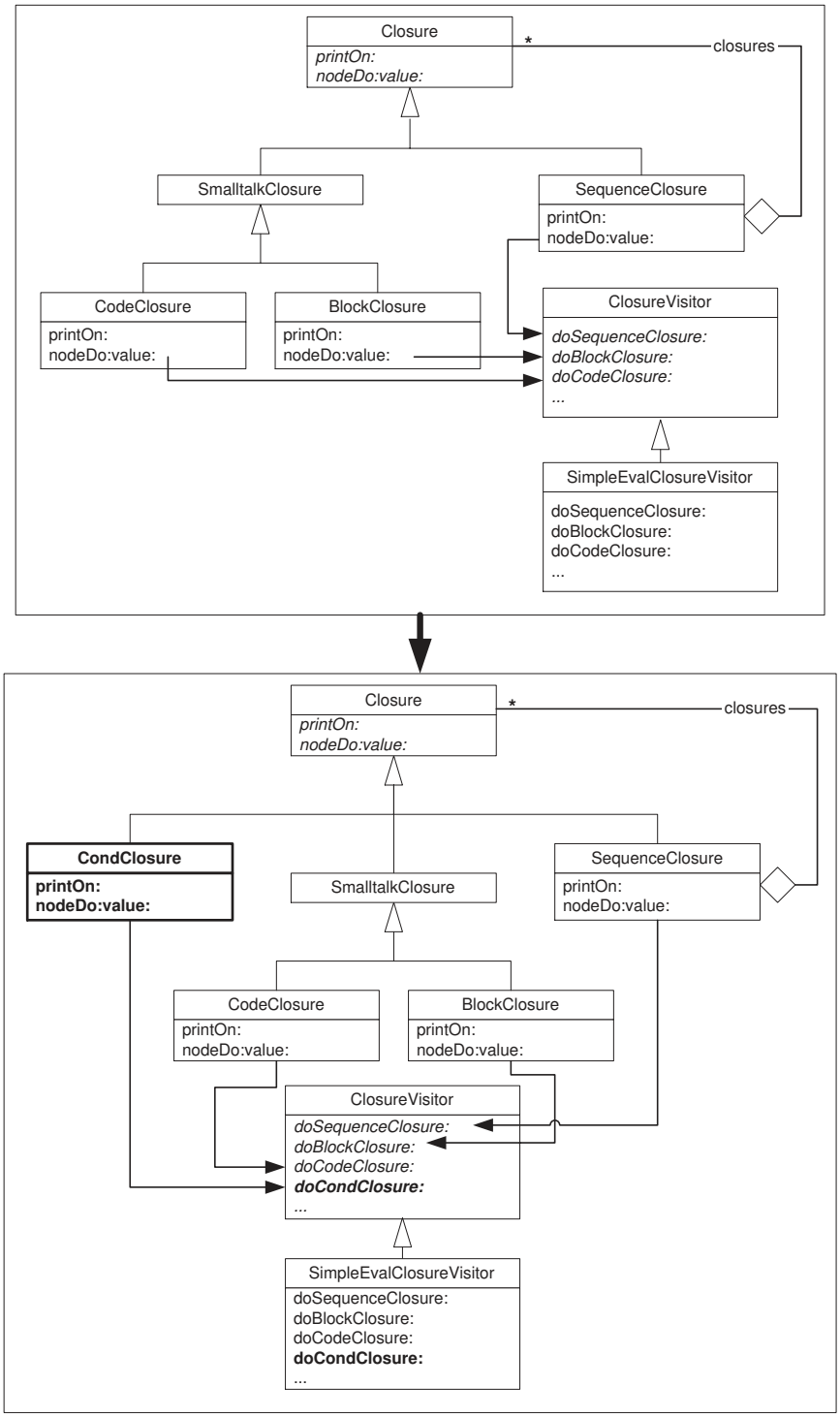


Figure 3.18: Changes to the `CondClosure` class

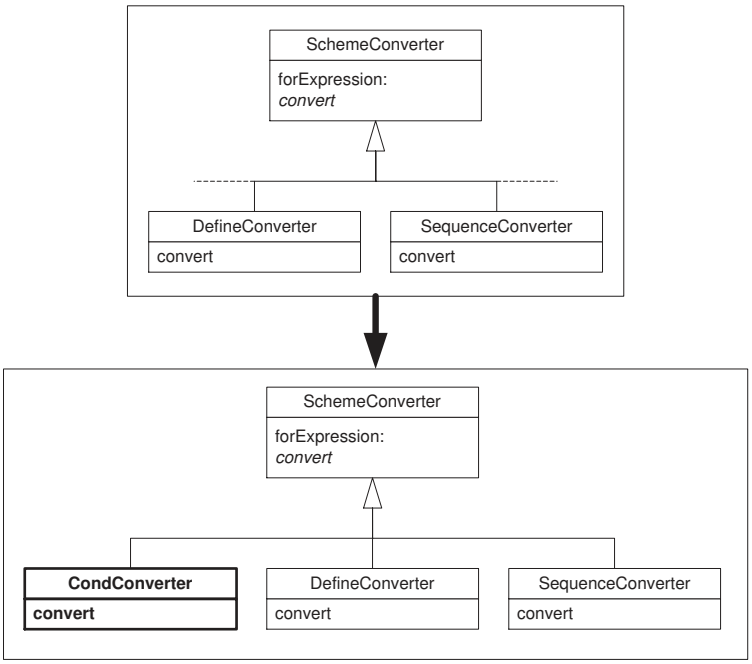


Figure 3.19: Changes to the `CondConverter` class

we should check if there are any design patterns in which this class plays a role in order to ensure that it adheres to the constraints imposed by these patterns.

The `SchemeConverter` hierarchy plays a role only in the *ConverterTM* design pattern instance (Section 3.4.7). All concrete subclasses of the `SchemeConverter` class should thus override this method and provide it with the appropriate implementation. Since the `CondConverter` class is such a concrete subclass, it should define and implement a `convert` method. Figure 3.19 shows the result after changing the `SchemeConverter` hierarchy.

Discussion

As can be observed, information about the design pattern instances used in the framework is of great help in performing the evolution correctly. Without this information, we would only be able to add the `CondHandler` class to the appropriate hierarchy, and we would only know that all abstract methods defined by the hierarchy should be overridden in that particular class. We would not be able to infer what the implementation of these methods should look like, nor could we know which other classes should be used to implement the appropriate behavior.

By considering the design pattern instances in which the `SpecialFormHandler` hierarchy participates, we can infer much more information. We were able to provide the `newClosure` and `newConverter` methods with a concrete implementation, because we explicitly knew they participated in an instance of the *Factory Method* design pattern. Moreover, we were able to infer that two new classes should be added in response to adding these methods: a `CondConverter` class to the `SchemeConverter` hierarchy and a `CondClosure` class to the `Closure` hierarchy. By examining the design pattern instances in which these classes participate, we were able to identify which methods these classes in their turn needed.

Figure 3.20 shows all changes that have been applied to add the *cond* special form.

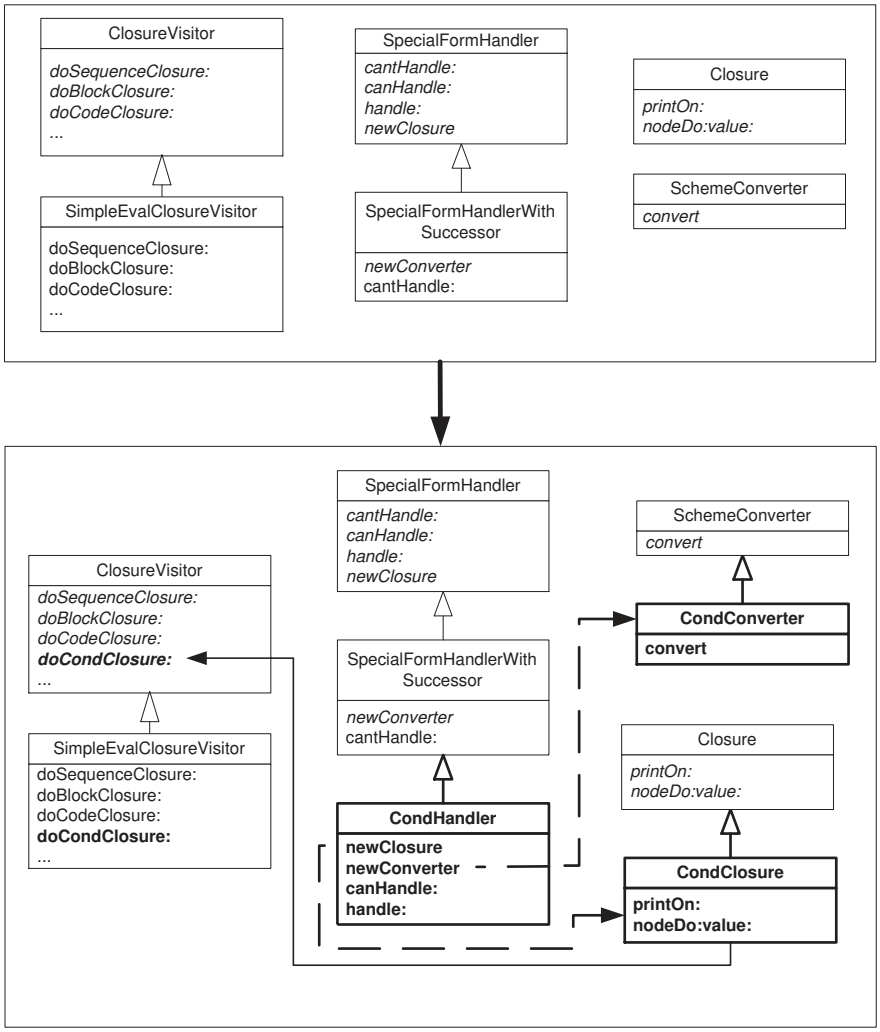


Figure 3.20: Adding a new special form to the Scheme interpreter

Chapter 4

Problem Statement & Proposed Solution of the Dissertation

In this chapter, we will first elaborate upon the problem that we want to tackle in this dissertation. This discussion will be based upon generalizations of the example evolution presented in the previous chapter. Afterwards, we will explain the specific solution we propose to solve the problems identified before.

4.1 Problem Statement

4.1.1 Identification and Propagation of Changes

When evolving a framework, we should first of all identify which changes should be made to its design and implementation. To identify the changes that are necessary, we should first know which parts of the framework should be changed (change propagation), and afterwards derive the specific changes that those parts should undergo (change identification).

Change Propagation

Evolution will bring about various changes to the existing design of the framework. Classes, methods and variables may be added or removed, relationships between classes changed, class hierarchies reorganized or method implementations adapted. When performing such changes, their impact should be assessed, so as to ensure that the framework still adheres to the intended design and exhibits the appropriate behavior. Often, this will require that a particular change be followed by a number of additional changes. As such, one simple change may propagate through the whole framework, a phenomenon that is known as the *ripple effect* [YCM78]. A good example of the ripple effect in the evolution example presented in Chapter 3 was when the `SpecialFormHandler` hierarchy was extended with a new class, which in turn required to extend the `Closure` and `SchemeConverter` class hierarchies.

Due to incomplete, inappropriate or outdated documentation, it is quite cumbersome to identify those parts of a framework that are affected by a particular change, since it is not clear how different parts are related.

Change Identification

While change propagation mainly deals with identifying the parts of a framework that are subject to a particular change, change identification tries to infer the specific kind of changes that should be applied to those parts. This requires us to track down the individual classes and methods that should be changed and determine which specific changes should be applied to them.

Adding a new class to a particular class hierarchy, for instance, requires us to infer where in that hierarchy we should place this class. We should decide which class should become the superclass of the new class, and which classes should be the subclasses. Furthermore, we should discover which other classes the new class should use in order to implement the appropriate collaborations. Also, we should determine which methods of the new class' superclass should be overridden, and, conversely, which methods should not be overridden. All methods that we override should be provided with an appropriate implementation, which means that we should find out which other methods they need to call. Clearly, this requires detailed knowledge about the responsibilities of the classes and methods involved and the specific collaborations and interactions between them.

For example, while we may have identified that the `SpecialFormHandler, Closure` and `SchemeConverter` hierarchies need to be changed, we still need to know how they should be changed. When we introduced a new `CondHandler` class in the `SpecialFormHandler` class hierarchy, we first found out that it should be added as a subclass of the `SpecialFormHandlerWithSuccessor` class, and that it should have no subclasses. Then, we figured out that classes in this class hierarchy collaborate with classes in the `Closure` and `SchemeConverter` hierarchy. As those hierarchies did not define the appropriate classes with which the `CondHandler` class should collaborate, we extended the hierarchies with a `CondClosure` and a `CondConverter` class respectively. Furthermore, we provided implementations for the `newClosure` and `newConverter` methods in the `CondHandler` class, that returned new instances of the corresponding classes, and we defined the appropriate methods in the `CondClosure` and `CondConverter` classes.

4.1.2 Avoiding Design Drift

When instantiating a framework, care should be taken that the resulting application does not violate the design dictated by the framework. Even when evolving the framework, which may change the design, the developer should make sure to preserve the appropriate design constraints [vGB01]. When this is not the case, subsequent versions of the framework and its applications may suffer from the problem of design drift, as was explained in Section 2.1.4. In order not to violate the intended design, the developer should be able to identify the affected parts correctly, and should know all changes that should be made to those parts. Even when using a change propagation and identification algorithm, this is not as straightforward as it may seem. Such algorithms are never 100 % accurate, and appropriate documentation that could be consulted is often lacking, incomplete or outdated [BD99, BGK98, Bro90]. It may thus come as no surprise that it is easy to introduce errors and inconsistencies into the framework, or its applications.

Simply making sure that the documentation always reflects the current state of the implementation does not suffice to solve the above mentioned problems. It is still up to the developer to read and interpret this documentation, and implement the required changes correctly. What is needed is a way to explicitly document the design constraints, so that they can be used actively to check for inconsistencies. This requires that these constraints can be expressed in a formal way, and that they are causally linked to the implementation of the framework, so that they can be checked automatically.

A particular example of a design constraint in the Scheme framework is that each subclass of the `SpecialFormHandler` class should use a corresponding class of the `Closure` hierarchy. This constraint forced us to extend the `SpecialFormHandler` class with a `CondHandler` class, and also add a `CondClosure` class to the `Closure` hierarchy. Such constraints are rarely explicitly documented, or perhaps only in natural language narrative. This prohibits their use in a supporting tool.

4.1.3 Support for Software Merging

Frameworks are evolved manually, most of the time, due to the lack of appropriate development environments that support the evolution process. As such, the various changes that are made to the framework and its implementation are not explicitly documented. This makes it difficult to provide automated support for software merging, since most approaches for detecting possible merge

conflicts are based on mutually comparing the various changes that have been applied [Men02].

In the reuse contract approach (see Section 2.3.3, experiments were conducted to recover changes after the facts. However, only lower-level changes could be detected, such as changes in method implementations, for example. Most of these lower-level changes are however the result of applying a more high-level change. Unfortunately, it is near to impossible to group automatically a number of lower-level changes into a higher-level change that reveals the intention of the evolution. In the example evolution presented in the previous chapter, several low-level changes have been applied: classes were added and a number of methods were defined in those classes. It is however impossible to infer automatically that the `printOn:` method was added to the `CondClosure` class because we wanted to add a new special form to the framework, for example.

Even when using an environment that supports evolution, such as the Refactoring Browser, providing support for software merging is not that straightforward. The Refactoring Browser does not explicitly log the refactorings that are applied, so they cannot be used by a merge conflict detection algorithm. We can easily imagine that the browser is extended so that the refactorings are logged, however. While in theory this would allow us to define a conflict detection algorithm, the scalability of the approach would prohibit practical usage. Many refactorings exist [Fow99], and mutually comparing such a high number of refactorings is quite cumbersome. Furthermore, most refactorings are not formally defined, with the exception of those in [Opd92, Rob99, Tic01]. This makes it extremely difficult to define the conditions under which refactorings applied in parallel give rise to merge conflicts.

To motivate the need for software merging, consider the following two evolutions of the Scheme framework that are applied in parallel. One developer decides to implement the `printOn:` method, defined in the `ScExpression` hierarchy, as a visitor (see Figure 4.1). He introduces a new `PrintOnVisitor` class, as a subclass of the `AbstractASTEnumerator` class, and defines the appropriate methods in this class. Those methods copy the implementation of the various `printOn:` methods defined in subclasses of `ScExpression`. Additionally, he changes the implementation of the `printOn:` method in the `ScExpression` class itself so that it uses the new visitor and removes all implementations on the subclasses. At the same time, another developer decides to extend the `ScExpression` hierarchy, and introduces a `ScQuoteExpression` class that represents quoted expressions (see Figure 4.2). Among the methods that this new class should implement are the `printOn:` method and a `nodeDo:` method. The developer provides the first method with the appropriate implementation that prints a textual representation of the object. The second method should call a specific method in the `AbstractASTEnumerator` hierarchy, which requires defining a new method in this hierarchy, `doQuoteExpression:`. The developer provides the appropriate implementation for this method in all concrete subclasses of the `AbstractASTEnumerator` hierarchy.

Both evolutions of the framework should be merged into a single version. Figure 4.3 shows the merge results. Clearly, a number of merge conflicts occur. First of all, the `PrintOnVisitor` class does not provide an implementation for the `doQuoteExpression:` method. Second, the `ScQuoteExpression` class still contains an implementation for the `printOn:` method, which it should not. Rather, it should reuse the implementation of `printOn:` defined in the `ScExpression` class. Because it does not do so, a runtime error will occur if we try to ask a `ScQuoteEpression` object for its textual representation, since it will not use the appropriate visitor object.

The first merge conflict can easily be detected by an ordinary compiler, since the `PrintOnVisitor` class is a concrete subclass of the `AbstractASTEnumerator` class, and should thus override all abstract methods of that class. The second merge conflict, however, is not intercepted by an ordinary compiler, and can only be detected by mutually comparing the different evolutions that were applied.

4.2 Solution Proposed By The Dissertation

To alleviate the problems of change propagation and identification, design drift and software merging, we propose to use design patterns as a means to document explicitly a framework's

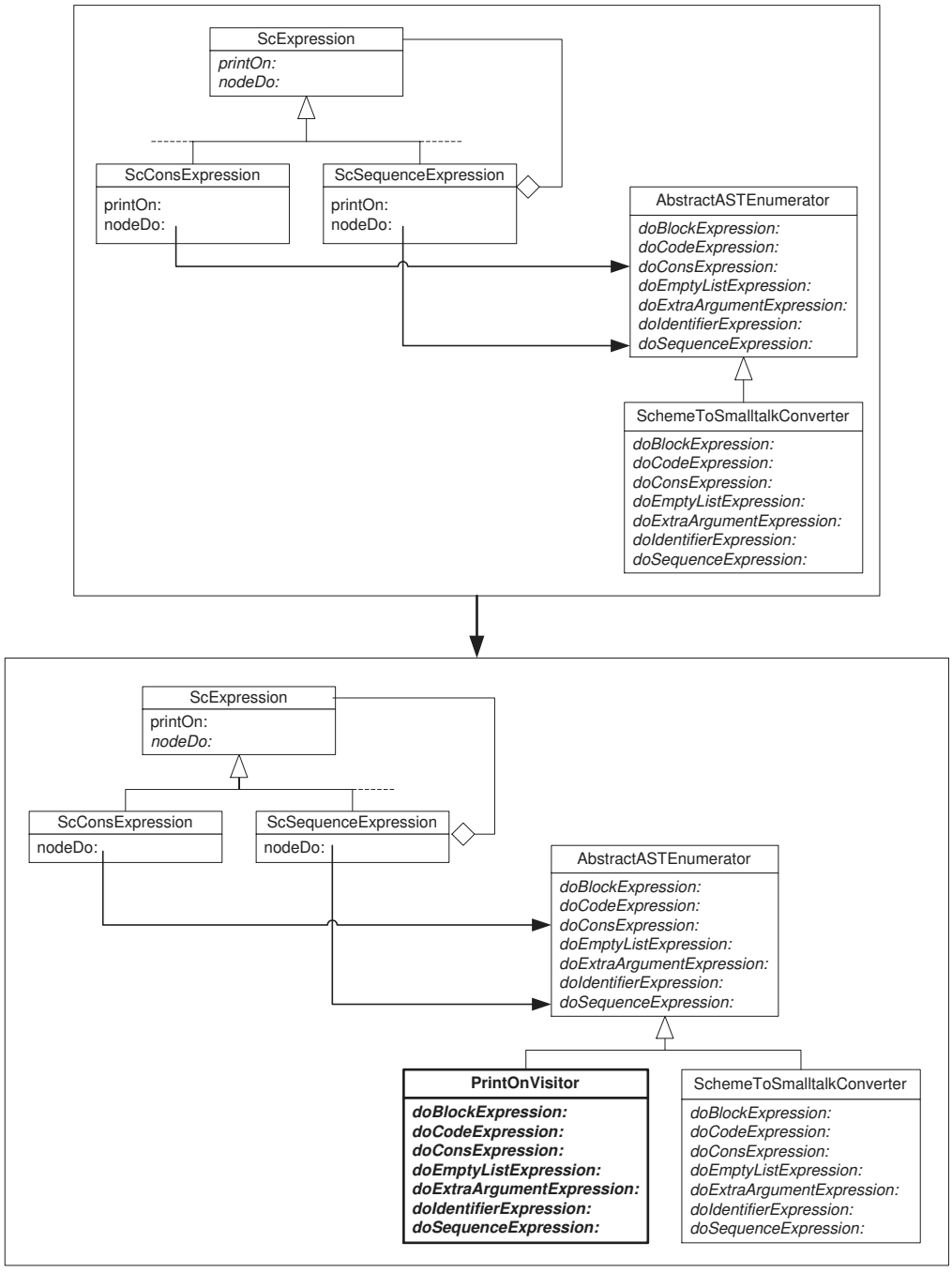


Figure 4.1: Implementing the `printOn:` method with the *Visitor* design pattern

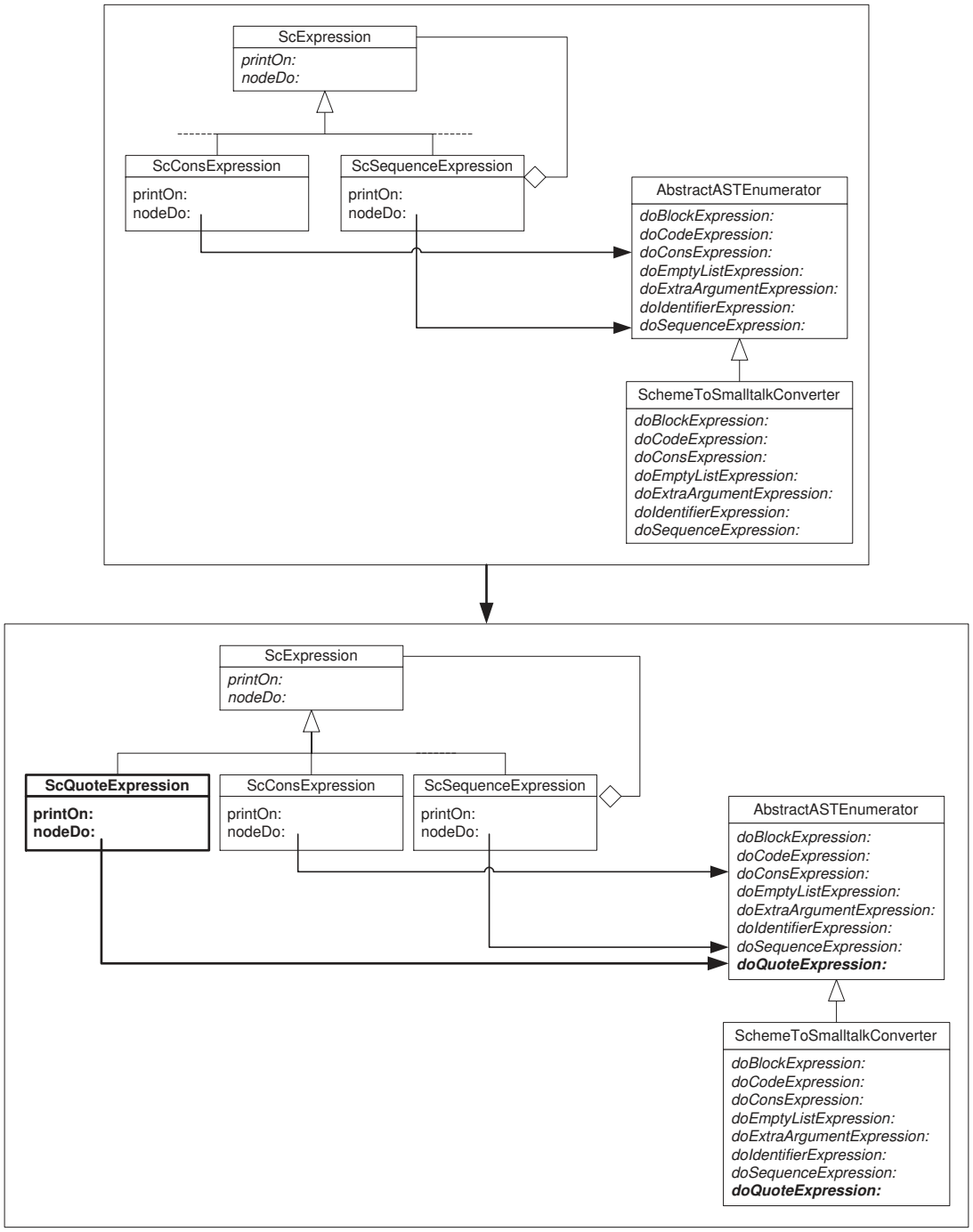


Figure 4.2: Introducing a `ScQuoteExpression` class

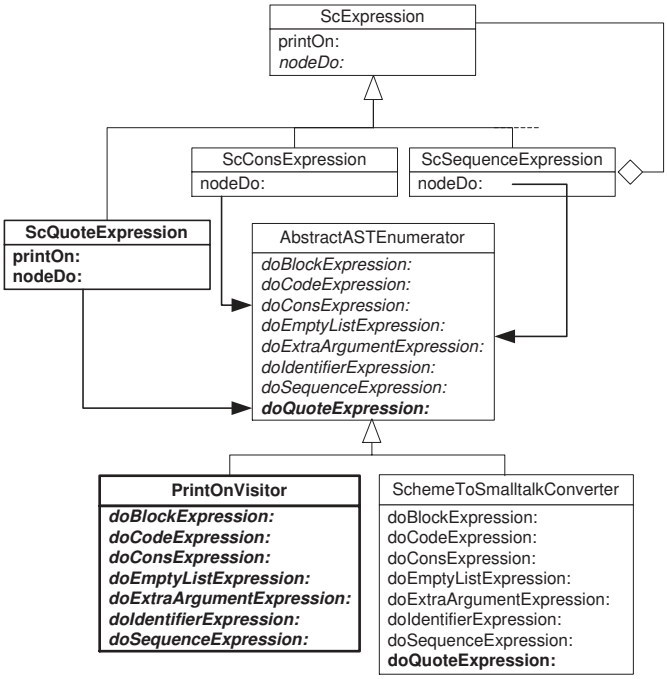


Figure 4.3: Result of merging both evolutions

design, its instantiation and evolution, and use this information in an active way. In this section, we will explain how design patterns provide exactly the kind of knowledge we require and show how this information can be employed for our purposes.

4.2.1 Explicit Design Documentation

It has already been shown that design patterns can be used to document a framework [BJ94, Pre94, Sch95]. Moreover, research also shows that maintenance tasks on a framework are completed faster and introduce fewer errors if the design pattern occurrences are explicitly documented [PU98]. Therefore, we firmly believe that design patterns convey exactly the kind of information that we are looking for. We attribute this to the following properties held by design patterns:

- Design patterns offer information about particular parts of the design and the structure of the framework at a level above the implementation level. Instead of focusing on individual classes and methods, they concentrate on the relationships between different classes, their respective interfaces, how their instances collaborate and the specific method interactions that implement such collaboration.

- Design patterns are used to implement the hot spots and frozen spots of a framework. In this way, they document the specific ways in which a framework can be extended, and provide information about the extensions that are more difficult to achieve. As such, they allow to identify the changes that are necessary to the design of a framework to incorporate a particular evolution, up to a certain extent.

This information is exactly what we are looking for when evolving a framework, be it for identifying the classes and methods that need to be changed, the specific changes that they should undergo or how these changes should be implemented and propagated. Design patterns and their instances help to understand the overall structure and the rationale behind a particular design, and thus provide insight on how it can and should be evolved.

4.2.2 Explicitly Representing Instantiation and Evolution

In order to be able to define a merge conflict detection algorithm , the particular changes that are applied to a framework should be explicitly logged. Moreover, we want to report possible problems at a high-level of detail, so that information about the intent of a change is available and it becomes easier for a developer to identify the cause and the solution for a particular conflict. In our approach, we will instantiate and evolve a framework by means of design pattern transformations and refactorings. Furthermore, we will explicitly document the application of these high-level transformations, which enables us to define a merge conflict detection algorithm.

High-Level design pattern transformations

Although refactorings already provide high-level transformations, we want to take this one step further. While it is acknowledged in the literature that design patterns can be introduced by using refactorings [RBJ97, TB95, O'C01], those refactorings themselves fail to employ information conveyed by design pattern instances. Since a design pattern specifies how a particular aspect of a design can be extended, it implicitly defines a *design pattern transformation*, which specifies the changes that are necessary to implement such extension and which can be applied on its instances. Refactorings can be combined to form *design pattern transformations* that can be used to evolve design pattern instances in such a way that their constraints are not violated (similar to the design pattern transformations provided by Florijn [FMvW97] and [MT01]).

When such a design pattern transformation adds a class to a particular class hierarchy, for example, it will consult the documentation to check the design pattern instances in which this class will participate. Based on this information, it can derive which methods this class should implement, and under some circumstances even generate an implementation for these methods automatically. For example, a *Composite* design pattern defines a transformation that adds a new leaf class to a class hierarchy that participates in an instance of that design pattern. Besides adding the class to the hierarchy, this transformation also specifies that all method participants of the design pattern instance should be defined by the class. As an example, Figure 4.4 shows how an *addLeaf* design pattern transformation is applied to the *CompositeExpression* design pattern instance of the Scheme framework, to introduce a `ScQuoteExpression` class. Note how this class provides implementations for the `printOn:` and `analyze` methods, since this is required by the design pattern constraints.

A *design pattern transformation* defined by a design pattern thus adds a participant to an instance of that design pattern, or removes a participant from such an instance. At the same time, it performs additional changes to the implementation, that are necessary to ensure that the design pattern's constraints are still satisfied. We will explicitly define such design pattern transformations for each design pattern, so that a developer can use them just as he uses refactorings in the Refactoring Browser. Note that we could have considered other transformations on design patterns as well, such as transformations that refine or coarsen method participants. While it would be straightforward to add such transformations, this can be considered future work.

It follows from the fact that design pattern transformations are only capable of adding or removing participants, that they can not be used to replace a design pattern instance by another instance, or to internally reorganize the structure of a design pattern instance. For example, no design pattern transformations exist that change the inheritance relationships between classes, or shifts method implementations up or down the class hierarchy. As such, design pattern transformations are mostly useful to support anticipated evolution. To support unanticipated evolution as well, refactorings should be incorporated, since these are generally applicable program transformations. Clearly, since instantiation can be seen as a special kind of anticipated evolution (see Section 2.1.4), design pattern transformations are able to support it as well. In short, design pattern transformations can be used to support anticipated evolution as well as instantiation, but do not support unanticipated evolution.

Furthermore, much in the same way that a combination of design pattern instances describes a framework's complete design, a combination of design pattern transformations can define a

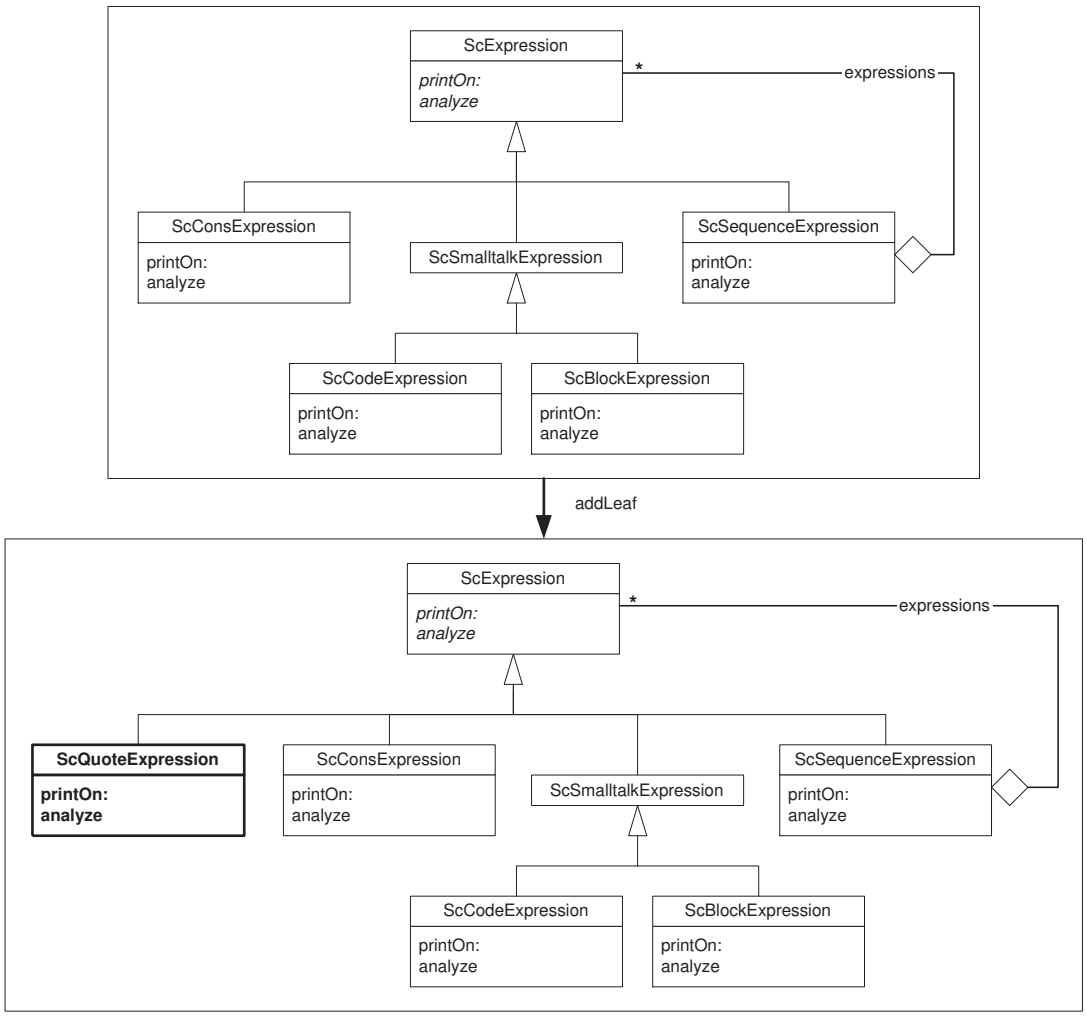


Figure 4.4: An *addLeaf* design pattern transformation performed on the `CompositeClosure` *Composite* design pattern instance

framework instantiation or *framework evolution* transformation. Such framework-specific transformations are high-level operations that implement a complete instantiation or evolution of the framework. As a concrete example, we can define an *addSpecialForm* framework-specific evolution transformation for the Scheme framework. A developer can use this operation to add a new special form to the framework's implementation automatically. The *addSpecialForm* transformation itself is defined in terms of the appropriate design pattern transformations, that add the necessary participants to the appropriate design pattern instances.

4.2.3 Formal Design Constraints

To alleviate design drift as much as possible, to guide a developer when manually evolving a framework and to support unanticipated evolution up to a certain extent, we will explicitly document the design constraints of the framework. Design patterns can also be used for this purpose [MT01].

A design pattern constrains the implementation of a framework. Participants in a design pattern instance should adhere to the specific relationships, collaborations and interactions defined by the design pattern. When a new class is added to the framework, for example, it will participate in one or more design pattern instances, since every important class hierarchy is a participant in one or more of those instances. If the constraints of the design pattern are explicitly documented, we can use them actively to verify that a developer provided a correct implementation of the class and its methods, according to the rules of the design pattern. If this is not the case, we can warn him of this fact, so that he can take appropriate action.

In order to allow tool support for automatically verifying whether a framework's implementation satisfies the necessary constraints, it is absolutely necessary that these constraints can be formally specified. This in turn requires a formal description of the design pattern for which these constraints hold. In Chapter 5 we will provide such a formal model, which includes a formal definition of an abstraction of design patterns (metapatterns) and their associated constraints. It is important to note that constraints are associated with design patterns, and hold for all of the design pattern instances. As such, we only need to specify these constraints once for each design pattern, and we can reuse them in any other framework.

As an example, we can formally and explicitly document the constraint that each subclass of the `SpecialFormHandler` class hierarchy should have a corresponding class in the `Closure` hierarchy, by specifying that those two hierarchies participate in an instance of the *Factory Method* design pattern. One of the constraints of this design pattern is that each subclass of the `SpecialFormHandler` class should implement the `newClosure` method, which should instantiate a specific subclass of the `Closure` hierarchy. By means of the formal specification of this constraint, we can detect if a developer forgets to implement this method, or if he does not provide it with the correct behavior and does not instantiate an appropriate subclass.

Explicit design constraints are also a means to provide limited support for unanticipated evolution. As already explained, no design pattern transformations exist that allow a developer to internally reorganize a design pattern instance, as such changes are impossible to anticipate beforehand. Reorganizing a class hierarchy, changing inheritance relationships and moving method implementations around thus remains a manual task. Luckily, the design pattern constraints remain useful after such changes have been applied, to verify whether a design pattern instance is still correct with respect to the appropriate design rules. E.g. the constraints can be used to check whether the inheritance relationships are still valid, classes implement the appropriate methods, and so on. The support for unanticipated evolution provided by these constraints is limited, however, since a developer can entirely destroy a design pattern instance, or can replace it with another one, which would render the constraints useless.

Merge Conflict Detection

Explicitly documenting the design pattern transformations and refactorings that are applied to a framework allows us to define an operation-based merge conflict detection algorithm, as shown in [MT01]. Because we know the individual transformations that have been applied, we can

compare them one by one, and define the conditions under which they give rise to a merge conflict when applied in parallel.

Consider the example presented in Section 4.1.3, where an *addLeaf* design pattern transformation was applied to the *CompositeExpression* instance of the *Composite* design pattern. This operation adds a new leaf class participant, `ScQuoteExpression`, to the `ScExpression` class hierarchy. At the same time, another developer applies an *addConcreteVisitor* design pattern transformation and a *pullUpMethod* refactoring to the *ExpressionVisitor* instance of the *Visitor* design pattern. The intent here is to assemble the printing behavior scattered over the different classes in the `ScExpression` hierarchy into one single class, the `PrintVisitor` class.

We already saw that merging both evolutions into one single version leads to a number of merge conflicts. We can detect these by comparing the design pattern transformations and refactorings that are applied, and deducing the conditions under which they give rise to a conflict if applied in parallel.

Figure 4.5 shows the merge conflict that arises because two design pattern transformations are applied in parallel: the *addConcreteVisitor* and the *addConcreteElement* design pattern transformations. The former adds a new visitor class to the `AbstractASTEnumerator` hierarchy, whereas as a result of the latter a new `doQuoteExpression:` method is added to that same hierarchy. This results in a merge conflict where the newly introduced visitor class does not implement the newly introduced method.

Figure 4.6 illustrates how applying a refactoring and a design pattern transformation in parallel may lead to a merge conflict. An *addLeaf* design pattern transformation is applied to a instance of the *Composite* design pattern, while at the same time a *pullUpMethod* refactoring is applied that changes the internal structure of that same instance. As a result, in the merged version, the newly introduced leaf class does not conform to the new structure of the class hierarchy.

As we will show in Chapter 7, we can provide a basis for detecting possible merge conflicts due to parallel refactorings or design pattern transformations, by identifying and formally defining the conditions under which they influence one another.

4.2.4 Support for Change Propagation

Design patterns can also provide help to propagate a particular change through the framework and infer its impact on the rest of the framework in this way. This is due to the fact that:

- design pattern instances cover a substantial amount of the frameworks classes and methods. They can thus be used as a basis for implementing a change propagation algorithm that defines how a change to one particular part of a framework, represented by a design pattern instance, propagates to the other parts, represented by other instances.

- design pattern instances can, and most likely will, overlap, since classes and methods can participate in more than one instance. An example of such an overlapping was shown in Section 3.4.5, where instances of the *Visitor* and *Composite* design patterns mostly consist of the same participants. Figure 4.7 depicts this situation.

A design pattern transformation applied on a design pattern instance will make sure that the constraints of the corresponding design pattern remain satisfied. However, this does not necessarily mean that the constraints of overlapping design patterns are satisfied as well. Additional transformations may be required on those overlapping instances, in order for the implementation to be conform with the design pattern's constraints.

Consider again the example of the overlapping of the *Visitor* and *Composite* design pattern instances. If we apply an *addLeaf* design pattern transformation on the instance of the *Composite* design pattern, to extend the `ScExpression` hierarchy with a new `ScQuoteExpression` class, we should apply an *addVisitorMethod* design pattern transformation on the instance of the *Visitor* design pattern as well, in order to introduce the `doQuoteExpression:` method in the `AbstractASTEnumerator` hierarchy.

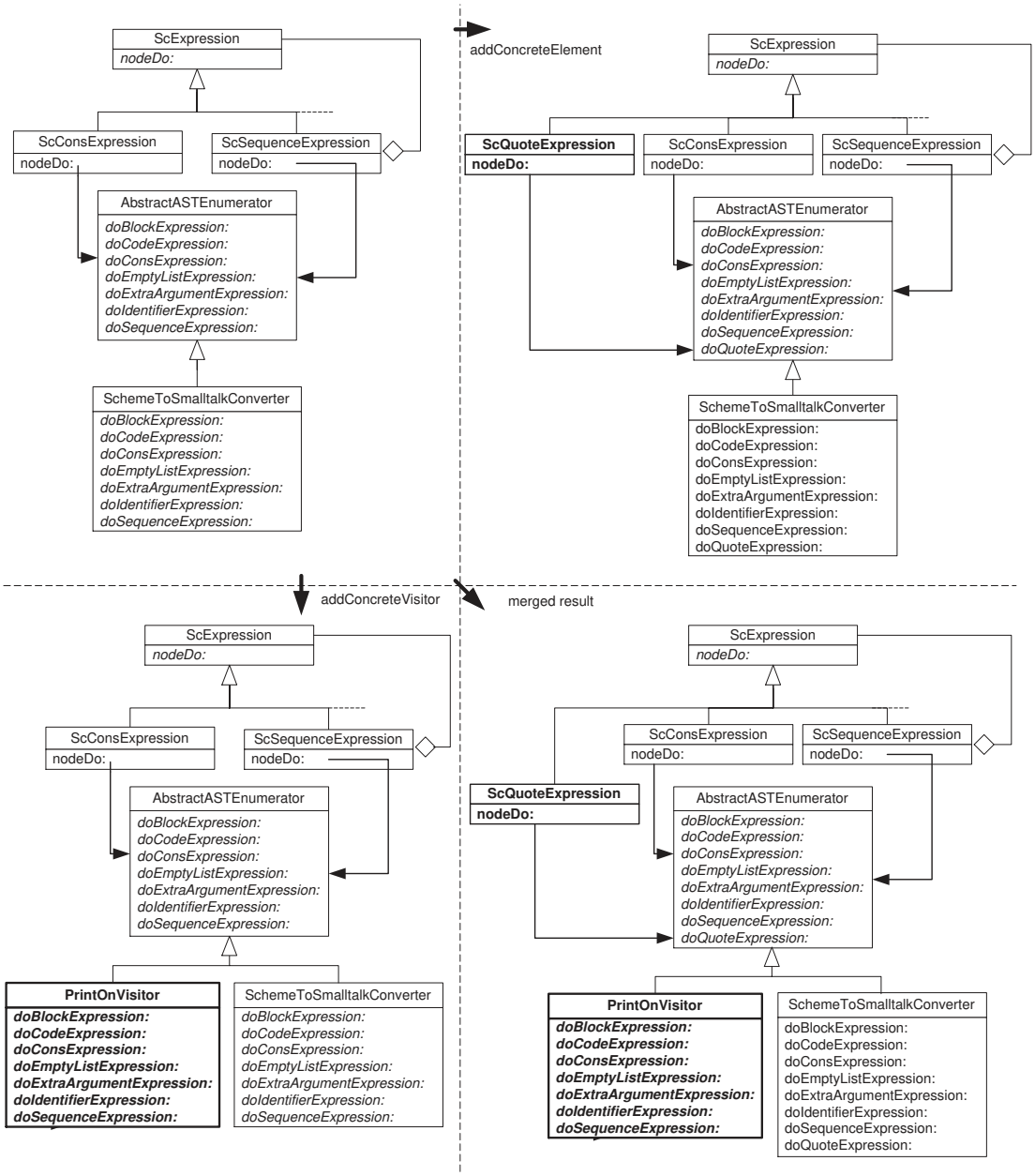


Figure 4.5: Merge conflict when applying a *addConcreteVisitor* design pattern transformation in parallel with an *addLeaf* design pattern transformation

In Chapter 5, we will show how we can detect overlapping of design pattern (or more specifically, metapattern) instances, and determine the conditions under which design pattern transformations give rise to additional design pattern transformations on overlapping instances. As such, we can implement a change propagation algorithm, based on information about design patterns and the design pattern transformations defined on them.

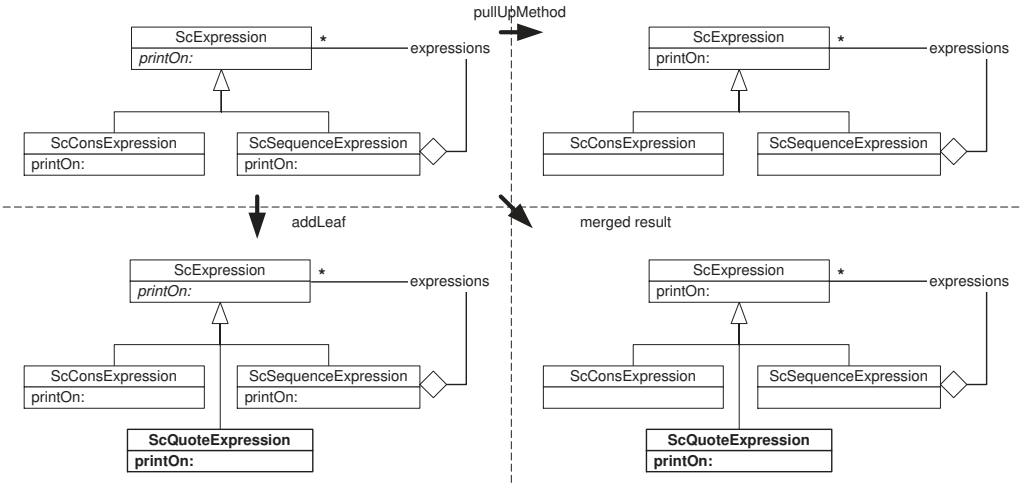


Figure 4.6: Merge conflict when applying a *pullUpMethod* refactoring in parallel with an *addLeaf* design pattern-specific transformation

4.3 Approach of the Dissertation

4.3.1 The Metapattern Approach

Tool support involving design patterns first of all requires a formal definition of those design patterns. Without a decent formalism, such patterns can only be described by means of natural language, informal design diagrams and sample code templates. This makes them inherently ambiguous and prohibits automated tool support. Several formal models were discussed in Section 2.3.4. The approach of [EHY98] appeals most to us, as it provides a simple and concise formal model, that can be expressed in a declarative way.

Second, the kind of tool support we want to provide for framework-based development requires a suitable abstraction of design patterns, due to the following reasons:

- we want to represent instantiation and evolution by means of high-level design pattern-specific transformations. Since many design patterns exist, and new ones keep being discovered, many such transformations would need to be defined and the tool would have to be updated constantly.

- our merge conflict detection algorithm is operation-based, and mutually compares the transformations that have been applied. This is quite cumbersome given the multitude of transformations. The fact that new transformations need to be integrated often would even jeopardize the scalability of our approach.

- our change propagation algorithm will be defined in terms of overlapping of design pattern instances. Without an abstraction, we should consider how each design pattern can overlap with each other design pattern. The large number of design patterns would clearly make this approach unmanageable, especially if new design patterns need to be integrated often.

A decent abstraction is thus clearly indispensable. Furthermore, instantiation and evolution activities require information about the structure and the collaborations between classes and methods, and involves changing these aspects. A suitable abstraction should convey such information.

Metapatterns [Pre95, Pre94, Pre97] provide exactly the kind of abstraction of design patterns we are looking for. As was discussed in Section 2.3.5, metapatterns classify design patterns into a small set of categories, according to their *Structure, Participants* and *Collaborations*. Such

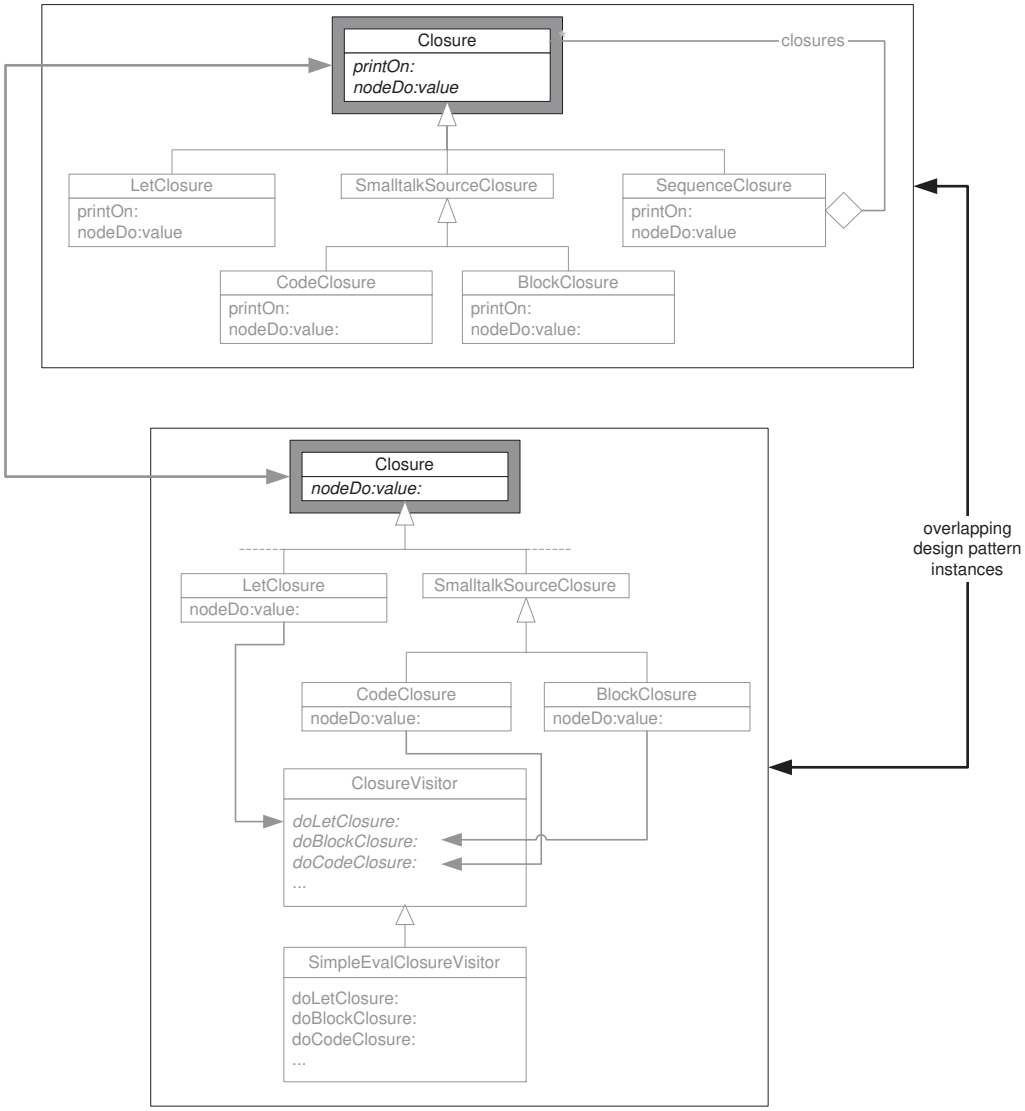


Figure 4.7: Overlapping of *Composite* and the *Visitor* design pattern instances

properties convey exactly the information we are looking for and can easily be defined formally, whereas other properties of design patterns (such as the *Applicability* and *Trade-offs*) are harder to formalize. Moreover, the small number of possible metapatterns ensures the scalability and manageability of our approach. Only a small number of transformations would need to be defined, which avoids the need to constantly update tools when new design patterns (and thus new design pattern-specific transformations) are discovered, and which greatly simplifies the definition of an operation-based merge conflict detection algorithm. Likewise, the number of possible ways in which such a limited set of metapatterns can overlap is quite small, which reduces the complexity of our change propagation algorithm.

The fact that metapatterns are an abstraction of design patterns naturally indicates that some information conveyed by a design pattern is lost. Many important properties of design patterns, such as *Intent* and *Applicability*, are not considered by metapatterns. This should not be a serious hindrance however. We merely use metapatterns as a basis for providing automated tool support for framework-based development. By no means do we exclude the use of design patterns for other purposes, such as documenting the framework and explaining the rationale behind some specific design decisions. Furthermore, it should be noted that not all design patterns can be captured in full detail as a combination of metapatterns. Rather, metapatterns are capable of expressing a significantly large part of the available design patterns, and are therefore nevertheless very useful as an abstraction.

Differences with Existing Metapatterns

The formal definition of metapattern we will provide in Chapter 5 differs slightly from the original definition presented by Pree [Pre95] in a number of ways:

- In the original definition, metapatterns consist of four participants: template and hook classes and template and hook methods defined in those template and hook classes (see Section 2.3.5). In our definition, we explicitly include class hierarchies, by introducing the concepts of template and hook hierarchies. The original definition already mentions that the hook class participant should be subclassed in order to override the appropriate hook method participants. In essence, the hook class participant in the original definition is thus similar to the hook hierarchy participant in our definition. A template hierarchy participant is however not considered in the original definition.

- Originally, seven different types of metapatterns are defined (see Section 2.3.5). Due to the fact that we introduce template hierarchy participants, we are able to define two new types: the *1:1 Hierarchy Connection* and the *1:1 Hierarchy Creation* metapatterns. Just like the different types of metapatterns that were already defined, these two new metapatterns allow us to capture important structures in a framework's implementation and form an appropriate abstraction for a number of design patterns. The definition of these new metapatterns and a discussion of their usefulness will be shown in later sections.

- While Pree merely presented a detailed discussion of the existing metapatterns in [Pre95], we provide a complete framework for metapatterns that is formally underpinned. This framework permits to define metapatterns that vary from the originally defined metapatterns in a number of ways, and should be easy to extend in a straightforward way with new metapatterns should the need arise. The framework is based on five *fundamental* metapatterns, that form the basis for all other metapatterns that exist.

In Chapter 5, we will first present a formal definition for the five fundamental metapatterns, and afterwards define the existing metapatterns in terms of the fundamental metapatterns. Moreover, the formal definition of a fundamental metapattern will be accompanied by a definition of the metapattern-specific transformations that can be applied upon its instances. Just like metapatterns form the basis for a formal definition of design patterns, these metapattern-specific transformations form the basis for formally defining design pattern-specific transformations.

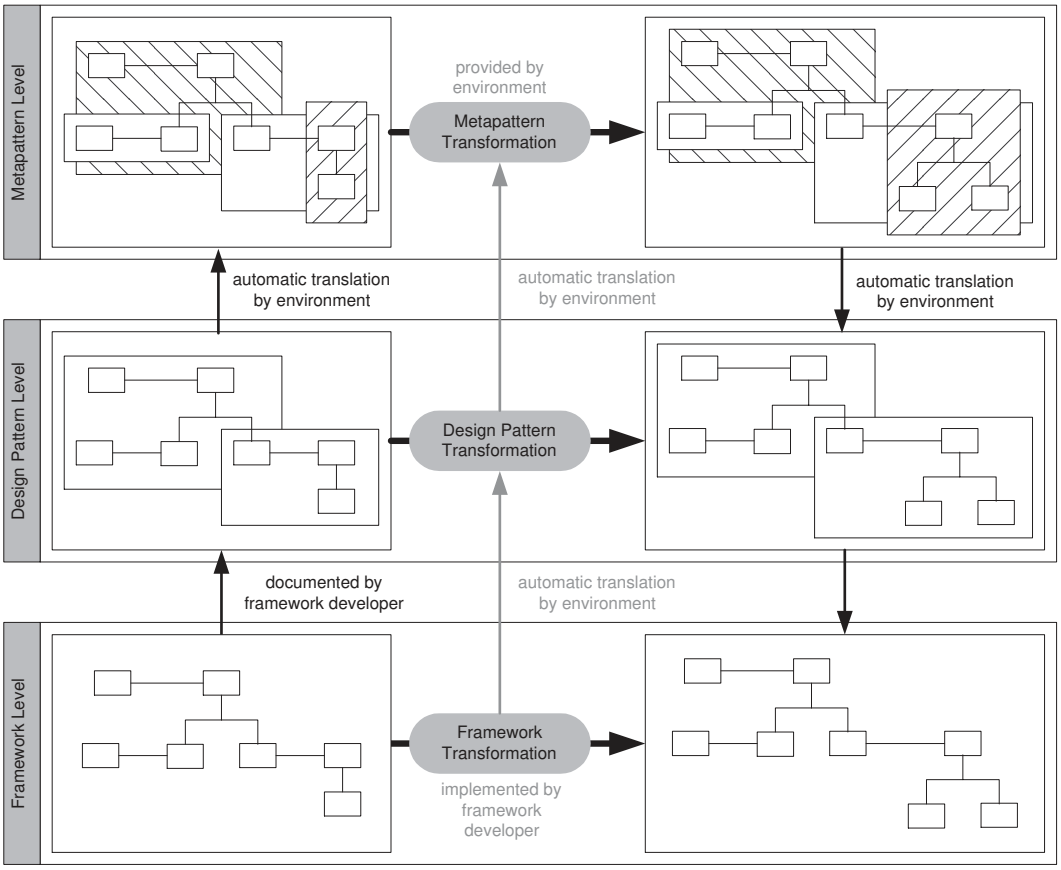


Figure 4.8: Our approach for supporting framework-based development

4.3.2 General Overview of the Approach

Figure 4.8 shows the approach we propose to build an environment for supporting framework-based development. The approach consists of three levels: the *framework, design pattern* and *metapattern* level. The information at the framework and design pattern level is always translated to the metapattern level, where the interesting work is performed. It is at this level that changes are made to the implementation, that the documentation is updated automatically and that the design constraints are verified. The resulting information is then again promoted to the higher levels of the approach. We will now highlight some of the most important features in more detail.

Framework Level

At the framework level, the framework developer is responsible for documenting a framework's design by means of the design pattern instances occurring in it, and for defining appropriate framework instantiation and evolution transformations. The design pattern documentation can be browsed by application developers or other framework developers to understand the inner workings of the framework, and they can use the framework transformations to instantiate or evolve the framework in a predefined way. These high-level operations guide them in performing their task by interacting with them and generating appropriate skeleton code that can be edited later on.

We have chosen a non-invasive approach to documenting design pattern instances, since annotations are provided in a separate environment. Another option would consist of making design patterns first class entities in the programming language itself [Bos98]. Such would however pro-

hibit the use of our approach on already existing frameworks, since no existing and widely-used programming language supports design patterns as first class entities. Moreover, it would sacrifice the flexibility of our approach, since we would be tied to design pattern information only, whereas in a non-invasive approach, other kinds of information could be used as well. Last, we would be forced to abandon the metapattern abstraction, since the programming language would support design patterns, and not metapatterns, since the former contain much more information and are far more familiar to developers. As such, the scalability and manageability of such an approach can not be guaranteed.

Design Pattern Level

At the design pattern level, the supporting environment provides design pattern-specific transformations, that can also be used by application or framework developers. These transformations are defined on the various design patterns, and apply changes to their instances. As such, these operations are independent of a particular framework, just as refactorings, and thus need to be implemented only once and can be provided as a library that can be reused.

A framework developer or an application developer simply uses the design pattern-specific transformations, and need not worry about their internal implementation. The framework-specific transformations that are defined for a framework, however, should be implemented by the framework developer. When defining such transformations, he should specify how these are to be translated into design pattern-specific transformations on the appropriate instances. This translation depends on the specific framework and the specific design pattern instances occurring in it. Therefore, the translation needs to be specified once per framework-specific transformation. Nonetheless, whenever another developer invokes such an operation, the supporting environment will be able to take care of the translation automatically, based on the specifications provided by the framework developer.

Metapattern Level

As explained above, tools based on design patterns should use a suitable abstraction to ensure scalability and manageability. To this extent, we use metapatterns as an abstraction of design patterns, and we include a metapattern level in our approach. It is at this level of the approach that the interesting work is performed. Design information is present in terms of metapattern instances and the specification of these instances is automatically derived from the specification of the design pattern instances at the design pattern level. Since this translation is fixed and independent of a particular framework, the environment includes a library that specifies how this should be achieved for each design pattern-metapattern(s) combination.

Just as design patterns define design pattern-specific transformations, metapatterns define metapattern-specific transformations. These are implemented at the metapattern level, perform the actual changes to the implementation of the framework, and automatically update the documentation appropriately. The design pattern-specific transformations, provided at the design pattern level, are automatically translated into appropriate metapattern-specific transformations by the environment, based on translation information provided in a library. This library contains the implementation of the metapattern-specific transformations, as well as a specification of how the design pattern-specific transformations should be mapped onto them. This library is reusable, since metapattern–specific transformations are independent of a particular framework, as is the translation from design pattern-specific transformations to metapattern-specific transformations.

Besides the metapattern-specific transformations, the metapatterns also define the constraints that a framework's design should satisfy. It is thus also at the metapattern level that the design constraints of a framework are specified and that the implementation is verified according to these constraints. The specification of design constraints is thus also independent of a particular framework. Therefore, these constraints should also be implemented only once and included in a reusable library.

To summarize, the only responsibilities of a framework developer are providing a specification of the design pattern instances used in the framework, and implementing the appropriate framework-specific transformations. He should not implement design pattern- and metapattern-specific transformations, as these are provided in a library and can be reused. Moreover, the translation algorithms that map design pattern instances onto metapattern instances and vice versa and the algorithms that map design pattern-specific transformations to metapattern-specific transformations are also implemented only once and can be reused by the environment across different frameworks.

We would like to stress once again that one of the major goals of the dissertation is to prove that an environment for supporting framework-based development can be constructed, based on the approach sketched above. Clearly, this requires us to first test the feasibility of the approach, discover its strengths and weaknesses and fine-tune it where possible. Our main focus in this dissertation thus lies in dealing with language-engineering issues first, as opposed to mere software engineering issues. As we have explained above, however, this does by no means imply that we will construct a new programming language or extend an existing one.

Chapter 5

A Formal Model for Metapatterns

In this chapter, we define a formal model for representing information about a framework's design. This model is based upon metapatterns, and formally defines the various existing metapatterns as well as the corresponding metapattern-specific transformations. Furthermore, we will also define the conditions under which metapattern instances overlap and show how this affects their operations.

5.1 Introduction

As was explained in Section 4.3.1, we will use metapatterns as an abstraction of design patterns to ensure the scalability and manageability of our approach to support framework-based development. In this chapter, we will define a formal framework that can be used to formalize the metapatterns defined by Pree [Pre95], as well as other metapatterns that may exist. This formal definition of metapatterns is largely similar to the formal definition of design patterns in LePUS [EHY98].

The formal framework is based on the definition of five *fundamental* metapatterns, which will be given first, together with a definition of the metapattern-specific transformations. These transformations form the basis of the design pattern-specific and framework-specific transformations that can be used to instantiate and evolve a framework. Afterwards, we will show how existing metapatterns can be defined in terms of these fundamental metapatterns. Furthermore, we will define a change propagation algorithm based on the metapattern-specific transformations and on the different ways in which metapattern instances can overlap.

5.2 Preliminaries

5.2.1 Notation

In the remainder of this chapter, metapatterns will be presented both textually and graphically. The textual representation serves as the formal definition, while the graphical representation is used to provide an intuitive corresponding picture.

In the textual representation, a single class will be denoted with a capital letter (such as T or C). A variable is represented by a small letter v, while a small letter m represents a method. Furthermore, we will use subscripts in order to distinguish between different methods. Curly braces are used to denote a set of methods ($\{m_1, m_2, m_3\}$ represents a set containing three methods, for example). In case of a singleton set with only one method, we will omit the curly braces (e.g. we use m for representing the set $\{m\}$). Furthermore, each method will be preceded by a class that shows where that method is defined. For example, $C :: \{h_1, h_2\}$ means that the class C defines two methods h_1 and h_2. When it is not explicitly known which or how many methods are contained in the set, we can use $C :: \mathcal{M}$, where $\mathcal{M}$ represents a set of methods. Note that there can be no confusion between a symbol denoting a class or a symbol denoting a set of methods although both

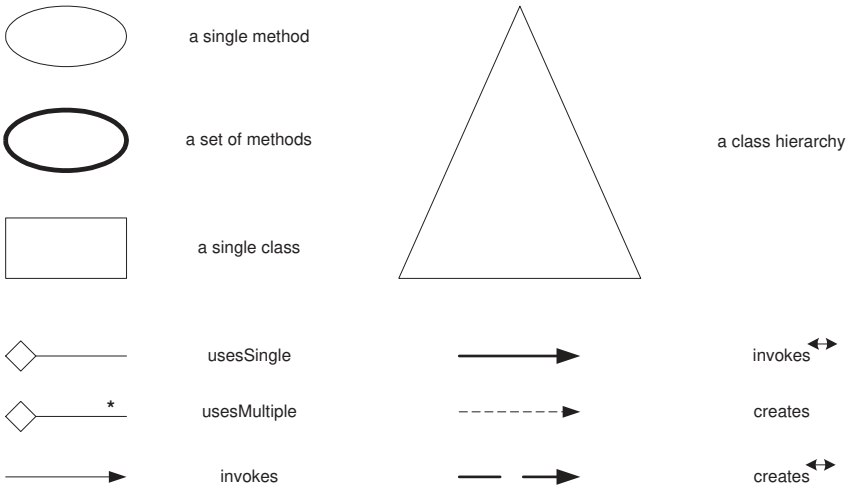


Figure 5.1: Graphical Notation Symbols for metapatterns

are represented with a capital letter: a set of methods is always preceded by the class in which these methods are defined.

The graphical notation symbols that are used are depicted in Figure 5.1. For each participant of a metapattern, a corresponding graphical symbol is defined, as is the case for (most of) the relations that can hold between these participants. These relations will be defined in Section 5.2.2.

5.2.2 General Definition of a Metapattern

A metapattern is defined as follows:

Definition 7 *A metapattern is a tuple $\langle P, R \rangle$, where P is a set of participants and R is a set of relations (or constraints) that hold between these participants.*

The set of participants P contains the classes, methods and variables that make up the structure of the metapattern. More specifically, a participant can either be a single class, a hierarchy of classes (see Section 5.2.3), a single method, a set of methods or a variable. Moreover, each participant is tagged with the specific role it fulfills in the metapattern instance (for example, a class hierarchy $\mathcal{H}$ that plays the role of *hookHierarchy* participant is denoted by $hookhierarchy(\mathcal{H})$).

The relations (or constraints) part R of a metapattern description specifies how the different participants of that metapattern are related and how they interact with each other. These relations should be adhered to at all times in order to preserve a correct metapattern structure. The definition of all the relations that can hold between two participants can be found in Tables 5.1, 5.2 and 5.3. In these definition, we consider $\mathbb{C}$ as the set of all classes defined in the framework, $\mathbb{M}$ as the set of all methods and $\mathbb{V}$ as the set of all variables defined by those classes. This includes instance and class variables, as well as formal parameters that are passed to methods. Local or temporary variables are not considered, as they do not form part of the structure of a metapattern. A brief description of each of these relations follows. We should point out that most of these relations are not completely formally specified. Rather, an intuitive definition is given in natural language. However, they can be formally defined as well, since they can be mapped in a straightforward way onto the basic concepts underlying any object-oriented programming language (e.g. message sending, method definition, etc.). Such formal definitions are outside the scope of this dissertation, however.

understandsMessage $\subseteq \mathbb{C} \times \mathbb{M}$ This relation specifies that a certain class `C` understands a certain

message `m`. This means that the class C itself, or one of its superclasses, provides a concrete implementation for the corresponding method (that was not cancelled).

definesMethod $\subseteq \mathbb{C} \times \mathbb{M}$ A class `C` defines a method `m` if it declares this method in its interface. The method may be an abstract method, or may provide an implementation.

definesVariable $\subseteq \mathbb{C} \times \mathbb{V} \times \{variable, formal\}$ A class `C` defines a variable `v` if this variable is defined as an instance or a class variable in the class, or if it is passed as a formal parameter to one of the methods of the class.

creates $\subseteq \mathbb{M} \times \mathbb{C}$ This relation holds between a method and a class and signifies that the method creates an instance of that class and returns it.

usesSingle $\subseteq \mathbb{C} \times \mathbb{C}$. This relation can hold between two class participants and specifies that an instance of the first class refers to exactly one instance of the second class.

usesMultiple $\subseteq \mathbb{C} \times \mathbb{C}$ The *usesMultiple* relation holds between two class participants and indicates the fact that an instance of one class can refer to many instances of another class. The relation does not specify how many instances can be referred to.

inherits $\subseteq \mathbb{C} \times \mathbb{C}$ This relation holds between two classes and reflects the fact that the first class is a direct subclass of the second class. A transitive variant of this relation exists, $inherits^{\star}$, which also specifies that a class C_1 is a descendant of another class C_2, but which does not require that class to be a direct subclass.

invokes $\subseteq \mathbb{M} \times \mathbb{M}$ This relation holds between two methods and specifies that the first method directly calls the second. Note that this relation is a static relation, which means that it is not guaranteed that the first method actually calls the second at run-time. For instance, if the call occurs inside the true branch of a conditional statement which always returns false, the call will never occur at runtime. This does not concern us however, since we are merely interested in static information, which is sufficient for achieving the kind of evolution support we envision.

There are two variants of the *invokes* relation. The $invokes_r$ relation specifies that a method calls itself recursively. The $invokes^{\hookrightarrow}$ relation indicates that a method calls one and only one other method and each method can only be called by exactly one other method.

Although we presented a discussion of the relations as if they only hold for single participants, a number of these relations is also defined for sets of participants. For example, the *understandsMessage* relation holds between a class and a method, but can also hold between a set of classes and a method or between a class and a set of methods. Such relations then simply state that each class in this set implements the given method or all methods are implemented by the single class, respectively. A similar definition with universal quantification is given for all other relations that can hold between primitive participants or sets of participants, except for the *invokes* relation. (see Tables 5.1, 5.2 and 5.3). The definition of the *invokes* relation for sets of methods uses an existential quantifier (see Table 5.2). This is because all methods in the set $\mathcal{M}_1$ should not call all methods in the set $\mathcal{M}_2$, but should call at least one of those methods. Furthermore, not every method in the set $\mathcal{M}_2$ should be called by a method in $\mathcal{M}_1$.

In the graphical notation for metapatterns (see Figure 5.1), we have symbols for all but three relations. For one, the *understandsMessage* and *definesMethod* relationships have no explicit graphical representation. Instead, these relationships are specified graphically by means of overlapping symbols. A method symbol occurring inside a class symbol, for example, means that the class defines the method. Second, the recursive variant of the *invokes* relationship, $invokes^{\star}$ has no graphical counterpart, as it is obvious from the figure if a method invokes itself recursively on another instance. We should note that recursive methods on the same instance are not considered, as these do not occur in any kind of metapattern. The model can however be extended in a straightforward way to allow such recursive methods.

$$
\begin{array}{lcl}
uses \subseteq \mathbb{C} \times \mathbb{C} \times \{single, multiple\} & & \\
uses(C_1, C_2, single) & = & usesSingle(C_1, C_2) \\
uses(C_1, C_2, multiple) & = & usesMultiple(C_1, C_2) \\
 & & \\
usesSingle \subseteq \mathbb{C} \times \mathbb{C} & & \\
usesSingle(C_1, C_2) & = & \text{class } C_1 \text{ references a single instance of class } C_2 \\
 & & \\
usesMultiple \subseteq \mathbb{C} \times \mathbb{C} & & \\
usesMultiple(C_1, C_2) & = & \text{class } C_1 \text{ references a number of instances of class } C_2 \\
 & & \\
inherits \subseteq \mathcal{P}(\mathbb{C}) \times \mathbb{C} & & \\
inherits(\{C_1\}, C_2) & = & \text{class } C_1 \text{ inherits directly from class } C_2 \\
inherits(\mathcal{C}, C_2) & = & \forall C_1 \in \mathcal{C} : inherits(C_1, C_2) \\
 & & \\
inherits^\star \subseteq \mathcal{P}(\mathbb{C}) \times \mathbb{C} & & \\
inherits^\star(\{C_1\}, C_2) & = & inherits(\{C_1\}, C_2) \\
inherits^\star(\{C_1\}, C_2) & = & \exists C_3 \in \mathbb{C} : inherits(\{C_1\}, C_3), inherits^\star(\{C_3\}, C_2) \\
inherits^\star(\mathcal{C}, C_2) & = & \forall C_1 \in \mathcal{C} : inherits^\star(\{C_1\}, C_2)
\end{array}
$$

Table 5.1: Primitive relations between classes

$$
\begin{array}{lcl}
invokes \subseteq \mathcal{P}(\mathbb{M}) \times \mathcal{P}(\mathbb{M}) \times \{self, v\} & & \\
invokes(\{m_1\}, \{m_2\}, self) & = & \text{method } m_1 \text{ invokes method } m_2 \text{ on the same instance} \\
invokes(\{m_1\}, \{m_2\}, v) & = & \text{method } m_1 \text{ invokes method } m_2 \text{ via the variable v} \\
invokes(\mathcal{M}, \{m_2\}, x) & = & \forall m_1 \in M : invokes(m_1, m_2, x) \\
invokes(\{m_1\}, \mathcal{M}, x) & = & \forall m_2 \in \mathcal{M} : invokes(\{m_1\}, \{m_2\}, x) \\
invokes(\mathcal{M}_1, \mathcal{M}_2) & = & \forall m_1 \in \mathcal{M}_1, \exists m_2 \in \mathcal{M}_2 : invokes(\{m_1\}, \{m_2\}, x) \\
 & & \\
invokes_r \subseteq \mathcal{P}(\mathbb{M}) \times \mathcal{P}(\mathbb{M}) \times \{self, v\} & & \\
invokes_r(\mathcal{M}_1, \mathcal{M}_2, x) & = & invokes(\mathcal{M}_1, \mathcal{M}_2, x), \mathcal{M}_1 \cap \mathcal{M}_2 \neq \emptyset \\
 & & \\
invokes^{\leftrightarrow} \subseteq \mathcal{P}(\mathbb{M}) \times \mathcal{P}(\mathbb{M}) \times \{self, v\} & & \\
invokes^{\leftrightarrow}(\mathcal{M}_1, \mathcal{M}_2, x) & = & \forall m_1 \in \mathcal{M}_1, \exists! m_2 \in \mathcal{M}_2 : invokes(\{m_1\}, \{m_2\}, x), \\
 & & \forall m_2 \in \mathcal{M}_2, \exists! m_1 \in \mathcal{M}_1 : invokes(\{m_1\}, \{m_2\}, x)
\end{array}
$$

Table 5.2: Primitive relations between methods

$understandsMessage \subseteq \mathcal{P}(\mathbb{C}) \times \mathcal{P}(\mathbb{M})$		
$understandsMessage(\{C\}, \{m\})$	$=$	class C understands message m (maybe indirectly)
$understandsMessage(\mathcal{C}, \{m\})$	$=$	$\forall C \in \mathcal{C} : understandsMessage(\{C\}, \{m\})$
$understandsMessage(\{C\}, \mathcal{M})$	$=$	$\forall m \in \mathcal{M} : understandsMessage(\{C\}, \{m\})$
$understandsMessage(\mathcal{C}, \mathcal{M})$	$=$	$\forall C \in \mathcal{C}, \forall m \in \mathcal{M} : understandsMessage(\{C\}, \{m\})$
$definesMethod \subseteq \mathcal{P}(\mathbb{C}) \times \mathcal{P}(\mathbb{M})$		
$definesMethod(\{C\}, \{m\})$	$=$	class C defines method m
$definesMethod(\mathcal{C}, \{m\})$	$=$	$\forall C \in \mathcal{C} : definesMethod(C, m)$
$definesMethod(\{C\}, \mathcal{M})$	$=$	$\forall m \in M : definesMethod(\{C\}, \{m\})$
$definesMethod(\mathcal{C}, \mathcal{M})$	$=$	$\forall C \in \mathcal{C}, \forall m \in M : definesMethod(\{C\}, \{m\})$
$definesVariable \subseteq \mathcal{P}(\mathbb{C}) \times \mathbb{V} \times \{variable, formal\}$		
$definesVariable(\{C\}, v, variable)$	$=$	class C defines variable v as an instance variable
$definesVariable(\mathcal{C}, v, variable)$	$=$	$\forall C \in \mathcal{C} : definesVariable(\{C\}, v, variable)$
$definesVariable(\{C\}, v, formal)$	$=$	class C defines v as a formal parameter in one of its methods
$definesVariable(\mathcal{C}, v, formal)$	$=$	$\forall C \in \mathcal{C} : definesVariable(\{C\}, v, formal)$
$creates \subseteq \mathcal{P}(\mathbb{M}) \times \mathcal{P}(\mathbb{C})$		
$creates : (\{m_1\}, \{C_1\})$	$=$	method m_1 creates an instance of class C_1
$creates : (\mathcal{M}, \mathcal{C})$	$=$	$\forall m \in \mathcal{M}, \exists! C \in \mathcal{C} : creates(\{m\}, \{C\})$

Table 5.3: Primitive relations between classes, methods and variables

$root \subseteq \mathbb{H} \rightarrow \mathbb{C} : \mathcal{H} \rightarrow C$	where C is the maximal element of $\mathcal{H}$ for the *inherits* relation
$leafs \subseteq \mathbb{H} \rightarrow \mathcal{P}(\mathbb{C}) : \mathcal{H} \rightarrow \{C_1, C_2, \ldots, C_n\}$	where C_i is a minimal element of $\mathcal{H}$ for the $inherits^*$ relation
$hierarchy \subseteq \mathbb{C} \rightarrow \mathbb{H} : C \rightarrow \mathcal{H}$	where $\mathcal{H}$ is the class hierarchy generated by C

Table 5.4: Preliminary definitions

5.2.3 Class Hierarchies

Besides classes, methods and variables, an important part of a framework consists of the different class hierarchies that it defines. Since this is such an important concept, we want to include it in our descriptions of metapatterns and in the formal model.

We define the set of all class hierarchies of the framework as $\mathbb{H}$, which is a subset of $\mathcal{P}(\mathbb{C})$, where $\mathcal{P}$ stands for the powerset, or the set of all subsets, and $\mathbb{C}$ is the set of all classes defined in the framework. If C is a class in the framework, we will use *hierarchy(C)* to denote the hierarchy of classes generated by C, i.e. class C together with all its direct and indirect descendants. A class hierarchy $\mathcal{H} \in \mathbb{H}$ always defines an $inherits \subseteq \mathbb{C} \times \mathbb{C}$ relation which forms a partial order. It has a unique maximal element $root(\mathcal{H})$ and a set of minimal elements $leafs(\mathcal{H})$ (see Table 5.4). Intuitively, the maximal element of a class hierarchy corresponds to the root of that hierarchy, while the set of minimal elements corresponds to the set of leaf classes of the hierarchy. Note that leaf classes are not required to be direct subclasses of the root class.

A hierarchy of classes will be denoted textually by a $\mathcal{H}$ symbol. A class hierarchy implementing a set of methods will be represented by a hierarchy symbol preceding that set: $\mathcal{H} :: \mathcal{M}$. As already mentioned, in the case of a class, $C :: \mathcal{M}$ means that all methods included in the set $\mathcal{M}$ are implemented by the class C. In the case of class hierarchies, however, $\mathcal{H} :: \mathcal{M}$ does not mean that each class in the hierarchy $\mathcal{H}$ implements all methods in the set $\mathcal{M}$. Instead, it means that all methods of $\mathcal{M}$ are defined in the root of the class hierarchy as methods that are either abstract or that provide a default implementation, and a concrete implementation exists for these methods in each leaf class of the hierarchy, i.e. each leaf class "understands" these methods. This does not necessarily mean that that leaf class actually provides that implementation, however. It could just as well be defined in one of the leaf class's superclasses. More specifically:

$$\mathcal{H} :: \mathcal{M} \Leftrightarrow definesMethod(\{root(\mathcal{H})\}, \mathcal{M}) \text{ and } understandsMessage(leafs(\mathcal{H}), \mathcal{M})$$

Graphically, a hierarchy will be represented by means of a large triangle symbol (see Figure 5.1). Furthermore, since the *understandsMessage* and *definesMethod* relations are not explicitly represented graphically, a symbol denoting a single method (or a set of methods) appearing inside a hierarchy symbol means that the root of that hierarchy defines the method(s), and each concrete leaf class of the hierarchy provides a concrete implementation.

5.3 Formal Definition of Fundamental Metapatterns

In this section, we will provide the basis for the formal definition of metapatterns and their associated transformations. As we will see, these transformations can be defined completely in terms of the participants of a metapattern. Since many of the metapatterns defined by Pree [Pre95] have the same participants, we are able to provide a meaningful extra abstraction by means of five *fundamental* metapatterns. The definition of a fundamental metapattern is highly parameterized,

so as to make it as generic as possible. In a later section, we will see how such generic definitions allow us to formally define all of Pree's metapatterns, as well as a number of other useful metapatterns that are not defined by Pree.

Because these fundamental metapatterns are parameterized, it is difficult to represent them graphically. We will thus not provide graphical illustrations of the fundamental metapatterns in this section, but we will do so for the existing metapatterns in later sections. We also note that the names of the fundamental metapatterns are generalizations of the names of the metapatterns defined by Pree.

5.3.1 The *Unification* Fundamental Metapattern

Definition

The *Unification* fundamental metapattern consists of three participants: a hierarchy $\mathcal{H}$, a set of template methods $\mathcal{H} :: \mathcal{M}_t$ and a set of hook methods $\mathcal{H} :: \mathcal{M}_h$. The template class and the hook class of this metapattern are one and the same class: the root class of the hierarchy $\mathcal{H}$. This class implements all of the template methods $\mathcal{H} :: \mathcal{M}_t$, each of which calls one or more of the hook methods from $\mathcal{H} :: \mathcal{M}_h$. These hook methods are defined by the root of the hierarchy and are provided with a concrete implementation for all concrete leaf classes of the hierarchy. Recall that the definition of *invokes* specifies that each *templateMethod* participant should call at least one *hookMethod* participant, and that not every *hookMethod* participant should be called by a *templateMethod* participant. The metapattern can thus be defined formally as follows:

$unificationFundamentalMP(\mathcal{H}, \mathcal{H} :: \mathcal{M}_t, \mathcal{H} :: \mathcal{M}_h)$ with

participants :

$hookhierarchy(\mathcal{H})$
$templatemethods(\mathcal{H} :: \mathcal{M}_t)$
$hookmethods(\mathcal{H} :: \mathcal{M}_h)$

constraints :

$understandsMessage(root(\mathcal{H}), \mathcal{H} :: \mathcal{M}_t)$
$definesMethod(root(\mathcal{H}), \mathcal{H} :: \mathcal{M}_h)$
$understandsMessage(leafs(\mathcal{H}), \mathcal{H} :: \mathcal{M}_h)$
$invokes(\mathcal{H} :: \mathcal{M}_t, \mathcal{H} :: \mathcal{M}_h, self)$

Transformations

We can define six transformations on this metapattern:

- The *addHookClass* operation takes a class C as an argument and adds this class to the leafs of the hook hierarchy $\mathcal{H}$ ($C \in leafs(\mathcal{H})$). Before this operation was applied, the class C was not part of the hook hierarchy ($C \notin \mathcal{H}$). The *removeHookClass* operation also takes a class C that is part of the *hookHierarchy* participant ($C \in leafs(\mathcal{H})$) and removes this class from the leafs of the hook hierarchy ($C \notin \mathcal{H}$).

- The *addHookMethod* operation has a method m, that is not part of the *hookMethod* participant ($m \notin \mathcal{H} :: \mathcal{M}_h$), and adds this method to the set of hook methods ($m \in \mathcal{H} :: \mathcal{M}_h$). Its *removeHookMethod* counterpart removes its argument m from the same set of hook methods (($m \notin \mathcal{H} :: \mathcal{M}_h$). Before this operation was applied m formed part of this set ($m \in \mathcal{H} :: \mathcal{M}_h$).

- The *addTemplateMethod* operation is similar to the *addHookMethod* operation, except that it adds its argument m to the set of template methods of the metapattern ($m \in \mathcal{H} :: \mathcal{M}_t$). The *removeTemplateMethod* on the other hand removes its argument m from the set of template methods ($m \notin \mathcal{H} :: \mathcal{M}_t$.

As can be seen, these transformations are entirely expressed in terms of operations on the metapattern's participants. Each operation either adds or removes a specific entity from a particular participant. Moreover, it can be observed that the transformations are completely orthogonal. The operations that add a particular entity are each defined on a different participant of the metapattern, as are the operations that remove an entity. Consequently, the operations will never interfere with each other.

Also note that there are no *addTemplateClass* and *removeTemplateClass* operations defined on the *Unification* pattern. It makes little sense to define an *addTemplateClass* operation, as it is the specific definition of this metapattern that it only contains one template class (which happens to be unified with the root of the hook hierarchy). For the same reason, we can not define a *removeTemplateClass* operation, as this would destroy this instance of the metapattern.

An important issue we want to stress is that the transformations should make sure that the metapattern constraints remain satisfied at all times. For example, when an *addHookClass* transformation adds a class to the leafs of the *hookHierarchy* participant, it should ensure that this class understands all *hookMethod* participants, since this is required by the constraints of the metapattern. This is possible, since we know the kind of metapattern a transformation is applied to, and we know the constraints of this metapattern. In a practical setting, the transformation will not only be responsible for adding a class to a hierarchy, but must also ensure that the class understands the appropriate methods. Similarly, since the constraints specify that each *templateMethod* participant should call at least one *hookMethod* participant, applying an *addTemplateMethod* transformation will automatically invoke an *addHookMethod* transformation, if a new *hookMethod* participant should be added.

5.3.2 The *Connection* Fundamental Metapattern

Definition

The *Connection* fundamental metapattern has five participants: a template class T, a hook hierarchy $\mathcal{H}$, a set of template methods $T :: \mathcal{M}_t$, a set of hook methods $\mathcal{H} :: \mathcal{M}_h$ and a reference variable v. This variable is defined in the *templateClass* participant, and can hold a reference to a single instance of a leaf class of the hook hierarchy $\mathcal{H}$, or to a number of instances of such classes. Furthermore, the *templateClass* participant implements a number of template methods $T :: \mathcal{M}_t$. Each of these template methods uses the object referenced by the variable v to call one or more hook methods. These hook methods $\mathcal{H} :: \mathcal{M}_h$ are defined by the root of the hook hierarchy $\mathcal{H}$, and a concrete implementation is present in the hierarchy for all its leafs. Formally:

$connectionFundamentalMP(T, \mathcal{H}, T :: \mathcal{M}_t, \mathcal{H} :: \mathcal{M}_h, v, Multiplicity, Association)$ with

participants :

$referencevariable(v)$
$templateclass(T)$
$hookhierarchy(\mathcal{H})$
$templatemethods(T :: \mathcal{M}_t)$
$hookmethods(\mathcal{H} :: \mathcal{M}_h)$

constraints :

$understandsMessage(T, T :: \mathcal{M}_t)$
$definesMethod(root(\mathcal{H}), \mathcal{H} :: \mathcal{M}_h)$
$definesVariable(T, V, Association)$
$understandsMessage(leafs(\mathcal{H}), \mathcal{H} :: \mathcal{M}_h)$
$uses(T, root(\mathcal{H}), Multiplicity)$
$invokes(T :: \mathcal{M}_t, \mathcal{H} :: \mathcal{M}_h, v)$

As can be seen, the definition of this fundamental metapattern is parameterized with a *Multiplicity* and an *Association* parameter. This provides the definition with a degree of flexibility that allows it to be used as a basis for the definition of a number of similar metapatterns. The

Multiplicity parameter can have two values assigned, *single* and *multiple*, and specifies in this way whether an instance of the *templateClass* participant refers to a single object of a *hookClass* participant, or to any number of such objects. The *Association* parameter, on the other hand, is used to determine whether the reference held by the *templateClass* participant is through an instance variable or through a formal parameter. As such, it can also accept two values: *variable* and *formal*.

The reference held by the *templateClass* participant is used by all *templateMethod* participants to call hook methods on objects of classes in the *hookHierarchy* participant. If this reference is defined by means of an instance variable in the *templateClass* participant, all template methods of that class have access to it. However, the reference can also be passed as a formal parameter. In that case, it should be possible to pass this reference as an argument each time a *templateMethod* participant is called. As such, each *templateMethod* participant should thus provide an appropriate formal parameter.

Transformations

Like for the *Unification* metapattern, six operations can be defined for the *Connection* metapattern.

- The *addHookClass* and *removeHookClass* operations respectively add and remove a class C to and from the leafs of the *hookHierarchy* participant $\mathcal{H}$ of this pattern ($C \in leafs(\mathcal{H})$ or $C \notin \mathcal{H}$).
- The *addHookMethod* operation adds a method m to the set of hook methods $\mathcal{M}_h$, while its counterpart, the *removeHookMethod* operation, removes a method m from this set ($m \in \mathcal{H} :: \mathcal{M}_h$ or $m \notin \mathcal{H} :: \mathcal{M}_h$).
- The *addTemplateMethod* and *removeTemplateMethod* operations respectively add and remove a template method m to and from the set of template methods $\mathcal{M}_t$ of this pattern ($m \in T :: \mathcal{M}_t$ or $m \notin T :: \mathcal{M}_t$).

Additionally, there are no *addTemplateClass* or *removeTemplateClass* operations defined for this metapattern. Like for the *Unification* metapattern, it makes little sense to define such operations, as instances of this metapattern should always contain exactly one *templateClass* participant.

Furthermore, the transformations should make sure the entity they add or remove from the metapattern instance satisfies all constraints defined by the metapattern, as was explained in Section 5.3.1.

5.3.3 The *Recursion* Fundamental Metapattern

Definition

The *Recursion* fundamental metapattern consists of five participants: a template class T, a hook hierarchy $\mathcal{H}$, a reference variable v, a set of template methods $T :: \mathcal{M}_h$ defined in class T and a set of hook methods $\mathcal{H} :: \mathcal{M}_h$ defined on the hierarchy $\mathcal{H}$. Actually, for all methods included in the set $T :: \mathcal{M}_h$, there is a method in the set $\mathcal{H} :: \mathcal{M}_h$ with the same signature, which is why the two sets have the same name.

In this metapattern, an instance of the template class T can either refer to exactly one instance or to a number of instances of leaf classes of the hook hierarchy $\mathcal{H}$. Such instances are referenced through the variable v, that is defined in the *templateClass* participant. This *templateClass* participant is itself included in the hook hierarchy $\mathcal{H}$, either as the root class or as a concrete leaf class. Furthermore, the root of the hook hierarchy $\mathcal{H}$ defines all hook methods as abstract methods, and all leaf classes provide a concrete implementation for these methods. Although it follows immediately that this also means that the class T provides a concrete implementation of these methods, we explicitly include this relation in the formal definition of the metapattern.

Additionally, the definition uses the $invokes_r$ relation to denote the fact that the *templateMethod* participants recursively call the *hookMethod* participants.

$recursionFundamentalMP(T, \mathcal{H}, T :: \mathcal{M}_h, \mathcal{H} :: \mathcal{M}_h, v, Multiplicity, Association)$ with

participants :

$referencevariable(v)$
$templateclass(T)$
$hookhierarchy(\mathcal{H})$
$templatemethods(T :: \mathcal{M}_h)$
$hookmethods(\mathcal{H} :: \mathcal{M}_h)$

constraints :

$definesMethod(root(\mathcal{H}), \mathcal{H} :: \mathcal{M}_h)$
$definesVariable(T, V, Association)$
$understandsMessage(leafs(\mathcal{H}), \mathcal{H} :: \mathcal{M}_h)$
$inherits^{\star}(T, root(\mathcal{H}))$
$understandsMessage(T, T :: \mathcal{M}_h)$
$uses(T, root(\mathcal{H}), Multiplicity)$
$invokes_r(T :: \mathcal{M}_h, \mathcal{H} :: \mathcal{M}_h, v)$.

Transformations

The following transformations that are defined on this metapattern:

- the *addHookClass* and *removeHookClass* operations are used to add or remove a class C respectively to and from the *hookHierarchy* participant of this pattern ($C \in leafs(\mathcal{H})$ and $C \notin \mathcal{H}$).
- the *addHookMethod* operation adds a hook method m to the set of hook methods $\mathcal{H} :: \mathcal{M}_h$ defined by the metapattern. Since the same set of methods is included twice as a participant of this pattern (once for the template class T and once for the hook hierarchy $\mathcal{H}$), we need to add the method m to the two sets ($m \in \mathcal{H} :: \mathcal{M}_h$ and $m \in T :: \mathcal{M}_h$).
- the same holds for the *removeHookMethod* and *removeTemplateMethod* operations: they remove a method m from the set of hook methods in the template class T and from the same set of methods defined on the hierarchy $\mathcal{H}$ ($m \notin \mathcal{H} :: \mathcal{M}_h$ and $m \notin T :: \mathcal{M}_h$).

Again, we do not provide *addTemplateClass* or *removeTemplateClass* operations, because they make no sense for this particular metapattern.

5.3.4 The *Hierarchy* Fundamental Metapattern

Definition

The *Hierarchy* fundamental metapattern also consists of five participants: two hierarchies $\mathcal{H}_1$ and $\mathcal{H}_2$, a single template method $\mathcal{H}_1 :: t$, a number of hook methods $\mathcal{H}_2 :: \mathcal{M}_h$ and a reference variable v. The root of the first hierarchy defines a template method t, which is overridden by each concrete leaf class of this hierarchy. Likewise, the root of the second hierarchy defines the hook methods $\mathcal{H}_2 :: \mathcal{M}_h$, which are provided with a concrete implementation for all leaf classes of the hierarchy. Furthermore, instances of classes in the first hierarchy can refer to instances of classes in the second hierarchy through the reference variable v. This can be achieved by defining an instance variable in the root of the hierarchy, which should be accessible to all leaf classes. Alternatively, the reference can be passed as an argument to the *templateMethod* participant as well. Each template method defined in the hierarchy $\mathcal{H}_1$ calls a different hook method defined on the hierarchy $\mathcal{H}_2$. This is reflected by means of the $invokes^{\leftrightarrow}$ relationship between the template method and the hook methods.

$hierarchyFundamentalMP(\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_1 :: t, \mathcal{H}_2 :: \mathcal{M}_h, v, Multiplicity, Association)$ with

participants :

$referencevariable(v)$
$templatehierarchy(\mathcal{H}_1)$
$hookhierarchy(\mathcal{H}_2)$
$templatemethods(\mathcal{H}_1 :: t)$
$hookmethods(\mathcal{H}_2 :: \mathcal{M}_h)$

constraints :

$definesMethod(root(\mathcal{H}_1), \mathcal{H}_1 :: t)$
$definesVariable(leafs(\mathcal{H}_1), V, Association)$
$understandsMessage(leafs(\mathcal{H}_1), \mathcal{H}_1 :: t)$
$definesMethod(root(\mathcal{H}_2), \mathcal{H}_2 :: \mathcal{M}_h)$
$understandsMessage(leafs(\mathcal{H}_2), \mathcal{H}_2 :: \mathcal{M}_h)$
$uses(root(\mathcal{H}_1), root(\mathcal{H}_2), Multiplicity)$
$invokes^{\leftrightarrow}(\mathcal{H}_1 :: t, \mathcal{H}_2 :: \mathcal{M}_h, v)$

Transformations

The transformations defined on this metapattern are the following:

- the *addHookClass* and *addTemplateClass* operations add a class C to the leafs of the hook hierarchy and the *templateHierarchy* participant respectively ($C \in leafs(\mathcal{H}_2)$ and $C \in leafs(\mathcal{H}_1)$).
- the *removeHookClass* and *removeTemplateClass* do exactly the opposite and remove a class C from the leafs of the hook hierarchy and the *templateHierarchy* participant of the metapattern ($C \notin \mathcal{H}_2$ and $C \notin \mathcal{H}_1$).
- the *addHookMethod* and *removeHookMethod* operations add and remove, a method m to and from the set of methods $\mathcal{H}_2 :: \mathcal{M}_h$ defined by the metapattern ($m \in \mathcal{H}_2 :: \mathcal{M}_h$ and $m \notin \mathcal{H}_2 :: \mathcal{M}_h$).

In this particular metapattern, there is no *addTemplateMethod* or *removeTemplateMethod* operation. The definition of the metapattern states that it has only one *templateMethod* participant *t*. As such, adding another template method is not possible, as there would then be two *templateMethod* participants. For the same reason, a *removeTemplateMethod* operation does not make much sense for this metapattern, as it would destroy its structure and the result would no longer adhere to the constraints imposed by the metapattern.

5.3.5 The *Creation* Fundamental Metapattern

Definition

The *Creation* fundamental metapattern consists of three participants: two hierarchies $\mathcal{H}_1$ and $\mathcal{H}_2$ and a single template method *t*. This method is defined in the root of the first hierarchy, and is provided with a concrete implementation in all of its leaf classes. For each of these leaf classes, the template method creates an instance of a different leaf class of the second hierarchy, as indicated by the $creates^{\leftrightarrow}$ relation:

$creationFundamentalMP(\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_1 :: t)$ with

participants :

$templatehierarchy(\mathcal{H}_1)$
$hookhierarchy(\mathcal{H}_2)$
$templatemethods(\mathcal{H}_1 :: t)$

constraints :

$$definesMethod(root(\mathcal{H}_1), \mathcal{H}_1 :: t)$$
$$understandsMessage(leafs(\mathcal{H}_1), \mathcal{H}_1 :: t)$$
$$creates^{\leftrightarrow}(\mathcal{H}_1 :: t, leafs(\mathcal{H}_2))$$

Transformations

These are the transformations that are applicable to this metapattern:

- the *addHookClass* operation adds a class C to the leafs of the *hookHierarchy* participant of the metapattern ($C \in leafs(\mathcal{H}_2)$). Likewise, the *addTemplateClass* operation adds a class C to the leafs of the *templateHierarchy* participant ($C \in leafs(\mathcal{H}_1)$).
- the *removeHookClass* operation removes a given class C from the leafs of the hook hierarchy ($C \notin \mathcal{H}_2$), while the *removeTemplateClass* does the same for the leafs of the template hierarchy ($C \notin \mathcal{H}_1$).

A similar reasoning as for the *Hierarchy* metapattern shows that it is not possible to define an *addTemplateMethod* or *removeTemplateMethod* operation on this metapattern. Furthermore, we do not even define *addHookMethod* or *removeHookMethod* operations, as the the metapattern does not contain any *hookMethod* participants.

5.3.6 Overview of Transformations on Metapatterns

An overview of the different transformations and the types of metapatterns on which they are applicable is presented in Table 5.5. As can be seen, when an operation that adds a particular entity to an instance is defined on a metapattern, the corresponding operation that removes such an entity is also defined. We can also observe that the *addHookClass* and *removeHookClass* operations are applicable on each type of metapattern, since all types have a *hookHierarchy* participant. The *addTemplateClass* and *removeTemplateClass* operations, on the other hand, are only applicable on the *Hierarchy* and *Creation* fundamental metapatterns, since these are the only ones that have a *templateHierarchy* participant and all other types of metapatterns must have exactly one *templateClass* participant. Furthermore, since the *Creation* fundamental metapatterns is the only metapattern not to include *hookMethod* participants, it is the only metapattern on which no *addHookMethod* and *removeHookMethod* operations are defined. Last, the *addTemplateMethod* and *removeTemplateMethod* operations are only defined on the *Unification, Connection* and *Recursion* fundamental metapatterns, since the definition of the other metapatterns explicitly states that they only contain one *templateMethod* participant.

5.4 The Existing Metapatterns

In this section, we will present the formal definition of the *Unification, 1:N Recursive Connection, 1:1 Hierarchy Connection* and *1:1 Hierarchy Creation* metapatterns defined by Pree [Pre95]. We refer to the appendix for a definition of all other metapatterns defined by Pree that are not considered here.

We will use the following template structure for our discussion. The *Definition* part will provide the formal definition of the metapattern in terms of one of the fundamental metapatterns, as well as an intuitive description of this definition in natural language. The next part will show the typical structure of a *templateMethod* participant in the particular metapattern under consideration. The structure is always more or less fixed for each type of metapattern, in that we know that a template method will always call one or more hook methods. Furthermore, we also know that the template method uses the *referenceVariable* participant of a metapattern (if it exists) as a receiver to send the hook messages to. Consequently, we can use the metapattern description in a clever way to generate skeleton code for template methods automatically. The last part of the discussion contains an example of a design pattern that uses the structure of the particular metapattern, and

	MP_1	MP_2	MP_3	MP_4	MP_5
$addHookClass$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
$addTemplateClass$	$\times$	$\times$	$\times$	$\surd$	$\surd$
$addHookMethod$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
$addTemplateMethod$	$\surd$	$\surd$	$\surd$	$\times$	$\times$
$removeHookClass$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
$removeTemplateClass$	$\times$	$\times$	$\times$	$\surd$	$\surd$
$removeHookMethod$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
$removeTemplateMethod$	$\surd$	$\surd$	$\surd$	$\times$	$\times$

MP_1 = *Unification* fundamental metapattern, MP_2 = *Connection* fundamental metapattern
MP_3 = *Recursion* fundamental metapattern, MP_4 = *Hierarchy* fundamental metapattern
MP_5 = *Creation* fundamental metapattern

Table 5.5: Overview of Transformations on Metapatterns

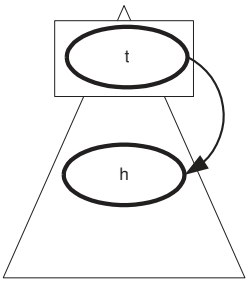


Figure 5.2: The *Unification* metapattern

shows how this design pattern instance can be described by means of our formal model. Where possible, we will use an example design pattern from our Scheme framework, as these have already been explained in detail in Chapter 3.

5.4.1 The *Unification* Metapattern

Definition

The *Unification* metapattern is depicted graphically in Figure 5.2. It consists of a number of *hookMethod* participants that are defined in a hierarchy participant, and a number of *templateMethod* participants that are defined in the root class of that hierarchy participant. As such, this metapattern can be defined in terms of the *Unification* fundamental metapattern and is defined formally as follows:

$$unificationMetapattern(\mathcal{H}, \mathcal{H} :: \mathcal{M}_t, \mathcal{H} :: \mathcal{M}_h)$$
$$::=$$
$$unificationFundamentalMP(\mathcal{H}, \mathcal{H} :: \mathcal{M}_t, \mathcal{H} :: \mathcal{M}_h)$$

Structure of the template methods

The template methods of this metapattern each have the same structure (shown below as Smalltalk code):

```
t
    ...
```

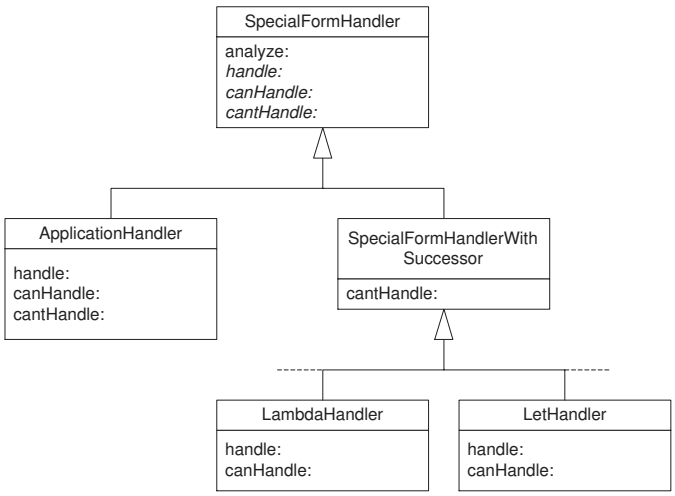


Figure 5.3: Example Design Pattern for the *Unification* metapattern

```
self h1.
...
self h2.
...
```

where `t` is an element of the set of template methods defined by the metapattern, and is defined in the root of the hierarchy $\mathcal{H}$. `h1` and `h2` are elements of the set of hook methods of this metapattern, are defined in the root of $\mathcal{H}$ and have a concrete implementation in all leaf classes of $\mathcal{H}$. Note that the template method `t` uses `self` as the receiver of the `h1` and `h2` messages, since this is required by the *invokes* relation of the *Unification* fundamental metapattern (see Section 5.3.1).

Example Design Pattern

A design pattern whose structure can be captured by this particular metapattern is the *Template Method* design pattern. As a concrete example, consider the particular instance of the *Template Method* design pattern from the Scheme framework depicted in Figure 5.3. The hierarchy depicted in this figure represents the *hookHierarchy* participant of the metapattern. The `analyze:` method defined in the `SpecialFormHandler` class plays the role of the *templateMethod* participant, and is the only template method occurring in this particular instance. It calls the `handle:, canHandle:` and `cantHandle:` methods, which represent the *hookMethod* participants.

Formally, we can represent this particular pattern instance as follows:

$$
\begin{aligned}
&hookhierarchy(hierarchy(SpecialFormHandler))\\
&templatemethods(SpecialFormHandler :: analyze:)\\
&hookmethods(hierarchy(SpecialFormHandler) :: \{handle:, cantHandle:,\\
&\quad canHandle:\})
\end{aligned}
$$

An alternative way to specify the *hookHierarchy* participant would be:

$$
\begin{aligned}
&hookhierarchy(\{SpecialFormHandler, ApplicationHandler,\\
&\quad SpecialFormHandlerWithSuccessor, LambdaHandler, LetHandler, \ldots\}
\end{aligned}
$$

But, as explained in Section 5.2.3, we can use the shorthand notation

$$hierarchy(SpecialFormHandler)$$

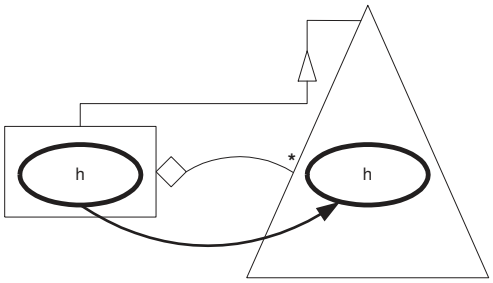


Figure 5.4: The *1:N Recursive Connection* metapattern

which is much more convenient. In a similar vein, the hook method participants could be specified as follows:

$$hookmethods(\{SpecialFormHandler :: handle :,\\ SpecialFormHandler :: canHandle :, SpecialFormHandler :: cantHandle :,\\ ApplicationHandler :: handle :, ApplicationHandler :: canHandle,\\ ApplicationHandler :: cantHandle, \ldots\})$$

which we can denote instead as follows:

$$hookmethods(\{SpecialFormHandler, ApplicationHandler,\\ LetHandler, \ldots\} :: \{handle :, canHandle :, cantHandle :\})$$

from which the obvious shorthand notation can be derived easily:

$$hookmethods(hierarchy(SpecialFormHandler) :: \{handle :, cantHandle :,\\ canHandle :\})$$

5.4.2 The *1:N Recursive Connection* Metapattern

Definition

The *1:N Recursive Connection* metapattern (see Figure 5.4) is defined in terms of the *Recursion* fundamental metapattern, and consists of five participants: a template class T, a hook hierarchy $\mathcal{H}$, a reference variable v, a set of hook methods M_h defined in class T and a set of hook methods M_h defined on the hierarchy $\mathcal{H}$. The class T itself is also part of the hierarchy $\mathcal{H}$.

In this metapattern, an instance of the template class T refers to multiple instances of a class in the hook hierarchy $\mathcal{H}$, which is denoted by the *multiple* annotation in the definition. The reference to an object of the hook hierarchy is via the reference variable v, which is defined as an instance variable in the *templateClass* participant. This is denoted by the *variable* annotation in the definition, which thus looks as follows:

$$oneToManyRecursiveConnectionMetapattern(T, \mathcal{H}, T :: \mathcal{M}_h, \mathcal{H} :: \mathcal{M}_h, V)\\ ::=\\ recursionFundamentalMP(T, \mathcal{H}, T :: \mathcal{M}_h, \mathcal{H} :: \mathcal{M}_h, V, multiple, variable)$$

Structure of the template methods

The structure of template methods in the template class T looks as follows:

```
h
   ...
   v do: [ :hookobject | hookobject h ].
   ...
```

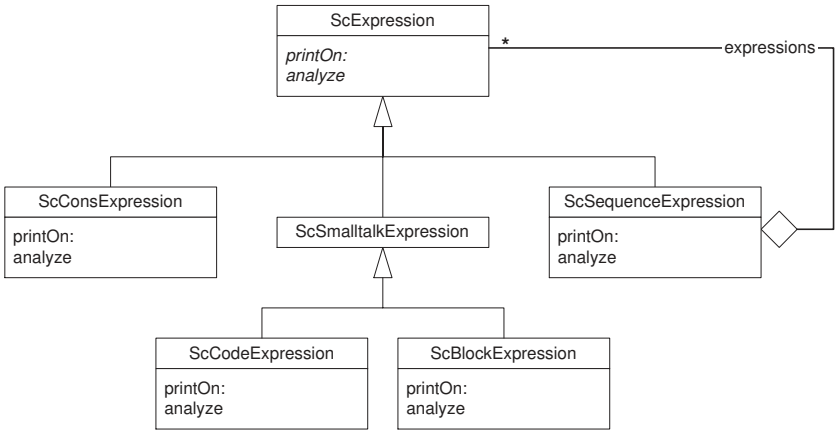


Figure 5.5: Example Design Pattern for the *1:N Recursive Connection* metapattern

where the `v` object contains a collection of all hook objects an instance of the template class T refers to, and h is an element of the set of template (or hook) methods, defined in the *hookHierarchy* participant.

Example Design Pattern

The *Composite* design pattern serves as an example of a design pattern that uses the structure of this metapattern. Consider the particular instance of this design pattern in Figure 5.5 that appears in the Scheme framework. The `ScSequenceExpression` represents the *templateClass* participant of the metapattern, while the `ScExpression` hierarchy itself represent the *hookHierarchy* participant. This hierarchy defines two methods, `printOn:` and `analyze`. The methods defined in the `ScSequenceExpression` class are the *templateMethod* participants of the metapattern, while the implementation of these methods in other classes of the hierarchy corresponds to the *hookMethod* participants. The *referenceVariable* participant of the metapattern is represented by the `expressions` instance variable defined in the `ScSequenceExpression` class.

The formal description of this particular instance is the following:

$$
\begin{aligned}
&hookhierarchy(hierarchy(ScExpression))\\
&hookmethods(hierarchy(ScExpression) :: \{analyze :, printOn :\})\\
&templateclass(ScSequenceExpression)\\
&templatemethods(ScSequenceExpression :: \{analyze :, printOn :\})\\
&referencevariable(expressions)
\end{aligned}
$$

5.4.3 The *1:1 Hierarchy Connection* Metapattern

Definition

The *1:1 Hierarchy Connection* pattern, as depicted in Figure 5.6, is defined in terms of the *Hierarchy* fundamental design pattern. It consists of two hierarchy participants, a single *templateMethod* participant, a number of *hookMethod* participants and a *referenceVariable* participant. Each leaf class in the first hierarchy refers to a single instance of a class in the second hierarchy through this reference variable. In this particular metapattern, this reference is passed as a formal parameter to the *templateMethod* participant. The formal definition is thus the following:

$$
\begin{gathered}
oneToOneHierarchyConnectionMetapattern(\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_1 :: t, \mathcal{H}_2 :: \mathcal{M}_h, V)\\
::=\\
hierarchyFundamentalMP(\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_1 :: t, \mathcal{H}_2 :: \mathcal{M}_h, V, single, formal)
\end{gathered}
$$

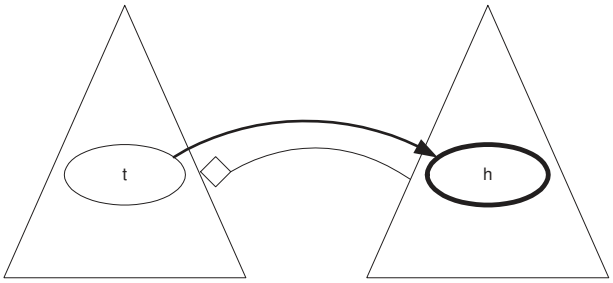


Figure 5.6: The *1:1 Hierarchy Connection* metapattern

Structure of the template methods

The typical structure of the template method t looks as follows:

```
t: v
   ...
   v h
   ...
```

where t is the template method defined in the *templateHierarchy* participant, `v` is the reference the template hierarchy holds to an object of the hook hierarchy and h is a member of the hook methods defined by the *hookHierarchy* participant. The implementation of the template method of other classes in the template hierarchy is similar, except for the fact that it calls another hook method.

Example Design Pattern

The *Visitor* design pattern uses the structure of this metapattern to implement its behavior. Consider the instance of this design pattern presented in Chapter 3 and depicted in Figure 5.7. In this instance, the `ScExpression` hierarchy plays the role of the *templateHierarchy* participant of the metapattern, while the `AbstractASTEnumerator` hierarchy corresponds to the *hookHierarchy* participant. The *templateMethod* participant of the metapattern is represented by the `nodeDo:` method, that is defined on the `ScExpression` hierarchy and all methods defined in the `AbstractASTEnumerator` hierarchy (such as `doBlockExpression:, doCodeExpression:` and so on) are *hookMethod* participants. The argument that is passed to the `nodeDo:` method corresponds to the reference variable of the metapattern, as it is used to connect the two hierarchies.

This instance can be formally specified in our model as follows:

$$
\begin{aligned}
&hookhierarchy(hierarchy(AbstractASTEnumerator))\\
&hookmethods(hierarchy(AbstractASTEnumerator) :: \{doBlockExpression :,\\
&\quad doCodeExpression : ...\})\\
&templatehierarchy(hierarchy(ScExpression))\\
&templatemethods(hierarchy(ScExpression) :: nodeDo :)\\
&referencevariable(aVisitor)
\end{aligned}
$$

5.4.4 The *1:1 Hierarchy Creation* Metapattern

Definition

The *1:1 Hierarchy Creation* metapattern, as shown in Figure 5.8, consists of three participants: two hierarchies and a template method. This metapattern corresponds to the *Creation* fundamental metapattern, and can be formally defined as follows:

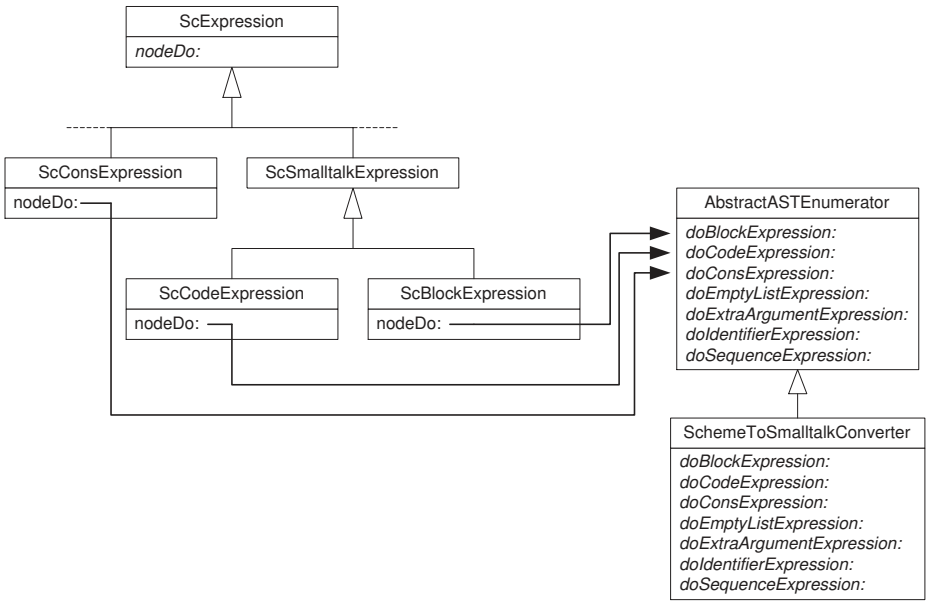


Figure 5.7: Example Design Pattern for the *1:1 Hierarchy Connection* metapattern

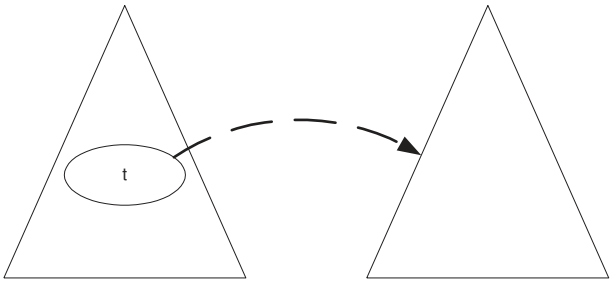


Figure 5.8: The *1:1 Hierarchy Creation* metapattern

$$oneToOneHierarchyCreationMetapattern(\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_1 :: t)$$
$$::=$$
$$creationFundamentalMP(\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_1 :: t)$$

Structure of the template methods

The structure of the template method of this metapattern looks as follows:

```
t
   ^H2LeafClass new
```

The template method t is defined as an abstract method in the *templateHierarchy* participant of the metapattern, and is provided with a concrete implementation for all leaf classes of that hierarchy. This implementation instantiates a concrete class of the *hookHierarchy* participant (the `H2LeafClass` class in the example), and returns it. Furthermore, these concrete methods each have a structure similar to the one above, except that they instantiate an object of a different class of the second hierarchy each time.

Example Design Pattern

The *Factory Method* design pattern is an example of a design pattern that uses the structure of this metapattern. One particular instance of this design pattern was presented in Chapter 3 and is depicted in Figure 5.9. In this instance, the `ScExpression` hierarchy represents the *templateHierarchy* participant of the metapattern, and the `Closure` hierarchy corresponds to the *hookHierarchy* participant. Furthermore, the `newClosure` method that is defined on the `ScExpression` hierarchy plays the role of the *templateMethod* participant, as it is responsible for creating the appropriate object from the `Closure` hierarchy.

We can represent this particular instance formally in the following way:

$$hookhierarchy(hierarchy(Closure))$$
$$templatehierarchy(hierarchy(ScExpression))$$
$$templatemethods(hierarchy(ScExpression) :: newClosure)$$

5.5 Overlapping of Metapatterns

A framework will inevitably use many instances of the presented fundamental metapatterns in its implementation. Furthermore, whereas the fundamental metapatterns do not overlap themselves, their instances will, since classes and methods in the framework can participate in more than one metapattern instance or in more than one design pattern instance. As metapattern-specific transformations are defined in terms of operations on the metapattern's participants, we should consider what happens when an operation is applied on a participant that already plays a role in another metapattern instance. In this section, we will define a change propagation strategy, based upon the metapattern-specific transformations and the possible ways in which metapattern instances can overlap.

5.5.1 An Illustrating Example

In Section 4.2.4, we presented a first example of overlapping design pattern instances. An instance of the *Visitor* and the *Composite* design pattern overlapped, because the `Closure` hierarchy participates in both instances. As another example, consider the situation depicted in Figure 5.10. It shows how instances of the *Composite* and *Factory Method* design patterns overlap. These design patterns correspond to the *Recursion* and *Creation* fundamental metapatterns. The overlapping is due to the fact that the `ScExpression` class hierarchy participates in both instances: it is a

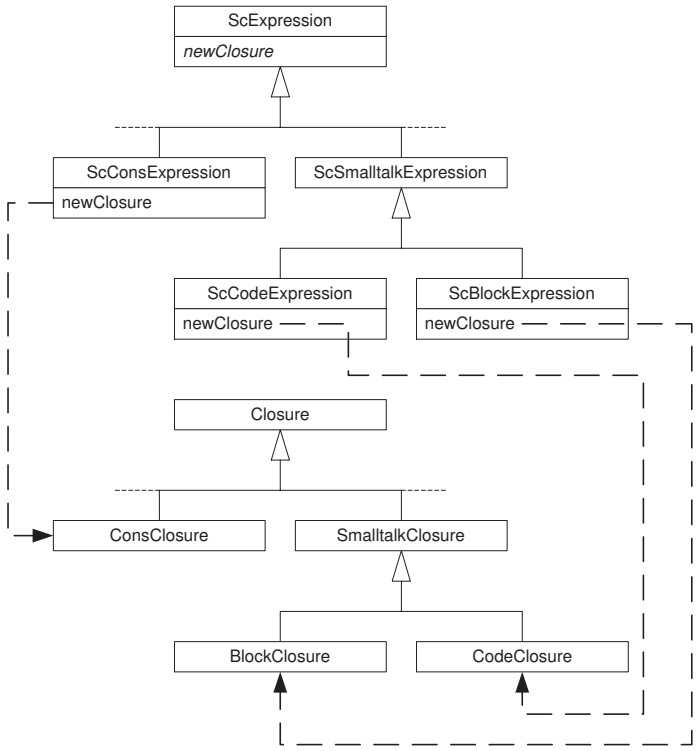


Figure 5.9: Example Design Pattern for the *1:1 Hierarchy Creation* metapattern

hookHierarchy participant in the *Recursion* metapattern instance, while it is a *templateHierarchy* participant in the *Creation* metapattern instance.

Consider now what happens when we apply an *addHookClass* transformation to the *Recursion* metapattern instance, to introduce a new `ScQuoteExpression` class into the `ScExpression` hierarchy. This operation will add the class to the leafs of the *hookHierarchy* participant, while additionally, it should make sure that this class adheres to the metapattern's constraints. Therefore the class should implement all *hookMethod* participants of this instance of the *Recursion* fundamental metapattern. In this case, there are two such participants: the `printOn:` and `analyze` methods.

While the constraints of the *Recursion* fundamental metapattern are satisfied, those of the *Creation* fundamental metapattern are not. Since the `ScExpression` class hierarchy participates in the *Creation* metapattern instance as a *templateHierarchy* participant, the newly added `ScQuoteExpression` should be added to the leafs of that participant and it should implement all *templateMethod* participants, as well. Thus, an *addTemplateClass* transformation should be applied on the *Creation* metapattern instance, which will take care of these issues.

We learn from this example that, whenever two metapattern instances overlap and we apply a transformation to one of these instances, we should consider the effect on the overlapping instance. When a transformation is applied to one metapattern instance, it will make sure that this metapattern's constraints are satisfied. It does not take into account the constraints of overlapping metapattern instances, however. To satisfy the constraints of the overlapping metapattern instances as well, additional transformation should be applied on them. What we need is a way to propagate a particular transformation that has been applied to one metapattern instance to all its overlapping instances. In what follows, we will define such a change propagation strategy, by considering the different ways in which metapattern instances can overlap and how transformations applied to one metapattern instance will give rise to transformations on overlapping instances.

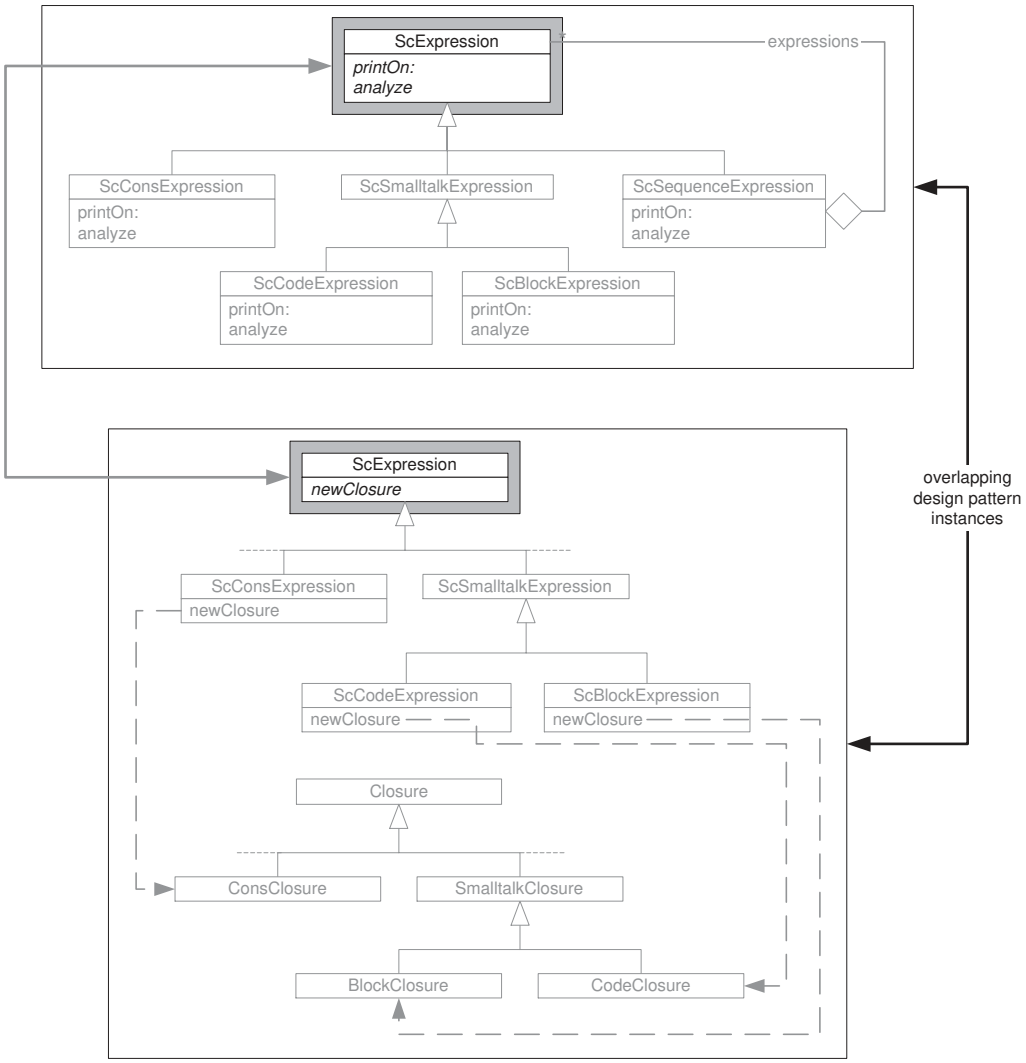


Figure 5.10: Instances of the *Recursion* and *Creation* fundamental metapatterns overlap

Note that, conceptually, both an *addHookClass* and an *addTemplateClass* transformation should be applied to the *Recursion* and *Creation* metapattern instances in the above example. This is required by our model, since the operations ensure that the `ScQuoteExpression` class is added as a participant to the appropriate metapattern instance and that it satisfies the desired constraints. In practice, however, applying both transformations would add the `ScQuoteExpression` class twice to the implementation, which is not what is expected. Therefore, when using the model in practice, we should ensure that the class is only added once, as we will explain in the next chapter. For the sake of the discussion in this chapter, however, we can safely ignore this technical difficulty. Note that the same holds for all other transformations.

5.5.2 Approach

We will first define the conditions under which a metapattern instance a overlaps with a metapattern instance b. We will define such overlapping in terms of the class and class hierarchy participants of the involved metapattern instances only. In this way, the conditions for overlapping become independent of the specific kind of metapatterns. As a result, when new metapatterns are added to the formal model, these conditions will not have to be changed. To simplify matters somewhat, we will extend the *inherits* relation to work for class hierarchies as well.

$$
\begin{array}{lcl}
inherits_h^\star \subseteq \mathbb{H} \times \mathbb{H} & & \\
inherits_h^\star(\mathcal{H}_1, \mathcal{H}_2) & = & inherits^\star(root(\mathcal{H}_1), root(\mathcal{H}_2)) \\
inherits_h^\star(\{C\}, \mathcal{H}) & = & inherits^\star(C, root(\mathcal{H}))
\end{array}
$$

Afterwards, we will consider the conditions under which a transformation applied to metapattern instance a will give rise to a transformation on the overlapping metapattern instance b. This in essence defines our change propagation algorithm. Due to their orthogonality, transformations that add a particular participant to a metapattern instance will never give rise to a transformation that removes a particular participant. Likewise, transformations involving class participants will never require transformations involving method participants. Therefore, we will consider transformations involving class participants separately from those involving method participants.

In the specification of the conditions for overlapping, we will need to be able to distinguish between transformations applied on different metapattern instances, as well as the different participants of those different instances. To this extent, we will use subscripts. For example, to denote an *addHookClass* transformation that is applied on an instance a, $addHookClass_a$ will be used. Similarly, to describe the *hookHierarchy* participant of metapattern instance a, $\mathcal{H}_{h,a}$ will be used, while the *templateHierarchy* participant of metapattern instance b will be denoted by $\mathcal{H}_{t,b}$.

5.5.3 Definition of Overlapping Conditions

There are many different ways in which two metapattern instances can overlap. In this section, we will consider these different ways and define the conditions when an overlapping may occur.

First of all, observe that only the *Connection* fundamental metapattern contains a single class participant (see Section 5.3.2). All other fundamental metapatterns contain class hierarchy participants. If we first consider only those fundamental metapatterns, and since we know that class hierarchies can overlap in only four different ways, we can define the conditions under which a metapattern instance a (of the *Unification, Recursion, Hierarchy* or *Creation* fundamental metapattern) overlaps with a metapattern instance b (of those same fundamental metapatterns):

Overlapping Condition 1 a class hierarchy is a *hookHierarchy* participant in instances a and b at the same time. The *hookHierarchy* participant of instance a may also be a part of the *hookHierarchy* participant of instance b. Formally: $inherits_h^*(\mathcal{H}_{h,a}, \mathcal{H}_{h,b})$.

Overlapping Condition 2 a class hierarchy is a *hookHierarchy* participant in instance a, while at the same time it is a *templateHierarchy* participant in instance b, or it forms part of that hierarchy. Formally: $inherits_h^*(\mathcal{H}_{h,a}, \mathcal{H}_{t,b})$.

Overlapping Condition 3 a class hierarchy is a *templateHierarchy* participant in instance a, and at the same time it is a *hookHierarchy* participant in instance b, or it forms part of that hierarchy. Formally: $inherits_h^*(\mathcal{H}_{t,a}, \mathcal{H}_{h,b})$.

Overlapping Condition 4 a class hierarchy is a *templateHierarchy* participant in instances a and b at the same time. The *templateHierarchy* participant of instance a may also be a part of the *templateHierarchy* participant of instance b. Formally: $inherits_h^*(\mathcal{H}_{t,a}, \mathcal{H}_{t,b})$.

These are the only conditions under which metapattern instances overlap by means of their class hierarchy participants. What remains is defining the conditions for overlapping of the *Connection* fundamental metapattern with the other metapatterns. Clearly, overlappings that involve the *hookHierarchy* participant of this fundamental metapattern are already covered by the above conditions. The only remaining ways of overlapping involve the *templateClass* participant. There are three different ways in which this participant can participate in two metapattern instances at once: an instance a of any kind of fundamental metapattern overlaps with an instance b of the *Connection* fundamental metapattern if:

Overlapping Condition 5 the class is a *templateClass* participant in instances a and b at the same time. Formally: $T_a = T_b$. In this case, instance a is an instance of the *Connection* fundamental metapattern, as well.

Overlapping Condition 6 a class hierarchy is a *hookHierarchy* participant in instance a, while the *templateClass* participant in instance b forms part of that hierarchy. Formally, this is expressed as follows: $inherits_h^*(\{T_b\}, \mathcal{H}_{h,a})$.

Overlapping Condition 7 similarly, a class hierarchy is a *templateHierarchy* participant in instance a, while the *templateClass* participant in instance b is part of that hierarchy. Formally: $inherits_h^*(\{T_b\}, \mathcal{H}_{t,a})$.

Note that these conditions specify how an instance of any kind of fundamental metapattern overlaps with an instance of the *Connection* fundamental metapattern, via the *templateClass* participant , but not vice versa. There are no interesting ways in which an instance of the *Connection* fundamental metapattern overlaps with an instance of any other fundamental metapattern through this participant. This is due to the specific nature of the *templateClass* participant: it is a single class and it contains only template methods. The only way that metapattern instances can overlap through a single class participant, is if this class participates in both instances. Therefore, an instance a of the *Connection* fundamental metapattern can only overlap with an instance b through the *templateClass* participant, if b is an instance of the *Connection* fundamental metapattern as well. Moreover, the only transformations that involve the *templateClass* participant are the *addTemplateMethod* and *removeTemplateMethod* operations. These two operations will however never require additional operations on overlapping metapattern instances, as we will discuss in the next section.

As an illustration of how these conditions can be used to detect overlappings of metapattern instances, consider the metapattern instances of Figure 5.10. These overlap according to *Overlapping Condition 2*. The `ScExpression` class hierarchy is a *hookHierarchy* participant in the *Recursion* metapattern instance and at the same time a *templateHierarchy* participant in the *Creation* metapattern instance. Note that these two instances also overlap according to *Overlapping Condition 7*, because the `ScExpression` hierarchy is a *templateHierarchy* participant in the *Creation* metapattern instance, and it contains the `ScSequenceExpression` *templateClass* participant of the *Recursion* metapattern instance. This does not necessarily have to be the case, however. It occurs here because the *templateClass* participant in an instance of the *Recursion* fundamental metapattern resides in the *hookHierarchy* participant of that instance as well.

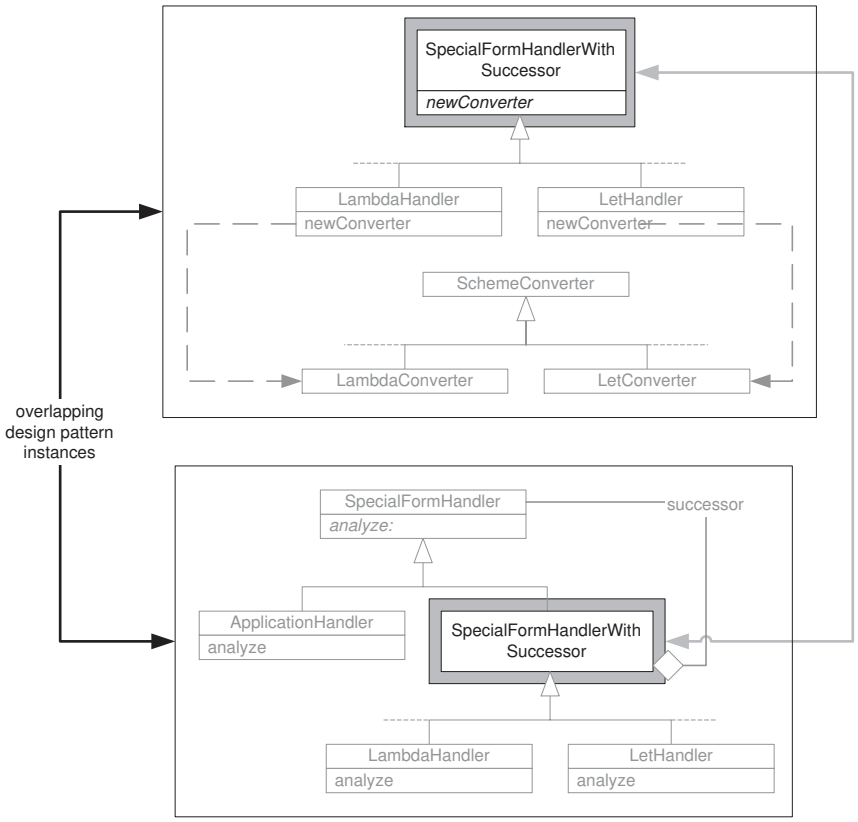


Figure 5.11: An instance of the *Recursion* fundamental metapattern overlaps with an instance of the *Creation* fundamental metapattern

$addHookClass_a(C) \Rightarrow addHookClass_b(C)$ if $inherits_h^*(\mathcal{H}_{h,a}, \mathcal{H}_{h,b})$ $addHookClass_a(C) \Rightarrow addTemplateClass_b(C)$ if $inherits_h^*(\mathcal{H}_{h,a}, \mathcal{H}_{t,b})$ $addTemplateClass_a(C) \Rightarrow addHookClass_b(C)$ if $inherits_h^*(\mathcal{H}_{t,a}, \mathcal{H}_{h,b})$ $addTemplateClass_a(C) \Rightarrow addTemplateClass_b(C)$ if $inherits_h^*(\mathcal{H}_{t,a}, \mathcal{H}_{t,b})$
$removeHookClass_a(C) \Rightarrow removeHookClass_b(C)$ if $inherits_h^*(\mathcal{H}_{h,a}, \mathcal{H}_{h,b})$ $removeHookClass_a(C) \Rightarrow removeTemplateClass_b(C)$ if $inherits_h^*(\mathcal{H}_{h,a}, \mathcal{H}_{t,b})$ $removeTemplateClass_a(C) \Rightarrow removeHookClass_b(C)$ if $inherits_h^*(\mathcal{H}_{t,a}, \mathcal{H}_{h,b})$ $removeTemplateClass_a(C) \Rightarrow removeTemplateClass_b(C)$ if $inherits_h^*(\mathcal{H}_{t,a}, \mathcal{H}_{t,b})$

Table 5.6: Transformations involving class participants

Figure 5.11 shows another example of how instances of the *Recursion* and *Creation* can overlap. This time, however, the *Creation* metapattern instance overlaps with the *Recursion* metapattern instance, according to *Overlapping Condition 3*. This example illustrates that two hierarchy participants need not necessarily be equal. In this case, the `SpecialFormHandler` hierarchy is a *hookHierarchy* participant in an instance of the *Recursion* fundamental metapattern, whereas the `SpecialFormHandlerWithSuccessor` hierarchy is a *templateHierarchy* participant in an instance of the *Creation* fundamental metapattern. Since the `SpecialFormHandlerWithSuccessor` class is a descendant of the `SpecialFormHandler` class, the two hierarchies, and thus the metapattern instances that contain them overlap.

5.5.4 Influence of Overlapping on Transformations

Based on the different ways in which (the class and class hierarchies of) metapattern instances overlap, we can derive how a transformation on metapattern instance a propagates to a metapattern instance b. We first consider only those transformations that add or remove a class participant from a metapattern instance. In the next section, we also consider those operations that involve method participants.

Transformations Involving Class Participants

The first part of Table 5.6 shows when *addHookClass* and *addTemplateClass* operations on instance a gives rise to *addHookClass* and *addTemplateClass* transformations on instance b. The

second part defines in a similar way how *removeHookClass* and *removeTemplateClass* operations on instance a may require *removeHookClass* and *removeTemplateClass* operations on instance b.

An *addHookClass* transformation on instance a will require an *addHookClass* transformation on instance b, if *Overlapping Condition 1* is satisfied This is denoted by means of the $inherits^*_h(\mathcal{H}_{h,a}, \mathcal{H}_{h,b})$ condition. If *Overlapping Condition 2* is satisfied, then an *addHookClass* on instance a will give rise to an *addTemplateClass* on instance b. Based on similar reasonings, we can derive how all other transformations influence one another.

Observe that only overlapping conditions 1 to 4 are used. Overlapping conditions 5 to 7 specify when an instance of any kind of fundamental metapattern overlaps with an instance of the *Connection* fundamental metapattern, via its *templateClass* participant. Under such conditions, a transformation involving class participants on the former instance will never give rise to a transformation on the latter instance, since the *Connection* fundamental metapattern does not define *addTemplateClass* or *removeTemplateClass* operations.

Transformations Involving Method Participants

Just as we did for class participants, we can define how transformations involving method participants propagate to overlapping metapattern instances (see Table 5.7).

A first important observation is that we do not consider how *addHookMethod* and *addTemplateMethod* transformations influence one another. This is simply because they don't. When we add a new method participant to a particular metapattern instance, we should not automatically add it to all overlapping instances as well. A method participant need not necessarily be included in all overlapping metapattern instances, since it is perfectly well possible that it participates in one metapattern instance, while it does not participate in any other instances. If the developer wants to include it as a participant in other instances as well, he should explicitly invoke the appropriate transformation on those instances.

Second, a *removeHookMethod* or *removeTemplateMethod* transformation on an instance a will only require an additional transformation on an overlapping metapattern instance b, if the method participant that is removed from instance a is also a method participant in instance b. This is included as an extra condition in all conditions, as can be seen in Table 5.7.

Third, the *removeTemplateMethod* transformation is only defined for the *Unification, Connection* and *Recursion* fundamental metapatterns. This is the reason why the condition under which a *removeTemplateMethod* operation on instance a gives rise to a similar operation on instance b does not involve class hierarchies, but only *templateClass* participants.

Last, a *removeTemplateMethod* transformation will never give rise to an *removeHookMethod* transformation. This is due to the specific nature of these operations and the participants upon which they are applied. A *templateMethod* participant is never overridden in the *Unification, Connection* or *Recursion* fundamental metapatterns. A *hookMethod* participant, on the other, is implemented across the *hookHierarchy* participant (e.g. the root of the hook hierarchy defines the method, and specific subclasses in the hierarchy provide a concrete implementation for all the leaf classes of the hierarchy). Therefore, the intent of the *removeTemplateMethod* operation is to remove a single method from a single class, whereas the *removeHookMethod* operation will remove a method from an entire class hierarchy (e.g. it will remove the method from all classes in the hierarchy that implement it). Clearly, when a *removeTemplateMethod* operation is applied, it should thus not invoke a *removeHookMethod* operation on overlapping metapattern instances, as it is not the intent to remove all these method implementations.

The reverse can occur, however, a *removeHookMethod* operation can invoke a *removeTemplateMethod* operation on overlapping metapattern instances, as a method can be a *hookMethod* participant in one metapattern instance and a *templateMethod* participant in another, overlapping one.

$$removeHookMethod_a(m) \Rightarrow removeHookMethod_b(m) \text{ if } inherits_h^*(\mathcal{H}_{h,b}, \mathcal{H}_{h,a}), m \in \mathcal{H}_{h,b} :: \mathcal{M}_h$$

$$removeHookMethod_a(m) \Rightarrow removeTemplateMethod_b(m) \text{ if } inherits_h^*(\{T_b\}, \mathcal{H}_{h,a}), m \in T_b :: \mathcal{M}_t$$
$$removeTemplateMethod_a(c) \Rightarrow removeTemplateMethod_b(m) \text{ if } T_a = T_b, m \in T_b :: \mathcal{M}_t$$

Table 5.7: Transformations involving method participants

5.6 Summary

In this chapter, we defined a formal framework for the definition of metapatterns. This framework provides a definition of five fundamental metapatterns, in terms of their participants and associated constraints. together with metapattern-specific transformations that can be applied on their instances. Furthermore, we formally specified the conditions under which instances of those fundamental metapatterns overlap, and showed how this allowed us to propagate changes. Please note that in subsequent chapters, we will use the term "metapattern" whenever we refer to "fundamental metapattern".

Chapter 6

Implementing the Metapattern Model Using Declarative Meta Programming

In the previous chapter, we presented a formal model for metapatterns and defined metapattern-specific transformations that can be applied on their instances. We also specified the conditions under which metapattern instances overlap and showed how this influences the application of transformations. In this chapter, we will show how this theoretical model can be implemented and used in practice to support framework-based development.

6.1 Introduction

The previous chapter presented a theoretical model for metapatterns and definitions of appropriate transformations that can be applied on their instances. In order to prove that a significant part of the structure of a framework can indeed be documented in this model, and that the model exposes important information for the instantiation and evolution of the framework, we need to test it in a practical setting. In this chapter, we will discuss how this formal model can be integrated into a standard development environment and in this way enables us to provide support for framework-based development.

We will first introduce the declarative meta-programming environment we will use to implement the model, Then, we will illustrate how a framework developer can document the design pattern instances that he uses. We will show how this information can be used by our supporting environment to detect possible design drift when evolving the framework manually. Furthermore, the environment enables supported evolution, by means of predefined framework and design pattern transformations that use the available information to apply changes to the framework's implementation in a semi-automatic manner.

The next part of this chapter contains an elaborate discussion of the implementation of the supporting environment. It shows how we use a declarative meta-programming environment to automatically translate design pattern instance specifications into metapattern instance specifications, and vice versa, and to implement metapattern constraints and metapattern-specific transformations.

6.2 Approach

Developers are more familiar with design patterns then they are with metapatterns, as this is what they use to develop software and what is documented in the different catalogs [GHJV94, ABW98, Lea96, MM97, CS95, BMR+96]. Furthermore, specifying information in terms of design

patterns is more concise, since a single design pattern instance may be mapped onto multiple metapattern instances, as we will see later on. Therefore, we will allow a developer to document a frameworks design by documenting the design pattern instances the framework uses. However, a tool that supports framework-based development needs to use information about metapattern instances. This information can be automatically derived from the design pattern information, however, which is what our supporting environment will be able to achieve, as was discussed in Section 4.3, and will be shown in Section 6.5.1.

A similar reasoning holds for the transformations that our theoretical model defines: according to our formal model, these transformations are defined for metapattern instances, while a developer is more familiar with design pattern-specific transformations, or even framework-specific transformations. Again, such framework- and design pattern-specific transformations can be defined and used by the developers and will be translated into metapattern-specific transformations automatically by our supporting environment. How this is achieved is shown in Section 6.5.3.

Another important aspect of the approach we propose is that the developer should provide design information manually as opposed to automatic extraction of such information. Our main motivation for this choice is that such extraction appears to be very difficult [CMO97]. Since a design pattern does not impose a particular implementation, but merely suggests a template, there are many different ways to implement it. Design pattern instances whose implementation deviates slightly from the standard template implementation may thus not be recognized by extraction tools. Furthermore, many design patterns have a similar structure, and differ mostly in some semantic aspect. We already saw in Section 2.3.5 that the structure of the *State* and *Strategy* design pattern look very similar. A tool can thus not automatically infer which of the two design patterns the developer intended to use. This is however extremely important since the design pattern determines the transformations that are applicable. Additionally, exploring a framework and inspecting each class and each method in detail in order to detect some pattern is a very time-consuming process. It is much easier and faster to simply check whether a particular specification is correct, as a tool performing such a task already knows where it should look and what it is looking for, and does not need to consider various design pattern specific implementation details.

Since we rely on the developer to provide design pattern instance specifications, we should take into account that errors may occur in this specification. Using the constraints defined by metapatterns however, we can detect possible inconsistencies. We can assess whether the specification is correct with respect to the implementation, and vice versa, check whether the implementation adheres to the specification as well.

6.3 A Declarative Meta-Programming Environment

6.3.1 Introduction

Declarative meta programming (DMP) is a technique, developed at the Programming Technology Lab, that uses a declarative programming language at the meta level to reason about and manipulate programs in some underlying base language. The goal of this effort is to build state of the art software tools and environments that support the software development and maintenance process in all of its aspects.

The declarative nature of DMP allows us to specify all sorts of information about a framework in a simple concise and intuitive way. Furthermore, declarative programming focuses on what a program is supposed to do, rather on how it achieves this task. Building supporting tools thus becomes a lot easier.

DMP is a meta-programming approach, since the tools are at the meta level with respect to the source code they want to reason about or manipulate. The flexibility of a meta-programming approach allows us to build many different kinds of tools, that all belong to one of the following categories:

- verification of source code to some higher-level description (for example, conformance checking to coding conventions [MMW01], to design models [Wuy98] or to architectural descrip-

tions [Men00])

- extraction of information from source code (for example, visualization, software understanding, browsing, generation of higher-level models or documentation, measurements, quality control and so on)

- transformation of source code (for example, refactoring, translation, re-engineering, evolution [MT01], optimization [TDM99] and so on)

- generation of source code [DV98, Wuy01]

A concrete DMP environment has an explicit link to the source code, which greatly facilitates all these activities. Information can be extracted easily and is always up to date. Existing code can be transformed in a relatively straightforward way and new code can be added easily. This clearly provides a major advantage over approaches that work with an external code repository.

One of the major advantages of using a DMP approach is that it offers us a turing complete programming language. We need the full computational power of a real programming language to implement the metapattern-specific transformations and the associated change propagation algorithm. This algorithm recursively computes the transformations that have to be applied, based on how metapattern instances overlap and which transformations are applied on those instances.

Many different flavors of DMP exist, depending upon both the base language and the metalanguage that are used. Most research in the area of DMP has been conducted using a logic programming language (SOUL) at the meta level, and a standard object-oriented programming language (Smalltalk) at the base level. We will discuss this setup in more detail in the following sections.

In the next section, we will provide a short overview of some specific details of the SOUL programming language, such as its specific syntax. In the remainder of this chapter, we will provide various examples of how the SOUL logic language can be used to describe design pattern instances, define design pattern constraints, transform design pattern specifications into metapattern specifications and implement the framework-, design pattern- and metapattern-specific transformations.

For more information about SOUL, and an extensive discussion of its code generation features and meta-programming capabilities, we refer to [DV98] and [Wuy01].

6.3.2 SOUL

SOUL is a logic programming language that closely resembles Prolog [DEDC96]. It allows us to assert logic facts, that represent information that is always true in a particular domain, and logic rules, that are used to derive new facts from existing ones. SOUL's syntax differs from Prolog's in a number of ways:

- logic variables are preceded by a question mark (such as `?x`) and need not start with a capital letter as in Prolog. This is due to the fact that SOUL is used to reason about (Smalltalk) programs, where class names are allowed to start with a capital for example, and we want to avoid the confusion between such names and variable names.

- lists are delimited by less-than ("<") and greater-than (">") symbols, rather then square brackets as in Prolog. The reason is that SOUL uses square brackets to represent Smalltalk terms and clauses, which we will explain shortly.

Besides these small syntactic differences, SOUL has a number of distinguishing features not present in a typical Prolog implementation, most notably the *quoted code terms*, *Smalltalk terms* and *Smalltalk clauses*.

Quoted code terms

Quoted code terms are special logic terms delimited by curly braces ("{" and "}"). They are mainly used to specify source code patterns, that can be parameterized by logic variables. These variables are substituted for a string representation of their actual values before interpretation.

As an example, consider the following logic program:

```
addNewClass(CondHandler,SpecialFormHandlerWithSuccessor).

generateAddClassCode({ ?super subclass: #?className }) if
   addNewClass(?className,?super),
   class(?super)
```

If we fire the query `if generateAddClassCode(?x)`, the variable `?super` will be bound to the symbol `SpecialFormHandlerWithSuccessor`, and the `?className` variable will be bound to the `CondHandler` symbol. These values are substituted for their corresponding variables in the quoted code term, and the variable `?x` in the original query will thus be bound to the string-like structure `SpecialFormHandlerWithSuccessor subclass: #CondHandler`. We can now execute this code from within SOUL to actually generate the `CondHandler` class as a subclass of class `SpecialFormHandlerWithSuccessor` in the implementation.

Smalltalk terms

Another important construct provided by SOUL is the *Smalltalk term*. A Smalltalk term is a logic term that can contain any Smalltalk expression. Such expressions are delimited by square brackets ("[" and "]") and are evaluated each time the logic interpreter unifies them with another term. As an example, consider the following logic program:

```
test([Array]).
if test(?x)
```

When launching the query, the variable `?x` is unified with the smalltalk term `[Array]`. As a result, the Smalltalk interpreter will be called to evaluate the expression *Array*, which results in the class object `Array` being returned [1]. As can be seen a Smalltalk term is a logic term containing a Smalltalk expression that is evaluated in Smalltalk upon unification, and returns the result to the SOUL environment. This result, a smalltalk object, is wrapped so that it can be used inside the SOUL environment. Smalltalk terms thus allow to refer to Smalltalk objects in the logic programming environment.

Smalltalk clauses

Smalltalk clauses are based on exactly the same idea as Smalltalk terms: they are delimited by square brackets and can contain an arbitrary Smalltalk expression, that can be parameterized by logic variables. The only difference is that a Smalltalk clause is expected to be used as a predicate. Hence, it is evaluated whenever it is encountered and is expected to return a boolean value. Smalltalk clauses enable the SOUL environment to access the underlying Smalltalk environment, and perform whatever computation at that level.

As an example, consider the following logic program:

```
write(?string) if [ Transcript show: ?string. true ].
if write(['Hello World'])
```

Launching the query will print "Hello World" on the transcript. Note how the Smalltalk clause `[ Transcript show: ?string. true ]` is parameterized by the logic variable `?string`, that will get bound during unification, and that it returns true as a value. `['Hello World']` is another example of a Smalltalk term whose value is computed when it is unified with the `?string` variable.

[1] Note that classes are first-class entities in Smalltalk, so they can be treated as any other object.

6.4 Support for Framework-based Development

6.4.1 Specification of Design Pattern Instances

Design pattern instances are described by using the technique of role modeling. Each design pattern defines a number of roles, that identify the responsibilities of the classes, methods and variables that make up the design pattern instance. These classes, methods and variables are the participants that implement the desired behavior for a specific design pattern instance. By using role modeling, all participants of a design pattern instance are mapped onto the roles defined by the corresponding design pattern.

The mapping of roles onto participants is specified in terms of two logic predicates: the *patternInstance/2* and *role/3* predicates [2]. The first predicate is used to identify an instance of a particular design pattern, while the second predicate is used to specify the actual mapping of roles onto participants.

In what follows, we will show how a developer should/can document three design patterns that are used in the Scheme framework of Chapter 3. The approach is of course general enough to cover any design pattern instance that should be documented. For each of these three design patterns, we will first identify the roles they define. Then, we will determine the source code entities that participate in the design pattern instances, and show how these are mapped onto the roles of the design pattern.

Specification of the *Composite* Design Pattern

The *Composite* design pattern has the following roles:

component The root class of the hierarchy that defines the *leaf* and *composite* participant classes of the design pattern. It defines the interface for all objects in the composition.

composite The class that represents composite objects in the composition. The *composite* participant should always be a (possibly indirect) subclass of the *component* participant.

leaf The classes that represent leaf objects in the composition. These classes are concrete subclasses of the *component* participant as well.

compositeMethod The methods that constitute the interface for objects in the composition. These methods are defined by the *component* participant and are overridden (if necessary) by the *leaf* and the *composite* participants. In the former case, they implement behavior for the primitive objects in the composition, while in the latter case they implement behavior for components having children.

compositeVariable The instance variable(s) defined in the *composite* participant that holds references to all objects in a composition.

A specification of a *Composite* design pattern instance should include all these roles and should specify which classes, methods and variables correspond to which roles. As a concrete example, consider the *CompositeExpression* design pattern instance occurring in the Scheme framework (as discussed in Section 3.4.4 and depicted in Figure 3.8). Figure 6.1 graphically depicts the mapping.

The *patternInstance/2* predicate is used in the following way to specify that *CompositeExpression* is an instance of the *Composite* design pattern:

```
patternInstance(CompositeExpression,compositeDP).
```

[2] Note that it is standard practice to append a number to the name of logic predicates indicating the number of arguments of the predicate. This is used to distinguish predicates with the same name but a different number of arguments.

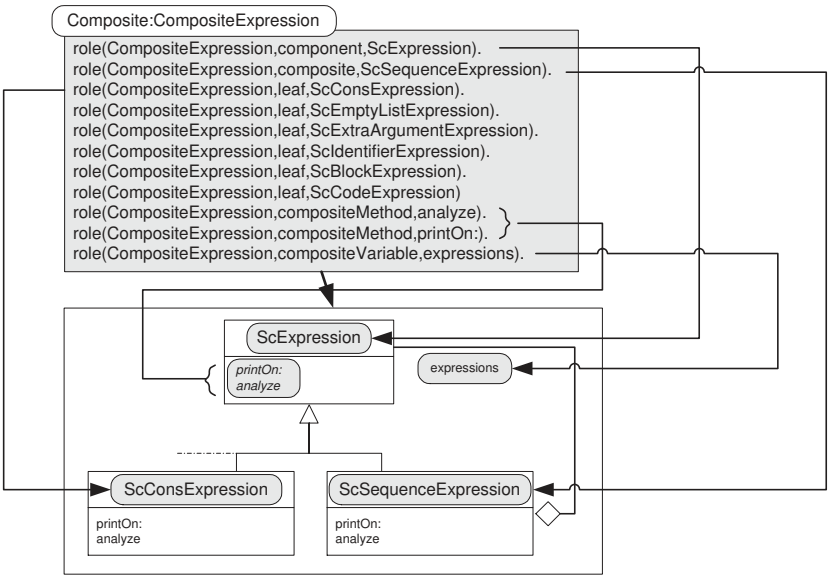


Figure 6.1: Mapping the roles of the *Composite* design pattern onto the participants of the *CompositeExpression* instance

The first argument of this predicate indicates the name of the design pattern instance. This name should be unique amongst all other design pattern instances occurring in the framework, as it is used as a key for identifying and assembling all of the instance's participants. The second argument denotes the design pattern of which this is an instance.

The *role/3* predicate maps the various roles of the *Composite* design pattern to the classes, methods and variables that implement the design pattern instance. The `ScExpression` class, for example, is mapped onto the *component* role and the `ScSequenceExpression` class to the *composite* role. The *role/3* predicate uses the unique name assigned to the design pattern instance to indicate which instance the role belongs to. By means of this name, the complete description of the design pattern can be assembled.

Specification of the *Factory Method* Design Pattern

Just as we did for the *Composite* design pattern, we first describe the five roles present in the *Factory Method* design pattern:

abstractCreator This class corresponds to the root of a class hierarchy, in which all *concreteClass* participants are concrete leaf classes. It implements the *factoryMethod* participant as an abstract method.

concreteCreator The *concreteCreator* role is played by all concrete classes in the class hierarchy defined by the *abstractCreator* participant. All *concreteCreator* participants provide a concrete implementation for the *factoryMethod* participant.

abstractProduct This class is the root of the hierarchy that contains the classes that are instantiated by the *factoryMethod* method.

concreteProduct The *concreteProduct* role is played by the concrete classes of the hierarchy defined by the *abstractProduct* participant. *concreteProduct* classes are instantiated by the *factoryMethod* methods defined in the *concreteCreator* classes.

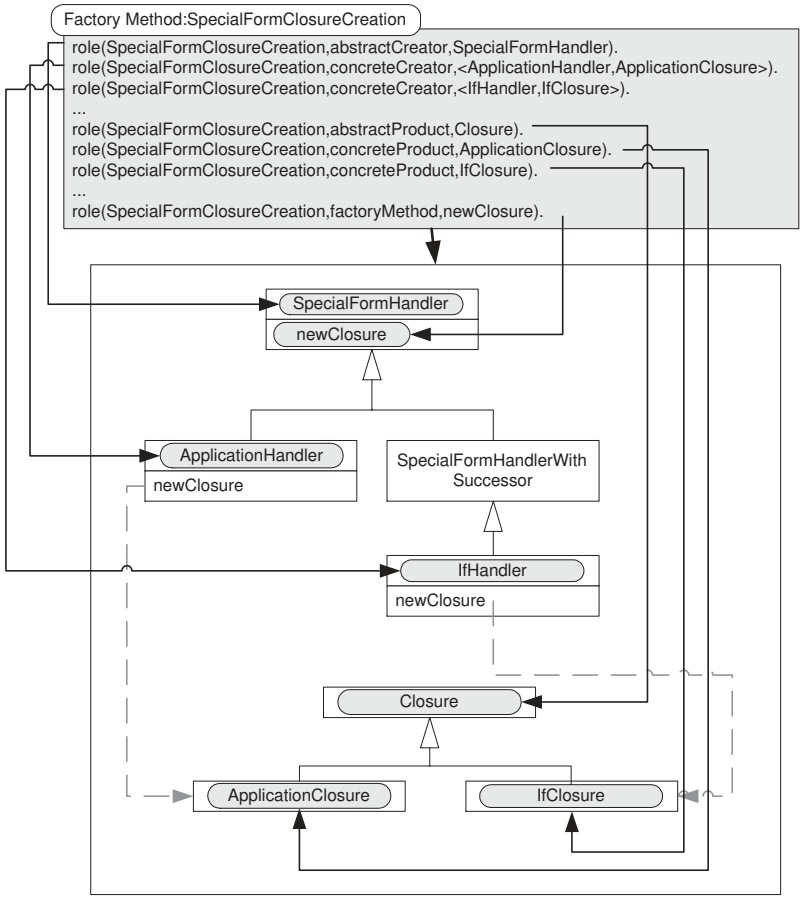


Figure 6.2: Mapping the roles of the *Factory Method* design pattern onto the participants of the *SpecialFormClosureCreation* instance

factoryMethod The *factoryMethod* participant is the method that instantiates and returns a *concreteProduct* object. The method is defined by the *abstractCreator* participant and is provided with subclass-specific behavior in all *concreteCreator* participants, if necessary.

Several instances of the *Factory Method* design pattern were identified in the Scheme framework (Section 3.4.3). The specification of the *SpecialFormClosureCreation* instance is depicted graphically in Figure 6.2 and the instance is specified as follows:

```
patternInstance(SpecialFormClosureCreation,factoryMethodDP).
```

Again, the name defined by the *patternInstance/2* predicate is unique amongst all other design pattern instances and is used by the *role* facts. The *role/3* predicate is used to map the five roles of the *Factory Method* design pattern to the particular implementation artifacts that constitute the design pattern instance. The `SpecialFormHandler` class is the root of a hierarchy, and is thus mapped onto the *abstractCreator* role. All concrete classes of this hierarchy are mapped onto *concreteCreator* roles, and the `newClosure` method is mapped onto the *factoryMethod* role.

One particular important aspect of this specification that deserves special attention is the mapping for the *concreteCreator* role. It differs from the rest of the specification in that it not only maps the *concreteCreator* role onto a particular class, but that it also includes the *concreteProduct* class that it creates. For example, the following fact:

```
role(SpecialFormClosureCreation,concreteCreator,<IfHandler,IfClosure>)
```

states that the `IfHandler` *concreteCreator* participant instantiates and returns the `IfClosure` *concreteProduct* participant. The reason why we specifically include this information in the specification is that *factoryMethod* participants are defined in all *concreteCreator* participants, and each instantiate a different *concreteProduct* participant. This is useful information that should be included in the specification.

Specification of the *Abstract Factory* Design Pattern

As a last example, we will show how an instance of the *Abstract Factory* design pattern can be described in our environment. This design pattern has four roles:

abstractFactory The *abstractFactory* participant is the root of a class hierarchy and defines an interface for creating a family of related objects.

concreteFactory The *concreteFactory* participants are concrete subclasses of the *abstractFactory* participant and provide a concrete implementation for the interface defined by the latter.

abstractFactoryMethod The *abstractFactoryMethod* participants are all the methods that are defined by the *abstractFactory* participant and that make up the interface for creating product objects.

abstractProduct The *abstractProduct* participants are the root classes of the class hierarchies that contain the *concreteProduct* participants. An *abstractProduct* participant is associated with each *abstractFactoryMethod* participant.

concreteProduct The *concreteProduct* participants are all classes that are instantiated by *concreteFactory* participants.

Only one instance of the *Abstract Factory* design pattern is present in the Scheme framework: the *ASTFactory* instance (see Section 3.4.1 for a description and Figure 3.2 for the structure of the instance). The specification of this instance can be found in Figure 6.3. The name of the instance is defined by using the *patternInstance/2* predicate as follows:

```
patternInstance(ASTFactory,abstractFactoryDP).
```

The `SchemeASTFactory` is mapped onto the *abstractFactory* role, the `DefaultSchemeASTFactory` to the *concreteFactory* role and seven *abstractFactoryMethod* participants are defined. Note how the specification of an *abstractProduct* role also includes the *abstractFactoryMethod* participant that is associated with it. This denotes the fact that the `newBlockExpression:` *abstractFactoryMethod* participant should instantiate a product from the `ScBlockExpression` hierarchy, for example.

Furthermore, the specification for a *concreteProduct* participant also includes the *concreteFactory* and *abstractFactoryMethod* participants that are associated with it. For example, the first of these facts states that the `newBlockExpression:` method in the `DefaultSchemeASTFactory` class creates an instance of the `ScBlockExpression` class.

Observe also that in this specification the classes in the `ScExpression` hierarchy play the role of *abstractProduct* and *concreteProduct* participant at the same time. When a new *concreteFactory* participant is added, the set of *concreteProduct* participants will be extended and will contain other classes as well. Moreover, the `ScExpression` class itself forms no part of the specification. The *Abstract Factory* design pattern does not require all *abstractProduct* participants to share a common superclass. Since it is the case here, this instance can be considered as a variable of the *Abstract Factory* design pattern.

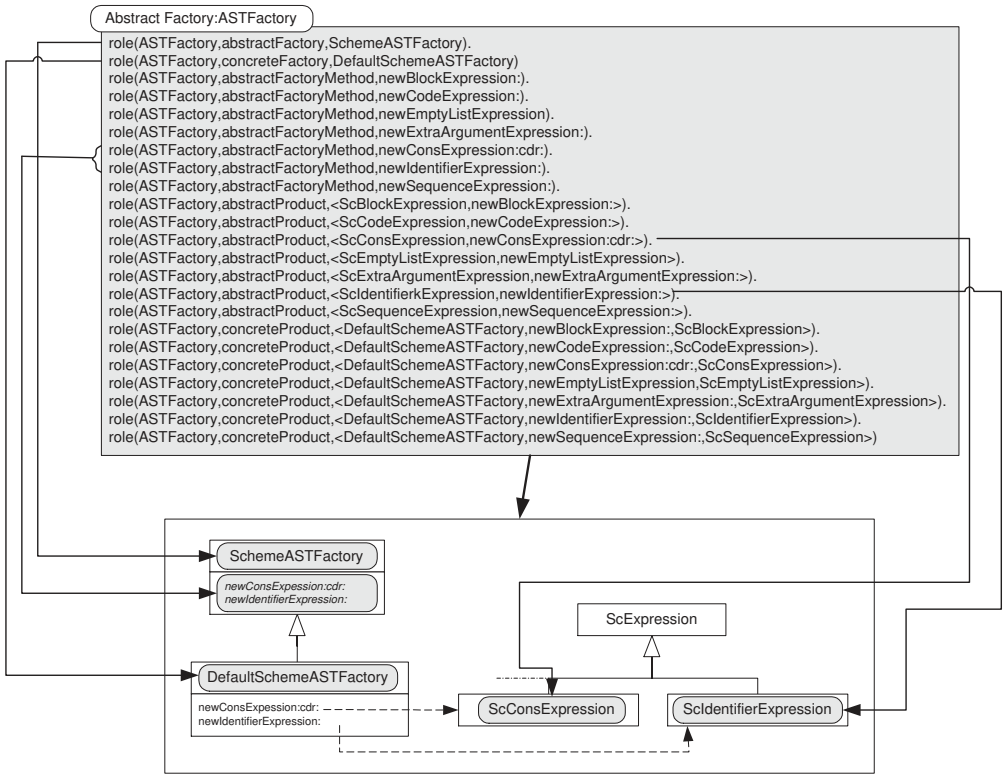


Figure 6.3: Mapping the roles of the *Abstract Factory* design pattern onto the participants of the *ASTFactory* instance

6.4.2 An Example of Manual Evolution

The evolution example presented in Chapter 3 was the addition of a *cond* special form to the Scheme framework. When performing this evolution by hand, a developer can accidently introduce a number of errors, due to a lack of adequate documentation and insufficient knowledge about the specific design of the framework.

As was explained in Chapter 3, a new `CondHandler` class, which is responsible for handling *cond* expressions, should be added to the `SpecialFormHandlerWithSuccessor` hierarchy. Figure 6.4 shows the different design pattern instance specifications in which the `SpecialFormHandler` hierarchy, and thus also the `CondHandler` class, participates. Now suppose that a developer correctly adds the `CondHandler` class as a subclass of the `SpecialFormHandlerWithSuccessor` class. Since this class participates in a number of design pattern instances, it should be registered as a new participant and it should implement the necessary methods in order not to violate the constraints of those design patterns and guarantee the correct behavior of the new special form.

If the developer is not aware that the particular design patterns of Figure 6.4 are used, he may forget to implement some of these methods. Furthermore, because some of the methods, most notably the `newConverter` and `newClosure` methods, are responsible for instantiating specific classes, additional classes may need to be added to other class hierarchies as well. If the developer does not apply these changes, Scheme interpreters built using the framework will produce runtime errors if a *cond* special form is used, since the implementation of the special form is not correct.

Such errors in the evolution can be detected by verifying whether the constraints of the design patterns that are affected by that particular change are still satisfied after the change. More specifically, if a developer adds a particular class to a hierarchy, our supporting environment should check the constraints of each design pattern in which this hierarchy participates. The constraints

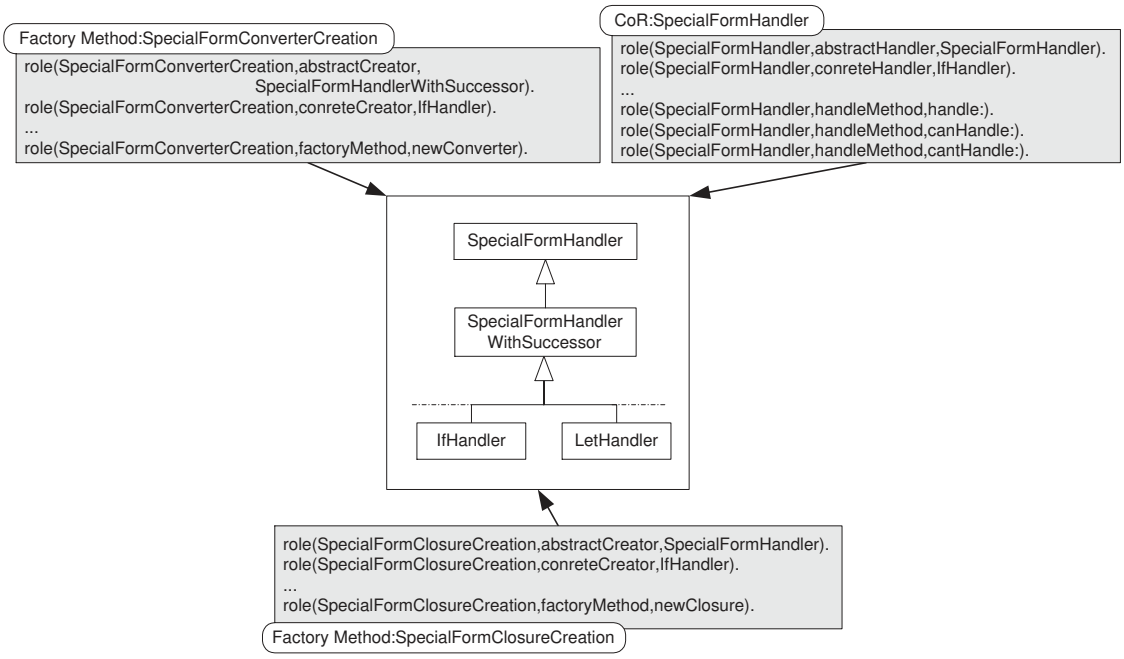


Figure 6.4: Design pattern instances in which the `SpecialFormHandler` hierarchy participates

of the design pattern are those that are associated with the corresponding metapattern(s). The implementation of these constraints will be discussed in a later section. In this case, the constraints of the *Chain of Responsibility*, the *Template Method* and the *Factory Method* design patterns, which correspond to the constraints of the *Recursion, Unification* and the *Creation* metapatterns, should be checked.

First of all, our environment checks whether the specification of the involved design pattern instances is still correct with respect to the implementation, If the developer correctly added the new classes to the framework, but did not register them as participants in the design pattern instances, the environment will detect this and notify him. For example, it will detect that the `CondHandler` class should be registered as a *concreteHandler* participant in the *SpecialFormHandler* design pattern instance, and as a *concreteCreator* in the *SpecialFormClosureCreation* and *SpecialFormConverterCreation* instances of the *Factory Method* design pattern. Similarly, it will detect that the `CondClosure` class should be registered as a *leaf* and *concreteElement* participant in the *CompositeClosure* and *ClosureVisitor* design pattern instances (see Figure 6.5) , and that the `CondConverter` class should be registered as a *concreteClass* participant in the *ConverterTemplateMethod* design pattern instance (Figure 6.6).

Second, once the classes have been registered as participants in the appropriate design pattern instances, the environment consults the instance's specifications to identify the methods these classes should implement. Then, it checks the implementation to verify whether the newly added classes effectively implement these methods. For example, if the developer did not provide the `CondHandler` class with an implementation for the `handle:`, `canHandle:` and `cantHandle:` method participants of the *SpecialFormHandler* design pattern instance, the environment notifies him and he should then implement them. For the `newClosure` and `newConverter` method participants, the environment is even able to check whether their implementation satisfies the constraints of the *Factory Method* design pattern: the first method should return an instance of a subclass of the `Closure` class, while the second should return an instance of a subclass of the `SchemeConverter` class. If these methods do not instantiate and return such an object, the environment reports this to the developer who can then take appropriate action.

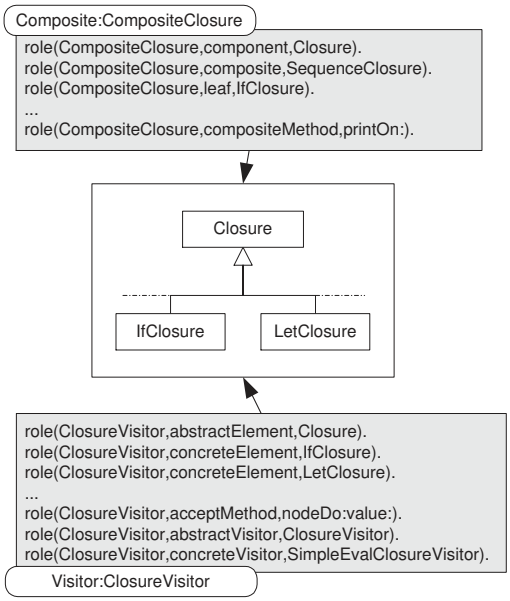


Figure 6.5: Design pattern instances in which the `Closure` hierarchy participates

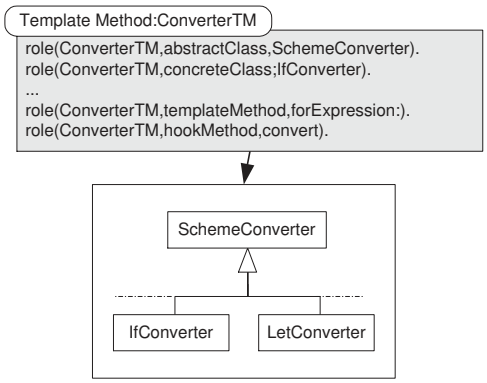


Figure 6.6: Design pattern instances in which the `SchemeConverter` hierarchy participates

A developer should continue making changes to the framework as long as the environment reports possible conflicts.

This example clearly illustrates that information about the design pattern instances used in a framework is valuable for a developer that evolves that framework. First of all, the mere fact that such documentation is present, provides a developer with important information about the design of the framework. Since he is aware of the different design patterns that are used in the framework, he will presumably introduce less errors when manually evolving it. Second, the documentation is used in an active way by the supporting environment to verify whether the implementation satisfies the constraints of the design patterns. If this is not the case, the developer is notified of the observed inconsistencies and should then take appropriate action.

It is important to note that there are two possible sources of inconsistencies: the environment either detects that the specification of a design pattern instance is not correct (for example, a class is not registered as a participant, but should be) or that the implementation does not conform to the specification (for example, a class does not define the required methods). Obviously, it is important to detect the latter kind of inconsistencies and report them to the developer, as they can lead to runtime errors. However, it is equally as important to detect the former kind of inconsistencies. Since our approach for supporting framework-based development is based upon documentation of design patterns, we should make sure that their specification is correct. If this is not the case, our environment will not be able to detect possible constraint violations. As we saw in the example, the environment is able to check whether a class implements the required methods, only if this class is registered as the appropriate participant in the corresponding design pattern instance. Moreover, supported evolution is also based on documentation of design patterns, which thus also requires correctness of the specification.

6.4.3 An Example of Supported Evolution

Instead of manually evolving a framework, a developer can also use the high-level transformations to achieve a more disciplined form of evolution. The transformations are defined so as to guide the developer when implementing the necessary changes, in order to guarantee that no constraints are violated. They thereby generate a skeleton implementation that can be finished further by the developer. Furthermore, the transformations are implemented in such a way that they automatically update the design pattern instance specification to include the new participants. In what follows, we will again consider the example of adding a *cond* special form to the Scheme framework and show how our supporting environment can guide a developer while performing this task.

The *addSpecialForm* transformation is defined so that it first asks the developer for a handler class that should be able to handle the new special form. The developer is presented with a dialog in which he can specify the name of that class [3]:

Name of concrete handler for the *cond* special form = `CondHandler`

The `CondHandler` class is now added automatically to the implementation as a subclass of the `SpecialFormHandlerWithSuccessor` class. Then, the *addSpecialForm* transformation consults the description of all design patterns in which this newly added class plays a role and continues its operation. It derives that the new class is a *concreteHandler* participant in a *Chain of Responsibility* design pattern instance and that it should thus implement all *handleMethod* participants of this instance (see Figure 6.4). It registers this class as such a participant in the design pattern instance specification and presents the developer with dialogs in which he has to specify an implementation for the `handle:`, `cantHandle:` and `canHandle:` methods:

Implementation for the `handle:` method in the `CondHandler` class = ...
Implementation for the `cantHandle:` method in the `CondHandler` class = ...

[3]Note that we provide a graphical user interface for our supported evolution environment, and that a developer will normally use this interface for evolving the framework. Furthermore, the interface is closely integrated with the standard development environment in order to provide appropriate browsing facilities on existing code. For the sake of discussion, we present these dialogs in a textual form here.

Implementation for the `canHandle:` method in the `CondHandler` class = ...

These methods are then added to the `CondHandler` class automatically.

The environment detects that the `CondHandler` class also participates in the *SpecialFormClosureCreation* instance of the *Factory Method* design pattern as a *concreteCreator* participant. This is due to the fact that the *SpecialFormHandler* design pattern instance overlaps with the *SpecialFormClosureCreation* instance, according to the definition of overlapping given in Section 5.5. The `CondHandler` class is thus registered as a *concreteCreator* participant in that design pattern instance. Each such participant should implement the *factoryMethod* participant, in this case, a `newClosure` method. This method should instantiate an object from a particular class, so the environment prompts the developer which class the `CondHandler` class should instantiate:

Name of concrete `Closure` class instantiated by the `CondHandler` class = `CondClosure`

Suppose that the developer specifies that the concrete closure class is `CondClosure`. Since this class does not yet exist in the framework, it is automatically added. The environment adds it as a subclass of the `Closure` class, since the specification of the *SpecialFormClosureCreation* design pattern instance specifies that this class is an *abstractProduct* participant of which all *concreteCreator* classes should inherit. Furthermore, the `CondHandler` class is automatically provided with a skeleton implementation for the `newClosure` method, that instantiates a `CondClosure` object. This implementation look as follows:

```
CondHandler>>newClosure
    ^CondClosure new
```

Likewise, the environment also derives that the `CondHandler` class participates as a *concreteCreator* participant in the *SpecialFormConverterCreation* instance of the *Factory Method* design pattern, and it thus registers this class a such a participant in the design pattern instance specification. This time, the class should implement a `newConverter` method which should instantiate a class from the `SchemeConverter` hierarchy. The environment thus again consults the developer:

Name of concrete `Converter` class instantiated by the `CondHandler` class = `CondConverter`

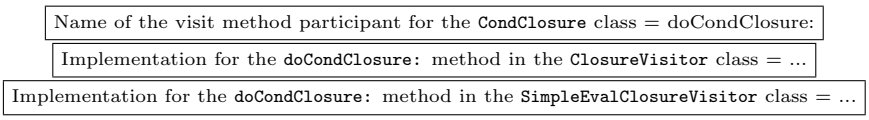
The environment also adds the `CondConverter` class to the framework automatically, and it derives from the documentation that it should be a subclass of the `SchemeConverter` class. Moreover, the `newConverter` method is defined in the `CondHandler` class that instantiates an object of the `CondConverter` class:

```
CondHandler>>newConverter
    ^CondConverter new
```

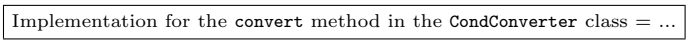
Since the *addSpecialForm* added two new classes to the framework, the environment should again check whether these classes participate in a design pattern instance. As it turns out, the `CondClosure` class participates in the *CompositeClosure* instance of the *Composite* design pattern as a *leaf* participant (see Figure 6.5). It should thus provide an implementation for all *compositeMethod* participants. In this case, there are two such participants, and the environment prompts the developer to provide a concrete implementation for them:

Implementation for the `printOn:` method in the `CondClosure` class = ...
Implementation for the `nodeDo:value:` method in the `CondClosure` class = ...

Furthermore, it also plays the role of *concreteElement* participant in the *ClosureVisitor* instance of the *Visitor* design pattern. With each such participant, a corresponding *visitMethod* participant is associated. Since the `CondClosure` class is a new class, the environment asks the developer for the name of the method that should be introduced in the `ClosureVisitor` hierarchy, and then asks him to provide an implementation for this method for all concrete visitor classes:



The `CondConverter` class participates in the *ConverterTM* instance of the *Template Method* design pattern (see Figure 6.6). It plays the role of a *concreteClass*, which should implement all *hookMethod* participants. In this instance, there is only one such method, the `convert` method, for which the environment asks an implementation:



As no more new classes are added, the *addSpecialForm* transformation finishes its execution. At this point, the implementation of the framework does not contain any constraint violation conflicts, since the transformation made sure that all design pattern instances are implemented and specified correctly. It should be noted that the implementation of the new special form is not yet finished. While the most important classes and methods are defined and provided with the appropriate behavior, it may be necessary to further finish the implementation. For example, when being asked to provide an implementation for the `handle:` method participant, a developer may have specified an implementation that calls additional helper methods. If these methods do not yet exist in the framework, they are not added automatically by our environment. The developer should thus add them manually. Note that this is not a restriction of our approach, however. The environment could check whether all necessary methods are already present in the framework, and if not, continue prompting the developer to provide the appropriate implementation for them.

6.5 Implementation

The previous section demonstrated the support for framework-based development provided by our environment. In this section, we will discuss how such support can be implemented. In Chapter 4, Figure 4.8, we already explained the different steps that are involved: implementation of metapattern specific transformations (Section 6.5.5), translation of framework-specific transformations into design pattern-specific transformations, who in turn are translated into metapattern-specific transformations (Section 6.5.3) and translation of design pattern instance specifications into metapattern instance specifications (Section 6.5.1), and vice versa (Section 6.5.4).

6.5.1 Translation of Design Pattern Instances to Metapattern Instances

In this section, we will explain how our declarative meta-programming environment automatically translates a design pattern instance specification into the corresponding metapattern instance specification(s). Such a mapping can be defined for any design pattern-metapattern combination, but due to space restrictions, we only consider the three example design pattern instances presented in Section 6.4.1.

Translation of the *Composite* design pattern

The *Composite* design pattern is an instance of the *Recursion* metapattern, defined in Section 5.3.3. In order to translate the design pattern description into a metapattern description, the roles of the *Composite* design pattern need to be mapped onto the roles of the *Recursion* metapattern. In this particular case, the mapping is quite straightforward: each design pattern role is mapped onto a corresponding metapattern role, as can be seen in Figure 6.7. In this figure, design pattern and metapattern roles are represented by rounded rectangles, while participants are represented by means of small circles.

We should note that in the formal model of Chapter 5, metapatterns were defined in terms of hierarchy participants, such as *hookHierarchy* and *templateHierarchy* participants. The specification of a metapattern instance should be more precise, and therefore models such participants by

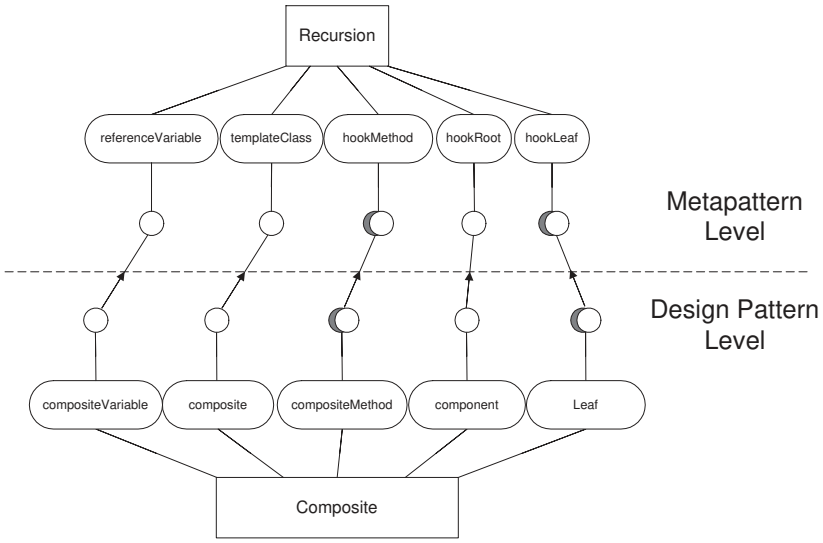


Figure 6.7: Mapping of roles from the *Composite* design pattern onto roles of the *Recursion* metapattern.

means of the appropriate *root* and *leaf* participants. A *hookHierarchy* participant of a *Recursion* metapattern, for example, is specified by means of *hookRoot* and *hookLeaf* participants. The former represents the root of the *hookHierarchy* participant, while the latter represents all its concrete leaf classes. Note that we could have chosen to simply specify the *root* participant of a hierarchy, and let our environment automatically derive the *leaf* participants (e.g. all concrete subclasses of the *root* participant). However, the complete specification of a metapattern instance would then have to be calculated every time it is needed, and this could be a very time-consuming process. Moreover, as we saw in Section 6.4.2, the environment detects and notifies a developer when he accidently forgets to register a class as a particular participant in a design pattern instance. This makes the developer aware of the fact that the class participates in a design pattern instance. If participants are calculated automatically, this additional benefit would disappear.

Similar to the description of a design pattern instance (Section 6.4.1), a metapattern instance's description is specified in terms of the *mpPatternInstance/2* and *mpRole/3* predicates. The translation is implemented by using logic rules that convert *patternInstance* and *role* facts into appropriate *mpPatternInstance* and *mpRole* facts. These logic rules are implemented once for each design pattern-metapattern combination, and are provided in a library with our supporting environment. The translation is thus performed completely automatically without any developer intervention.

For example, to define an instance of the *Recursion* metapattern that corresponds to a *Composite* design pattern instance, we can use the following rule:

```
mpPatternInstance(?instance,recursionMP) if
   patternInstance(?instance,compositeDP).
```

which states that each instance of the *Composite* design pattern defines an instance of the *Recursion* metapattern. As can be seen, we simply use the unique name of the design pattern instance to identify the metapattern instance. This is allowed because this name is unique amongst all metapattern instances. When a design pattern instance is mapped onto multiple metapattern instances, this simple approach can not be applied, since the name of all these metapattern instances would then not be unique. We will show how we overcome this problem when illustrating how our environment translates instances of the *Abstract Factory* design pattern.

Likewise, we can easily translate information specified by the *role/3* predicate into the appropriate *mpRole/3* facts:

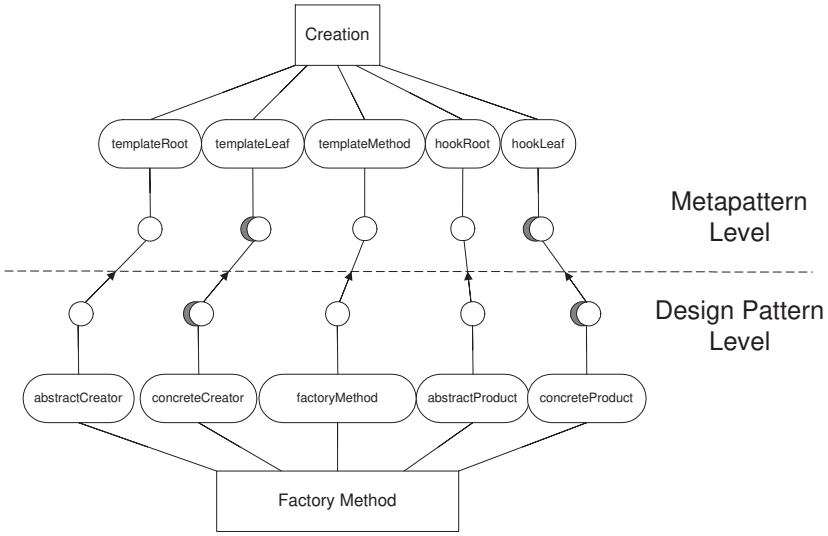


Figure 6.8: Mapping of roles from the *Factory Method* design pattern onto roles of the *Creation* metapattern.

```
mpRole(?instance,?role,?participant) if
   patternInstance(?instance,compositeDP),
   mpRoleCompositeDP(?instance,?role,?participant).

mpRoleCompositeDP(?instance,templateClass,?class) if
   role(?instance,composite,?class).

mpRoleCompositeDP(?instance,hookRoot,?class) if
   role(?instance,component,?class).

mpRoleCompositeDP(?instance,hookLeaf,?class) if
   role(?instance,leaf,?class).

mpRoleCompositeDP(?instance,hookMethod,?selector) if
   role(?instance,compositeMethod,?selector).

mpRoleCompositeDP(?instance,templateMethod,?selector) if
   role(?instance,compositeMethod,?selector).

mpRoleCompositeDP(?instance,referenceVariable,?variable) if
   role(?instance,compositeVariable,?variable)
```

As can be observed, an instance of the *Composite* design pattern is translated in a straightforward way into an instance of the *Recursion* metapattern. The *mpRoleCompositeDP* rules map the *composite* design pattern role onto a *templateClass* metapattern role, for example, while the *leaf* design pattern role is mapped onto the *hookLeaf* metapattern role.

Translation of the *Factory Method* design pattern

The *Factory Method* design pattern can be mapped in a straightforward way onto a *Creation* metapattern. This mapping takes the form as depicted in Figure 6.8.

An instance of the *Factory Method* design pattern automatically defines an instance of the *Creation* metapattern, as follows:

```
mpPatternInstance(?instance,creationMP) if
   patternInstance(?instance,factoryMethodDP).
```

Once again, we simply share the name between the design pattern instance and the metapattern instance, since it remains unique for all metapattern instances.

The mapping of design pattern roles to metapattern roles is implemented by the following rules, which translate the *role/3* predicate into the *mpRole/3* predicate:

```
mpRole(?instance,?role,?participant) if
   patternInstance(?instance,factoryMethodDP),
   mpRoleFactoryMethodDP(?instance,?role,?participant).

mpRoleFactoryMethodDP(?instance,hookLeaf,?class) if
   role(?instance,concreteProduct,?class)

mpRoleFactoryMethodDP(?instance,hookRoot,?class) if
   role(?instance,abstractProduct,?class)

mpRoleFactoryMethodDP(?instance,templateRoot,?class) if
   role(?instance,abstractCreator,?class)

mpRoleFactoryMethodDP(?instance,templateLeaf,?class) if
   role(?instance,concreteCreator,?class)

mpRoleFactoryMethodDP(?instance,templateMethod,?selector) if
   role(?instance,factoryMethod,?selector).

mpRole(?instance,creates,?tclass,?hclass) if
   patternInstance(?instance,factoryMethodDP),
   role(?instance,concreteCreator,<?tclass,?hclass>),
```

The *concreteProduct* design pattern role is mapped onto a *hookLeaf* metapattern role, for example. Observe that in the description of the metapattern instance, extra information is included by means of a *mpRole/4* predicate. In a description of a *Factory Method* design pattern instance, the *concreteCreator* role includes extra information about the particular *concreteProduct* participant that is instantiated by the concrete creator class. For example, a specification of the *SpecialFormHandlerClosureCreation* instance includes the following fact, to denote that a `IfHandler` *concreteCreator* participant instantiates a `IfClosure` *concreteProduct* participant (see Section 6.4.1):

```
role(SpecialFormClosureCreation,concreteCreator,<IfHandler,IfClosure>)
```

The specification of the metapattern instance, corresponding to the *SpecialFormClosureCreation* instance, would use the *mpRole/4* predicate to ensure that such important information does not get lost during the translation:

```
mpRole(SpecialFormClosureCreation,templateLeaf,IfHandler).
mpRole(SpecialFormClosureCreation,creates,IfHandler,IfClosure)
```

One could argue that we could have equally well used the *mpRole/3* predicate in the same way as the *role/3* predicate to specify the *concreteCreator* role in the metapattern description, as follows:

```
mpRole(SpecialFormClosureCreation,templateLeaf,<IfHandler,IfClosure>)
```

The specific reason we decided to use multiple predicates, however, is to preserve consistency between the *mpRole/3* predicates. First of all, consistency greatly facilitates reasoning about a specification, as we don't need to take into account different forms of one and the same predicate. Second, different design patterns have different roles, and can use the *role/3* predicate in any which way they want to describe these roles accurately. Two different design patterns can be mapped onto one and the same metapattern however. In order to ensure that this can be done in a consistent way, we should agree on a standard use of the *mpRole/3* predicate, that is used in a metapattern instance description.

For example, the *composite* participant of an instance of the *Composite* design pattern is specified as follows:

```
role(CompositeExpression,composite,ScSequenceExpression)
```

and is mapped onto the *templateLeaf* participant of the *Recursion* metapattern:

```
mpRole(CompositeExpression,templateLeaf,ScSequenceExpression)
```

Similarly, a *concreteCreator* participant in an instance of the *Factory Method* design pattern is specified as follows:

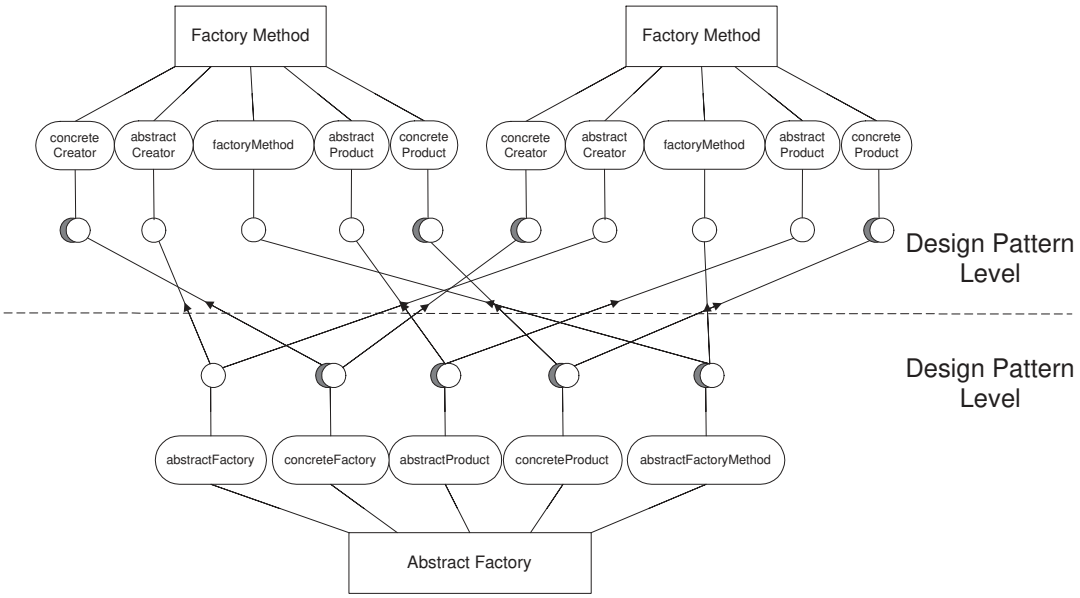


Figure 6.9: Mapping of roles from the *Abstract Factory* design pattern onto roles of the *Factory Method* design pattern

```
role(ExpressionClosureCreation,concreteCreator,<ScSequenceExpression,SequenceClosure>)
```

and would then be mapped onto a *templateLeaf* participant of an instance of the *Creation* metapattern as follows:

```
mpRole(ExpressionClosureCreation,templateLeaf,<ScSequenceExpression,SequenceClosure>)
```

As can be seen, the specification of a *templateLeaf* participant has a different format depending on the kind of metapattern that we are dealing with. When reasoning about such specifications, we are thus obliged to take such differences into account, which would make the reasoning process much more difficult. When a transformation adds a method participant, for example, it should gather all class participants to which an implementation of this method should be added. If the specification of these class participants differs depending on the metapattern, we should implement the transformation for each particular metapattern, so that it takes these differences into account. If we use a standard format, however, we can simplify the implementation of the transformations. Therefore, we will make use of extra predicates whenever this is required to preserve important information.

Translation of the *Abstract Factory* design pattern

While the *Composite* and *Factory Method* design patterns could be mapped easily onto appropriate metapatterns, this is not the case for the *Abstract Factory* design pattern. There is no immediately suitable metapattern candidate that has roles corresponding to those of the design pattern. We can however see an instance of the *Abstract Factory* design pattern as a collection of instances of the *Factory Method* design pattern. Instances of the latter design pattern can be mapped easily on instances of the *Creation* metapattern. As shown in Figure 6.9, an instance of the *Abstract Factory* design pattern is thus mapped onto several instances of the *Factory Method* design pattern.

Conceptually, the *ASTFactory* design pattern instance as described in Section 6.4.1 can be translated into a number of *Factory Method* design pattern instances as follows:

```
patternInstance(ASTFactory1,factoryMethodDP).
role(ASTFactory1,abstractProduct,ScConsExpression).
```

```
role(ASTFactory1,concreteProduct,ScConsExpression).
role(ASTFactory1,abstractCreator,SchemeASTFactory).
role(ASTFactory1,concreteCreator,DefaultSchemeASTFactory).
role(ASTFactory1,factoryMethod,newConsExpression:cdr:).

patternInstance(ASTFactory2, factoryMethodDP).
role(ASTFactory2,abstractProduct,ScCodeExpression).
role(ASTFactory2,concreteProduct,ScCodeExpression).
role(ASTFactory2,abstractCreator,SchemeASTFactory).
role(ASTFactory2,concreteCreator,DefaultSchemeASTFactory).
role(ASTFactory2,factoryMethod,newCodeExpression:).

..
```

As can be seen, a new instance of the *Factory Method* design pattern is defined for each *abstractFactoryMethod* role in a *Abstract Factory* design pattern instance. Each of these design pattern instances has a unique name (*ASTFactory1, ASTFactory2*, ...), which is generated automatically. Additionally, the mapping of *Abstract Factory* design patterns roles to *Factory Method* design pattern roles is done in the following way (as can be seen in Figure 6.9):

- The *abstractFactory* role of the *Abstract Factory* design pattern instance is mapped onto the *abstractCreator* role in each *Factory Method* design pattern instance.
- Similarly, the *concreteFactory* role of the *Abstract Factory* design pattern instance is mapped onto the *concreteCreator* role in each *Factory Method* metapattern instance.
- Each *abstractFactoryMethod* participant in the *Abstract Factory* design pattern instance is mapped onto a *factoryMethod* participant in the appropriate *Factory Method* metapattern instance.
- Each *abstractProduct* participant in the *Abstract Factory* design pattern instance is mapped to an *abstractProduct* participant in the appropriate *Factory Method* metapattern instance.
- Each *concreteProduct* participant in the *Abstract Factory* design pattern instance is mapped to a *concreteProduct* participant in the appropriate *Factory Method* metapattern instance.

The result is that each *Factory Method* design pattern instance contains the same *abstractCreator* and *concreteCreator* participants, but contains different *factoryMethod, abstractProduct* and *concreteProduct* participants.

The logic rules that implement this translation are actually very similar to those implementing a design pattern instance to metapattern instance translation. First of all, a number of *Factory Method* design pattern instances are defined based on the definition of an *Abstract Factory* design pattern instance as follows:

```
patternInstance(?newInstance,factoryMethodDP) if
   patternInstance(?instance,abstractFactoryDP),
   generateNewInstance(?instance,?newInstance)
```

The *generateNewInstance/2* predicate is responsible for generating a unique name for each *Factory Method* instance, based on the name of the *Abstract Factory* instance. The algorithm we use for generating this new name makes sure that the original name of the design pattern instance can easily be recovered. The *Abstract Factory* design pattern specification can then be translated into the appropriate *Factory Method* design pattern specifications by means of the following rules:

```
role(?newInstance,abstractProduct,?class) if
  patternInstance(?instance,abstractFactoryDP),
  role(?instance,abstractProduct,<?class,?selector>),
  generateNewInstance(?instance,?selector,?newInstance).

role(?newInstance,abstractCreator,?class) if
   patternInstance(?instance,abstractFactoryDP),
   role(?instance,abstractFactory,?class),
   generateNewInstance(?instance,?newInstance)

role(?newInstance,concreteCreator,?class) if
```

```
   patternInstance(?instance,abstractFactoryDP),
   role(?instance,concreteFactory,?class),
   generateNewInstance(?instance,?newInstance).

role(?newInstance,factoryMethod,?selector) if
   patternInstance(?instance,abstractFactoryDP),
   role(?instance,abstractFactoryMethod,?selector),
   generateNewInstance(?instance,?selector,?newInstance).

role(?newInstance,concreteProduct,<?factory,?class>) if
  patternInstance(?instance,abstractFactoryDP),
  role(?instance,concreteProduct,<?factory,?selector,?class>),
  generateNewInstance(?instance,?selector,?newInstance).
```

Note that the specification of a *concreteProduct* participant in a *Factory Method* design pattern only contains the *concreteCreator* participant that is responsible for creating it, and not the specific method that actually creates it, as is the case for instances of the *Abstract Factory* design pattern. Instances of the *Factory Method* design pattern contain only one *factoryMethod* participant, as opposed to instances of the *Abstract Factory* design pattern. This sole participant is already included in the *Factory Method* design pattern specification via the *factoryMethod* role, so it can safely be omitted from the *concreteProduct* specification.

6.5.2 Defining Metapattern Constraints

By using the declarative meta-programming environment we can define the constraints associated with metapatterns as logic rules. These check whether the implementation of the framework conforms to the specification of the metapattern instances, or vice versa, check that the specification is correct with respect to the implementation. This allows us to verify whether a developer specified a particular design pattern correctly, and also provides support for (anticipated as well as unanticipated) evolution by checking whether the appropriate constraints still hold after a number of changes have been applied.

In this section, we provide several examples of how constraints defined by metapatterns are specified in the declarative meta-programming environment. We first define some general constraints, that hold for most of the metapatterns. Later on, we will also define constraints that are specific to particular metapatterns. To do so, we use the three examples design patterns (or their corresponding metapatterns) presented earlier.

General Constraints

Verifying correctness of a template/hook hierarchy specification Each metapattern has at least one hierarchy participant: the *hookHierarchy* participant. Some metapatterns, most notably the *Creation* and *Hierarchy* metapatterns, even have an additional *templateHierarchy* participant. In a metapattern instance specification, such hierarchies are described in terms of the *hookRoot, hookLeaf, templateRoot* and *templateLeaf* roles respectively as we saw earlier.

In order to verify whether a metapattern instance description conforms to the actual source code, we need to check whether the hierarchy participants are specified correctly. In concreto, this means that the classes registered as leaf participants should exist in the implementation and should be subclasses of the appropriate root participant. If this is not the case, we regard them as incorrectly specified leaf participants. A constraint that checks whether the *leaf* participants of a particular metapattern instance are indeed subclasses of the *root* participant looks as follows, for example:

```
leafButNoSubclass(?instance,?rootRole,?leafRole, ?violators) if
[1] mpRole(?instance,?rootRole,?root),
[2] findall(incorrectLeafParticipant(?leaf,?root),
[3]         and(mpRole(?instance,?leafRole,?leaf),
[4]             not(inheritsStar(?leaf,?root))),
            ?violators).
```

This rule first fetches the class that is registered as the root participant of the class hierarchy (line 1). Then, it gathers all classes that are registered as leaf participants but do no inherit

(maybe indirectly) from the root participant (line 3 and 4). Such classes are included in a list, held by the variable `?violators` and are reported to be incorrect leaf participants (line 2). The *inheritsStar/2* predicate actually implements the *inherits** relation, defined in Chapter 5. It makes use of Smalltalk clauses to consult the implementation of the framework and to verify whether the leaf class participant is actually a descendant of the root class participant.

A correct specification of a class hierarchy not only means that all leaf participants are effectively subclasses of the root participant, but also requires that each concrete subclass of the root participant is registered as a leaf participant. This constraint can be checked by the following rule:

```
concreteSubclassButNoLeaf(?instance,?rootRole,?leafRole,?violators) if
[1] mpRole(?instance,?rootRole,?root),
[2] findall(shouldBeLeafParticipant(?cClass),
[3]         and(concreteSubclass(?root,?cClass),
[4]             candidateLeaf(?instance,?cClass),
[5]             not(mpRole(?instance,?leafRole,?cClass))),
          ?violators)
```

It first fetches the root participant of the metapattern instance (line 1), and then gathers all concrete subclasses of this participant that are candidate leaf classes but not registered as such (lines 3, 4 and 5). A class that conforms to these conditions is reported to be a class that should be registered as a leaf participant in the metapattern instance (line 2).

The *candidateLeaf/2* predicate is used to check whether a certain class is a candidate for being a leaf participant or not. Normally, all concrete classes in a class hierarchy are candidate leaf participants, except when we are dealing with a *Recursion* metapattern instance. In that case, the class that is registered as the *templateClass* participant is a concrete class in the hierarchy, but may not be registered as a leaf participant.

We can use the two constraints defined above to define a predicate *correctHierarchy/2* that checks whether a specific hierarchy is correctly specified as follows:

```
correctHierarchy(?instance,?rootRole,?leafRole,?violators) if
   leafButNoSubclass(?instance,?rootRole,?leafRole,?v1),
   concreteSubclassButNoLeaf(?instance,?rootRole,?leafRole,?v2),
   append(?v1,?v2,?violators).
```

Verifying correctness of method participant implementors. Another example of a generic constraint that must hold for several metapatterns is the following. A number of metapatterns define method participants that should be implemented throughout a particular hierarchy participant. For example, all *hookMethod* participants of a *Unification* metapattern instance should be implemented by the *hookHierarchy* participant. According to the definition of the metapatterns, this means that a method participant should be defined in the root of that hierarchy, and that each leaf participant should understand that method. This constraint is expressed by the following rules:

```
notDefinedByRoot(?instance,?rootRole,?methodRole,?violators) if
[1] mpRole(?instance,?rootRole,?root),
    findall(notDefinedByRoot(?root,?selector),
[2]         and(mpRole(?instance,?methodRole,?selector),
[3]             not(definesMethod(?root,?selector))),
[4]         ?violators).

notUnderstoodByLeaf(?instance,?leafRole,?methodRole,?violators) if
   findall(?v1,
           and(mpRole(?instance,?leafRole,?leaf),
               findall(notUnderstoodByLeaf(?leaf,?selector),
                       and(mpRole(?instance,?methodRole,?selector),
                           not(understandsMessage(?leaf,?selector))),
                       ?v1)),
           ?v2),
   flatten(?v2, ?violators)
```

The *notDefinedByRoot/4* rule fetches the root class of the class hierarchy participant in a metapattern instance (line 1), and checks whether all *method* participants are defined by that root class (line 2 and 3). Those method participants that violate this rule are reported and maintained in a list (line 4). The *notUnderstoodByLeaf/4* rule does something similar and checks whether all

leaf participants can understand all required method participants. Those leaf participants that do not understand particular method participants are again registered in a list. Note that the *definesMethod/2* and *understandsMessage/2* predicates actually implement the $definesMethod$ and $understandsMessage$ relations defined in Chapter 5. These predicates are defined in terms of Smalltalk clauses that consult the implementation to check whether a class actually defines a certain method or understands a particular message.

The general version of this constraint can thus be implemented by the *correctMethods/5* predicate as follows:

```
correctMethods(?instance,?rootRole,?leafRole,?methodRole,?violators) if
   notDefinedByRoot(?instance,?rootRole,?methodRole,?v1),
   notUnderstoodByLeaf(?instance,?leafRole,?methodRole,?v2),
   append(?v1,?v2,?violators).
```

In the following sections, we will show how the above predicates are used in conjunction with other constraints to specify the constraints of some particular metapatterns.

Constraints for the *Recursion* metapattern

The *Recursion* metapattern (Section 6.5.1) has only one hierarchy participant: the *hookHierarchy* participant. The constraints of this metapattern specify that each *hookMethod* participant should be implemented across this hierarchy. Furthermore, a constraint that is specific to this metapattern is that the *templateClass* participant should be present in the *hookHierarchy* participant. The general constraint that should hold for the *Recursion* metapattern is specified by the following rule:

```
patternConstraint(?instance,?violators) if
   mpPatternInstance(?instance,recursionMP),
   correctHierarchy(?instance,hookRoot,hookLeaf,?v1),
   correctMethods(?instance,hookRoot,hookLeaf,hookMethod,?v2),
   templateClassInHookHierarchy(?instance,?v3),
   templateClassDefinesTemplateMethods(?instance,?v4),
   recursiveTemplateMethods(?instance,?v5),
   append(?v1,?v2,?v3,?v4,?v5,?violators)
```

The *correctHierarchy/4* predicate is passed the *hookRoot* and *hookLeaf* specific roles as arguments, in order to specify that the *hookHierarchy* participant should be checked. Similarly, the *hookRoot, hookLeaf* en *hookMethod* roles are passed as arguments to the *correctMethod/5* predicate to check whether the *hookMethod* participants are implemented correctly in the *hookHierarchy* participant. The *templateClassInHookHierarchy/2* predicate is responsible for checking that the *templateClass* participant actually is a class in the *hookHierarchy* participant, while the *templateClassDefinesTemplateMethods/2* and the *recursiveTemplateMethods/2* predicates are used to verify whether the *templateClass* participant defines all *templateMethod* participants and whether the *templateMethod* participants call themselves recursively.

Constraints for the *Creation* metapattern

A specification of the *Abstract Factory* design pattern is translated into a specification of several *Factory Method* design pattern instances, which are in turn translated into *Creation* metapattern instances. The *Creation* metapattern has two hierarchy participants: a *hookHierarchy* participant and a *templateHierarchy* participant. As such, it should be checked whether the specification of both these hierarchies is correct, which is achieved by calling the *correctHierarchy/4* predicate twice: once with arguments *hookRoot* and *hookLeaf* roles, and once with arguments *templateRoot* and *templateLeaf* roles. Furthermore, the *templateMethod* participant of a *Creation* metapattern instance should be implemented in the appropriate way across the *templateHierarchy* participant. This is achieved by calling the *correctMethods/5* with the appropriate arguments. The constraint specification for the *Creation* metapattern thus looks as follows:

```
patternConstraint(?instance, ?violators) if
   mpPatternInstance(?instance,creationMP),
```

incorrectLeafParticipant	a leaf participant is not a subclass of the root participant
shouldBeLeafParticipant	a subclass of the root participant is not registered as a leaf participant
templateClassPtcptNotInHookHierarchy	a *templateClass* participant is not part of the *hookHierarchy* participant
notUnderstoodByLeaf	a leaf participant does not understand some method participant
notDefinedByRoot	a root participant does not define some method participant
noCallToHookMethod	a *templateMethod* participant does not call a *hookMethod* participant
noClassCreation	a *templateMethod* participant does not instantiate a class
noHookClassCreation	a *templateMethod* participant does not instantiate a *hookLeaf* participant
notImplementedByTemplateClass	a *templateClass* participant does not define a *templateMethod* participant
nonRecursiveTemplateMethod	a *templateMethod* participant does not call itself recursively

Table 6.1: Constraint Violation Conflicts

```
correctHierarchy(?instance,hookRoot,hookLeaf,?v1),
correctHierarchy(?instance,templateRoot,templateLeaf,?v2),
correctMethods(?instance,templateRoot,templateLeaf,templateMethod,?v3),
templateMethodsInstantiateHookClassParticipants(?instance,?v4),
append(?v1,?v2,?v3,?v4,?violators)
```

As it name suggests, the *templateMethodsInstantiateHookClassParticipants/2* predicate is used to verify whether each implementation of the *templateMethod* participant instantiates a class that is registered as a *hookLeaf* participant in the same metapattern instance.

Discussion

In the above sections, we only defined the constraints for the *Recursion* and *Creation* metapatterns. Naturally, the constraints for the *Unification, Connection* and *Hierarchy* metapatterns can be defined in a similar way. Table 6.1 contains a list of the constraint violation conflicts that can be detected, by means ot the constraints of all metapatterns. Some of these conflicts can occur for any kind of metapattern (the *notUnderstoodByLeaf* conflict, for example), while other conflicts are specific to a particular metapattern (the *noHookClassCreation* conflict can only occur for the *Creation* metapattern). Moreover, the metapattern-specific constraints can be used as a basis for defining design pattern-specific constraints as well, much like the general constraints are used to define metapattern-specific ones. These design pattern-specific constraints can include more restrictions on the method bodies, for example.

6.5.3 Translation of Framework-Specific Transformations to Metapattern-Specific Transformations

As was discussed in Section 4.3 and depicted in Figure 4.8, the high-level transformations that are defined for the Scheme framework, such as the *addSpecialForm* transformations, should be translated automatically into lower-level transformations in order to be able to apply them. Therefore, framework-specific transformations are automatically translated into design pattern-specific transformations first. In turn, these design pattern-specific transformations are further translated into

more abstract metapattern-specific transformations that perform the actual changes. In the following sections, we will explain how a framework developer can specify how the *addSpecialForm* framework-specific transformation is translated into an appropriate metapattern-specific transformation, which will add the necessary classes and methods to the framework and update the documentation.

Framework-Specific Transformations

The *addSpecialForm* transformation is defined specifically for the Scheme framework and is responsible for making changes to the framework that add a new special form. A developer invokes this transformation via the user interface of the development environment. The environment will prompt him to specify the name of the special form that he wants to add and will assert the following fact into the logic repository:

```
operation(SchemeFramework,addSpecialForm,cond)
```

The *operation/3* predicate is used to represent information about the transformations that are applied to the implementation. The first argument identifies the framework to which the transformation is applied, the second argument identifies the name of the transformation, while the third argument includes its arguments. As we will see in Chapter 7, explicitly logging the transformations in this way enables us to provide support for software merging.

Design Pattern-Specific Transformations

Framework-specific transformations are translated into design pattern-specific transformations that operate on design pattern instances occurring in the framework.

Recall from Chapter 3 that special forms in the Scheme framework are represented by classes in the `SpecialFormHandler` hierarchy. Adding a new special form thus requires extending this hierarchy with a new class. The *addSpecialForm* framework-specific transformation should thus be translated into a design pattern-specific transformation that adds a new class to this hierarchy. As can be observed from Figure 6.4, the `SpecialFormHandler` hierarchy participates in a number of design pattern instances. The *addSpecialForm* framework-specific transformation can thus be translated into a design pattern-specific transformation of any of the corresponding design patterns. In this particular case, the framework developer that implemented the *addSpecialForm* transformation opted to translate it into an *addConcreteHandler* design pattern transformation on the *Chain of Responsibility* design pattern:

```
operation(SpecialFormHandler,addConcreteHandler,<?class,SpecialFormHandlerWithSuccessor>) if
   operation(SchemeFramework,addSpecialForm,?name),
   askUser('Name of concrete handler for the ?name special form', ?class)
```

This rule translates the *addSpecialForm* framework-specific transformation into an *addConcreteHandler* design pattern-specific transformation on the *SpecialFormHandler* instance. Once again, the *operation/3* predicate is used to explicitly log the fact that this transformation is being applied, and which contains the information necessary to perform the transformation. In this case, the *askUser/2* predicate is used to consult the developer and ask him to specify a name for the class that will represent the *cond* special form, and the class should be added as a subclass of the `SpecialFormHandlerWithSuccessor`.

Other framework-specific transformations, such as the *addNewExpressionType* transformation, can be translated into the appropriate design pattern-specific transformations in a similar way. Note that it is not necessarily always the case that one framework-specific transformation is translated into one design pattern-specific transformation. It is perfectly well possible that two or more design pattern-specific transformations are required. This can be easily achieved, however, by simply specifying two or more logic rules, as appropriate.

Metapattern-Specific Transformations

Design pattern-specific transformations in their turn are translated into metapattern-specific transformations that actually make the necessary changes to the implementation. For example, the *Chain of Responsibility* design pattern is an instance of the *Unification* metapattern. As such, the *addConcreteHandler* transformation that is defined for the *Chain of Responsibility* design pattern is defined in terms of the *addHookClass* transformation on the *Unification* metapattern as follows:

```
mpOperation(?instance,addHookClass,<?class,?superclass>) if
   patternInstance(?instance,chainOfResponsibilityDP),
   operation(?instance,addConcreteHandler,<?class,?superclass>)
```

Just like for the translation of framework-specific transformations to design pattern-specific transformations, it is not necessarily the case that a design pattern-specific transformation is translated into a single metapattern-specific transformation. Furthermore, observe that the above rule does not explicitly mention the *Unification* metapattern. However, the *patternInstance/3* predicate is mapped onto a *mpPatternInstance/3* predicate, which includes this information (see Section 6.5.1), and thus it is ensured that the *addConcreteHandler* transformation on the *Chain of Responsibility* design pattern is mapped onto an *addHookClass* transformation on the *Unification* metapattern..

Note that in the formal model of Chapter 5, the *addHookClass* transformation only has one argument, a class `C`. In practice, we should also specify the class from which this class should be derived. To this extent, the *addHookClass* transformation above has an extra argument that specifies that superclass.

6.5.4 Translation of Metapattern Instances to Design Pattern Instances

The transformations of metapattern instances are implemented in such a way that they automatically update the metapattern instance specification to include the new participants or remove obsolete ones. After the transformations have been performed, this updated specification needs to be translated back into a corresponding design pattern instance specification. Since design pattern instance participants in most cases are not mapped exactly onto metapattern instance participants, this translation is not as straightforward as it may seem. The purpose of this section is thus to show how this reverse mapping can be implemented. We will do this by means of the three example design pattern instances that we already used earlier in this chapter. Once again, other mappings can be specified in a similar way.

Translation of the *Composite* design pattern

As shown in Section 6.5.1, each participant of the *Composite* design pattern can be mapped onto a similar participant of the *Recursion* metapattern. Therefore, the reverse mapping is also straightforward, as is depicted in Figure 6.10, which represents the situation where a *removeHookMethod* and an *addHookClass* transformation have been performed. In the figure, the original participants of the metapattern instance that are not influenced by the evolution are represented by shaded circles, the participant that is crossed is the participant that is removed by the *removeHookMethod* transformation, while the participant in black is added by the *addHookClass* transformation.

This mapping is implemented by the following logic rules, which map *mpRole* facts onto *role* facts:

```
role(?instance,leaf,?class) if
   patternInstance(?instance, compositeDP),
   addedRole(?instance,hookLeaf,?class),
   not(removedRole(?instance,hookLeaf,?class)).

role(?instance,compositeMethod,?selector) if
   patternInstance(?instance, compositeDP),
   addedRole(?instance,hookMethod,?selector),
   not(removedRole(?instance,hookMethod,?selector))
```

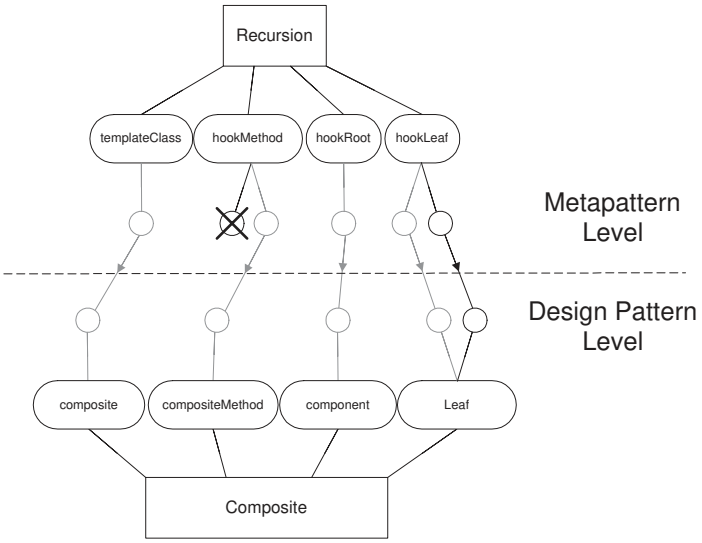


Figure 6.10: Mapping an instance of the *Recursion* metapattern onto an instance of the *Composite* design pattern.

These rules make use of the *addedRole/3* and *removedRole/3* predicates to construct an updated *Composite* design pattern specification. As we will see in Section 6.5.5, these predicates are asserted by the transformations to specify that a particular participant has been added or removed.

Note that, as was the case for the rules implementing the design pattern instance to metapattern instance mapping, these rules only need to be specified once and are provided as a library with our supporting environment.

Translation of the *Factory Method* design pattern

Figure 6.11 shows an instance of the *Creation* metapattern upon which an *addTemplateClass* and a corresponding *addHookClass* transformation has been applied to, and shows how this instance is mapped onto an instance of the *Factory Method* design pattern. What is not shown in this picture is how this mapping ensures that a specification of the *concreteCreator* role contains all necessary information. The rules that implement this mapping reveal this in more detail:

```
role(?instance,concreteCreator,<?creator,?product>) if
   addedRole(?instance,templateLeaf,?creator),
   addedCreates(?instance,?creator,?product),
   patternInstance(?instance, factoryMethodDP).

role(?instance, concreteProduct, ?product) if
   addedRole(?instance,hookLeaf,?product),
   patternInstance(?instance, factoryMethodDP)
```

Like was the case for the mapping of the *Recursion* metapattern to the *Composite* design pattern, these rules make use of the *addedRole/3* and *removedRole/3* predicates to determine which participants were added and/or removed. Furthermore, the first rule searches the logic database for *addedCreates/3* facts to make sure the specification of a *concreteCreator* participant includes the *concreteProduct* participant it creates.

Translation of the *Abstract Factory* design pattern

An instance of the *Abstract Factory* is mapped onto multiple instances of the *Factory Method* design pattern, which are then translated into appropriate *Creation* metapattern instances, as was shown in Section 6.5.1.

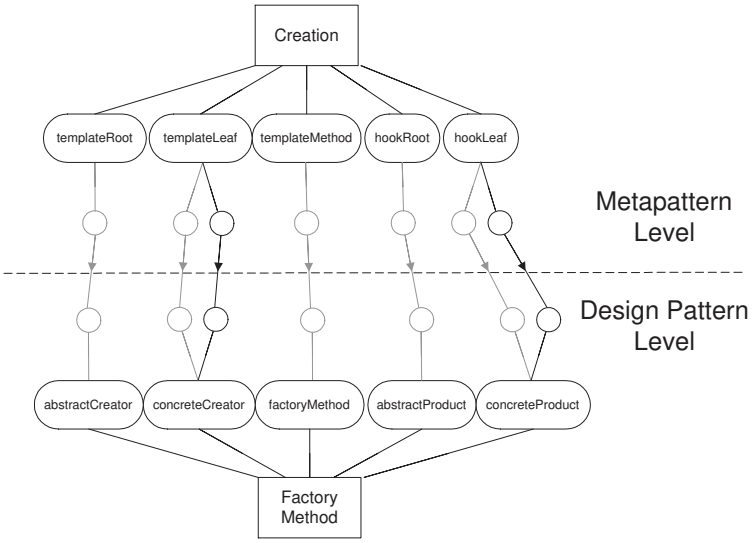


Figure 6.11: Mapping of a *Creation* metapattern instance to a *Factory Method* design pattern instance

Figure 6.12 shows the reverse mapping after an *addAbstractFactoryMethod* transformation has been applied. This transformation actually requires introducing a new instance of the *Factory Method* design pattern, as illustrated by the figure.

Recreating an updated instance of the *Abstract Factory* design pattern from all the *Factory Method* instances is implemented as follows:

```
role(?instance,concreteProduct,<?factory,?selector,?class>) if
   role(?fmInstance,concreteProduct,?class),
   role(?fmInstance,concreteCreator,?factory),
   role(?fmInstance,factoryMethod,?selector),
   patternInstance(?fmInstance, factoryMethodDP)
   extractOldInstance(?fmInstance,?instance),
   patternInstance(?instance, abstractFactoryDP).

role(?instance, abstractFactoryMethod, ?selector) if
   role(?fmInstance,factoryMethod,?selector),
   patternInstance(?fmInstance,factoryMethodDP),
   extractOldInstance(?fmInstance,?instance),
   patternInstance(?instance, abstractFactoryDP)
```

Introducing a new *abstractFactoryMethod* participant requires adding an abstract method to the *abstractFactory* participant and concrete methods to the *concreteFactory* participants. With each concrete factory class and factory method, a *concreteProduct* participant is associated, since a factory method in a concrete factory class instantiates an object of this concrete product. The specification of the *concreteProduct* role includes all this information, which can be gathered by looking for appropriate *concreteCreator* and *factoryMethod* facts that are included in the specification of a *Factory Method* design pattern.

Other transformations that are defined for the *Abstract Factory* design pattern, such as the *addConcreteFactory* or *removeFactoryMethod* transformations, are handled in a similar way.

6.5.5 Implementation of Metapattern-Specific Transformations

In this section, we will explain how the transformations on metapatterns are actually implemented in the declarative meta-programming environment and how they perform their task. This task consists of adding the appropriate participants to the implementation of the metapattern instance, updating the specification of the instance with the new information, checking if additional transformations on related metapatterns are necessary and performing those transformations if so. In

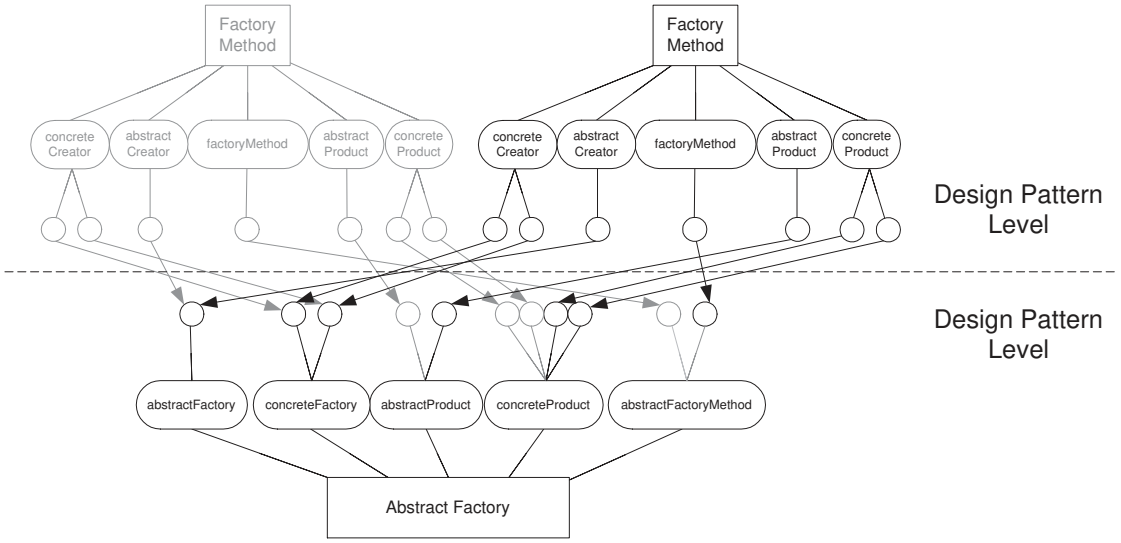


Figure 6.12: Adding an *abstractFactoryMethod* participant to the *Abstract Factory* design pattern

this section, we will only present the implementation of an *addHookClass* transformation. This transformation is applicable on all metapatterns defined in Chapter 5, and thanks to the fact that each metapattern instance specification has a similar form (as was discussed in Section 6.5.1), we only need to provide a single implementation, which is applicable for all kinds of metapatterns. Other metapattern-specific transformations, such as *addTemplateClass* or *addHookMethod* are implemented in a similar way, although these sometimes need to be tuned for the specific kind of metapattern upon which they are applied.

As was already explained, the *addHookClass* transformation has only one argument in our formal model: the class `C` that should be added as a *hookLeaf* participant. In practice, we also need to specify a superclass for this class `C`. Therefore, the *addHookClass* transformation requires two arguments: a class C and a class S from which C will be derived.

The transformation is implemented in three different phases

1. In the first phase, a new class participant is added to the implementation as a subclass of the specified superclass, and the metapattern instance specification is updated to include the new *hookLeaf* participant.

2. In the second phase, the new class participant is updated so that it implements all necessary methods. This actually consists of two steps.

   (a) First, the metapattern instance specification is consulted and a concrete method is added to the class for each *hookMethod* participant of the instance.

   (b) Second, all overlapping metapattern instances in which the newly added class will participate are determined. This is achieved by checking the enabling conditions for overlapping, as defined in Chapter 5. Then, the specification of the overlapping metapatterns is consulted and a concrete method is added to the class for each method participant of the overlapping metapattern instance that the class should implement.

3. In the third phase, additional transformations are performed on related metapattern instances. Such additional transformations may be necessary because new classes and methods have been added in the previous phases, and this may require other transformations so as to complete the implementation and make the metapattern instance conform with its constraints. In the evolution example presented in Chapter 3, additional transformations

were necessary to add a `printOn:` method on a `CondClosure` class and a `convert` method to a `CondConverter` class, for example.

These different phases are each implemented by a number of logic rules that depend on one another. In what follows, we will explain the implementation of these logic rules in more detail.

The *evolve/3* predicate

The *evolve/3* predicate is used to generate code for a specific transformation, such as *addHookClass*. It is responsible for implementing the three phases mentioned above:

```
evolve(?instance,addHookClass,{ ?addClassCode ?updateClassCode ?evolvePatternsCode }) if
[1] mpOperation(?instance,addHookClass,<?leaf,?super>),
[2] addClassParticipant(?instance,?leaf,?super,?addClassCode),
[3] evolveHookClass(?instance,?leaf,?updateClassCode),
[4] evolveOverlappingPatterns(?instance,?evolvePatternsCode),
[5] assert(addedRole(?instance,hookLeaf,?leaf))
```

On line 3, the *addClassParticipant/4* is called, which will generate code that adds the class held by the `?leaf` variable as a subclass of the class held by the `?super` variable.

For example, in Figure 6.13, the specification of the *SpecialFormHandler* metapattern instance is consulted (in particular, the facts that are in italics are used), so that the new `CondHandler` class can be added to the implementation as a subclass of the `SpecialFormHandlerWithSuccessor` class and can be registered as a new participant in the metapattern description (this is denoted by bold facts in the figure).

The predicates *evolveHookClass/3* and *evolveOverlappingPatterns/3* used at lines 4 and 5 are used to implement phases 2 and 3 of the transformation. The implementation of these predicates is explained in detail in the following sections. Line 6 adds an *addedRole/3* fact to the logic repository, to assert that a new leaf class has been added to the specified design pattern instance. This is necessary to be able to implement the mapping from metapattern to design pattern instances, as discussed in Section 6.5.4.

Recall that SOUL provides quoted code terms, delimited by { and }, that can be used to hold code. In the implementation of the transformations, we used these terms to generate Smalltalk expressions, that will later on be evaluated and that add the appropriate code to the implementation of the framework. For example, if we execute the following query:

```
if evolve(SpecialFormHandler,addHookClass,?code)
```

the `?code` variable will contain a chunk of Smalltalk expressions, that look as follows:

```
{ SpecialFormHandlerWithSuccessor
    subclass: #CondHandler
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Scheme-Handlers'.
(Smalltalk at: #CondHandler) compile: 'handle: anObject ... ' classified: 'generated methods'.
(Smalltalk at: #CondHandler) compile: 'cantHandle: anObject ...' classified: 'generated methods'.
...
Closure
    subclass: #CondClosure
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Scheme-Closures'.
(Smalltalk at: #CondClosure) compile: 'printOn: anObject ... ' classified: 'generated methods'.
... }
```

When these Smalltalk expressions are executed, the `CondHandler` and `CondClosure` classes will be added to the appropriate class hierarchies, and the appropriate methods will be defined in them. The developer can then fine-tune this code as necessary, by adding instance or class variables, or moving methods to their appropriate categories, for example.

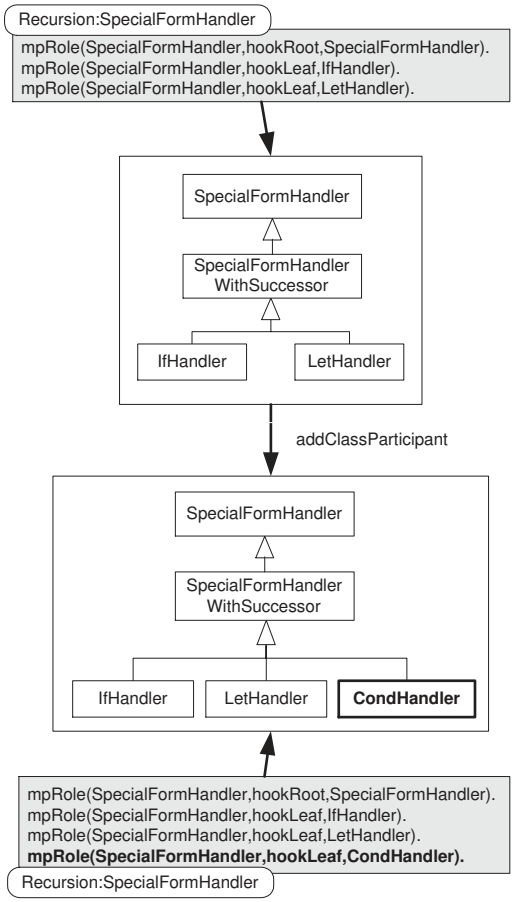


Figure 6.13: The *addClassParticipant/4* adds a *hookLeaf* participant `CondHandler` to metapattern instance *SpecialFormHandler*

The *evolveHookClass/3* predicate

The *evolveHookClass/3* predicate is responsible for implementing phase 2 of the *addHookClass* transformation. As such, its task is to update the class that was added to the implementation in phase 1 by defining the appropriate methods in it. The predicate is implemented as follows:

```
evolveHookClass(?instance,?leaf,?code) if
   findall(?c,
           updateHookClassParticipant(?instance,?leaf,?c),
       ?allCode),
   appendCode(?allCode,?code).
```

It uses the *findall/3* predicate to accumulate all the code generated by the *updateHookClassParticipant/3* predicate. There are a number of different rules that implement the *updateHookClassParticipant/3* predicate, since there are two different ways in which a *hookLeaf* participant must be updated, corresponding to steps 2a and 2b in the above explanation.

The first and simplest form of updating a *hookLeaf* participant that is added to a metapattern instance is consulting that metapattern instance's specification and determining which *hookMethod* participants should be implemented by the class. The *updateHookLeafParticipant/3* thus assembles all *hookMethod* participants of a metapattern instance *?instance* (through the *findall/3* predicate), and generates code that defines all these methods in the newly added class (by invoking the *addMethodParticipants/3* predicate):

```
updateHookClassParticipant(?instance,?leaf,?code) if
   updateHookLeafParticipant(?instance,?leaf,?code).

updateHookLeafParticipant(?instance,?leaf,?code) if
   findall(?selector,mpRole(?instance,hookMethod,?selector),?selectors),
   addMethodParticipants(?selectors,?leaf,?code)
```

As a concrete example, consider Figure 6.14. It depicts the situation where a `CondHandler` class is added to the *SpecialFormHandler* metapattern instance and the specification of this instance is consulted to see which methods need to be added to this class. As it turns out, this instance has three *hookMethod* participants: `handle:`, `canHandle:` and `cantHandle:` methods. Thus an event will be triggered that asks the developer to provide an implementation for these methods. Then, these implementations will be added to the class.

Another task of the *evolveHookClass/3* predicate is to detect overlapping metapattern instances and make sure the appropriate method participants of these instances are also defined in the newly added class. To this extent, two additional *updateHookClassParticipant* rules are defined:

```
updateHookClassParticipant(?instance1,?leaf,?code) if
   overlap(?instance1,?instance2,hookRoot,hookRoot),
   registerNewParticipant(?instance2,hookLeaf,?leaf),
   updateHookLeafParticipant(?instance2,?leaf,?code).

updateHookClassParticipant(?instance1,?leaf,?code) if
[1] overlap(?instance1,?instance2,hookRoot,templateRoot),
[2] registerNewParticipant(?instance2,templateLeaf,?leaf),
[3] updateTemplateLeafParticipant(?instance1,?instance2,?leaf,?code).
```

The first rule handles the case where a *hookHierarchy* participant in one metapattern instance is also a *hookHierarchy* participant in another metapattern instance, or forms a subhierarchy (this corresponds to *Overlapping Condition 1* in Section 5.5.3). This is determined by means of the *overlap/4* predicate. In the running example of the `CondHandler` class, this situation does not occur.

The second rule handles the case where a *hookHierarchy* participant of one metapattern instance is a *templateHierarchy* participant in another metapattern instance, or a subhierarchy of that participant (this corresponds to *Overlapping Condition 2* in Section 5.5.3). If this situation occurs, the *templateMethod* participant of the second metapattern instance should be added to the newly added class. This is achieved by calling the *updateTemplateLeafParticipant/4* predicate (line 3), which is implemented by two logic rules:

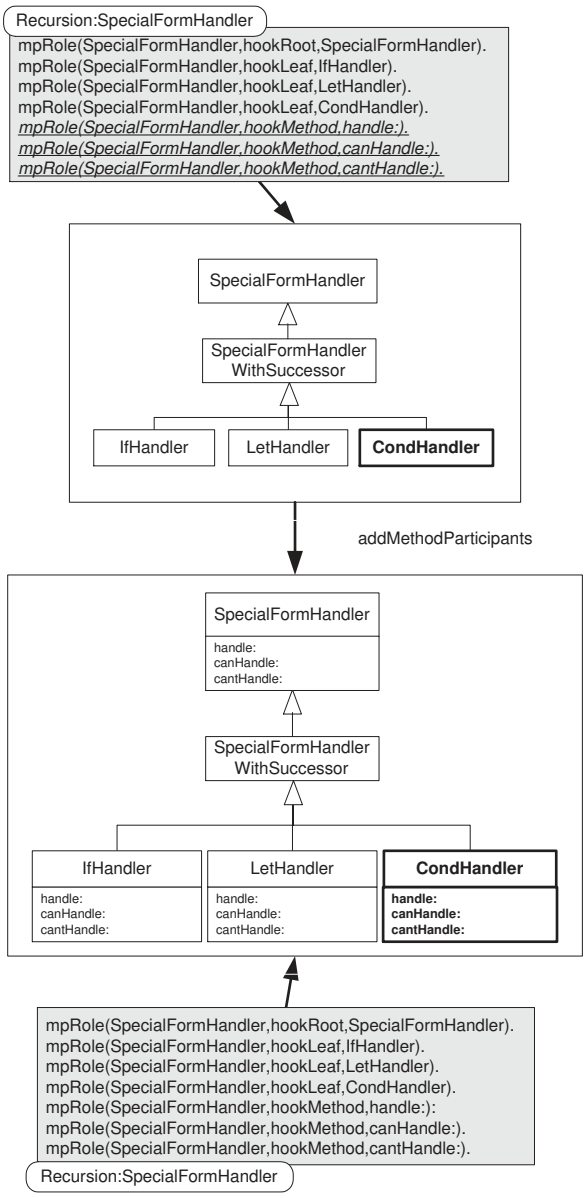


Figure 6.14: All *hookMethod* participants of instance *SpecialFormHandler* are added to the `CondHandler` class

```
updateTemplateLeafParticipant(?instance1,?instance2,?leaf,?code) if
   mpPatternInstance(?instance2,hierarchyMP),
   mpRole(?instance2,templateMethod,?selector),
   addMethodParticipant(?leaf,?selector,?code),
   registerAddHookMethod(?instance1,hookLeaf,?leaf,?selector,?instance2).

updateTemplateLeafParticipant(?instance1,?instance2,?leaf,?code) if
   mpPatternInstance(?instance2,creationMP),
   mpRole(?instance2,templateMethod,?selector),
   addMethodParticipant(?leaf,?selector,?code),
   registerAddHookClass(?instance1,hookLeaf,?leaf,?selector,?instance2)
```

Both rules consult the metapattern instance specification to retrieve the *templateMethod* participant, and generate code that adds this method to the newly added class. This is however not the only thing these transformations perform. As we have seen in Chapter 5, when adding a *templateClass* participant to a metapattern instance, this always requires additional transformations to be performed on this instance. In the case of a *Hierarchy* metapattern instance, an additional *addHookMethod* transformation may be necessary, while in the case of a *Creation* metapattern instance, an *addHookClass* transformation may be needed. This is the reason why there are two rules that implement the *updateTemplateLeafParticipant/4* predicate: one is specific for the *Hierarchy* metapattern and one for the *Creation* metapattern. The *registerAddHookMethod* and *registerAddHookClass* predicates are responsible for registering that such extra transformations need to be performed.

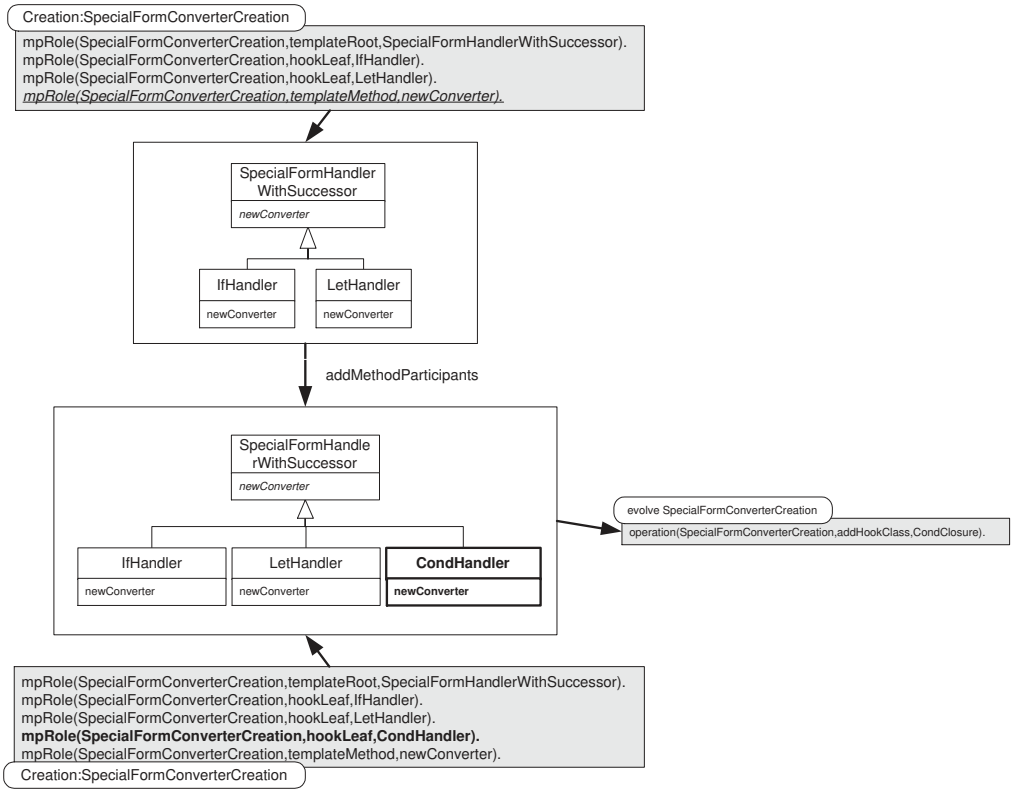


Figure 6.15: The *templateMethod* participant of metapattern instance *SpecialFormConverterCreation*, which overlaps with instance *SpecialFormHandler*, is added to the `CondHandler` class

As a concrete example, consider the situation depicted in Figure 6.15. The *addHookClass* transformation determines that the *SpecialFormConverterCreation* metapattern instance overlaps with the *SpecialFormHandler* metapattern instance according to *Overlapping Condition 2*. Therefore, the *templateMethod* participant of the *SpecialFormConverterCreation* instance, the `newConverter`

method, should be added to the newly added `CondHandler` class. Since the *SpecialFormConverterCreation* instance is an instance of the *Creation* metapattern, its *templateMethod* participant should instantiate a specific class. The development environment thus consults the developer who determines that the `newConverter` method should return an object of the `CondConverter` class. Since this class does not yet exist in the implementation, an additional *addHookClass* transformation is required on the *SpecialFormConverterCreation* metapattern instance.

A similar situation occurs for the *SpecialFormClosureCreation* metapattern instance (see Figure 6.16), where a `newClosure` template method is added to the `CondHandler` class, which should return an object of the `CondClosure` class. This latter class should be added to the implementation via an additional *addHookClass* transformation on the *SpecialFormClosureCreation* instance.

The *overlap/4* predicate, which is used to determine whether two metapattern instances overlap, implements the overlapping conditions defined in Section 5.5.3.

One particular situation that we did not address here is when the *SpecialFormHandler* metapattern instance overlaps with an instance of the *Hierarchy* metapattern. This would require us to add the *templateMethod* participant of the latter instance to the `CondHandler` class, and register that an additional *addHookMethod* transformation is required, to add a *hookMethod* participant that corresponds to the newly added *templateLeaf* participant. This is not much different from the overlappings presented above, however, and the implementation for such situation would be similar to the one above.

The *evolveOverlappingPatterns/2* predicate

As we have seen in the previous section, the *updateTemplateLeafParticipant/4* predicate asserts a number of facts to the logic database that specify that additional transformations are necessary. For example, facts were added to assert the need for an extra *addHookClass* transformation on the *SpecialFormClosureCreation* and *SpecialFormConverterCreation* instances of the *Creation* metapattern. Based on these facts, the *evolveOverlappingPatterns/2* predicate will perform these transformations one after the other.

In the case of our example, two extra transformations will thus be applied:

- an *addHookClass* transformation on the *SpecialFormConverterCreation* instance, which will add a `CondConverter` class as a *hookLeaf* participant.
- an *addHookClass* transformation on the *SpecialFormClosureCreation* instance, which adds a `CondClosure` class as a *hookLeaf* participant.

In their turn, these transformations will make sure these classes are actually added to the implementation and that they define the appropriate methods. For example, a `printOn:` and `nodeDo:value:` method will be defined for the `CondClosure` class, since this class participates in the *CompositeClosure* metapattern instance, in which this method is a *hookMethod* participant (see Figure 6.5). After these transformations have been performed, no more additional transformations are required, so the process stops and the evolution is finished.

6.6 Conclusion

In this chapter, we have shown how an environment that supports framework-based development can be built, based on the model defined in Chapter 5. We illustrated how this environment provides support for the specification of design pattern instances, and for manual as well as supported evolution. In the second part of the chapter, we explained how the approach put forward in Chapter 4 was used to implement the environment. We showed how design pattern instances can be translated into metapattern instances, and vice versa, how metapattern-specific transformations can be implemented in a declarative meta-programming environment, how framework-specific and design pattern-specific transformations can be defined and how metapattern constraints can be specified. We illustrated the main features of the approach and the environment by means of

Figure 6.16: The *templateMethod* participant of instance *SpecialFormClosureCreation*, which overlaps with instance *SpecialFormHandler*, is added to the `CondHandler` class

the Scheme framework. In Chapter 8, we will present a discussion of the results obtained when performing experiments on a real-world case study.

Chapter 7

Support for Software Merging

In previous chapters, we focused on support for framework-based development by means of enforceable design constraints and automated high-level transformations. Besides being of great value for developers that evolve or instantiate a framework, this kind of support also allows us to provide support for the process of software merging. In this chapter, we will elaborate on this issue. We show how different merge conflicts, that are due to developers modifying the framework simultaneously, can be detected and consider the different ways in which they can be resolved.

7.1 Introduction

When different developers evolve the same version of framework independently of one another, different versions of the framework are created, each one incorporating a different change. All these versions need to be merged into one single version again, that incorporates all changes. This situation is depicted in Figure 7.1. Merging different versions of a framework (or any software system in general) can give rise to various *merge conflicts*. This is due to the fact that developers rely on particular assumptions about the framework when evolving it. Because the framework is modified in parallel, some of these assumptions may no longer be valid.

Because we represent instantiation and evolution in terms of transformations on design patterns, and because we encourage developers to use high-level transformations, we can provide elaborate support for software merging. This support includes automatically detecting and reporting possible merge conflicts. Moreover, because evolution is expressed at a high-level, the reported conflicts and proposed solutions are also reported at a high level. As such, it becomes easier to identify the particular reasons for a certain conflict, because the intentions and the assumptions of the different evolutions are made explicit. This also makes it easier to resolve the conflict in the appropriate way.

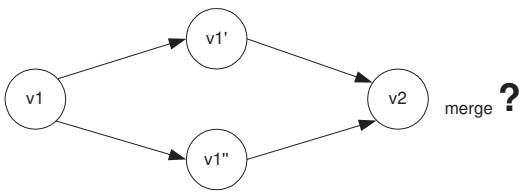


Figure 7.1: Merging parallel changes

7.2 Approach

The approach we propose to detect possible merge conflicts is an operation-based merge algorithm that is an extension of similar techniques proposed by [Men99, Luc97, SLMD96], and can be considered an extension of it. In our approach, changes are applied to a framework by using high-level transformations. By carefully analyzing these transformations, we can automatically infer the conditions under which two of them influence one another when applied in parallel, and hence give rise to a merge conflict. Besides identifying these conflicts, in some cases we can even specify the actions that should be taken in order to resolve them.

We will construct a conflict table to compare metapattern-specific transformations and identify the conditions that give rise to a conflict when these are applied in parallel. This ensures that the approach remains scalable, as there are only a limited number of metapattern-specific transformations. Furthermore, these transformations are the basis for both design pattern-specific and framework-specific transformations. As such, a developer can still use these more intuitive and high-level transformations, and will still get reports of possible conflicts.

Besides the metapattern-specific transformations, we will also consider other transformations that can be applied to a framework. Metapattern-specific transformations only deal with the metapattern instances in the framework and are mainly useful for supporting anticipated evoluition. Many other high-level transformations can be applied that do not influence the metapattern instances, but do change the design of the framework and can thus be used to support unanticipated evolution as well. Such transformations are called refactorings, and will be included in our conflict tables as well.

Naturally, we can not detect each and every possible merge conflict. Only when the evolution (and instantiation) of the framework can be expressed as a series of metapattern-specific transformations or refactorings will we be able to detect such conflicts. When two developers manually evolve the framework at the same time, conflicts can be introduced but will remain undetected by our approach.

7.2.1 Considered Refactorings

Many possible refactorings exist, and the more important and widely used ones are gathered in a catalog [Fow99]. In order not to sacrifice the scalability of the approach, we will not consider each and every possible refactoring. Instead, we will consider the *low-level* refactorings defined in [Opd92, Rob99, Tic01]. These refactorings are formally defined, so we know their effect upon the implementation, and can be combined to form *higher-level* refactorings [Opd92, OR93, JO93] that can be used to evolve a framework in many different ways.

What follows is a list of these low-level refactorings.

- *addClass(className, superclass, subclasses)*: this refactoring adds a new class named `className` as a subclass of class `superclass`. Furthermore, it changes the superclass of all classes included in the *subclasses* list to the new `className` class.
- *renameClass(class, newName)*: the *renameClass* refactoring renames the class `class` to `newName`. All references to the old name are automatically updated as well.
- *removeClass(class)*: this refactoring removes the class `class` from the framework. This is only possible when no references to this class exist (the class may not have any subclasses, for example).
- *addMethod(class, selector, body)*: this refactorings adds a method `selector` to the class `class` and provides an implementation *body* for it. This is only allowed if this method does not override a method in the class' superclass, and it is not implemented by any of the class' subclasses.
- *renameMethod(class, selector, newName)*: this refactoring renames the method `selector` to `newName` in the class `class`. It also updates all references to the original `selector` and

all overriding methods. We assume that this refactoring is applied to the highest class in the class hierarchy that defines the method. In other words, the method does not override a method of any superclass of class `class`.

- *removeMethod(class, selector)*: this refactoring removes the method `selector` from the class `class`. No references to `selector` may exist in the framework, meaning that the operation is prohibited when other methods call this method, override it, or are overridden by it.

- *pullUpMethod(class, selector)*: this refactoring removes the method `selector`, defined by subclasses of class `class`, in the class hierarchy and defines it in class `class` itself. The method's implementation in all subclasses is removed.

- *pushDownMethod(class, selector)*: this refactoring performs the opposite operation of the *pullUpMethod* refactoring: it pushes a method `selector` down in the class hierarchy, and defines it in all direct subclasses of class `class`.

- *moveMethod(class, selector, destClass, newSelector)*: this refactoring moves the method `selector` from class `class` to the class `destClass`. At the same time, this method is renamed to `newSelector`, and is replaced in the old class by an implementation that simply forwards the call to the new method.

- *addParameter(class,selector,name)*: this refactorings adds an extra parameter with name *name* to method `selector` in class `class`. At the same time, the parameter is added to all methods that override this method and all call sites are updated with a default value as the extra parameter. Like for the *renameMethod* refactoring, we require that this refactoring is applied with respect to the highest class in the class hierarchy that defines the method. The method in class `class` does not override a method in a superclass of class `class`.

- *removeParameter(class,selector,name)*: this refactorings removes the parameter *name* from the method `selector` in class `class`. At the same time, the parameter is removed from all overriding methods and the call sites are updated so that they no longer use the parameter. Like for the *addParameter* refactoring, we require that this refactoring is applied to the highest class in the class hierarchy that defines the method.

- *addVariable(class, varName, initClass)*: this refactoring adds an instance variable `varName` to the class `class` and initializes it with an object of class `initClass`.

- *removeVariable(class, varName)*: this refactoring removes the variable `varName` from the class `class`.

- *renameVariable(class, varName, newName)*: this refactoring renames the variable `varName` of class `class` to `newName`.

- *pullUpVariable(class, varName)*: this refactoring removes the variable `varName` from all subclasses of class `class` and defines the variable in `class` itself.

- *pushDownVariable(class, varName)*: this refactoring removes the variable `varName` from class `class` and defines it in all of its direct subclasses. This is only allowed when the class `class` does not contain any references to the variable.

- *moveVariable(class, varName, destClass)*: this refactoring moves the variable `varName` from class `class` to the class of variable `destVarName`. This is only possible when there exists a one to one relation between these two classes.

7.2.2 Conflict Categories

We can classify the different kinds of merge conflicts that we can detect into four different categories:

naming merge conflicts occur when two transformations introduce a class or a method with the same name or when one transformation renames an existing class or method and another transformation adds a class or method with the same name.

constraint violation merge conflicts are all conflicts that can be detected by checking the constraints of the metapattern instances occurring in the framework. Remember from Chapter 5 that we specifically introduced metapattern-specific transformations to avoid introducing constraint violation conflicts. Applying one single such transformation will thus never give rise to a constraint violation conflict, but when applying many such transformations in parallel, such conflicts can occur again, as we will demonstrate.

structural merge conflicts signal structural inconsistencies in the design of the framework. A class may still override a method that has already been removed from its superclass, for example, or similar participants in a design pattern instance are implemented in a completely different way. While structural conflicts do not affect the behavior of the framework, it is important to detect the inconsistencies in the design. Otherwise, the framework may drift away further from the intended design after a couple of iterations.

behavioral merge conflicts are conflicts that cause the framework to behave in unforeseen and unpredictable ways. Whereas the structure and the design of the framework may appear correct, a behavioral conflict signals a possible error in the behavior of some methods. Although such conflicts are situated at the method level, they can be caused by design level transformations, as we will see.

7.3 Merge Conflicts due to Parallel Application of Metapattern-Specific Transformations

In this section, we will start by mutually comparing the metapattern-specific transformations defined in Chapter 5, define the conditions under which merge conflicts can arise when they are applied in parallel and determine possible ways of resolving these conflicts.

Eight different metapattern-specific transformations exist for the five fundamental metapatterns we identified: *addHookClass (aHC), addTemplateClass (aTC), addHookMethod (aHM), addTemplateMethod (aTM), removeHookClass (rHC), removeTemplateClass (rTC), removeHookMethod (rHM)* and *removeTemplateMethod (rTM)*. Remember that not all of these transformations are applicable on all five fundamental metapatterns. This does not concern us here: we assume that a metapattern-specific transformation is applied on the appropriate metapattern instance.

We will compile a conflict table that allows us to compare such transformations one by one. In order to keep this table compact, we will represent the eight transformations listed above as only four transformations: *addLeaf, removeLeaf, addMethod* and *removeMethod*. An *addLeaf* transformation thus represents either an *addHookClass* or an *addTemplateClass* transformation, and similarly for the other operations. This has no influence on the applicability or completeness of our approach. The conflict tables can easily be expanded when information about the specific transformations is necessary. In the discussions that follow, this is not strictly necessary, however.

For each merge conflict that we identify, we will formally define the conditions under which it occurs. An example of such a conflict will be provided, in order to explain it more clearly. In these examples, we will use design pattern-specific transformations although conflicts are detected by comparing metapattern-specific transformations. Since the former transformations can be translated into the latter, this is no problem however. Furthermore, we will always consider

	$addLeaf_b$	$addMethod_b$	$removeLeaf_b$	$removeMethod_b$
$addLeaf_a$	NM	C	–	OM
$addMethod_a$	C	NM	–	–
$removeLeaf_a$	–	–	–	–
$removeMethod_a$	OM	–	–	–

NM = naming conflict, C = constraint violation conflict
OM = obsolete method conflict

Table 7.1: Comparing Metapattern-Specific Transformations

transformations applied to one particular metapattern instance. Strictly speaking this is not absolutely required. If transformations are applied on overlapping metapattern instances, they can give rise to the same merge conflict. The formal definition of the conflict will take this into account, whereas the examples don't, for the sake of clarity and simplicity.

7.3.1 Conflict Table

Table 7.1 relates the different metapattern-specific transformations. Those on the left are all applied on a metapattern instance a, while those at the top are applied on a metapattern instance b (which may or may not be the same instance). An entry in the table defines the kind of merge conflict that occurs whenever the conditions for that conflict are satisfied. Only three different kinds of conflicts can occur: *naming, constraint violation* and *obsolete method* merge conflicts. In the following sections, we will discuss these conflicts in more detail and provide a formal definition that includes the conditions under which they occur.

7.3.2 Naming Merge Conflicts

Naming conflicts occur when two developers independently introduce an artifact, such as a class or a method, with the same name. A framework can not contain two classes with the same name. Likewise, a naming conflict involving methods can only occur when two developers add a method with the same name to the same class. It is allowed to introduce methods with the same name in different classes, of course, as this does not pose an problems when merging the different versions.

Example Conflict

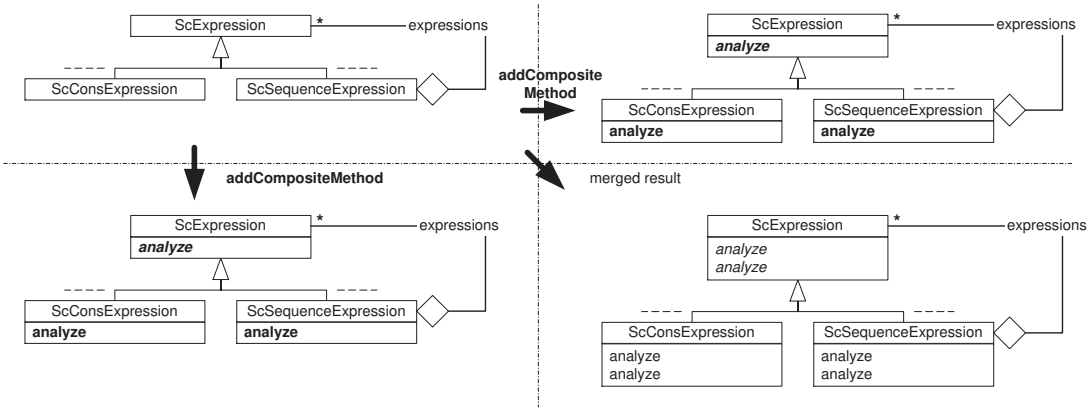


Figure 7.2: A *naming* merge conflict

As an example of a *naming* merge conflict, consider the situation depicted in Figure 7.2. Two developers simulteanously and independently from one another introduce an `analyze` method in the same class hierarchy by means of an *addCompositeMethod* transformation. The merged result in the figure shows how the classes contain two methods with the same name. In practice, however, this is not possible, since classes can not contain two method definitions with the same name. Most likely, when merging, the effect of the transformation that is applied first will be undone by applying the second transformation.

Note that, although two developers introduce a method with the same name, it is not guaranteed that this method actually serves the same purpose. It may wel be that one `analyze` method's responsibility is to translate the expression into a closure object (as explained in Chapter 3), whereas the other method's responsibility is to analyze whether the abstract syntax tree representing the expression is well-formed. In the former case, the conflict can be resolved by simply choosing one of the two implementations. In the latter case, however, one of the two methods should be renamed.

Formal Definition

Table 7.2 contains a formal definition of the conditions under which *naming* merge conflicts occur when adding class participants to a metapattern instance. Table 7.3 contains a formal definition of the conditions that give rise to a naming conflict involving method participants. The $\|$ operator denotes that the two transformations are applied in parallel.

As can be seen from these definitions, a *naming* merge conflict involving methods occurs when a method participant with the same name is added to instance a and an instance b at the same time, and these instances overlap (e.g. instance a overlaps with instance b, or vice versa). This overlapping is specified in terms of the $inherits_h^*$ relation that was defined in Chapter 5. Note that in these definitions, an *addMethod* transformation actually represents either an *addHookMethod* or an *addTemplateMethod* transformation. As such, the conditions in Table 7.3 actually cover all four possible combinations of both these transformations. Also note that the notation $\mathcal{H}_a$ actually denotes the *hookHierarchy* participant of metapattern instance a when an *addHookMethod* transformation is involved, but denotes the *templateHierarchy* participant when an *addTemplateMethod* operation is involved. The kind of transformation that is applied actually determines which hierarchy participant is involved.

We should note that the occurrence of *naming* merge conflicts depends upon the particular programming language that is used. In a programming language with namespaces (such as Java, for example), it is allowed to introduce classes with the same name, as long as they reside in a different namespace. It is not allowed to have classes with the same name in the same namespace. Furthermore, some programming languages also allow method overloading (C++ and Java, for example). This means that methods with the same name may be defined in the same class, as long as their number of arguments differ. If we require that the transformations always work with the fully qualified class name (e.g. the name takes into account the namespace of the class) and the signature of the method (e.g. we do not only consider the method name, but also the different types of its arguments), the conditions for *naming* merge conflicts are still correct and independent of the particular programming language that is used.

Discussion

Although naming conflicts are very basic conflicts, there is no automatic way for resolving them. It is impossible to infer automatically whether the two artifacts that are introduced actually serve the same purpose. For example, when two classes with the same name are introduced, it is not necessarily the case that they represent the same thing.

The only valid solution to a naming conflict is thus to consult the original developers that introduced both artifacts. They should look into the situation and decide to either use one of both classes if they do represent the same thing or perform a renaming operation when they don't.

$$addLeaf_a(leafName_1) \parallel addLeaf_b(leafName_2) \Rightarrow \\ namingConflict(leafName_1) \\ if \\ leafName_1 = leafName_2$$

Table 7.2: Condition giving rise to a naming conflict when adding class participants in parallel

$$addMethod_a(m_1) \parallel addMethod_b(m_2) \Rightarrow \\ namingConflict(m_1) \\ if \\ m_1 = m_2 \wedge \\ (inherits_h^*(\mathcal{H}_a, \mathcal{H}_b) \vee \\ inherits_h^*(\mathcal{H}_b, \mathcal{H}_a))$$

Table 7.3: Conditions giving rise to a naming conflict when adding method participants in parallel

7.3.3 Constraint Violation Merge Conflicts

Even though we introduced transformations to help a developer in avoiding constraint violation conflicts, such conflicts can still arise when different developers apply such transformations in parallel.

Example Conflict

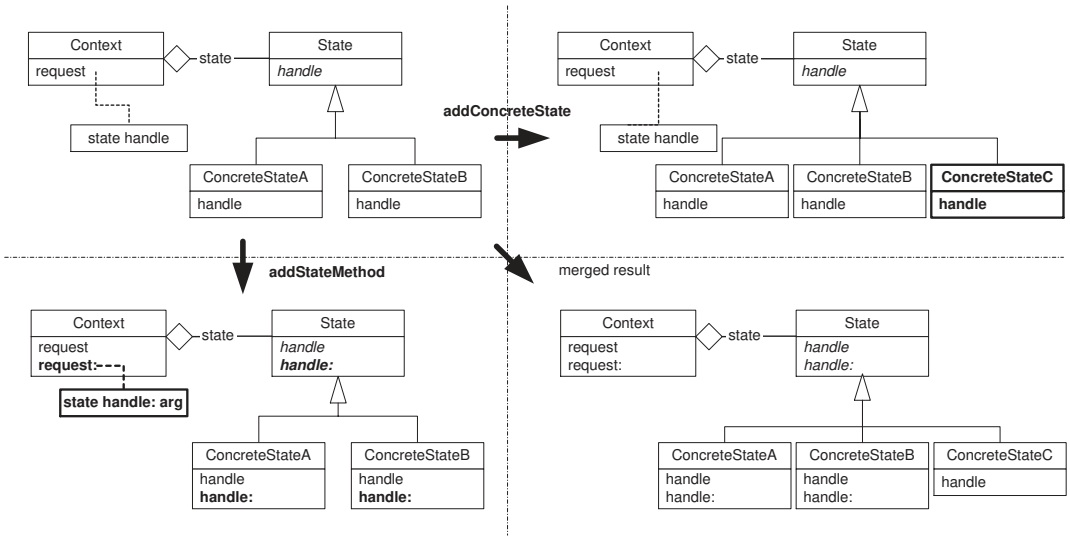


Figure 7.3: A *constraint violation* merge conflict

Figure 7.3 contains an example of a *constraint violation* merge conflict. An instance of the *State*

$$addLeaf_a(leafName) \parallel addMethod_b(selector) \Rightarrow$$
$$constraintViolationConflict(leafName, selector)$$
$$if$$
$$inherits_h^*(\mathcal{H}_a, \mathcal{H}_b)$$

Table 7.4: Condition giving rise to a *constraint violation* merge conflict

design pattern is evolved in two different ways. One developer evolves it by applying the *State* specific *addConcreteState* transformation to add a new *concreteState* participant, `ConcreteStateC` to the design pattern instance. Independently, another developer uses the *addStateMethod* transformation, that is also specific to the *State* design pattern, to add a new state-specific method, `handle:`, to the same instance.

When merging these two particular versions of the *State* design pattern, a *constraint violation* merge conflict arises: the newly added class `ConcreteStateC` does not provide an implementation for the newly added method `handle:`, since the class did not yet exist in the version where the *addStateMethod* transformation was applied.

Formal Definition

Table 7.4 contains the formal definition of the condition that gives rise to a *constraint violation* merge conflict. As can be seen, such a conflict occurs when one developer adds a class participant to a particular metapattern instance, and another developer simultaneously adds a method participant to the same instance or to an overlapping instance.

Observe that only overlapping conditions 1 to 4 of Section 5.5.3 are considered in the definition. Overlapping conditions 5 to 7 deal with overlapping of an instance of any kind of metapattern with an instance of the *Connection* metapattern, via the latter's *templateClass* participant. The *Connection* metapattern does not define an *addTemplateClass* transformation, and its *templateMethod* participant is not overridden in subclasses. As such *constraint violation* merge conflicts can only occur if the *hookHierarchy* participant of such metapattern instances is involved, and this is covered by overlapping conditions 1 to 4.

The particular constraint violation conflict that occurs is a *notUnderstoodByLeaf* conflict, that arises when a selector is registered as a method participant in some metapattern instance, but is not understood by a class that is a leaf participant of that instance (see Chapter 6).

Discussion

As opposed to *naming* merge conflicts, *constraint violation* merge conflicts can be resolved automatically, by simply imposing a strict order on the application of the transformations. First, all transformations that add method participants should be applied, and only afterwards those transformations that add class participants are allowed. In the example above, if we would first apply the *addStateMethod* transformation, followed by the *addConcreteState* transformation, the `handle:` method would be registered as a *stateMethod* participant in the instance of the *State* design pattern, and thus the developer would be forced to provide an implementation for it in the `ConcreteStateC` class.

7.3.4 Obsolete Method Merge Conflicts

Example Conflict

Figure 7.4 contains an example of an *obsolete method* merge conflict. Once again, an instance of the *State* design pattern evolves in two different ways. The first evolution again performs an

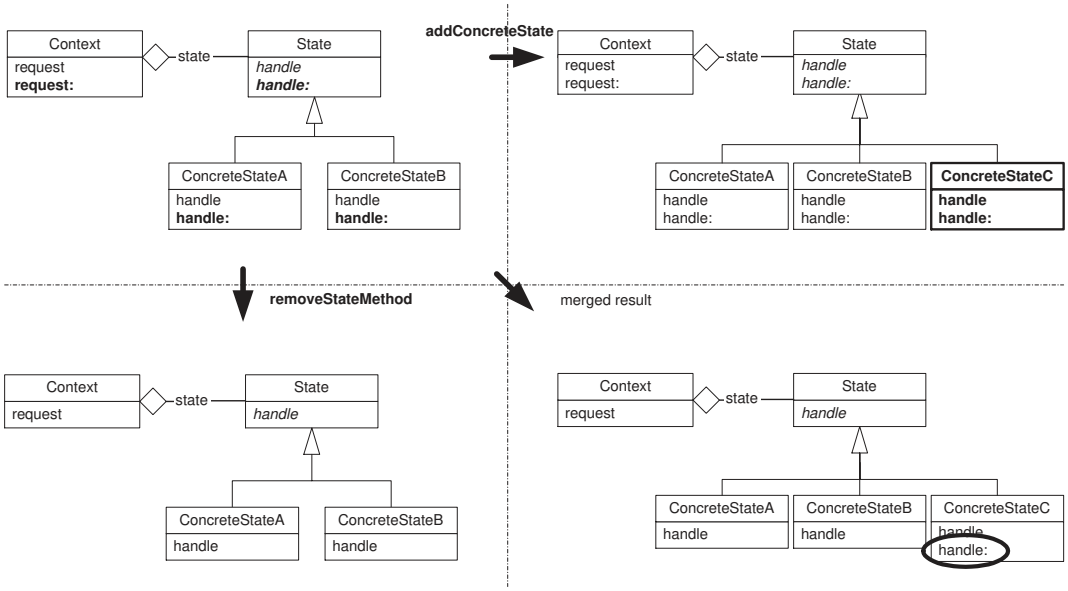


Figure 7.4: An *obsolete method* merge conflict

$$addLeaf_a(leafName) \parallel removeMethod_b(selector) \Rightarrow$$
$$obsoleteMethodConflict(leafName, selector)$$
$$if$$
$$inherits_h^*(\mathcal{H}_a, \mathcal{H}_b) \wedge definesMethod(class(leafName), selector)$$

Table 7.5: Conditions giving rise to an *obsolete method* merge conflict

addConcreteState transformation, which adds a new `ConcreteStateC` *concreteState* class participant to the instance. This time, the second transformation, *removeStateMethod*, removes the *stateMethod* participant `handle:` from the instance. As a result, in the version of the design pattern instance incorporating the two evolutions, the newly introduced *concreteState* class participant `ConcreteStateC` still contains a *stateMethod* participant, that was removed from all other *concreteState* participants by the *removeStateMethod* transformation.

Formal Definition

A formal definition of the condition that gives rise to an *obsolete method* merge conflict can be found in Table 7.5. Such conflicts occur when one developer adds a class participant to a particular metapattern instance, while another developer removes a method from the instance or an overlapping instance at the same time.

The overlapping of metapattern instances is again specified by means of the $inherits_h^*$ relationship. As was the case for *constraint violation* merge conflicts, *obsolete method* merge conflicts only need overlapping conditions 1 to 4 (see Section 7.3.3).

Discussion

Obsolete method merge conflicts are structural merge conflicts. Obviously, the fact that a class implements a method that is never called, does not mean that the intended design of the framework

is violated. Most likely, the framework will still behave as intended. However, the conflict should be reported, because obsolete methods pollute the interface of a class, and nothing prevents developers to use such methods in the future if they so desire. This clearly is not what is intended, since the aim actually was to remove this method.

To resolve an *obsolete method* merge conflict, it suffices again to impose an order on the application of transformations. This time, the transformations that add class participants should be applied before any transformation that removes method participants. In the case of our example, this would mean that the *addConcreteState* transformation is applied before the *removeStateMethod* transformation. Doing so, the `ConcreteStateC` class will also be subject to the *removeStateMethod* transformation, which will thus remove the `handle:` method.

7.3.5 Summary

Merge conflicts arise due to the parallel application of transformations on overlapping metapattern instances. We identified three different kinds of conflicts: *naming, constraint violation* and *obsolete method* merge conflicts. *Naming* merge conflicts can not be resolved automatically, and the developers should be consulted in order to solve the problem. Both *constraint violation* and *obsolete method* merge conflicts can be resolved by imposing an order on the transformations. First, those transformations that add method participants should be applied, then, transformations that add class participants and afterwards those that remove method participants. In this way, the merged version of the framework will incorporate all changes and will not contain merge conflicts of those kinds.

7.4 Merge Conflicts due to Parallel Application of Metapattern transformations and Refactorings

Now that we have defined the merge conflicts that can occur when applying metapattern transformations in parallel, in this section, we will identify the conflicts that occur when applying a metapattern transformation and a refactoring in parallel, and define the conditions under which this gives rise to merge conflicts. We will only present a detailed discussion of some of the conflicts in this chapter, and refer to the appendix for the definition of all conflicts not discussed here.

It is important to note that we will not mutually compare refactorings in order to discover when their parallel application will lead to a merge conflict. We will only compare refactorings to metapattern-specific transformations. While our approach is general enough for comparing refactorings as well, we believe this is beyond the scope of this dissertation. Furthermore, Roberts already analyzed the dependencies between refactorings in his doctoral dissertation [Rob99]. This allows him to detect conflicts between a group of refactorings that are applied, which is a first step towards support for software merging based on refactorings.

It should be noted that some of the refactorings bear resemblance to some of the metapattern transformations. For example, both the *addClass* refactoring and the *addHookClass* transformation add a particular class to the framework. However, the transformations perform some additional changes as well: they register the participant they add, or unregister the participant they remove, and make sure other changes are performed, as needed, to ensure no constraint violation conflicts occur. In what follows, we will assume that if a particular refactoring is applied, it is the intention of the developer to effectively apply this refactoring, and not to apply some evolution transformation.

7.4.1 Conflict Table

Table 7.6 compares the metapattern transformations (at the top of the table) to the refactorings (on the left side of the table). Whenever two transformations applied in parallel may give rise to a merge conflict, the table entry contains the kind of conflict that occurs. As can be seen, we

	$addLeaf_a$	$addMethod_a$	$removeLeaf_a$	$removeMethod_a$
$addClass$	N/PIC	–	–	–
$renameClass$	N	–	–	–
$removeClass$	–	–	–	–
$addMethod$	–	N	RM	–
$renameMethod$	–	N	–	–
$removeMethod$	–	–	–	–
$moveMethod$	–	N	MD	MO
$pullUpMethod$	OM	–	OG	–
$pushDownMethod$	MA	–	–	–
$addParameter$	MP	–	–	–
$removeParameter$	OP	–	–	–
$addVariable$	–	–	RM	–
$renameVariable$	–	–	–	–
$removeVariable$	–	–	–	–
$moveVariable$	–	–	MD	–
$pullUpVariable$	OV	–	–	–
$pushDownVariable$	VA	–	–	–

N = naming merge conflict, OG = overly general method merge conflict
MA = method absence merge conflict, VA = variable absence merge conflict
OV = orphan variable merge conflict , OM = orphan method merge conflict
RM = remove merge conflict, MD = missing destination merge conflict
MO = missing origin merge conflict, MP = missing parameter merge conflict
OP = obsolete parameter merge conflict, PIC = possible incorrect superclass merge conflict

Table 7.6: Comparing Metapattern transformations and Refactorings

identified twelve possible merge conflicts. We will only discuss four of these conflicts in detail here. We refer to the appendix for a discussion of the remaining conflicts.

7.4.2 Possible Incorrect Superclass Merge Conflicts

Example Conflict

A concrete example of a *possible incorrect superclass* merge conflict is depicted in Figure 7.5. The upper left hand part of the figure shows the situation where all handler classes in the Scheme framework simply inherit from the `SpecialFormHandler` class. This design does not take into account the difference between the `ApplicationHandler` class and all other classes: the former class does not have a successor, while all others do. To improve the design, one developer decides to insert a new `SpecialFormHandlerWithSuccessor` class into the hierarchy, as a subclass of all classes except the `ApplicationHandler` class. The resulting design is depicted in the upper right side of the figure. At the same time however, another developer decides to add a new `CondHandler` leaf class to the same class hierarchy, by means of an *addConcreteHandler* transformation. The result is depicted in the lower left part of the figure.

When merging both versions, a *possible incorrect superclass* merge conflict occurs, because the `CondHandler` class should really be a subclass of the new `SpecialFormHandlerWithSuccessor` class, instead of subclassing `SpecialFormHandler` directly.

Formal Definition

Table 7.7 shows the condition that gives rise to a *possible incorrect superclass* merge conflict. As can be seen, such a conflict is caused by applying an *addClass* refactoring in parallel with an

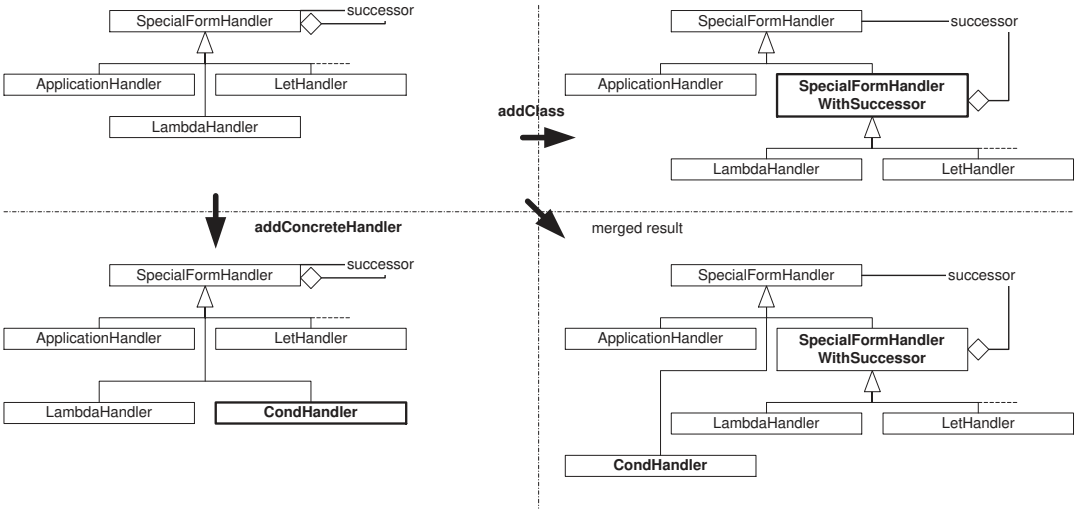


Figure 7.5: A *possible incorrect superclass* merge conflict

$$addClass(className, superclass, subclasses) \parallel addLeaf_a(leafName) \Rightarrow possibleIncorrectSuperclass(className, leafName)$$
$$if$$
$$inherits_h^*(\{superclass\}, \mathcal{H}_a)$$

Table 7.7: Condition giving rise to a *possible incorrect superclass* merge conflict

addTemplateClass or *addHookClass* transformation (represented by an *addLeaf* transformation). The *addClass* refactoring can insert a class anywhere into a class hierarchy, thereby redefining the superclass of already existing subclasses to the newly introduced class. When at the same time a second new class is added to the same class hierarchy, it may well be that this second class should be a subclass of the first class as well.

Discussion

A *possible incorrect superclass* merge conflict is a structural merge conflict. If such a conflict occurs, similar classes are implemented in a different ways: classes that were already present in the framework become subclasses of the class introduced by the *addClass* refactoring, whereas only the class that is added by the transformation is not a subclass of that class, but should presumably be.

Resolving this kind of conflict automatically is not possible. First of all, we can not automatically decide whether the class added by the transformation should be a subclass of the class added by the refactoring. This is something which only the developers responsible for the changes can decide. Second, simply imposing an order on the transformations will not automatically resolve the conflict. Clearly, the *addLeaf* transformation must be applied first, to add the new class to the class hierarchy participant of the metapattern instance. Afterwards, the *addClass* refactoring can be applied. However, the class added by the transformation is not included in the arguments of the refactoring. Therefore, it will not be a subclass of the class introduced by the refactoring, and the conflict still remains to be solved.

7.4.3 Overly General Method Merge Conflicts

Some methods of a class, such as template methods, provide a default implementation for an algorithm, and are supposed to be reused by the different subclasses of the class. The behavior of such a method must thus be appropriate for all subclasses. As we will see, when applying particular transformations, the behavior of this method may take into account many more classes than it should. In that case, we say the method is *overly general*.

Example Conflict

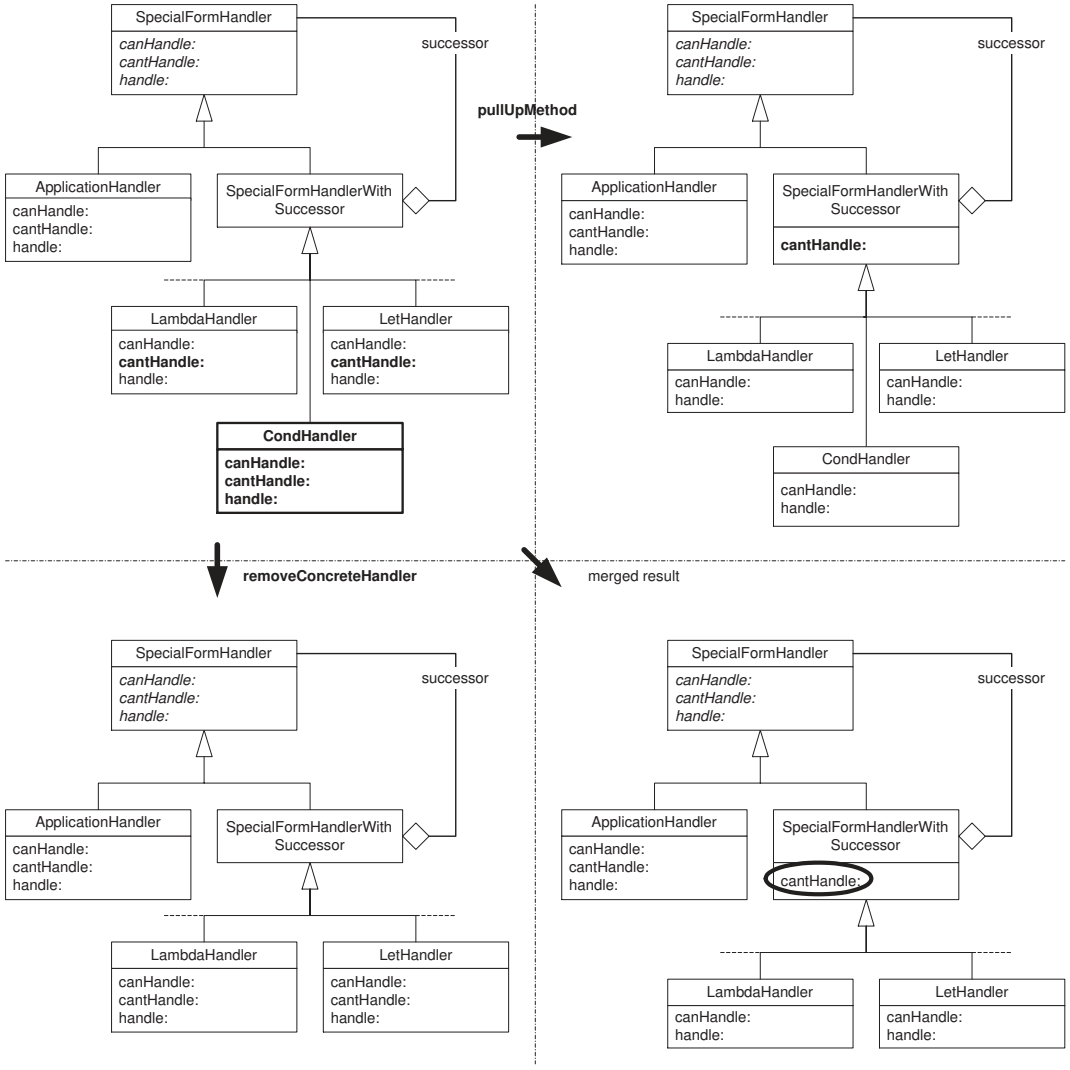


Figure 7.6: An *overly general method* merge conflict

A particular example of an *overly general method* merge conflict is depicted in Figure 7.6. It is taken from the Scheme framework discussed in Chapter 3, in particular, the *Chain of Responsibility* design pattern instance that occurs in it.

One developer decides to apply a *removeConcreteHandler* transformation, to remove the `CondHandler` class as a *concreteHandler* participant from the design pattern instance. At the same time, another developer decides to apply the *pullUpMethod* refactoring, to pull up the

$$pullUpMethod(class, selector) \parallel removeLeaf_a(leafName) \Rightarrow overlyGeneralMethodConflict(class, selector)$$
$$if$$
$$inherits^*(class(leafName), class) \wedge definesMethod(class(leafName), selector)$$

Table 7.8: Conditions giving rise to an *overly general method* merge conflict

`cantHandle:` method from the subclasses of class `SpecialFormHandlerWithSuccessor`. In doing so, this method should be generalized, so that it implements the appropriate behavior for all the subclasses. This generalization will take into account that the implementation should work for the `CondHandler` class as well. However, since another developer removed this class, in the merged version of the software, this is not strictly necessary. The implementation of the `cantHandle:` in the merged version could be overly general, and should thus be inspected and changed as needed.

Formal Definition

The formal definition of the condition that lead to an *overly general method* merge conflict is contained in Table 7.8. Such a conflict occurs when the *pullUpMethod* refactoring is applied in unison with a metapattern transformation that removes a class participant from a metapattern instance. Only if the class participant that is removed and the class to which the method is pulled up are related via inheritance, the conflict occurs. More specifically, the class participant that is removed should be a (possible indirect) subclass of the class to which the method is pulled up. This is reflected in the condition by making use of the $inherits^*$ relation.

Discussion

Overly general method merge conflicts are classified as behavioral merge conflicts. The conflict occurs because the behavior of the method that is pulled up may have to be adapted, to reflect the new situation where there are less subclasses than originally anticipated. Clearly, this concerns the behavior of the framework, and not its structure.

Overly general method merge conflicts can not easily be resolved in an automatic manner. It is difficult to assess automatically how general the method that is pulled up is, and if it effectively contains behavior that takes into account the class that was removed, and that should be removed. The developer should thus have a look at the implementation and decide whether it needs adaptation or not.

7.4.4 Method/Variable Absence Merge Conflicts

Example Conflict

Figure 7.7 shows an example of a *variable absence* merge conflict. An instance of the *Composite* design pattern in the Scheme framework is evolved in two ways. One developer decides to push the `lineNumber` instance variable down the `ScExpression` class hierarchy by using the *pushDownVariable* transformation. At the same time, another developer applies the *addLeaf* transformation, that is specific to the *Composite* design pattern, and adds a class `ScQuoteExpression` as a leaf participant to the instance. In the merged result, we observe the the class that was added by the second developer does not contain the `lineNumber` variable.

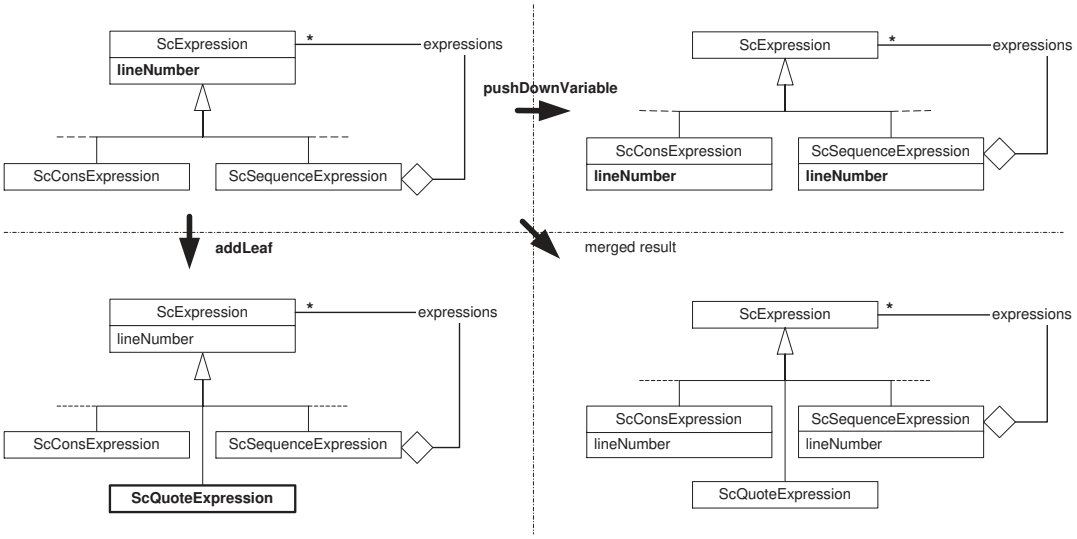


Figure 7.7: A *variable absence* merge conflict

$$pushDownMethod(class, selector) \parallel addLeaf_a(leafName) \Rightarrow methodAbsenceConflict(leafName, selector)$$
$$if$$
$$class = root(\mathcal{H}_a)$$

Table 7.9: Condition giving rise to a *method absence* merge conflict

Formal Definition

A *method absence* merge conflict occurs when a *pushDownMethod* is applied in parallel with an *addHookClass* or an *addTemplateClass* transformation (see Table 7.9). In the case of an *addHookClass* transformation, applied to a metapattern instance a, the conflict occurs when the class from which the method is pushed down acts as the root of the *hookHierarchy* participant of instance a. Similarly, in case of an *addTemplateClass* transformation, the conflict arises whenever the class from which the method is pushed down is the root of the *templateHierarchy* participant of instance a. Both participants are conveniently denoted as $root(\mathcal{H}_a)$.

In a similar way, Table 7.10 defines the condition that gives rise to a *variable absence* merge conflict. The only difference compared with the *method absence* merge conflict is that a *pushDownVariable* refactoring operation is performed, as opposed to a *pushDownMethod* refactoring.

Discussion

The *method absence* merge conflict is a structural conflict, as is the *variable absence* merge conflict. The parallel application of the specific refactoring (*pushDownMethod* or *pushDownVariable*) and the transformation (*addHookClass* or *addTemplateClass*) leads to a situation in which similar artifacts are implemented in an inconsistent way. The example presented in the previous section clearly showed that all leaf participants in the *CompositeExpression* instance of the *Composite* design pattern define a variable `lineNumber`, except the `ScQuoteExpression` class. This can be considered an inconsistency in the design, which is why we classify such conflicts as structural conflicts.

$$pushDownVariable(class, variable) \parallel addLeaf_a(leafName) \Rightarrow variableAbsenceConflict(leafName, variable)$$
$$if$$
$$class = root(\mathcal{H}_a)$$

Table 7.10: Condition giving rise to a *variable absence* merge conflict

Method absence or *variable absence* merge conflicts are caused because a new class is introduced into a class hierarchy, while at the same time this class hierarchy is reorganized in some or other way. Clearly, if we first add the new class to the hierarchy and only then reorganize it, the new class will take part in the reorganization, and the conflict will disappear. These kind of merge conflicts can thus again be resolved by imposing an order on the application of the transformations. First, the *addLeaf* transformation should be applied, and only afterwards the *pushDownMethod* or *pushDownVariable* refactoring. In the case of our example, this ordering would ensure that the `lineNumber` variable would also be pushed down to the `ScQuoteExpression` class.

Note that we could have considered a *method absence* merge conflict as a constraint violation conflicts instead of as a structural merge conflict, but only if the method that is pushed down in the class hierarchy participates in a design pattern instance, which need not necessarily be the case, However, if this was the case, a *notUnderstoodByLeaf* constraint violation conflict would be reported: a class registered as a leaf participant in some design pattern instance would not implement a method registered as a method participant in that same instance. A *variable absence* merge conflict can not be considered a constraint violation conflict, since design patterns (or metapatterns) do not define constraints involving instance variables.

Since we want to report *method absence* merge conflicts under all circumstances, and a refactoring can push down a method that does not participate in a design pattern instance, we consider this conflict to be a structural conflict instead of a constraint violation conflict.

7.4.5 Missing/Obsolete Parameter Merge Conflicts

Example Conflict

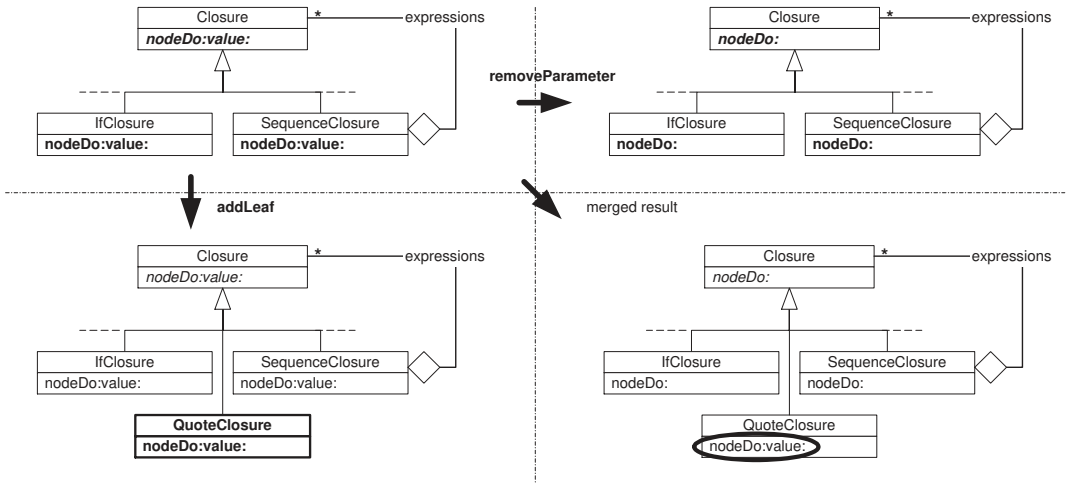


Figure 7.8: An *obsolete parameter* merge conflict

$$addParameter(class, selector, name) \parallel addLeaf_a(leafName) \Rightarrow$$
$$missingParameter(class, selector)$$
$$if$$
$$inherits^*(class(leafName), class), \wedge selector \in \mathcal{M}_a \wedge$$
$$implements(class(leafName), selector)$$

$$removeParameter(class, selector, name) \parallel addLeaf_a(leafName) \Rightarrow$$
$$obsoleteParameter(class, selector)$$
$$if$$
$$inherits^*(class(leafName), class) \wedge selector \in \mathcal{M}_a \wedge$$
$$implements(class(leafName), selector)$$

Table 7.11: Conditions giving rise to a *missing parameter* or an *obsolete parameter* merge conflict

Figure 7.8 shows an example of an *obsolete parameter* merge conflict. In this particular case, it is caused because one developer decides to remove a parameter that is passed to the `nodeDo:value:` method in the `Closure` class hierarchy. At the same time, another developer applies an *addLeaf* transformation and thus extends the same class hierarchy with a new `QuoteClosure` class. The transformation will make sure that the newly added class adds all necessary methods, and will thus ask the developer to provide an implementation for the `nodeDo:value:` method to be defined in the `QuoteClosure` class.

The end result of merging both changes is shown in the lower right part of the figure. Clearly, the `QuoteClosure` class contains a method `nodeDo:value:`, whereas all other classes residing in the same class hierarchy implement an `nodeDo:` method, without the extra parameter.

Note that such conflicts can not be detected by an ordinary compiler, since it does not know that the `nodeDo:value:` and `nodeDo:` methods are actually related and have the same responsibilities.

Formal Definition

Whenever a *addParameter* refactoring is applied in parallel with an *addHookClass* or *addTemplateClass* transformation, a *missing parameter* merge conflict can arise. If, for example, the *addParameter* adds a parameter to a *hookMethod* participant in a metapattern instance, and a new *hookLeaf* participant is added to that hierarchy that implements that method participant, the conflict occurs. The reason is that the new *hookLeaf* participant implements a *hookMethod* participant with the old signature, which lacks the particular parameter that is added by the *addParameter* refactoring. Such conflict also occurs when a parameter is added to a *templateMethod* participant and a *templateLeaf* participant is added.

The *obsolete parameter* merge conflict is similar to the *missing parameter* merge conflict, except that it occurs whenever a *removeParameter* refactoring is applied in unison with an *addHookClass* or *addTemplateClass* transformation.

The formal definition of the conditions that give rise to both conflicts are listed in Table 7.11. Note that an *addParameter* or *removeParameter* refactoring is always applied to the highest class in the class hierarchy that defines the method (as was discussed in Section 7.2.1). Therefore, the conditions do not include the case where this class resides in the hook or template hierarchy participant of the metapattern instance, as this is not possible.

Discussion

Both *missing parameter* and *obsolete parameter* merge conflicts are structural merge conflicts. The resulting design of the merged version of the framework clearly contains inconsistencies. In

the example presented above, all `Closure` classes implement a `nodeDo:` method, whereas only the `QuoteClosure` class implements a `nodeDo:value:` method and does not implement a `nodeDo:` method. Clearly, this is a structural inconsistency in the design.

Missing parameter as well as *obsolete parameter* merge conflicts can be resolved if we imply a strict order on the transformations. First, the transformation that adds a class to the framework should be applied. This class will then provide an implementation for all appropriate methods with the appropriate signature. Then, the *addParameter* or *removeParameter* refactoring can be applied, which will change the signature of the involved method. In this way, the signature of this method in the newly added class will also be changed and the conflict will disappear.

7.5 Summary

In this chapter, we have shown how merge conflicts due to the parallel application of metapattern-specific transformations and refactorings can be detected and resolved. A total of 15 important merge conflicts were defined, based on mutually comparing the metapattern-specific transformations and refactorings that were applied in parallel. We classified these conflicts into four categories: naming, constraint violation, structural and behavioral merge conflicts. We observed that structural and constraint violation merge conflicts can be resolved by imposing an order on the application of the transformations and refactorings. Naming and behavioral merge conflicts on the other hand are harder to resolve automatically. In the case of a naming merge conflict, it is clear that automatically renaming an entity is not a good option. Behavioral merge conflicts concern the behavior of methods, and our model does not include the necessary information for resolving them.

Chapter 8

Validation

In this chapter, we will perform experiments on a real-world framework to validate the claims made in previous chapters. We will discuss how our formal model and the high-level transformations that it defines can be used in practice and allows us to provide valuable support for framework-based development. The concrete framework we use to validate our approach is HotDraw, a popular and successful object-oriented framework.

8.1 Introduction

Up until now, we illustrated the main ideas of our approach using the Scheme framework. This framework was perfectly well suited as a proof of concept, since it was relatively small and we developed it ourselves and thus knew all its nitty-gritty details. In order to validate our approach and the claims we put forward in previous chapters, we should consider performing experiments on a real-world framework.

HotDraw [Bra95] is a small-scale framework in the domain of structured drawing editors. It is a very popular, successful and well-documented framework, that is considered as the reference example of how framework-based development can leverage its touted benefits. Not surprisingly, numerous applications have been derived from the HotDraw framework, amongst others class diagrams editors, music composition editors and blueprint editors. Many different versions of the HotDraw framework exist, for many different programming languages (VisualWorks Smalltalk, Squeak, Java, ...).

We choose HotDraw as a case to perform some initial experiments, exactly because we focus on language engineering issues first, as opposed to mere software engineering issues. As such, we require a controlled setting to allows us to experiment with our environment that supports framework-based development. This controlled setting ensures that we can assess the strengths and weaknesses and that we can further extend and fine tune our environment as appropriate. This would be much harder, or even impossible, with a large, complex and mostly undocumented industrial framework, that is subject to various changes and evolutions during our experiments.

8.1.1 Approach

For the purpose of studying evolution, we will consider versions 4.0 and 4.5 of the HotDraw framework, implemented in VisualWorks Smalltalk. The former version was developed in 1996, and evolved into the latter in 1999. We will first show how version 4.0 can be documented by means of design patterns and illustrate how this documentation can be used actively to derive a concrete application from the framework, that fits into the framework's design and adheres to all appropriate design constraints. Then, we will use the design pattern constraints to verify whether the implementation of the framework corresponds to its intended design. As it turns out, various design flaws and inconsistencies will be detected, that were not identified by the original developers

of the framework. Afterwards, we will discuss the specific transformations and refactorings that were (assumedly) applied to evolve version 4.0 into version 4.5, and assess how these changes impact the framework itself, as well as its existing customizations. Once again, we will observe that some conflicts that are reported by our environment remained unnoticed to the developers.

It is important to note that we actually simulate how version 4.0 was evolved into version 4.5. Unfortunately, there is no explicit and detailed documentation available about this process. We do not know which changes were applied manually, and which changes were applied through refactorings. Moreover, since framework- and design pattern-specific transformations were not available at the time the framework was evolved, most certainly, these have not been used. Many of the changes that we identified can be expressed as design pattern-specific transformations or refactorings, however, as we will illustrate. Additionally, version 4.5 of the HotDraw framework is correct, e.g. instances have been derived which exhibit the appropriate behavior. As such, presumably there are no behavioral merge conflicts between the two versions of the framework, only structural inconsistencies and constraint violations. This does not mean that our techniques are not valuable, however. Without any decent support, detecting and resolving such conflicts is a time-consuming and error-prone process. Proof is that our techniques reported conflicts, that remained undetected by the original framework developers.

8.2 Overview of the HotDraw framework

This section presents an overview of the design and implementation of the HotDraw framework. We will not provide an in-depth discussion. Rather, we first present a general high-level overview of the framework and how it evolved, and afterwards discuss the design pattern instances it uses. For a more detailed discussion about the framework, we refer to John Brant's Masters Thesis [Bra95].

8.2.1 General Overview

Each version of HotDraw comes with an implementation of the framework itself, as well as a number of example applications that show how the framework can be used. In version 4.0, the framework consists of 79 classes, whereas the sample applications contain 35 additional classes. Version 4.5 only contains 39 classes, and the sample applications amount to 30 classes. These numbers indicate that the framework itself has evolved considerably, since only half of the classes remain in the latest version. The size of the example instances more or less remained stable.

At the global level, we can identify two reasons for the simplification of the framework. First of all, version 4.0 uses the *SkyBlue* and *ColbaltBlue* general constraint solver packages to implement constraints between figures in a drawing. In version 4.5, this package is removed entirely and the internal Smalltalk dependency mechanism is used instead. Second, HotDraw uses tools to manipulate figures, and the architecture of these tools has been simplified drastically between the two versions of the framework. Together, these two changes are largely responsible for the decrease in number of classes.

HotDraw has a number of important class hierarchies: the `Figure, Drawing, DrawingEditor` and `Command` hierarchies. Every application derived from the HotDraw framework consists of a drawing (or possibly a number of drawings), which contains a number of figures. A figure is a visual element that can be drawn on the screen, such as a rectangle, a circle or an ellipse. The `Figure` hierarchy is the most important and most complex hierarchy of the HotDraw framework. This is due to the fact that HotDraw comes with a large number of predefined figure classes, which are all subclasses of the abstract `Figure` class. The `Drawing` class hierarchy is a subhierarchy of the `Figure` hierarchy and contains the various kinds of drawings that applications define. Typically, an application provides its own subclass of the `Drawing` class to represent its particular kind of drawing. The `DrawingEditor` class plays the role of the view in the *Model-View-Controller* paradigm. It opens the window for a specific application and is responsible for composing the views in that window. Each application should have a specific drawing editor which contains the particular drawing that can be edited, together with a list of tools that can be used to

manipulate the drawing and the figures in it. The `Command` class hierarchy defines a number of classes that each implement a particular action that can be performed upon a drawing or a figure. A number of readily available commands are already provided, such as the `ConnectionCommand` and `BackspaceCommand`, but applications can easily define their own specific commands.

8.2.2 Design Patterns in the HotDraw framework

The *Template Method* design pattern

The HotDraw framework uses three instances of the *Template Method* design pattern: the *figureTM, cachedFigureTM* and *drawingEditorTM* instances.

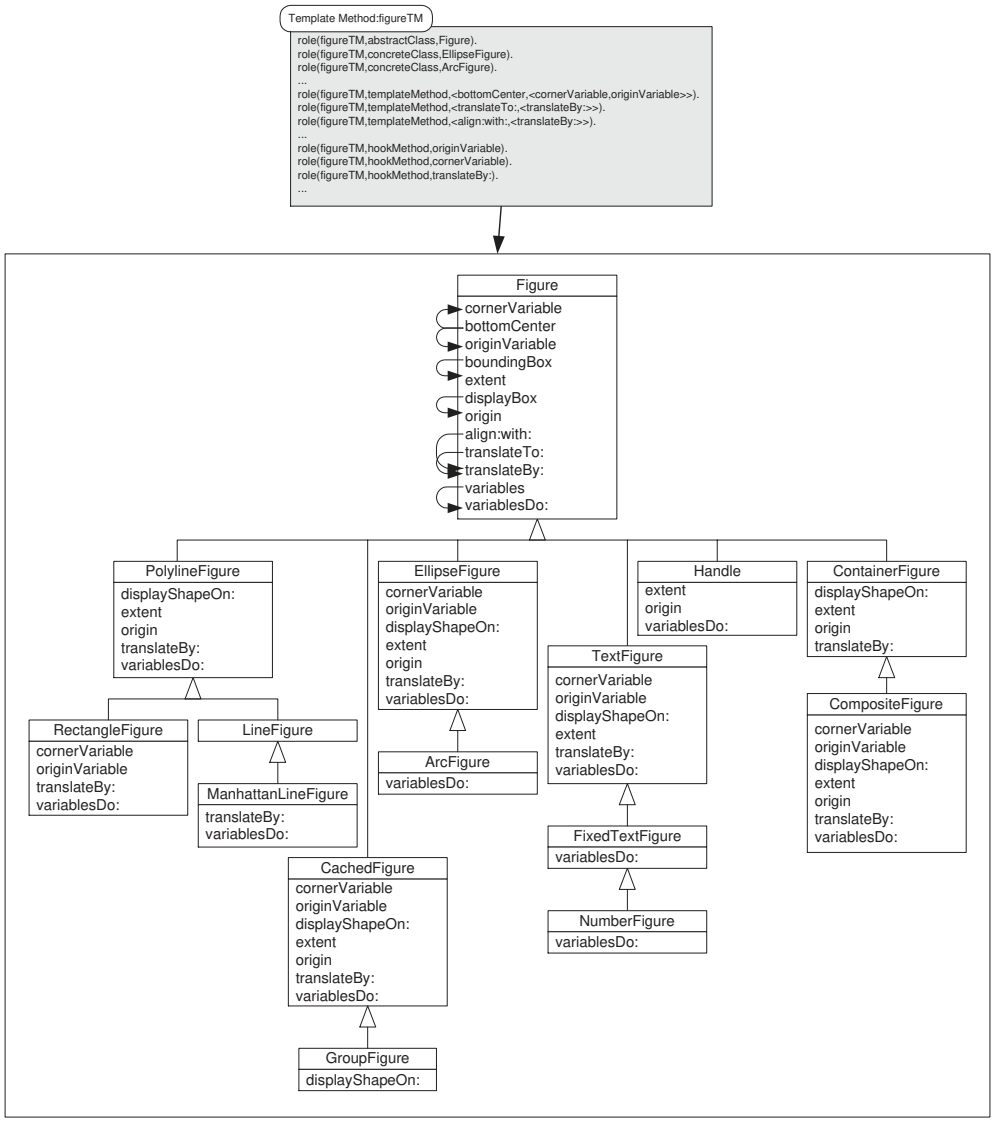


Figure 8.1: The *figureTM* instance of the *Template Method* design pattern

In the *figureTM* instance, whose (formal) specification is depicted in Figure 8.1, the *hookMethod* participants are methods that deal with constraints (the `variablesDo:` method), displaying the

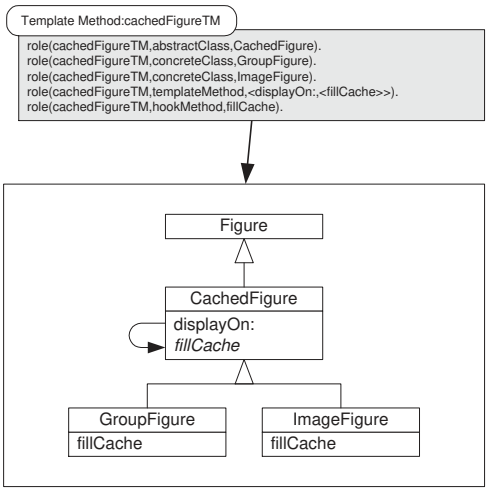


Figure 8.2: The *cachedFigureTM* instance of the *Template Method* design pattern

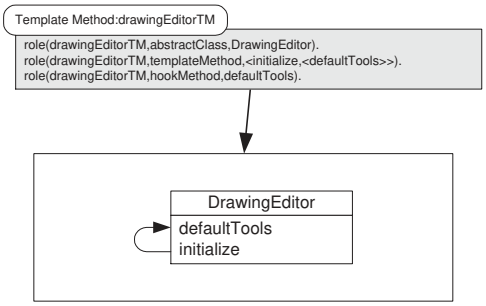


Figure 8.3: The *drawingEditorTM* instance of the *Template Method* design pattern

figure on a mask's graphics context (the `displayShapeOn:` method), moving the figure (the `translateBy:` method) and computing the bounds of the figure (the `extent` and `origin` methods). As can be seen, not all subclasses of the `Figure` class override all of these *hookmethod* participants. This is no problem, as the implementation of most of these methods in the `Figure` class provides a default behavior. A number of *templateMethod* participants are also defined in the `Figure` class, that each call the appropriate *hookMethod* participants.

The specification of the *cachedFigureTM* instance is depicted in Figure 8.2. The `CachedFigure` class, which resides in the `Figure` hierarchy, represents complex figures. By keeping a reference to the pixmap of their display in a local cache, it can be ensured that such figures display much faster. To implement this kind of behavior, the `displayOn:` method of the `CachedFigure` class relies on an abstract `fillCache` method, that is responsible for filling the cache. This method needs to be overridden by concrete subclasses that need to provide their own behavior.

The *drawingEditorTM* instance, whose specification is depicted in Figure 8.3, is used to enable each application to provide its own kind of tools. The `DrawingEditor` class holds the list of tools that are defined for a particular drawing. Each kind of drawing may need different kinds of tools, and thus the `DrawingEditor` class defines a method `defaultTools` that returns a list of tools appropriate for the drawing that is manipulated. Subclasses of the `DrawingEditor` class can override this method to register additional tools, if this is required.

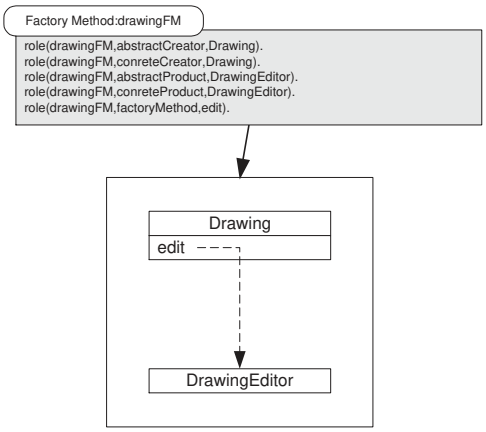


Figure 8.4: The *drawingFM* instance of the *Factory Method* design pattern

The *Factory Method* design pattern

Two instances of the *Factory Method* design pattern can be found in the framework: the *drawingFM* and *drawingEditorFM* instances.

The *drawingFM* instance is used to connect the `Drawing` hierarchy with the `DrawingEditor` hierarchy. Each application defines its own `Drawing` and `DrawingEditor` classes, and a drawing object should be opened in the appropriate drawing editor. Therefore, each `Drawing` class defines an `edit` method, that instantiates and opens the corresponding `DrawingEditor` class. The specification of this design pattern instance is depicted in Figure 8.4.

Conversely, each `DrawingEditor` class has an associated `Drawing` class. When a new drawing editor is opened, it should thus contain an instance of the appropriate drawing object. Therefore, the `DrawingEditor` class implements a `drawingClass` method that returns the class of the drawing that should be instantiated. This method should be overridden by all concrete subclasses of the `DrawingEditor` class. Figure 8.5 shows the specification of this instance.

Note that the `edit` and the `drawingClass` methods are actually related. The implementation of the `edit` method in the class returned by the `drawingClass` method should instantiate an instance of the class that defines that `drawingClass` method. This is a constraint that should hold at all times, but it can not be expressed by means of a design pattern constraint. As such, violations of this constraint will not be detected by our environment. Note that it is possible to express this constraint in our environment, since it is open to extensions. However, as opposed to the metapattern constraints, this constraint is specific to the HotDraw framework. It should thus be added by the framework developers, and it can not be reused in other frameworks.

The *Command* design pattern

There are two instances of the *Command* design pattern: the *readerCommand* and the *drawingEditorCommand* instances, which are depicted in Figure 8.6 and 8.7. In the former instance, a `Command` object is told by a `Reader` object to perform its action, by sending it the `effect` method. Each subclass of the `Command` class can override this method in order to provide the appropriate behavior. The latter instance is used to allow a command to be undone, by invoking its `unexecute` method. This method can be overridden in subclasses of `Command` as well.

Note how their is a lot of duplication in the specification of both design pattern instances. This is due to the fact that the `Command` class hierarchy needs to be specified twice, since it participates in both instances. This problem can be alleviated to some extent. We currently require a developer to provide the complete specification manually. The environment could provide some help, however, and prompt him for the root of the class hierarchy that needs to be specified. It would then

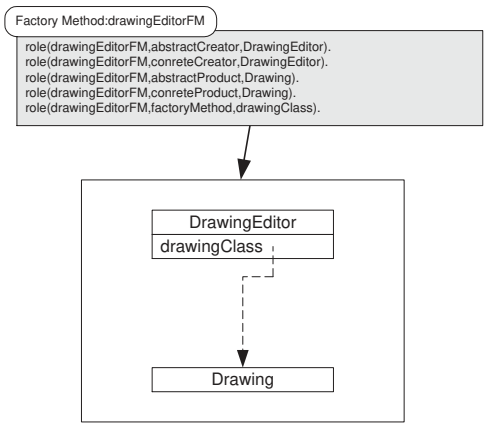


Figure 8.5: The *drawingEditorFM* instance of the *Factory Method* design pattern

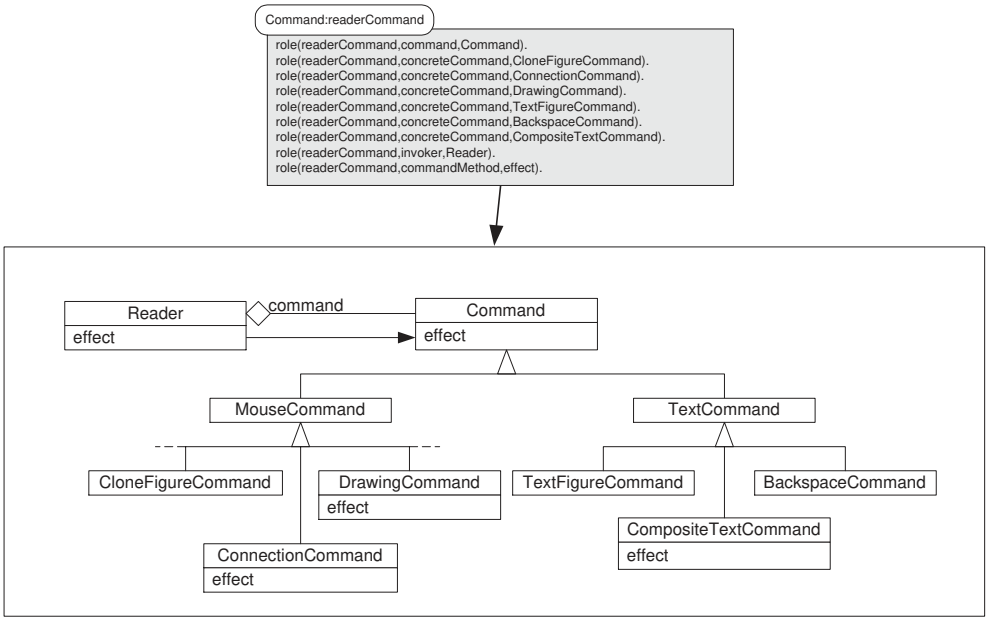


Figure 8.6: The *readerCommand* instance of the *Command* design pattern

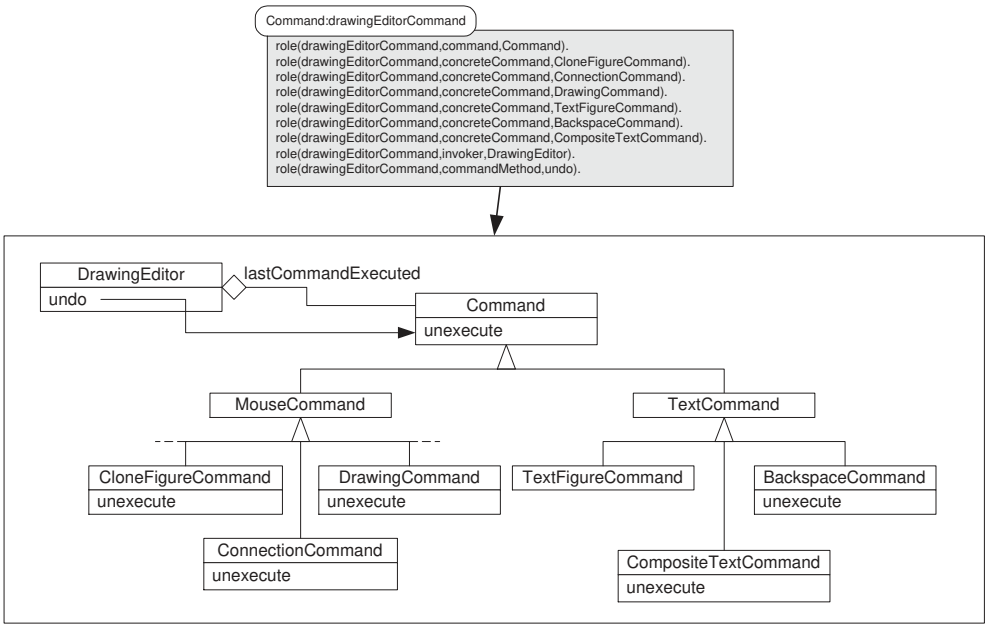


Figure 8.7: The *drawingEditorCommand* instance of the *Command* design pattern

compute the specification automatically based on this information. This is possible since full access to the source code of the framework is provided. The result of this computation can then be presented to the developer, who can accept it as is, or can modify it as needed.

The *Composite* design pattern

Three instances of the *Composite* design pattern are used in the framework: *compositeCommand, compositeTextCommand* and *compositeFigure*.

The `Command` class hierarchy contains mouse commands and text commands. The former are responsible for manipulating figure objects with the mouse, whereas the latter are used for manipulating text figures. Mouse commands are implemented by the `MouseCommand` class hierarchy, while text commands reside in the `TextCommand` hierarchy. Both class hierarchies use an instance of the *Composite* design pattern (as can be seen from the specifications shown in Figures 8.8 and 8.9) to allow us to define more complex commands by combining simpler ones. A `MouseCommand` object should implement the `initialize:` and `moveTo:` methods, which are responsible for the initialization of the command and for taking appropriate action when the mouse (and thus the figure upon which the command acts) is moved. A text command should also implement an initialization method (`initialize` in that case) and a `keyboardEvent:` method, that determines what should be done when the user uses the keyboard.

The *compositeFigure* instance of the *Composite* design pattern is used in the `Figure` class hierarchy, as depicted in Figure 8.10. The `ContainerFigure` class is an abstract class that represents a figure that can contain any number of other figures, and that can be manipulated as a whole. Different concrete subclasses of this class exist, in the framework (the `Drawing` class, for example) as well as in concrete applications (the `PERTEvent` class, for instance). The interface of this instance of the design pattern contains the following methods: the `translateBy:, displayOn:, displayShapeOn:, extent, origin, figureAt:` and `figuresIn:` methods. These have a standard implementation in the `ContainerFigure` class, and are overridden in the many concrete subclasses of the `Figure` hierarchy.

Note again how there is a lot of duplication in the specification of the *compositeFigure* instance and that of the *figureTM* instance. Once again, this is due to the fact the same classes and methods

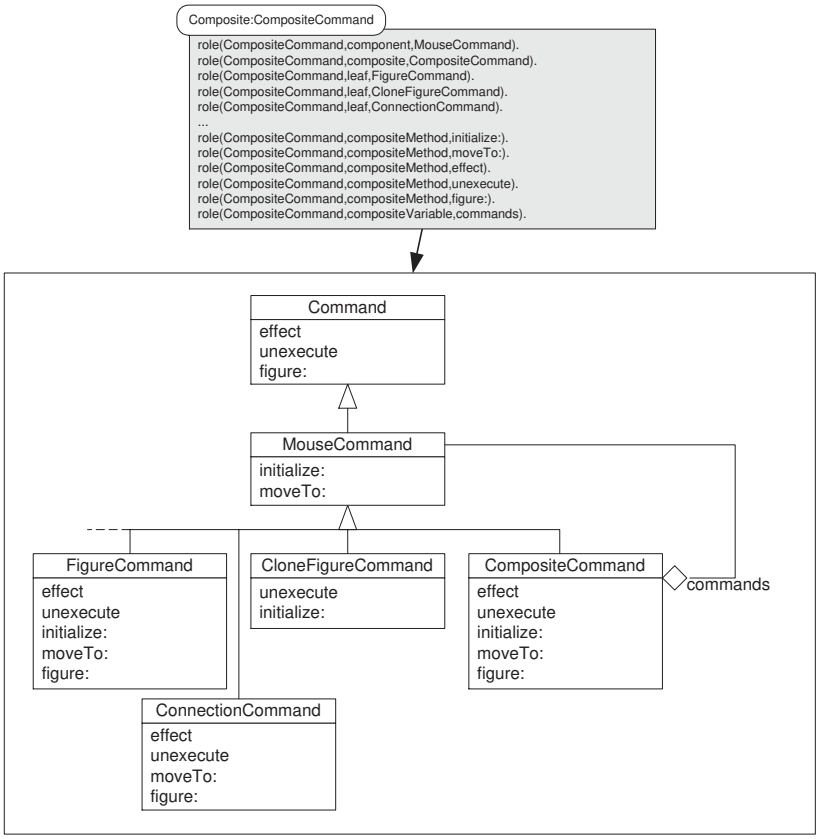


Figure 8.8: The *compositeCommand* instance of the *Composite* design pattern

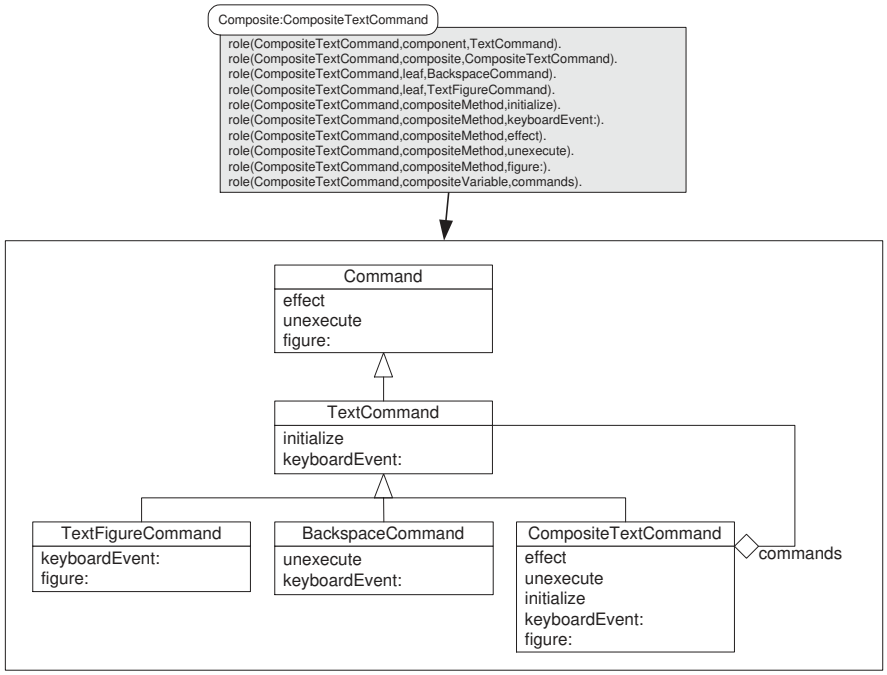


Figure 8.9: The *compositeTextCommand* instance of the *Composite* design pattern

participate in these instances.

The *Proxy* design pattern

The `LayeredContainerFigure` class hierarchy uses an instance of the *Proxy* design pattern, whose specification is shown in Figure 8.11. Figures in a drawing are not stored in a `Drawing` object. Instead, such an object maintains a reference ot a `ConstraintDrawing` object, that contains those figures together with their associated constraints. Methods in the `Drawing` class' interface simply delegate to the appropriate method of the `ConstraintDrawing` class.

The *Model-View-Controller* design pattern

The *Model-View-Controller* design pattern is used to separate the model from the way it is represented to the user (the view) and from the way in which the user controls it (the controller). In the *mvcDrawing* instance that is used in the framework (depicted in Figure 8.12), the `Drawing` class represents the model, the `DrawingView` class the view and the `DrawingController` class the controller. An instance of the `DrawingView` class maintains a reference to a `Drawing` object and its `displayModelOn:clipped:` method uses this reference to call the `displayOn:` method that effectively displays the drawing on the screen. A `DrawingController` object receives mouse and keyboard events and translates these into messages that the model object understands. Upon receipt of these messages, the model takes appropriate action and updates its views appropriately (the `damageRegion, repairDamage` and `update:with:from:` methods) .

8.2.3 Discussion

Since % of the classes in the HotDraw framework participates in one or more design pattern instances, and since each class hierarchy is documented by means of these design pattern instance specifications, we can conclude that the documentation of the framework's design by means of design patterns is quite accurate. As such, if those hierarchies are extended with a new class,

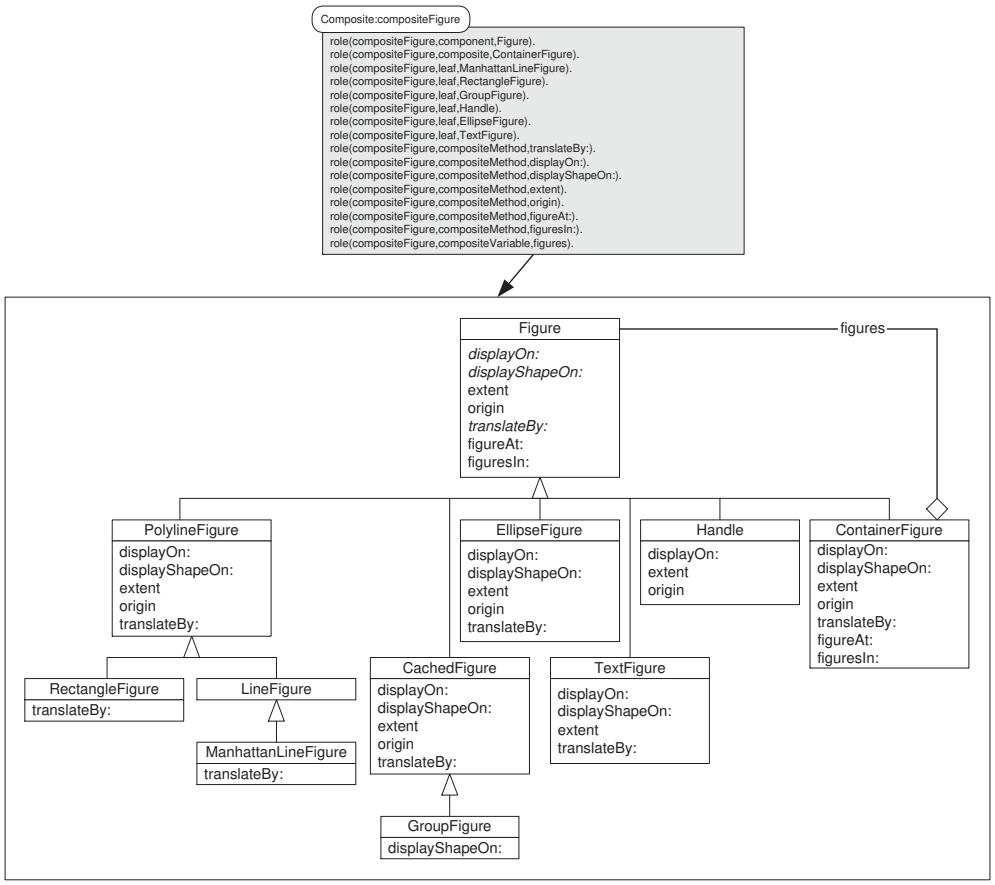


Figure 8.10: The *compositeFigure* instance of the *Composite* design pattern

the documentation provides useful information about where the new class should be placed in the hierarchy and which methods it should at least implement. This will be illustrated in detail in Section 8.3.2.

The design pattern instances capture most of the important relationships between class hierarchies, classes and methods in the framework. As such, the design pattern constraints can be used to detect when these relationships are not adhered to after changes have been applied manually. Furthermore, our change propagation algorithm will be very effective, since it is entirely based upon the explicit documentation of these relationships.

As we have illustrated, some relationships can not be documented by means of design patterns, however. This can be due to an inferior design choice of the developers, or to the fact that no design pattern exists that can be used to model the specific relationship. For example, there exists a relationship between the `Drawing` and the `Figure` class (as shown in Figure 8.13), because a drawing object is responsible for displaying a figure's handles. There is no design pattern that can describe this particular relationship, however. Thus, no support will be provided to check the correctness of this relationship after changes have been applied manually, nor will the transformations automatically make sure that the developer keeps this relationship satisfied. Since our supporting environment is extensible, a framework developer could add framework-specific constraints and could define appropriate ad hoc transformations.

We also observed that some of the design pattern specifications contained a lot of duplication. This occurs in particular when a large class hierarchy (such as the `Figure` hierarchy) participates in

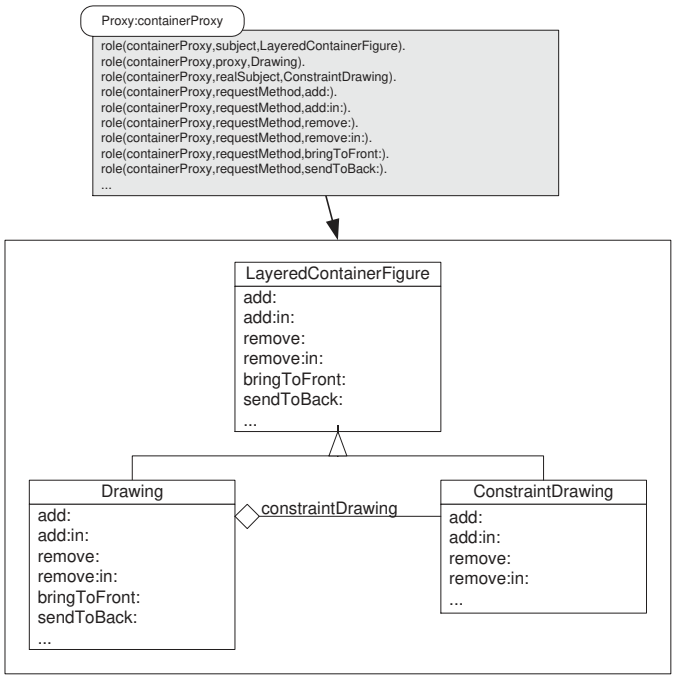


Figure 8.11: The *containerProxy* instance of the *Proxy* design pattern

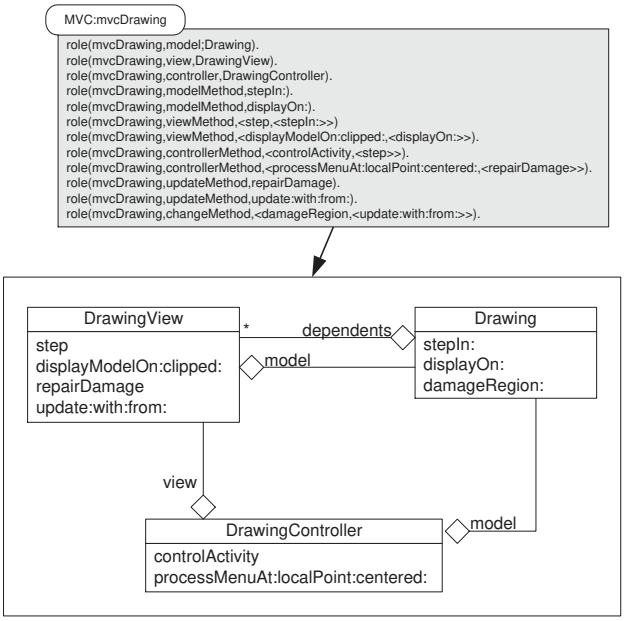


Figure 8.12: The *mvcDrawing* instance of the *Model-View-Controller* design pattern

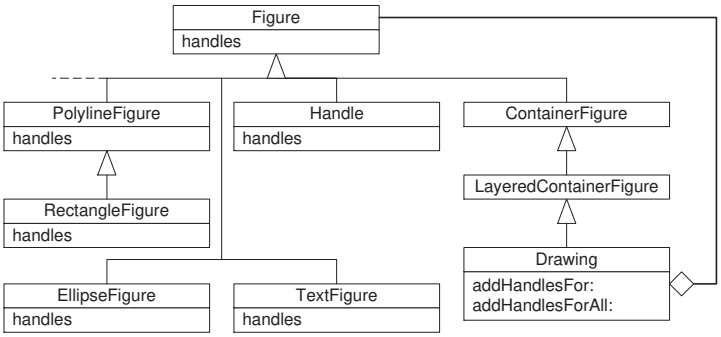


Figure 8.13: Relationship between the `Drawing` and the `Figure` classes

many design pattern instances simultaneously. Such duplication is hard to avoid without changing our formal model. However, by extending the environment, the overhead of specifying design pattern instances can be lessened. Based on a small amount of basic information, the environment could compute a specification automatically, and could present the result to the developer for inspection. This developer can then either accept the specification as is or can modify it as needed.

8.3 Support for Framework Instantiation

Now that we have seen which design patterns are used in the HotDraw framework and how they can be documented, we will show how this information helps us when deriving a concrete application. We will first explain the framework-specific instantiation transformations that exist. Afterwards, we will illustrate how these framework-specific instantiation transformations can be used to help a developer when instantiating the framework, and thereby show how they are defined in terms of design pattern-specific transformations.

8.3.1 Framework Instantiation Transformations

A typical application derived from the HotDraw framework should provide subclasses for the `DrawingEditor` and `Drawing` classes. Furthermore, if the application uses specialized figures, these should be defined as subclasses of the `Figure` class, or one of its subclasses. In that case, corresponding tools that instantiate and manipulate such figures should be provided as well. This boils down to providing appropriate subclasses of the `MouseCommand` and `TextCommand` classes. Thus, the following framework-specific instantiation transformations would be useful for an application developer, to guide him through the instantiation process and make sure he ends up with a correct instance.

`addFigure(FigureClass, Superclass)` adds a new class `FigureClass` as a subclass of the class `Superclass`. This `Superclass` class should reside in the `Figure` class hierarchy.

`addCachedFigure(FigureClass, Superclass)` adds a new class `FigureClass` as a subclass of the `Superclass` class. This superclass should be a subclass of the `CachedFigure` class, or that class itself.

`addCompositeFigure(FigureClass)` adds a new class `FigureClass` as a subclass of the `CompositeFigure` class.

`addDrawing(DrawingClass)` adds a new class `DrawingClass` as a subclass of the `Drawing` class.

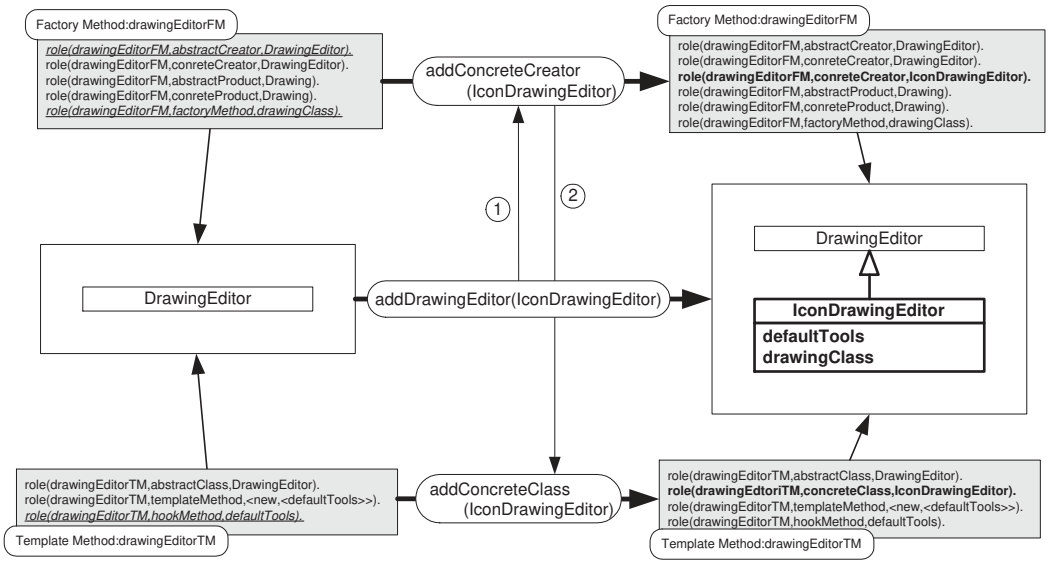


Figure 8.14: Adding the `IconDrawingEditor` class

`addDrawingEditor(DrawingEditorClass)` adds a new class `DrawingEditorClass` as a subclass of the class `DrawingEditor`.

`addMouseCommand(MouseCommandClass,Superclass)` adds a new class `MouseCommandClass` as a subclass of the given `Superclass` class, which should reside in the `MouseCommand` class hierarchy.

`addTextCommand(TextCommandClass,Superclass)` adds a new class `TextCommandClass` as a subclass of the given `Superclass` class. This `Superclass` class should reside in the `TextCommand` class hierarchy.

These framework-specific instantiation transformations are defined by a framework developer, who should also specify how they are to be mapped onto the appropriate design pattern-specific transformations. For example, the *addCachedFigure* transformation is mapped onto an *addConcreteClass* transformation that operates on the *cachedFigureTM* instance of the *Template Method* design pattern. Remember from Section 4.3 that, when a framework-specific transformation is applied, this is translated automatically into a design pattern-specific transformation, which in turn is translated into a metapattern-specific transformation that performs the actual changes to the source code.

8.3.2 Deriving an Instance

In this section, we will show how the *VisualCreation* instance, that is provided as a sample application with the HotDraw framework, can be derived by using the framework-specific instantiation transformations. This instance implements a graphical editor that allows a user to easily build and integrate new tools for the specific instance of the HotDraw framework that he wants to implement. In what follows, we will show how our supporting environment interactively guides the application developer while he implements the application.

Adding a new drawing editor

To add the `IconDrawingEditor` class as a subclass of `DrawingEditor`, the application developer invokes the *addDrawingEditor* framework-specific instantiation transformation. This transforma-

tion will require the developer to provide an implementation for two important methods that the new class should override: the `defaultTools` and the `drawingClass` methods.

Figure 8.14 shows how the environment first translates the *addDrawingEditor* framework instantiation transformation into an *addConcreteCreator* design pattern-specific transformation on the *drawingEditorFM* instance (step 1). This transformation registers the `IconDrawingEditor` class as a *concreteCreator* participant in the design pattern instance and asks the developer to provide an implementation for the `drawingClass` method. Afterwards, the environment searches for all overlapping design pattern instances and invokes the appropriate design pattern-specific transformation upon them (step 2). In this case, the *drawingEditorFM* instances overlaps only with the *drawingEditorTM* instance. As such, an *addConcreteClass* design pattern-specific transformation is applied to this instance, which registers the `IconDrawingEditor` class as a *concreteClass* participant in the design pattern instance, and asks the developer to provide an implementation for the `defaultTools` method participant, which is then added to the `IconDrawingEditor` class.

Adding a new drawing

A specific drawing editor class is always associated with a particular drawing class. As such, the application developer instantiating the *VisualCreation* instance could also use the *addDrawing* framework-specific instantiation transformation to add a `IconDrawing` class. However, he does not have do to so explicitly, because the `DrawingEditor` and `Drawing` class hierarchies are connected via the *drawingEditorFM* design pattern instance (see Section 8.2.2). The *addConcreteCreator* transformation that was applied to this instance will automatically invoke an *addConcreteProduct* transformation, since each *concreteCreator* participant should be associated with a *concreteProduct* participant (this is a constraint imposed by the *Factory Method* design pattern).

In this particular case, the *addConcreteProduct* transformation will ask the developer for the specific drawing class that corresponds to the `IconDrawingEditor` and that should be returned by the `drawingClass` factory method. The result is that a new class `IconDrawing` class is added to the instance as a subclass of the `Drawing` class, and that this class is added as a *concreteProduct* participant in the design pattern instance (see Figure 8.15).

In order to know which methods this new class should implement, our supporting environment checks in which design pattern instances the `Drawing` class participates. As it turns out, this class is a *concreteCreator* participant in the *drawingFM* instance, a *concreteModel* participant in the *mvcDrawing* instance, a *concreteClass* participant in the *figureTM* instance and a *leaf* participant in the *compositeFigure* instance. The appropriate transformations are thus applied on these instances (see Figure 8.15), and the environment asks the developer to provide an implementation for the appropriate method participants (those method participants are underlined and in italics in Figure 8.15). The `IconDrawing` class should only define the `edit` method, for the implementation of all other methods, it relies on its superclasses. This is a deficiency in our approach: the environment asks the developer to provide an implementation for 16 methods, whereas he only needs to specify one. This is due to the fact that the transformations assume that a leaf participant should always implement all appropriate method participants. Especially with large class hierarchies that participate in many design pattern instances, this turns out to be a rather simplistic view. The problem can only be alleviated by adopting a more realistic approach, where, a developer can specify which parts of the hierarchy need to implement which method participants, for example. This requires some extensions to the formal model however, which is considered future work.

Note that, the *drawingProxy* design pattern instance does not overlap with the *drawingEditorFM* instance, although the `Drawing` class participates in both instances. None of the overlapping conditions of Section 5.5.3 are satisfied, however.

Adding new figures

The *VisualCreation* application introduces six new types of figures. Three of these figures are subclasses of the `CachedFigure` class, and as such are added to the instance by means of the

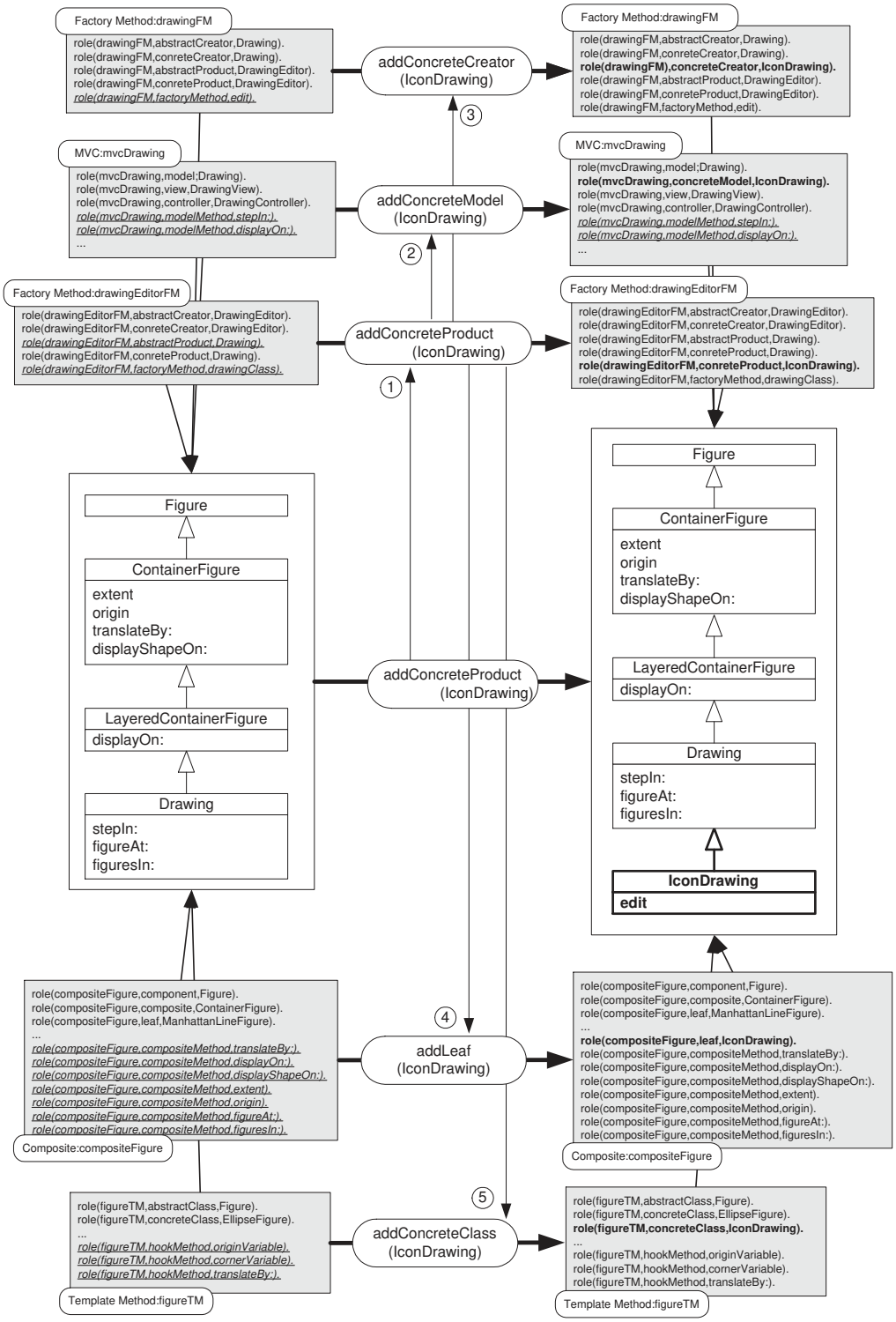


Figure 8.15: Adding a new drawing class

addCachedFigure instantiation transformation. The other three figures are subclasses of the `RectangleFigure` and are thus added by applying the *addFigure* instantiation transformation.

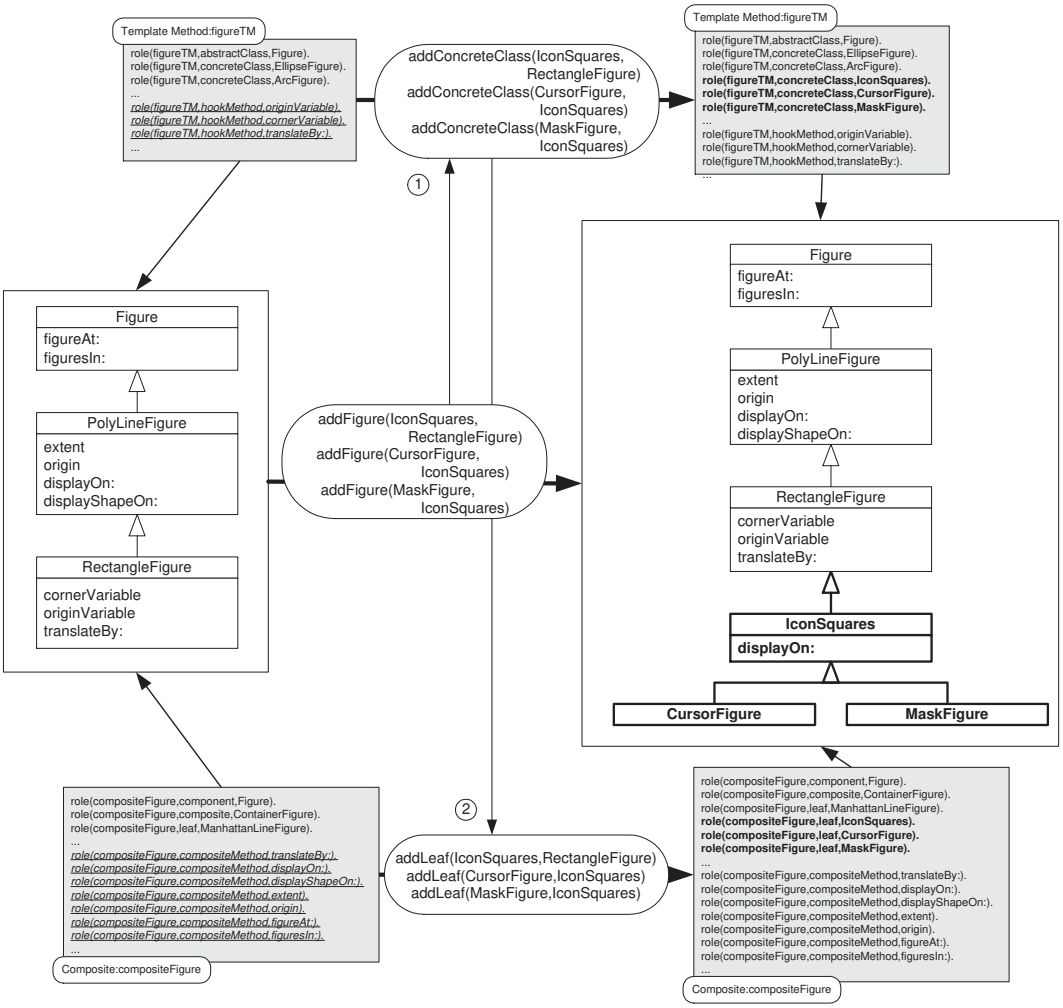


Figure 8.16: Adding a new figure class

As can be seen in Figure 8.16, the *addFigure* framework instantiation transformation is translated into an *addConcreteClass* design pattern-specific transformation that operates on the *figureTM* instance. This will add `IconSquares` as a subclass of `Rectangle`, and afterwards add `MaskFigure` and `IconFigure` as subclasses of `IconSquares`. It will also register these three new classes as *concreteClass* participants in the design pattern instance.

Afterwards, the supporting environment considers all design pattern instances that overlap with the *figureTM* instance. In this case, there is only one such instance: the *compositeFigure* instance. The *addConcreteClass* transformation will thus give rise to an *addLeaf* transformation on this instance, and as a result, the three new classes will be registered as *leaf* participants in it.

While applying these design pattern-specific transformations, not only are the three classes

registered as participants in the appropriate instances, the application developer is also asked to provide an implementation for the method participants occurring in those instances. For the `IconSquares` class, he only provides an implementation for the `displayOn:` method and relies on the implementation in the superclasses for all other methods. The `MaskFigure` and `IconFigure` class do not need to define additional methods.

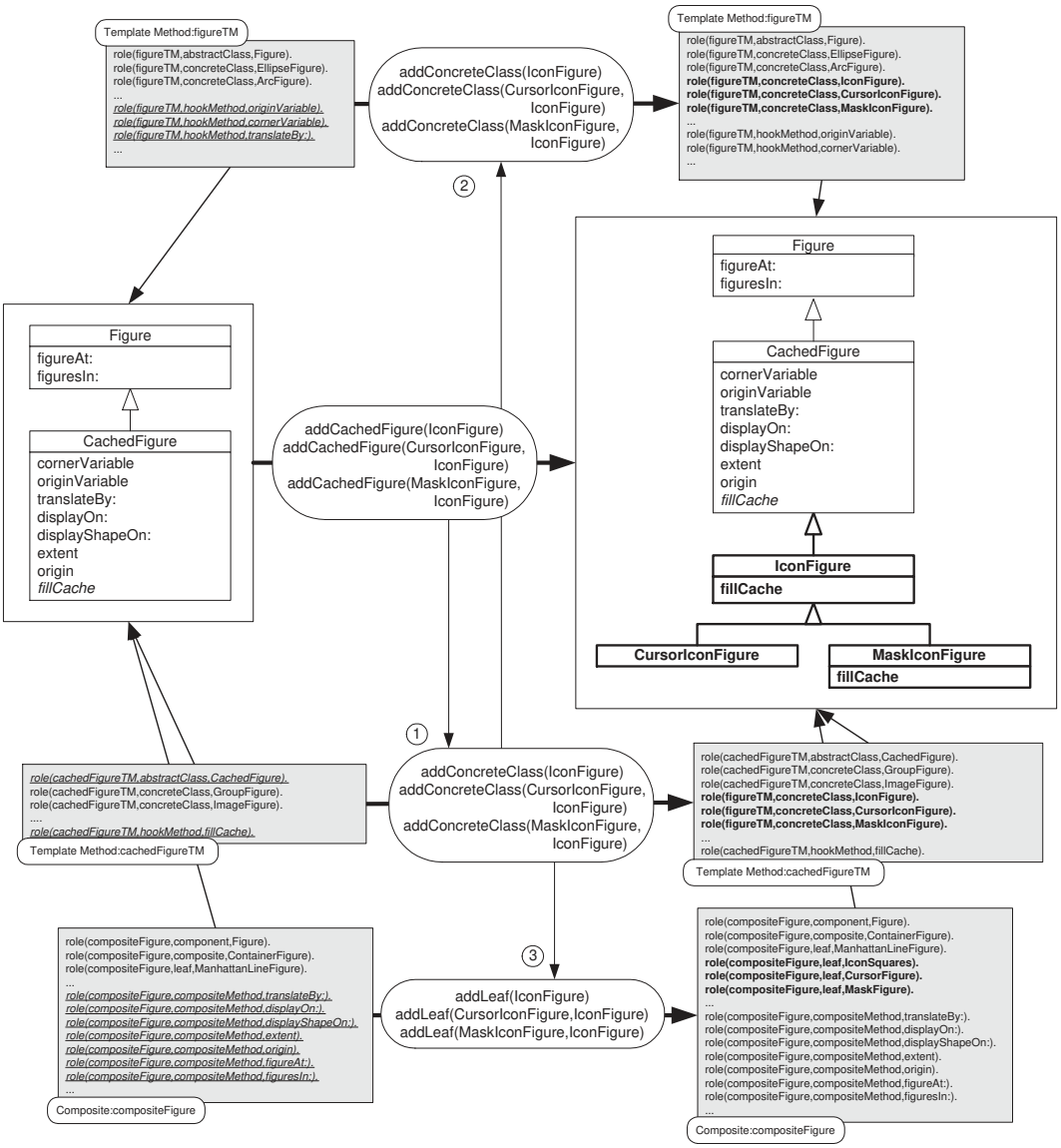


Figure 8.17: Adding a new cached figure class

Figure 8.17 shows how the *addCachedFigure* framework instantiation transformation is applied to introduce the `IconFigure, MaskIconFigure` and `CursorIconFigure` classes as subclasses of the `CachedFigure` class. This framework instantiation transformation is translated into an *addConcreteClass* design pattern-specific transformation that operates on the *cachedFigureTM* instance. It adds the three classes as *concreteClass* participants to the instance, and asks the developer to provide an implementation for the `fillCache` method participant. The `IconFigure` and `MaskIconFigure` classes provide an appropriate implementation, whereas the `CursorIconFigure` class can simply use the implementation provided by the `IconFigure` class and does not need to

provide one of its own.

Furthermore, the *cachedFigureTM* design pattern instance overlaps with a number of other design pattern instances. As such, applying an *addConcreteClass* transformation gives rise to a number of other transformations on these overlapping instances. As can be seen in Figure 8.17, an *addConcreteClass* transformation is applied upon the *figureTM Template Method* design pattern instance, and an *addLeaf* transformation is applied on the *compositeFigure* instance. All these transformations register the three new classes as the appropriate participants in the design pattern instances, and consult the developer to provide an implementation for the required methods. As it turns out, the implementation of all methods in the superclasses (`CachedFigure` and `Figure`) is already appropriate for the new classes, and no additional methods need to be defined in the new classes.

Once again, in both cases, the environment asks the developer for an implementation for a large number of methods (13 and 14 methods respectively), while he only provides one implementation for such method in each case.

Adding a new mouse command

New mouse commands are added to an instance of the framework by means of the *addMouseCommand* framework instantiation transformation. This transformation is translated into an *addLeaf* design pattern-specific transformation on the *Composite* design pattern, as can be seen from Figure 8.18. An `IconFigureCommand` is thus added as a leaf participant to the *compositeCommand* instance, and is added as a subclass of the `MouseCommand` class in the implementation. Furthermore, all leaf participants of the *compositeCommand* instance should implement the `effect, unexecute, initialize:` and `moveTo:` method participants, so the developer is asked to provide an implementation for them. While the `initialize:` and `moveTo:` methods do receive an appropriate implementation, the developer decides not to provide a concrete implementation for the `effect` and `unexecute` methods, but rather relies on the implementation of these methods in the `Command` superclass.

Next, the supporting environment considers all design pattern instances that overlap with the *compositeCommand* instance and determines that this is the case for the *readerCommand* and *drawingEditorCommand* instances of the *Command* design pattern. A *addConcreteCommand* transformation is thus applied upon these instances, which adds the `IconFigureCommand` as a *concreteCommand* participant. Concrete commands should provide an implementation for the method participants of these instances, in this case, the `effect` and `unexecute` methods. Since the developer already stated that these methods should not be overridden in the new command class, the transformation finishes.

Observe that the environment asks the developer twice for an implementation of the `effect` and `unexecute` methods. This is because those methods participate in two design pattern instances at once. This is again due to the simplistic implementation of the transformations: they do not take overlapping of design pattern instances into account when prompting the developer for method implementations. This can easily be changed. The environment could compute all methods that a class should implement beforehand, by considering overlapping instances, remove all duplicates and only then prompt the developer.

8.3.3 Discussion

Our approach to support framework instantiation can be seen as an "active cookbook" approach [PPSS95]. The instantiation transformations use the documentation in an active way to guide a developer when instantiating the framework. After he has invoked all of these instantiation transformations, the implementation of the *VisualCreation* instance contains a definition of all required classes, except one, the `ToolBuilder` class. This should come as no surprise. All important class hierarchies participate in one or more design pattern instances, and this allows a developer to use appropriate design pattern-specific transformations that add the required classes. The `ToolBuilder` class is an application-specific class, that does not rely on any framework class,

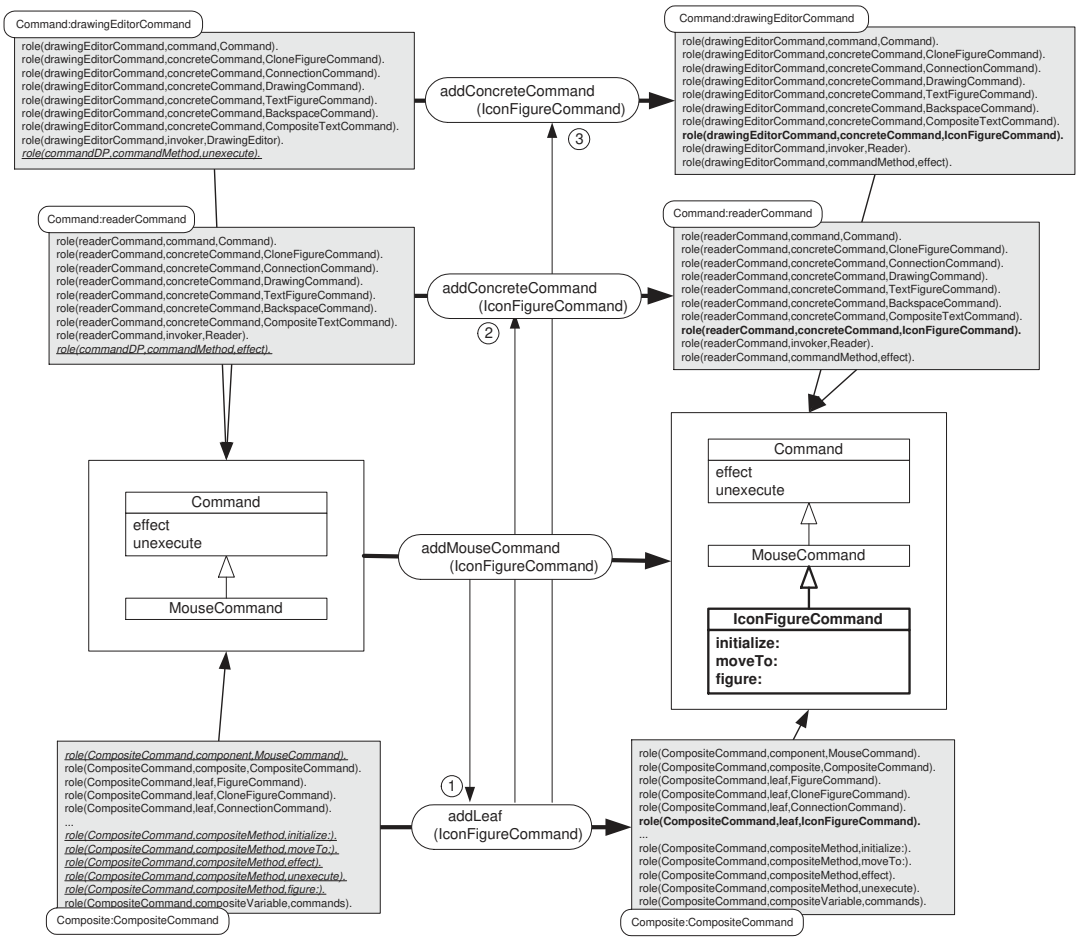


Figure 8.18: Adding a new mouse command class

does not participate in a design pattern instance and should thus be added manually. Also note how the design pattern-specific transformations make sure the specifications are kept up to date automatically.

Moreover, the classes added by the transformations already contain the appropriate method implementations for nearly all important methods they should define. Only some classes still require implementations for the `menu` and `handles` methods. The reason why our environment did not prompt the developer to implement these methods is that they do not participate in any design pattern instance. Naturally, other, class and application specific methods need to be added to finish the implementation of the classes. While we only showed how the environment guides an application developer in deriving the *VisualCreation* instance, the results can be generalized to other instances as well.

We can also observe that the only design pattern specific transformations that are used by an application developer are those that *add* class participants. This seems reasonable. In normal circumstances, an application is not allowed to *remove* classes from the framework, as this would have an impact on all other existing applications. For the very same reason, applications can not add or remove methods from the framework. In some specific cases, however, applications are allowed to change the framework. Embedded systems, for example, have to deal with limited resources, and may therefore want to reduce the overall size of an application by removing dead code and features of the framework that are not used, Typically, such applications work with a copy of the framework, which they can change without affecting other applications.

Although the framework and design pattern specific transformations can guide a developer, he still needs to be familiar with the design and the implementation of the framework to some extent. He should be able to identify where a class should be added in a class hierarchy, or which methods need an implementation and which methods can rely on an implementation in superclasses. This is no different from the situation where there is no guidance, however. Furthermore, the transformations do point him to the class hierarchies and method implementations that he should look at.

One particular impediment to our approach is that a developer is often asked to provide an implementation for method participants when this is not strictly necessary. This is caused by a deep class hierarchy with a large interface, many methods of which participate in a (number of) design pattern instance(s). Leaf classes of such deep hierarchies often only need to implement a small part of this interface, as most of its methods are already implemented at other levels of the hierarchy. In the case of the *VisualCreation* instance, this problem frequently occurred because the `Drawing` hierarchy is a subhierarchy of the `Figure` hierarchy, and therefore, all design pattern instances involving the latter hierarchy overlap with those instances involving the former. Subclasses of the `Drawing` class should normally only override method participants of a design pattern instance containing the `Drawing` hierarchy. If there was a way to represent such information in our model, this problem could be alleviated. At present, this is not possible however and resolving this issue remains future work.

A similar problem occurs because methods often participate in two (or more) overlapping design pattern instances. As a result, the environment will ask the developer for an implementation for those methods multiple times. This problem is due to the current, and naive, implementation of the transformations, however. It could be avoided if this implementation was changed so as to take such overlappings into account.

One particular important question that we did not address is whether the order in which transformations are applied matters. Since transformations on overlapping instances are invoked automatically, the developers is relieved from doing so explicitly and hence can not accidently forget to invoke one. It remains an open question whether we can compute an optimal order for the transformations, so that explicit developer intervention is reduced to a minimum.

8.4 Support for Framework Evolution

Besides being useful for framework instantiation, documentation based on design pattern specifications is also of great value when evolving the framework. In this section, we will elaborate upon this in more detail. We will first show how design drift can be avoided up to a certain extent, by checking design pattern constraints. Afterwards, we will show how the impact of specific evolutions on the framework itself and on its instances can be assessed.

8.4.1 Avoiding Design Drift

By checking the constraints associated with the design patterns used in the HotDraw framework, we are able to check whether its implementation actually adheres to the intended design. Constraint violation conflicts will be reported if the implementation is not conform with the design. In spite of the fact that HotDraw has been studied extensively, is well documented and has been used many times, we will see that it still contains some design flaws and inconsistencies that remained unnoticed and that our approach is able to detect these.

Constraint checking is especially useful when the framework is evolved manually, to ensure that a developer does not accidently break the design. In this particular case, there is no explicit information about how the first version of HotDraw was evolved in the second, nor what was done manually and what was done via automated transformations such as refactorings. By comparing the designs of both versions, we were able to infer which transformations and refactorings could have been applied. There is no way in which we can reconstruct the specific lower-level changes that were applied to the implementation manually, however. Furthermore, many design pattern

constraints are used to verify whether the design pattern specifications remain correct after a number of changes have been applied manually. They check whether a developer (un)registered the appropriate participants, for example. Such errors do not occur, because we documented the design patterns ourselves, and we can not consider manual evolution in our experiments.

Despite all these problems, we can still use the constraints to detect design inconsistencies "after the facts", as we will see in the following sections.

Conflicts in the framework's implementation

78 constraint violation conflicts are detected in the implementation of the framework. All of them occur because classes that are registered as *leaf* participants in a design pattern instance do not provide a concrete implementation for a particular method participant. They are thus *not understood by leaf* constraint violation conflicts.

This large number is partly due to the naive implementation of our constraints. *Not understood by leaf* constraint violation conflicts are reported with respect to the specific leaf participant that does not define a particular method participant. In many cases however, this conflict is reported for a number of classes that all share a common superclass. This suggests that it is in fact the particular subhierarchy that does not define the method, and as such, it suffices to report the conflict only once with respect to the common superclass. For example, 20 reported conflicts are due to the `Handle, TentativePositionHandle, ConstraintHandle, CommandHandle` and `IndexPositionHandle` classes that do not implement the `cornerVariable, originVariable, displayShapeOn:` and `translateBy:` methods. If we report this conflict with respect to the `Handle` class only, which is the common superclass of all the other classes, we would get only four reported conflicts, one for each method that is not understood.

Another reason why so many conflicts are reported is because design pattern instances overlap and the implementation of the constraints does not take this into account. Conflicts that are reported for a class and method combination in a particular design pattern instance, are also reported for the same class and method combination in another design pattern instance, if the two instances overlap and the class and method participate in both instances. For example, 9 conflicts are reported for the *readerCommand* and *drawingEditorCommand* design pattern instances, since several classes in the `Command` class hierarchy do not understand the `effect` and `unexecute` methods. The `Command` hierarchy participates in the *CompositeCommand* and *CompositeTextCommand* design pattern instances as well, as do the `effect` and `unexecute` methods (see Section 8.2.2). All conflicts that are reported for the *readerCommand* and *drawingEditorCommand* instances are thus also reported for either the *CompositeCommand* or the *CompositeTextCommand* instances. Of the 26 reported conflicts in the `Command` class hierarchy, only 17 are actually different.

When we change the implementation of the constraints to take the above considerations into account, the number of reported conflicts is considerably reduced from 78 to 34:

- 14 of these conflicts are considered to be real *not understood by leaf* conflicts. These occur because several concrete classes in the `Figure` hierarchy do not provide an implementation for the `cornerVariable` or `originVariable` methods, which are abstract methods in the `Figure` class. In Java, this would have been detected by the compiler, whereas in Smalltalk, this is not the case. A compiler can not always detect such conflicts however. For example, the `TextFigureCommand` class does not implement the `unexecute` and thus inherits the default do-nothing behavior defined in the `Command` class itself. This is clearly a bug, because when we add a text figure to a drawing, we will not be able to undo that action.

- 4 of the reported conflicts are not *not understood by leaf* conflicts, but do point at a flaw in the design. The `initialize` method that is defined in the `TextCommand` hierarchy, is never overridden in any of its concrete classes, except in the `CompositeTextCommand` class. This class is actually the *Composite* participant in an instance of the *Composite* design pattern. The implementation of the `initialize` method in this class thus simply contains a default implementation that forwards the message to the components contained within the class. The `initialize` method does thus not implement any real behavior and serves

no particular purpose. Similarly, the `effect` method that is defined in the `Command` class is never overridden in any subclass of the `TextCommand` class, except once again in the `CompositeTextCommand`, where it contains a default implementation. The question may thus be raised whether this method should be pushed down, so that it only exists in the `MouseCommand` hierarchy, where it does fit some purpose.

The 16 remaining conflicts are false positives, that occur because a particular leaf class relies on the default implementation of a method provided by another class higher up in the hierarchy.

Conflicts in the framework's instances

Constraint violation conflicts are also reported for the framework's instances. When considering the 7 instances that are provided with the framework, 14 *not understood by leaf* conflicts are detected. Of these, 5 of these conflicts are real *not understood by leaf* conflicts where a class does not define a method that it should. One conflict is due to a bad design: the `Drawing` class provides a `stepIn:` method, which should only be overridden by subclasses when they want to be animated. Since not every framework instance provides animation, this method is not always overridden. Consequently, the `IconDrawing` does not override this method, which results in an unnecessary conflict being reported. In the new version of the framework, a new class `AnimatedDrawing` is introduced, which contains all animation-specific behavior. The conflict will thus disappear in the new version.

The 8 remaining conflicts are once again false positives.

Summary

To summarize, approximately 60% (24 out of 41) of the reported constraint violation conflicts pointed at real flaws in the design of the framework, or in the implementation of the applications. This is quite surprising. We would expect that a framework that has been studied extensively, is very well documented and has been used many times, contained only very few such inconsistencies.

The remaining 40% of the detected conflicts were false positives. These conflicts often occur in deeply nested class hierarchies, where a leaf class relies on one of its superclasses to provide an implementation for a method, instead of defining that method itself. This problem is actually related to the problem of deep class hierarchies identified in Section 8.3.3, where it was argumented that it can not be easily avoided without incorporating extra information into the formal model.

The amount of false positives reported may provide a misleading picture, however. The design pattern constraints are mainly intended to be used in an interactive way when evolving a framework manually. In this case, we only checked the constraints after the facts. Most certainly, when version 4.0 of the framework was evolved into version 4.5, the developers inspected the code by hand and detected and fixed many conflicts. This is a time consuming process, however that can certainly benefit greatly from the automation we provide. Furthermore, the process is also error prone. Proof of this is the fact that we still discovered flaws in the design. In this light, we believe our results are still quite satisfactory and would show even better results if used interactively.

We do not have any indication about the time needed for the constraint checking process itself. This is exactly because the constraints were not used in an interactive manner. We can only note that checking *all* constraints is very time intensive. Therefore, when using the constraints in an interactive manner, we should only check those constraints that may be affected by a particular change. These are the constraints that are local to the place where a change has been applied, e.g. only the constraints that are associated with the design pattern instance that may be affected by the change. This may still take up quite some time, however, and more experiments are needed to determine if interactive use is feasible in practice.

8.4.2 Evolving the Framework

It has already been shown in numerous publications that refactorings can be used to evolve/improve the structure of a framework [Fow99, O'C01, Opd92, RBJ97, OCN99, OR93, JO93, TB95].

Since design pattern-specific transformations are nothing more then high-level refactorings, it is clear that they can be used for this purpose as well. Rather then presenting a detailed discussion of the refactorings and design pattern-specific transformations that have been applied to evolve the HotDraw framework, we will merely present a short overview of them, together with a short explanation.

Design Pattern-Specific Transformations and Refactorings

Design pattern-specific transformations are applied to the design pattern instances that occur in the framework. The transformations that we identified were all applied on only two instances: the *figureTM* and the *drawingEditorTM* instances.

The following transformations were applied on the *figureTM* instance, to change the framework's implementation and update the documentation accordingly:

- *addConcreteClass(RoundedRectangleFigure,RectangleFigure)*: introduces the `RoundedRectangleFigure` class in the framework, since it is used by many different applications.
- *removeConcreteClass(ManhattanLineFigure)* removes the `ManhattanLineFigure` class from the framework, as it is too application specific and should therefore not be part of the framework.
- *removeHookMethod(cornerVariable), removeHookMethod(originVariable)* and *removeHookMethod(variablesDo:)*: these methods formed part of the implementation of the constraint system, that is removed from the framework and replaced with the Smalltalk dependency mechanism.
- *addTemplateMethod(preferredBounds,computePreferredBounds)*: the `preferredBounds` method is an abstract method in the `VisualComponent` class, and was not overridden in the `Figure` hierarchy, as it should have been. Therefore, it is now implemented by the `Figure` class and relies on the `computePreferredBounds` hook method.

Similarly, the following transformations were applied on the *drawingEditorTM* instance:

- *addTemplateMethod(postBuildWith, windowName)*: this transformation was applied to introduce a `windowName` method in all subclasses of the `DrawingEditor` class. This method is responsible for returning the name of the window.
- *removeHookMethod(defaultTools)* and *addHookMethod(toolNames)*: as the implementation of the tools changed, the `defaultTools` method became obsolete and was replaced by a `toolNames` method.

Note that the last two transformations do not model the evolution that took place adequately and completely. The `initialize` method originally called the `defaultTools` method, and should now call the `toolNames` method instead. The above transformations do not reflect this. What is needed are *refineTemplateMethod(initialize, toolNames)* and *coarsenTemplateMethod(initialize, defaultTools)* transformations, that actually change the implementation of the `initialize` method appropriately. Such transformations are not included in our approach, however, and can thus only be applied manually.

Besides design pattern-specific transformations, ordinary refactorings were also applied to the framework. These refactorings mainly serve to restructure class hierarchies. The `Figure` class hierarchy, for example, was restructured by means of the following refactorings:

- *addParameter(Figure, menu, menuAt:)*: this refactoring is applied to provide the menu method with an extra parameter that is passed the current location of the mouse pointer. In this way, the menu can be displayed right next to this pointer.

- *pullUpMethod(Figure,translateBy:), pullUpMethod(Figure,origin)* and *pullUpMethod(Figure, extent)*: the implementation of these methods in the `Figure` hierarchy contained a lot of code duplication. Therefore, these methods were pulled up in the class hierarchy and the common behavior was factored out.
- *renameMethod(Figure, displayShapeOn:, displayFigureOn:)*: this refactoring is applied because `displayFigureOn:` is a more appropriate name for the method.

The `Drawing` class hierarchy has actually only undergone one single change. In the original version of the framework, the `Drawing` class contained a `stepIn:` method, that could be overridden by subclasses to provide animation. However, only a small part of the applications need this functionality. Therefore, the new version of the framework includes an `AnimatedDrawing` class, that should be subclassed by applications that want to provide animation. In order to apply this change, the following refactorings were performed:

- *addClass(AnimatedDrawing, Drawing)*: this refactorings adds the class `AnimatedDrawing` as a subclass of class `Drawing`. The idea is to separate all animation behavior into a separate class.
- *removeParameter(Drawing,stepIn:,step)*: this refactoring removes the parameter from the `stepIn:` method, as all drawings that include animation use the same bounding box in which this animation occurs.
- *pushDownMethod(Drawing, step)*: this refactoring pushes the `step` method down to the `AnimatedDrawing` class, where it belongs.

It should be noted that these refactorings do not update the design pattern specifications, even if this could be necessary. If the refactorings only reorganize the classes in a hierarchy, this is normally not a problem. Typically, leaf classes will remain leaf classes, and the root of the hierarchy will also remain. If the refactorings involve methods that participate in a design pattern instance, however, it could well be that the design pattern instance's specification will no longer reflect the current implementation. After a *pullUpMethod* refactoring has been applied, for example, the method involved should no longer be registered as a *hookMethod* participant, as it now contains a template implementation suited for all subclasses, and is no longer overridden.

In theory, the correctness of the specification can be examined by checking the constraints of the design patterns in which the arguments of the refactoring participate. Further research is needed in to validate this claim in practice, however.

Miscellaneous changes

Besides all changes mentioned above, a number of other important changes have been applied to the framework as well:

- Various classes have been subject to a renaming operation. The `ToolPaletteController` and `ToolPaletteView` classes have been renamed to `ToolbarController` and `ToolbarView`, the class `IndexPositionHandle` has been renamed to `IndexedTrackHandle`, for example.
- The superclass of the `RectangleFigure` class was changed from the `PolylineFigure` class to the `Figure`.class. Similarly, the `PERTEventFigure` class has become a subclass of the `ViewAdapterFigure` class instead of the `CompositeFigure` class.
- The `ContainerFigure` class hierarchy is reorganized completely. The functionality of the `ContainerFigure, LayeredContainerFigure, CompositeFigure` and `GroupFigure` class is merged into one single `CompositeFigure` class.
- The `Command` and `Reader` class hierarchies have been removed from the framework altogether. These hierarchies were mainly used for implementing tools, and since the `Tool` class has changed drastically between version 4.0 and 4.5 of the framework, they became obsolete.

With the exception of the renaming operations, neither of these changes can be expressed as refactorings or transformations. Obviously, this prohibits us from detecting some important merge conflicts, as we wil see later on.

8.4.3 Assessing the Impact on Existing Applications

In this section, the impact of the evolutions of the framework on its existing instances will be assessed. By mutually comparing the transformations that have been applied to evolve the framework with the transformations that have been used to construct the *VisualCreation* instance, 16 merge conflicts have been reported: 9 *constraint violation* merge conflicts, 2 *obsolete method* merge conflicts, 3 *missing parameter* merge conflicts and 1 *method absence* merge conflict. We will now discuss these conflicts in more detail, except the *missing parameter* merge conflict, which will be discussed in the next section.

Obsolete Method Merge Conflicts

Obsolete method merge conflicts were discussed in Section 7.3.4 and are caused when one transformation removes a method participant, while another transformation adds a new class participant at the same time. Two such conflicts occur in the *VisualCreation* instance as a result of the evolution of the framework. The `IconFigure` class provides an implementation for the `variablesDo:` method, which is actually removed from the framework by the *removeHookMethod* transformation. This situation is depicted in Figure 8.19. A similar conflict occurs because the *VisualCreation* instance introduces a new `IconDrawingEditor` class as a subclass of the `DrawingEditor` class, and thereby provides an implementation for the `defaultTools` method. This method was removed from the `DrawingEditor` class by means of a *removeHookMethod* transformation as well. As such, the `defaultTools` method in the `IconDrawingEditor` becomes useless, as it will never get called.

Note that it is not always the case that an *obsolete method* merge conflict arises due to the fact that new figure classes are added while at the same time methods are removed from the `Figure` class hierarchy. For example, the `originVariable` and `cornerVariable` methods are removed by means of a *removeHookMethod* design pattern-specific transformation, but the figure classes that are introduced for the *VisualCreation* application do not provide an implementation for these methods. Therefore, these classes do not contain any obsolete methods, and an *obsolete method* merge conflict is thus not reported.

Constraint Violation Merge Conflicts

The conditions that give rise to a *constraint violation* merge conflict were introduced in Section 7.3.3. During the evolution of the HotDraw framework, a new method `windowName` was introduced into the `DrawingEditor` class, that should return the name of the window for the application. Since this method was not present in the original version of the framework, the class `IconDrawingEditor` that was introduced in the *VisualCreation* application does not provide an implementation for it. This situation is depicted in Figure 8.20.

A similar situation occurs in the `Figure` hierarchy, where the evolved version of the framework introduces a `computePreferredBounds` method that should be overridden by concrete subclasses. Since this method was not present in the original version of the framework, all figures that are introduced by the *VisualCreation* instance (the `IconSquares, CursorFigure, MaskFigure, IconFigure, CursorIconFigure` and `MaskIconFigure` classes) do not provide an implementation for this method. Therefore, a *constraint violation* merge conflict is reported with respect to these classes, and the application developer should consider whether the introduced figures should provide an implementation or not.

Possible Incorrect Superclass Merge Conflicts

A *possible incorrect superclass* merge conflict occurs whenever a new class is introduced in the middle of a class hierarchy while at the same time a concrete class is added as a leaf of that

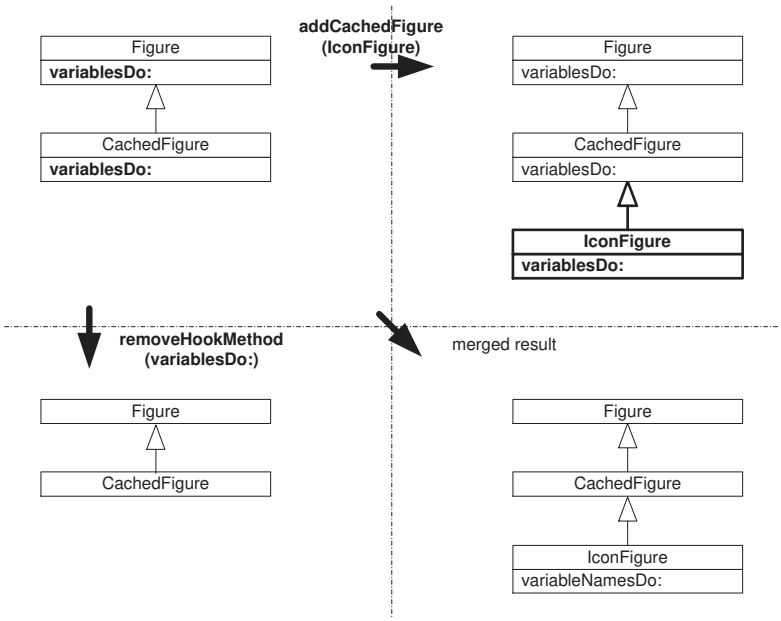


Figure 8.19: An *obsolete method* merge conflict

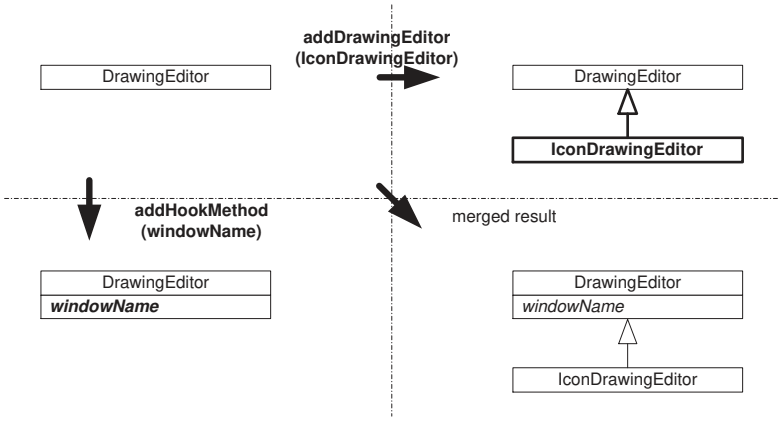


Figure 8.20: A *constraint violation* merge conflict

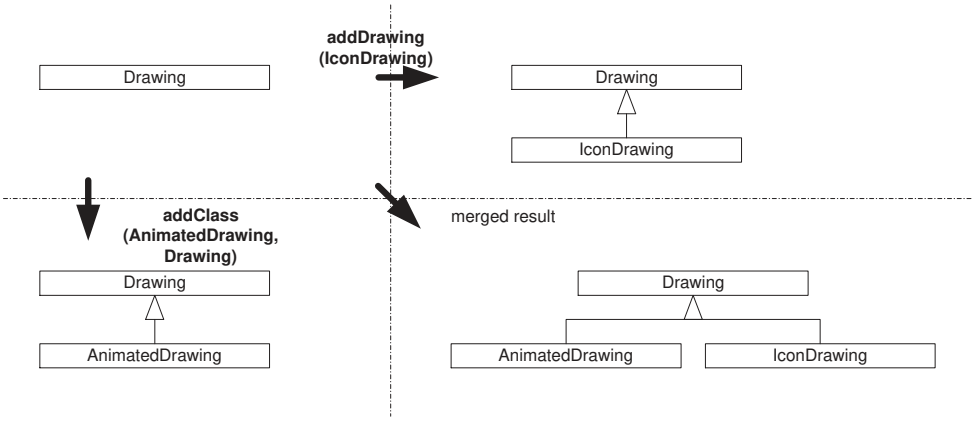


Figure 8.21: A *possible incorrect superclass* merge conflict

hierarchy (see Section 7.4.2).

In the case of the HotDraw framework, one particular evolution of it introduced a new `AnimatedDrawing` class as a subclass of the `Drawing` class, with the specific intent of isolating all animation behavior in this new class. However, the *VisualCreation* instance of the framework introduces a new concrete subclass `IconDrawing` of the `Drawing` class. Thus, a *possible incorrect superclass* merge conflict is reported (see Figure 8.21), and the application developer should consider whether the `IconDrawing` class should remain a subclass of `Drawing` or whether it should be changed to `AnimatedDrawing`. Since the `IconDrawing` does not provide animation, it should remain to be a subclass of `Drawing` in this case. In those applications that do provide animation, the `AnimatedDrawing` class should become the new superclass, however.

Method Absence Merge Conflicts

The conditions for a *method absence* merge conflict were defined in Section 7.4.4. In the case of the HotDraw framework and its *VisualCreation* instance, such a conflict occurs because the instance introduces a new `IconDrawing` as a subclass of `Drawing`. During evolution of the framework, however, the `step` method was pushed down from the class `Drawing` to all of its concrete subclasses. Since `IconDrawing` is such a concrete subclass, but is not part of the framework, it did not participate in the *pushDownMethod* refactoring and hence did not receive an implementation for the `step` method (see Figure 8.22).

This particular occurrence of the *method absence* merge conflict is not a real conflict. The *pushDownMethod* refactoring was applied to move the `step` method from the `Drawing` class to its `AnimatedDrawing` subclass. Both classes are framework classes and the `step` method is a framework method. The `IconDrawing` is an application-specific class, and should thus not participate in a *pushDownMethod* refactoring involving a framework method. Neither the refactorings, nor our merge conflict detection algorithm takes this consideration into account, however.

8.4.4 Assessing the Impact on the Framework

The framework itself also contains some merge conflicts as a result of its evolution. Our merge conflict detection algorithm reports 6 conflicts: 3 *obsolete method* merge conflicts, 1 *constraint violation* merge conflict, 1 *overly general method* conflict and 1 *missing parameter* merge conflict.

Obsolete Method and Constraint Violation Merge Conflicts

An *obsolete method* merge conflict occurs in the framework itself because one evolution introduces a new `RoundedRectangleFigure` class, while another evolution removes the `variablesDo:`

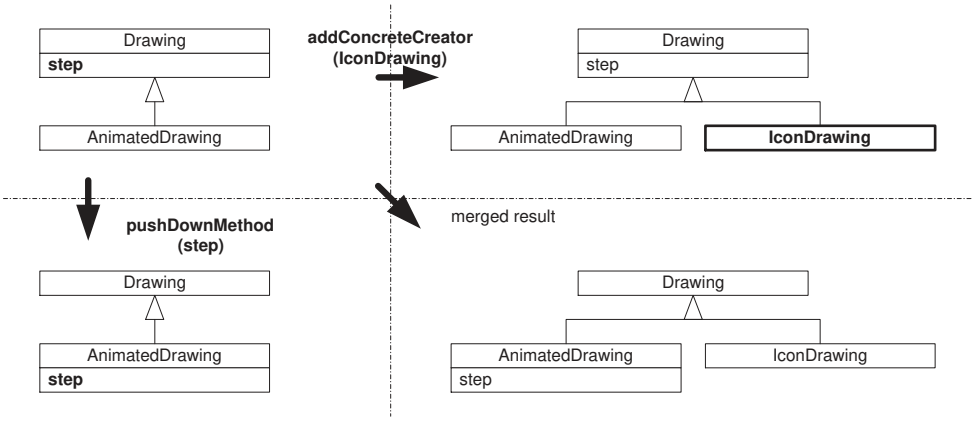


Figure 8.22: A *method absence* merge conflict

method from the `Figure` hierarchy. Likewise, a *constraint violation* merge conflict occurs because a new `computePreferredBounds` method is introduced into the `Figure` hierarchy, and the new `RoundedRectangleFigure` does not provide an implementation for it.

Missing Parameter Merge Conflicts

A *missing parameter* merge conflict occurs whenever an *addParameter* refactoring adds a parameter to a method participating in a design pattern instance, while at the same time this instance is extended with a new class participant (see Section 7.4.5). This new class participant will implement the method without the extra parameter.

This situation occurs when evolving the HotDraw framework: an *addConcreteClass* transformation is applied on the *figureTM* design pattern instance to add a new `RoundedRectangleFigure` class. This class provides an implementation of the `menu` method, that participates in the design pattern instance. At the same time, however, an *addParameter* refactoring is applied, to provide the `menu` method with an extra parameter, indicating the position where the menu should be displayed. When these two evolution steps are merged into one version, a *missing parameter* conflict occurs, as is depicted in Figure 8.23.

Overly General Method Conflicts

An *overly general method* merge conflict occurs whenever one particular evolution applies a *pullUpMethod* refactoring, while another evolution removes a particular class that implements the method to be pulled up (see Section 7.4.3).

A particular example of such a conflict in the HotDraw framework is depicted in Figure 8.24. One developer pulls up the `translateBy:` method, that is implemented by various classes in the `Figure` hierarchy, to the `Figure` class itself. At the same time, another developer removes the `ManhattanLineFigure` class from the framework. Since this class provides an implementation for the `translateBy:` method, the implementation of this method in the `Figure` class may take this into account, while this is no longer necessary. Therefore, a *overly general method* merge conflict is reported, and both developers should look into the situation in order to correct the problem.

Note that the combination of the *pullUpMethod* refactoring and the removal of a class does not always lead to a *overly general method* merge conflict. The `origin` and `extent` methods, for example, are also pulled up to the `Figure` class, whereas at the same time, the `ManhattanLineFigure` class is removed. However, since this latter class did not provide an implementation for those two methods, the implementation of the `origin` and `extent` methods in the `Figure` class will be correct.

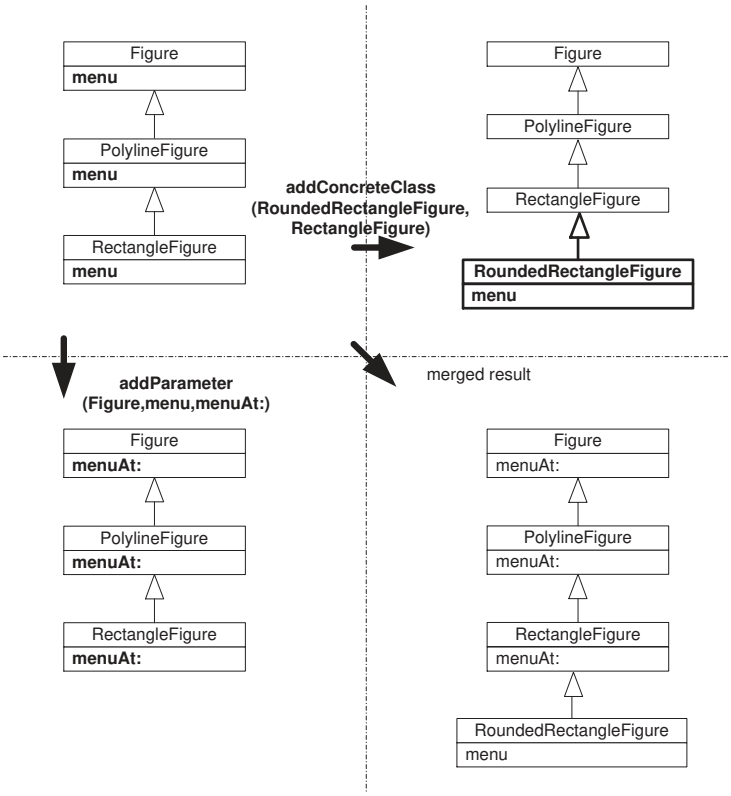


Figure 8.23: A *missing parameter* merge conflict

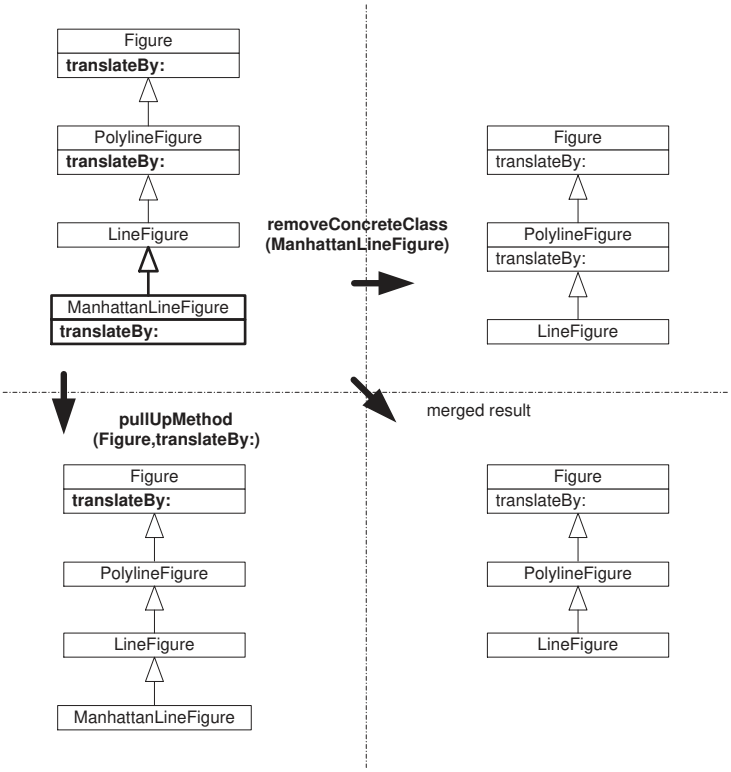


Figure 8.24: An *overly general method* merge conflict

8.4.5 Undetected Conflicts

As was explained in Section 8.4.2, some evolutions of the framework can not be expressed as refactorings or design pattern-specific transformations. Under these circumstances, some merge conflicts remain undetected. We identified two such conflicts, which are discussed next.

Change of superclass

The superclass of the `RectangleFigure` class was changed from the `PolylineFigure` class to the `Figure` class. This may create conflicts in the framework or in its instances.

In the *VisualCreation* instance, for example, an *addFigure* transformation was applied to add an `IconSquares` class as a subclass of `RectangleFigure`. In combination with the change of superclass, this leads to a conflict: it is not clear whether `IconSquares` should remain a subclass of `RectangleFigure`, or whether it now should subclass `PolylineFigure` directly instead. This depends upon which behavior the `IconSquares` should inherit from either of these two classes. This can only be decided by the developers responsible for the evolutions and therefore, a conflict should be reported. However, because the change can not be expressed as a refactoring, this conflict can not be detected and will thus not be reported.

Class Hierarchy Reorganization

In the new version of the framework, the functionality of the `ContainerFigure, LayeredContainerFigure, CompositeFigure` and `GroupFigure` classes was merged into one single `CompositeFigure` class. While this restructuring is an important change of the design of the framework, it can not be expressed by the refactorings incorporated in our approach. Presumably, this change is a combination of manual changes and a number of transformations and refactorings. We were not able to reconstruct these changes, so no conclusions can be drawn.

This change may however give rise to merge conflicts, in the framework, as well as in it's instances. For example, the *PERTChart* instance of the framework provides a `PERTEvent` class as a subclass of the original `CompositeFigure` class. Since this latter class now contains more behavior than its original version, the question should be raised whether the `PERTEvent` class should still be a subclass of the new `CompositeFigure` class. If we were able to report a possible merge conflict, we could point out this fact to the developers who could then take appropriate actions.

8.4.6 Discussion

In total, 22 merge conflicts were detected by our merge conflict detection algorithm, both in the framework as in the considered instance. These conflicts were based solely upon the refactorings and the design pattern-specific transformations discussed in Section 8.4.2 and the instantiation transformations of Section 8.3.2.

All except one conflict (the *method absence* merge conflict) were real conflicts that had to be resolved in order for the framework's design to be correct and consistent. The specific reason why the *method absence* conflict was reported is that the refactorings and the merge conflict detection algorithm do not distinguish between framework-specific and application-specific classes and methods. This issue should be considered further, as it may occur under other circumstances as well.

Of the 22 reported conflicts, only one conflict was a behavioral merge conflict: the *overly general method* merge conflict. Naming conflicts could not occur, of course, since both versions of the framework were correct and we merely simulated how transformations were applied in parallel. This is not the typical situation however, but is due to the fact that we simulated how the framework evolved. Presumably, many more behavioral conflicts and perhaps some naming conflicts were present in version 4.5 of the framework, but have been detected and removed manually by the developers. Just like we argumented for the constraint checking (see Section 8.4.1), this is

a time-consuming and error-prone process, for which automated support is quite valuable and feasible. It remains to be studied however how the approach behaves under interactive use.

A number of important conflicts remained undetected. This was due to the fact that it was impossible to express some particular evolutions of the framework by means of refactorings or design pattern-specific transformations. Since our merge conflict detection algorithm is entirely based upon comparing such transformations, it is no surprise that those conflicts were not detected. An approach to merge conflict detection that is based upon manual changes (such as [Luc97]) will also have a hard time detecting these conflicts, however, and will certainly not be able to report them at a high-level. To overcome such problems, we can incorporate more refactorings into the environment, which would enable us to represent more evolutions in terms of such high-level transformations. At the same time, however, this would influence the scalability of our merge conflict detection approach. This problem can thus only be alleviated if a suitable abstraction for refactorings is introduced, something which has not yet been achieved, and is certainly beyond the scope of this dissertation.

While we only considered one application derived from the framework, it is clear that other applications will be impacted as well. The results obtained from the above experiments leave us confident that the appropriate merge conflicts will be reported for these instances as well.

8.5 Conclusion

In this chapter, we have tested our supporting environment on a real-world framework. By using the environment for documenting design pattern instances, instantiating and evolving the framework, we were able to identify the strengths and weaknesses of our approach and were able to propose ways of alleviating them. Besides showing the feasibility of such an approach, we also proved the usefulness and necessity of an environment that support framework-based development. Many important design flaws, inconsistencies and merge conflicts were detected, both in the framework itself as in the example instance. All this despite of the fact that the HotDraw framework has been used many times and was studied extensively by many people. These results confirm our believe that we have established a first step towards an environment that supports framework-based development, and we are quite confident that the results can be generalized to large scale frameworks as well.

Chapter 9

Conclusion and Future Work

This chapter first summarizes the dissertation, and then presents our conclusions. Furthermore, we will discuss issues that remain to be solved and extensions that can be made to the approach.

9.1 Summary

The thesis statement we put forward at the beginning of this dissertation was the following:

Thesis. *Elaborate automated tool support for framework-based software development can be provided by explicitly documenting a framework's design in a formal way.*

In the rest of the dissertation, we have tried to prove this thesis.

To document the design of a framework, we used information about design patterns and their instances. This was motivated by the fact that design patterns are generally applicable, well-documented, well-understood, commonly used and that they expose useful information about the role and responsibilities of important classes and methods of a framework. Such information proved valuable when instantiating and/or evolving the framework. We implemented an environment that uses such information to actively support developers when they perform such tasks. The kind of support ranged from verifying the framework's design constraints, to automatically generate skeleton code for applications and detecting conflicting changes that were applied by different developers.

To be able to provide such elaborate automated support, a solid formal basis was required. Therefore, we first defined a (quasi-)formal model for metapatterns. Such metapatterns form an abstraction for many different design patterns, and can be formalized. They can thus be used as a basis for a formal definition of (the structure, participants and collaborations of) those design patterns. This formal model included the definition of five fundamental metapatterns, the constraints that these impose upon the framework's implementation, as well as high-level transformations that can be applied upon their instances, that change their implementation and update their documentation (semi-)automatically.

Based on the formal model and its associated transformations, we defined a change propagation strategy. This was achieved by first identifying the possible ways in which instances of the five fundamental metapatterns can overlap, and then determining the conditions under which transformations applied on one metapattern instance give rise to transformations on overlapping metapattern instances. Moreover, because the high-level transformations explicitly represent the kind of changes that are applied to a framework, we were capable of defining an operation-based merge conflict detection strategy. This allowed us to detect merge conflicts at a high level of abstraction, and suggest possible resolutions.

We integrated this formal model into a declarative meta-programming environment, and provided implementations for the metapattern constraints, transformations, change propagation and

merge conflict detection algorithms. In this way, we were able to show how support for framework-based development could be achieved in practice, based on our theoretic foundations. We illustrated how design pattern instances could be explicitly documented, and how this enables the environment to guide a developer when evolving or instantiating a framework. At the same time, the environment also ensures that the design pattern documentation remains up to date at all times. Such support is possible thanks to the fact that the environment can be programmed, so that it can automatically translate design pattern instance specifications into the appropriate metapattern instance specifications, and that it allows us to define framework-specific and design pattern-specific transformation in terms of the built-in metapattern-specific transformations.

Finally, we used this environment to document the design and the evolution of the HotDraw framework, This allowed us to test our approach in a practical setting, assess its usefulness and detect possible shortcomings. The (preliminary) results we obtained in this way were quite satisfying. The environment is able to guide a developer in constructing a correct instance of the framework, by generating a template application that needs to be filled in. Furthermore, it automatically detects various design flaws and inconsistencies in the framework's implementation, despite the fact that we considered a stable version of the framework, that has already been studied and reused many times. We also showed how the high-level transformations can be used to evolve the framework, and how this allowed us to assess the impact of these evolutions on the framework itself, as well as on its existing instances. The environment detected several important merge conflicts and pointed out the places in the framework and its instance were developer attention is required to resolve those conflicts.

9.2 Conclusions

Our approach to support framework-based development was validated on two different frameworks: the Scheme framework, presented in Chapter 3 and the HotDraw framework, presented in Chapter 8. We used the Scheme framework as a proof-of-concept throughout the dissertation, to illustrate the approach, present various explanatory examples and fine tune the approach whenever necessary. As a side effect, the usefulness of the approach was illustrated as well. The HotDraw framework, on the other hand, was used as a case study to effectively prove our claims and validate the approach. It was selected for two specific reasons. First, we required a controlled environment, in which we could study the evolution of two stable versions of a framework. Second, the HotDraw framework is widely recognized as a high-quality framework, that is well documented, has been reused many times and has been studied extensively. Studying and summarizing the results we obtained, we feel we can safely draw the following conclusions.

The most important conclusion of this dissertation is that building an environment that supports framework-based development, based upon a formal annotation of a framework's design, is both feasible and valuable. Our experiments clearly show that evolving and instantiating even a small-scale framework is a complex and difficult task. Automated support for checking design constraints, performing non-trivial changes and assessing the impact of particular changes is thus undoubtably necessary and valuable.

Explicitly expressing a framework's design constraints by means of metapattern constraints turns out to be very useful when the framework is evolved manually. As our experiments indicate, such constraints can be used to tackle the problem of design erosion. The supporting environment is able to check whether the framework's implementation still respects the appropriate constraints after a number of changes have been applied manually. Several constraint violations were identified in the framework's implementation and a number of design flaws and inconsistencies were detected, that apparently remained unnoticed by the framework developers. These are quite surprising results, given the fact that we only considered stable releases of the HotDraw framework, that have been studied extensively by many different people over the years.

We provided evidence that high-level transformations are valuable for automatically instantiating and evolving a framework. Framework-specific transformations help a developer to derive a correct instance from a framework, by guiding him through the process, pointing out the classes

and methods that have to be added and generating code automatically whenever possible. They thereby actually achieve the kind of support that is proposed by active cookbooks [PPSS95]. Design pattern-specific transformations can be considered as higher-level refactorings and therefore help a developer to evolve a framework in a more straightforward way at a higher level of abstraction. Our experiments showed however that such design pattern-specific transformations are only useful for dealing with anticipated evolution. A number of changes to which a framework is subject deals with internally reorganizing a class hierarchy, something which can not be achieved by applying a design pattern transformation. To alleviate this problem, we also included refactorings into our approach, thus showing that support for unanticipated evolution can also be provided to a certain extent.

The high-level transformations also enable us to reason about the evolution and instantiation of a framework at a high level of abstraction. We have illustrated how support for software merging, and in particular merge conflict detection, can be supplied, based on such transformations. Besides showing that such support is possible, our experiments also proved that it can be used in practice, and enabled us to automatically assess the impact of particular evolutions on the framework, as well as on existing applications. The supporting environment detected various merge conflicts in the HotDraw framework, and proposed adequate ways of resolving them.

We also extensively demonstrated that documenting a framework's design by means of design patterns is accurate and concise, and reveals significant information. All important class hierarchies and methods of both frameworks are covered (and thus documented) by the design pattern instances occurring in them. Furthermore, documentation based on design patterns explicitly describes the specific roles and responsibilities of the classes and methods involved. Such crucial and important information can be used by tools, as we have illustrated, but will also be of great value for a developer trying to understand the design and the inner workings of the framework. We believe the framework understanding process can be significantly improved by incorporating tools that allow developers to browse through this information in various ways. We do not have any hard evidence of this fact, however, as this was not the specific focus of this dissertation.

Our initial belief that metapatterns do indeed form a suitable abstraction of design patterns for the purposes of this dissertation was also confirmed. Due to the use of metapatterns, the scalability and manageability of our approach was ensured, without sacrificing its overall usefulness and practical applicability. Of course, we recognize that such an abstraction is only useful for specific kinds of tools, that deal with the structure, participants and collaborations of design patterns. Other tools that support working with design patterns, such as those that help a developer in identifying the design pattern to use, will not benefit as much from the metapattern abstraction, since metapatterns do not contain the appropriate information. We can imagine that other abstractions for those kinds of tools will emerge.

9.3 Achievements

In this section, we elaborate upon the artifacts that were produced in the context of this dissertation.

The most important artifact that was produced is the formal framework for the definition of metapatterns. This framework provides general, parameterized formal definitions for five fundamental metapatterns. These definitions can be used as a basis to provide formal definitions of Pree's metapatterns, as well as formal definitions of (some aspects of) design patterns. The framework is the most primitive building block of our approach, since without it, the scalability and manageability of the approach can not be guaranteed.

This formal model also includes a definition of metapattern-specific transformations, that can be used to construct design pattern-specific transformations, and a definition of metapattern instance overlappings. This enabled us to define change propagation and merge conflict detection algorithms. These algorithms can be used to detect the impact of a particular change upon the framework and its existing applications.

Based on this model and the derived algorithms, another important artifact was developed: we

have built a prototype of an environment that is able to support framework-based development. Thanks to the use of metapatterns, this environment is scalable and general, and can be used for any kind of framework. We integrated this environment into a standard development environment by using SOUL, a declarative meta-programming language.

Other useful artifacts were produced during our case-study: we documented the complete design of the HotDraw framework by means of design patterns, we provided framework-specific transformations that implement an active cookbook for this framework and we identified places in the framework that could be improved.

9.4 Limitations and Boundaries of the Approach

The approach to support framework-based software development we have proposed in this dissertation naturally has some limitations, not in the least because we took into account some important restrictions from the start. We will discuss the most important of these limitations and restrictions in more detail in the following sections.

9.4.1 Fully Supporting Unanticipated Evolution

As we have mentioned, design pattern-specific transformations can only be used to support anticipated evolution. This is due to the fact that, unlike refactorings, these transformations are not generally applicable, but rather rely on a specific design of the framework to be able to perform the needed changes. Consequently, such design pattern transformations are very well suited to support instantiation of the framework, where a design is simply reused, but are much less useful for reorganizing that design, for example.

In order to overcome this issue to some extent, and show that our approach is general enough to include unanticipated evolution as well, we incorporated refactorings in Chapter 7. We did not achieve full integration of these refactorings into our formal model of Chapter 5, however. Therefore, we can not claim that we are able to fully support unanticipated evolution. For example, refactorings could be used to reorganize participants in a design pattern instance, could destroy that instance altogether or could replace it with an instance of another design pattern. At present, we do not support such changes, meaning that these changes are not reflected in the design pattern instance documentation, which may thus become incorrect. The only solution to this problem consists of incorporating refactorings into the formal model, so that their effect upon a design pattern instance is defined in a precise manner. However, fully integrating all refactorings into the model is far from trivial, and would undoubtably increase the complexity of the model. This should be considered future work.

Other approaches to support unanticipated evolution, such as reuse contracts [Luc97, Men99] and intentional software views [MMW02], could also be considered. Since our approach bears much resemblance to the reuse contract approach, and can be considered an extension of it, integrating both approaches would be straightforward. The kind of support for unanticipated evolution offered by the reuse contract approach is rather low level, compared to the kind of support offered by refactorings, however. Integrating intentional software views into our approach would be much harder, since these are a completely different way to support evolution.

9.4.2 Other Software Development Processes

Frameworks are not the only way in which flexible and reusable software can be developed. Other software development processes exist, such as *component-oriented software development* [Szy97] and *product line engineering* [Bos00]. The question can be raised to which extent our approach is applicable in such domains.

In the component-oriented software development process, applications are constructed by reusing off-the-shelf components and combining them in a suitable way by writing appropriate

glue code. In this way, component-oriented software development closely resembles software development using black-box frameworks [RJ96]. Both development methods clearly face entirely different problems then a (white-box) framework-based development method. Application developers no longer have to consider the internal design and implementation details of the components, since they are not allowed to change them. Rather, they are concerned with picking the appropriate components for their purposes, and they should know how different components can be glued together. This requires information about the external behavior of the components, as well as information about the interfaces required and provided by these components. It is clear that the approach we have presented in this dissertation can not be used for such purposes, since it deals with entirely different information, and provides no support for gluing components together.

In product line engineering, applications (termed *products*) are implemented by means of a *product line*. A product line consists of a *product-line architecture* and a set of components implementing that architecture. A product is constructed by deriving a *product architecture* from the product-line architecture and subsequently instantiating and configuring the components, extending them with product-specific code where necessary. As can be seen, product line engineering differs from component-oriented programming in that the components in the product line architecture are not black-box, since they can be adapted by the application developer. Very often, these components take the form of small object-oriented frameworks. For these reasons, we believe our approach can be valuable in this domain as well. Since object-oriented frameworks are used to implement the components, it is clear that our approach can support customization of these components. Moreover, by slightly adapting our approach and incorporating extra information, such as e.g. architectural information, it should be possible to support deriving a product architecture from the product-line architecture as well. This issue should however be investigated further, and can be considered future work.

9.4.3 Other Information

Another restriction we considered throughout the dissertation, was to use only information about a framework's design. Consequently, we could only express the instantiation and evolution of a framework in terms of how its design was reused or how it changed over time. Other sources of information are available, however, and if we use them, such restrictions would not be present. We have two options in considering other kinds of information: we could use lower-level or higher-level information. In what follows we will elaborate upon these issues.

Information below the Design Level

Information at the level below the design level can consist of *coding conventions, programming idioms* [Cop92] and *best practice patterns* [Bec96], for example. Such information is closer to the implementation level, but still provides some extra abstraction. It is thus much more detailed than design information. This has the advantage that it can be used to provide extra support. For example, we can easily imagine that coding conventions and best practice patterns can be used to automatically generate appropriate method bodies, something a developer now has to specify manually. The disadvantage of such detailed information is that it does expose many more details. This means that such information could easily become unmanageable, could endanger the scalability of the approach and could make it language dependent.

Information above the Design Level

Information at a level above the design level includes architectural information and domain knowledge, for example.

Architectural information. A software architecture describes the overall structure of a framework (or any software system in general), abstracting away all implementation details and focusing only on a few concepts of interest and their mutual relationships. Architectural information thus includes descriptions of the various conceptual parts of the framework (e.g. the classes

and methods that conceptually belong together), as well as how these parts are related and collaborate to achieve certain behavior. As has been shown in [Men00, Flo00, Min97] and [MNS95], tools can be build that use such information to support various aspects of the software development process. Most certainly, the results obtained can be used to support framework-based development as well. For example, it is possible to check the conformance of a framework's implementation to its intended architecture, after this implementation has evolved in a number of ways. However, we believe that, due to the high-level nature of architectural information, it is impossible to provide the kind of advanced support offered in this dissertation, such as semi-automatic guidance of instantiation and evolution or detecting possible merge conflicts. This requires much more detailed information, as we have illustrated throughout this dissertation. Still, we believe it is worthwhile to investigate how the architectural information can complement the design information, and to find out if significantly more, or improved, support could be provided by incorporating such extra information.

Domain knowledge. Since frameworks are specific to a particular domain, it could be worthwhile to investigate how *domain knowledge* can be exploited to provide support for framework-based development. For example, as we have observed during our validation, several classes in the `Figure` class hierarchy do not need to override key methods (such as the `extent, origin, cornerVariable` and `originVariable` methods, for example). Nevertheless, our change propagation algorithm prompts the developer for an implementation every time a new class is added to this hierarchy. Classes in this hierarchy represent mathematical figures, however, and the methods implement mathematical properties and formulas. Since the hierarchy already includes all "basic" figures, such as `RectangleFigure` and `EllipseFigure`, these methods never need to be overridden again. Clearly, if we incorporate domain knowledge into our approach, such information can be made explicit, and this would enable us to refine our change propagation algorithm. Since domain knowledge, just like architectural information, is rather high level, we believe it can only be used as a complement to design information, for our specific purposes.

9.5 Future Work

Besides the issues discussed in the previous section, the approach we proposed in this dissertation can be extended and improved upon in various other ways.

Overhead of design pattern instance specification. As our experiments have shown, there is a slight overhead associated with the specification of design pattern instances. First, a design pattern instance may involve many different participants, that each need to be specified individually. Second, whenever such large instances overlap, the duplication that arises in their specifications may be an impediment. We already explained that this problem can be alleviated, by providing an appropriate interface that computes a design pattern specification automatically and presents the result to the developer for inspection. For example, the developer may only need to specify the root of a class hierarchy and the interface would be able to compute all leafs automatically. Moreover, many tools exist today that help a developer in introducing design pattern instances into the implementation. Such tools can be easily adapted so that they not only generate the appropriate code for the instance, but also generate the corresponding specification automatically.

Simplistic implementation of the metapattern-specific transformations. As our experiments have shown, the current implementation of the metapattern-specific transformations is too simplistic if used in a realistic setting. This implementation assumes that all newly added leaf classes should implement all appropriate method participants. As we have observed, this is often too strict a constraint when deep class hierarchies are involved. When a class is added deep down in such hierarchies, it can often rely on method implementations provided by its superclasses. It remains to be investigated how this issue can be dealt with. Presumably, more information as to which methods should always be overridden and which methods may be inherited should be included into the model in order to resolve the issue. As we have explained above, domain knowledge may help to prevent this problem.

Metapattern constraints. We identified a number of constraint violation conflicts that can occur when a developer manually evolves a framework. Such constraint violation conflicts were established upon the constraints that are associated with a particular metapattern. We did not show however that these conflicts are complete with respect to the metapattern constraints, e.g., that we effectively captured all constraint violation conflicts that can occur, based on the metapattern constraints. This remains to be proven.

Moreover, much in the same way that metapattern-specific transformations can be used to define design pattern-specific transformations, metapattern constraints can be used as a basis to define design pattern constraints. Since design patterns exhibit more information than metapatterns about specific relationships and collaborations, design pattern constraints could presumably detect more constraint violation conflicts. This in turn allows the environment to detect more constraint violation conflicts and thus supply more support for a developer.

Furthermore, the definition, and corresponding implementation, of many metapattern constraints is quite naive. As our experiments have shown, unimportant, or even faulty, constraint violation conflicts are reported when deeply nested class hierarchies are used. This is mainly due to the fact that the definition of the constraints is too coarse grained. They do not allow, for example, that leaf classes reuse the (default) implementation of a method defined in the root class of a hierarchy. It remains an open question whether these constraints can be fine tuned, so that they allow such reuse, while at the same time they can detect whenever a method is not overridden but actually should be. Actually, this problem is related to the problem associated with the transformations mentioned above. The extra information that we proposed to introduce into the model there, can also be used in this case to provide a more realistic implementation for the metapattern constraints.

Extensions to the model. The model we defined for metapatterns forms an important basis for our approach. Several remarks about and extensions and improvements to this model are possible:

- The model is only quasi-formal, in the sense that many definitions (such as those for the relations that can hold between participants) are only given in natural language. A complete model would also define these relations in a formal way. Such formal definitions are feasible, however, since the relations mainly deal with important constructs that occur in every object-oriented programming language. This was not the focus of our dissertation however.

- The metapattern-specific transformations only add or remove participants to and from metapattern instances. Other useful transformations are imaginable, such as a *refineTemplateMethod* and *coarsenTemplateMethod*, for example. The main motivation for adding such transformations is that many more evolutions could be expressed as high-level transformations, This would not only improve the interactive support for instantiation and evolution, but would also result in many more merge conflicts that can be detected. The price that has to be paid is that the model would become more complex: more transformations would need to be defined, and the impact of many more transformations on overlapping metapattern instances should be assessed. Furthermore, formal definitions for many more merge conflicts would have to be provided.

- We have not provided formal proofs that the overlapping conditions for metapatterns are complete, that the metapattern-specific transformations are indeed orthogonal and that those transformations are independent from the actual metapatterns. We have only provided informal discussions and relied on the reader's common sense. These issues thus remain to be investigated.

Scalability of the merge conflict detection algorithm. Although we used metapatterns and metapattern-specific transformations to ensure the scalability of the merge conflict detection algorithm, the approach can still be improved upon. Our operation-based merge conflict approach as it is now is entirely based upon the particular operations that are allowed, e.g. the metapattern-specific transformations and the refactorings. This makes the approach less scalable if more

metapattern-specific transformations are included in the model, or if more refactorings need to be incorporated. In those cases, we need to compare them to all existing transformations and refactorings and define when their parallel application induces a conflict. Although we believe this situation will not happen frequently, it is possible to define a merge conflict detection algorithm independently of the transformations and refactorings that are allowed. This can be achieved by first determining the assumptions upon which each individual transformation or refactoring relies and then considering how one transformation or refactoring breaks the assumptions of another one. Under such circumstances, a merge conflict can be reported. Whenever a new transformation or refactoring should be introduced, it then suffices to define the assumptions upon which it relies and which assumptions it may break, in order to incorporate it into the merge conflict detection algorithm.

Language independence. One particular issue we did not explicitly address is language independence. All of our experiments were performed on frameworks that were implemented in the Smalltalk programming language. In principle, the results we obtained should be generalizable to any other object-oriented programming language. Design patterns (and metapatterns) are mostly language independent, since only their implementation is specified in a particular programming language. The structure, participants and, to some extent, the collaborations of a particular design pattern instance are similar in many different programming languages. Moreover, the relations present in our formal model are present as basic constructs in any object-oriented programming language. We can thus describe a framework's design by means of design patterns, irrespective of the programming language in which it is implemented. As for the transformations, it would appear as if they are specific to a particular programming language, since they generate the actual code that updates the implementation. However, these transformations are easy to generalize as well, since they merely manipulate classes and methods, and they mostly rely on the developer for the actual method implementations. It would thus be easy to reimplement the transformations in a more generic way, so that they generate code in the appropriate language.

Validation on industrial cases. As we have stressed many times, the major focus of this dissertation was on language engineering issues. We mainly concentrated on showing the feasibility of building an environment that can support framework-based development. This required us to work in a controlled setting, where stable versions of a framework were available for study. As a result, industrial frameworks were ruled out initially. Nevertheless, the results we obtained strongly indicate that the support we are able to provide is valuable. The logical next step is thus to further test our approach on industrial frameworks. Naturally, this requires that our supporting environment is integrated into a standard development environment, much like the Refactoring Browser. We are currently working on a prototype of such a browser, that extends the Refactoring Browser with design pattern-specific transformations and that includes constraint checking and merge conflict detection algorithms. This environment will also allow us to test our approach in an interactive way, during evolution and instantiation. As was already explained, in this dissertation we only considered the evolutions that were applied in retrospect.

Appendix A

Formal Definition of Pree's Metapatterns

In Chapter 5, we already provided formal definitions for the Unification *and* 1:N Recursive Connection *metapatterns in terms of the fundamental metapatterns. We will now show how the other metapatterns identified by Pree in [Pre95] can be formally defined.*

A.1 The *1:1 Connection* Metapattern

Definition

The *1:1 Connection* metapattern, as depicted in Figure A.1, can be defined by means of the *Connection* fundamental metapattern. As such, it has five participants: a template class T, a hook hierarchy $\mathcal{H}$, a set of template methods M_t, a set of hook methods M_h and a reference variable v. In this particular metapattern, the *templateClass* participant refers to exactly one instance of a concrete hook class from the hook hierarchy $\mathcal{H}$, which is denoted by the *single* annotation in the definition. The reference is held by means of the reference variable v, which is defined as an instance variable in the *templateClass* participant. This is the reason for the *variable* annotation in the definition. The complete definition for the *1:1 Connection* metapattern thus looks as follows:

$$oneToOneConnectionMetapattern(T, \mathcal{H}, T :: \mathcal{M}_t, \mathcal{H} :: \mathcal{M}_h, V)$$
$$::=$$
$$connectionFundamentalMP(T, \mathcal{H}, T :: \mathcal{M}_t, \mathcal{H} :: \mathcal{M}_h, V, single, variable)$$

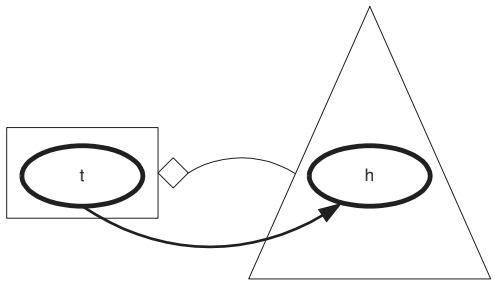


Figure A.1: The *1:1 Connection* metapattern

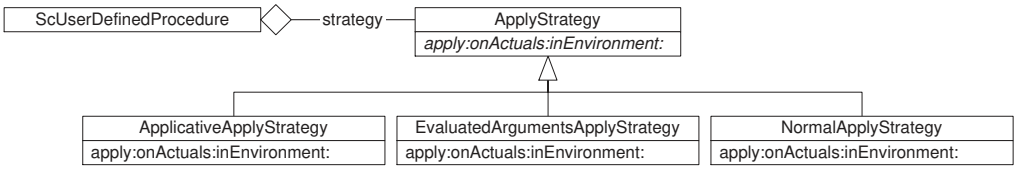


Figure A.2: Example Design Pattern for the *1:1 Connection* metapattern

A.1.1 Structure of the template methods

The typical structure of a template method defined for this pattern looks as follows:

```
t
   ...
   v h1.
   ...
   v h2.
   ...
```

where the method `t` is a member of the *templateMethod* participants defined by this pattern and `v` is the variable participant that holds the reference from the template class to an object of the hook hierarchy. Both of these participants are defined in the *templateClass* participant of the metapattern. The `h1` and `h2` methods are members of the hook methods of this metapattern and are defined in the root of the *hookHierarchy* participant, while provided with a concrete implementation for all leaf classes of that hierarchy.

A.1.2 Example Design Pattern

The *Strategy* design pattern is a prototypical example of a design pattern that uses this metapattern as its underlying structure. Figure A.2 shows an instance of this design pattern in the Scheme framework. The `ApplyStrategy` hierarchy represents the *hookHierarchy* participant $\mathcal{H}$ of the metapattern, the `ScUserDefinedProcedure` class corresponds to the *templateClass* participant, while the `apply:onActuals:inEnvironment:` method is the only *hookMethod* participant. Observe that there is no *templatemethods* participant, since the design pattern instance does not contain any such participants.

We can express this instance of the *Strategy* design pattern in a formal way as follows:

$$
\begin{aligned}
&hookhierarchy(hierarchy(ApplyStrategy))\\
&templatemethods(\{\})\\
&templateclass(ScUserDefinedProcedure)\\
&hookmethods(hierarchy(ApplyStrategy) :: apply : onActuals : inEnvironment :)\\
&referencevariable(strategy)
\end{aligned}
$$

A.2 The *1:N Connection* Metapattern

Definition

The *1:N Connection* metapattern (see Figure A.3) differs from the *1:1 Connection* pattern only in the number of references to objects of the hook hierarchy. As such, it can also be defined in terms of the *Connection* fundamental metapattern. In the *1:N Connection* metapattern, the template class T refers to multiple instances of classes in the hook hierarchy $\mathcal{H}$ through the reference variable v. This is denoted in the definition by means of the *multiple* annotation. The reference variable is defined in the template class participant as an instance variable once again, which is the reason for the *variable* annotation. The formal definition of this metapattern is thus:

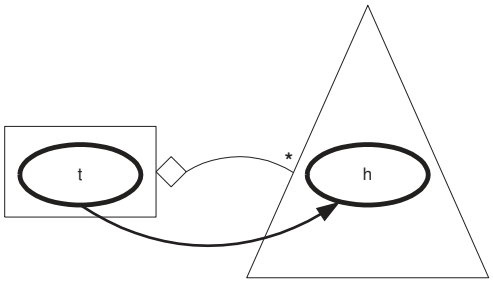


Figure A.3: The *1:N Connection* metapattern

$$oneToManyConnectionMetapattern(T, \mathcal{H}, T :: \mathcal{M}_t, \mathcal{H} :: \mathcal{M}_h, V)$$
$$::=$$
$$connectionFundamentalMP(T, \mathcal{H}, T :: \mathcal{M}_t, \mathcal{H} :: \mathcal{M}_h, V, multiple, variable)$$

A.2.1 Structure of the template methods

In this particular metapattern, the template methods typically iterate over the collection of hook objects contained in the variable `v`, and call the appropriate hook methods on each of them. In Smalltalk, such template methods might look as follows:

```
t
   ...
   v do: [ :hookobject | hookobject h1. hookobject h2: 1 ]
   ...
```

where the method `t` is defined in the template class participant and belongs to the set of template methods. The `v` object, which is also defined in the template class participant, contains a collection of all hook objects that an instance of the template class T refers to. The `h1` and `h2:` methods, defined in the hook hierarchy participant, belong to the set of hook methods $\mathcal{M}_h$.

A.2.2 Example Design Pattern

The *Observer* design pattern is the prototypical example of a design pattern that uses this metapattern as its underlying structure. A typical instance of this design pattern is depicted in Figure A.4. The `Subject` class, which plays the role of the template class participant of the metapattern, holds a reference to a number of `Observer` objects, through the instance variable `observers`. This instance variable represents the reference variable participant of the metapattern, while the `Observer` hierarchy represents the hook hierarchy participant. Whenever the state of a subject changes, the `notify` method is called, which notifies all the observers by sending them the `update` message. The `notify` method of the `Subject` class represents the template method participant of the metapattern, while the `update` method of the `Observer` hierarchy plays the role of hook method participant. Concrete observer classes override this *update* method in order to update themselves appropriately upon state changes.

In our formal model, we can express this instance of the *Observer* design pattern as follows:

$$hookhierarchy(hierarchy(Observer))$$
$$templatemethods(Subject :: notify)$$
$$templateclass(Subject)$$
$$hookmethods(hierarchy(Observer) :: update)$$
$$referencevariable(observers)$$

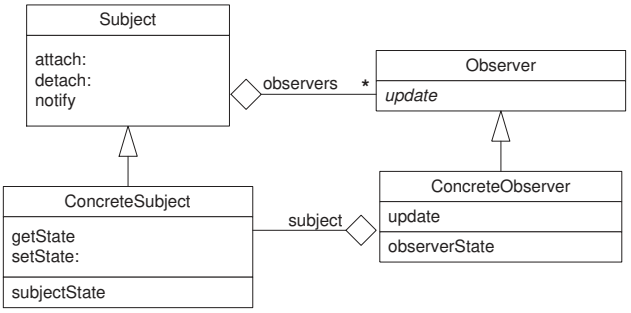


Figure A.4: Example Design Pattern for the *1:N Connection* metapattern

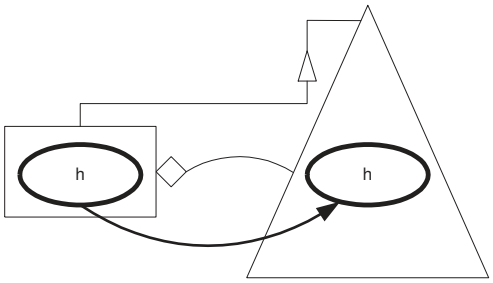


Figure A.5: The *1:1 Recursive Connection* metapattern

A.3 The *1:1 Recursive Connection* Metapattern

Definition

The *1:1 Recursive Connection* metapattern (see Figure A.5) is defined in terms of the *Recursion* fundamental metapattern, and consists of five participants: a template class T, a hook hierarchy $\mathcal{H}$, a reference variable v, a set of hook methods M_h defined in class T and a set of hook methods M_h defined on the hierarchy $\mathcal{H}$. The class T itself is also part of the hierarchy $\mathcal{H}$.

In this metapattern, the template class T refers to exactly one instance of a class in the hook hierarchy $\mathcal{H}$, which is denoted by the *single* annotation in the definition. The reference to an object of the hook hierarchy is via the reference variable v, which is defined as an instance variable in the template class participant. This is denoted by the *variable* annotation in the definition, which thus looks as follows:

$$oneToOneRecursiveConnectionMetapattern(T, \mathcal{H}, T :: \mathcal{M}_h, \mathcal{H} :: \mathcal{M}_h, V)$$
$$::=$$
$$recursionFundamentalMP(T, \mathcal{H}, T :: \mathcal{M}_h, \mathcal{H} :: \mathcal{M}_h, V, single, variable)$$

A.3.1 Structure of the template methods

The typical implementation of a template method for this metapattern looks as follows:

```
h
   ...
   v h
   ...
```

which shows that the template method h, defined in class T, recursively calls the hook method h, defined in the hierarchy $\mathcal{H}$. This is formally specified by the $invokes_r$ relation in the definition of the *Recursion* fundamental metapattern.

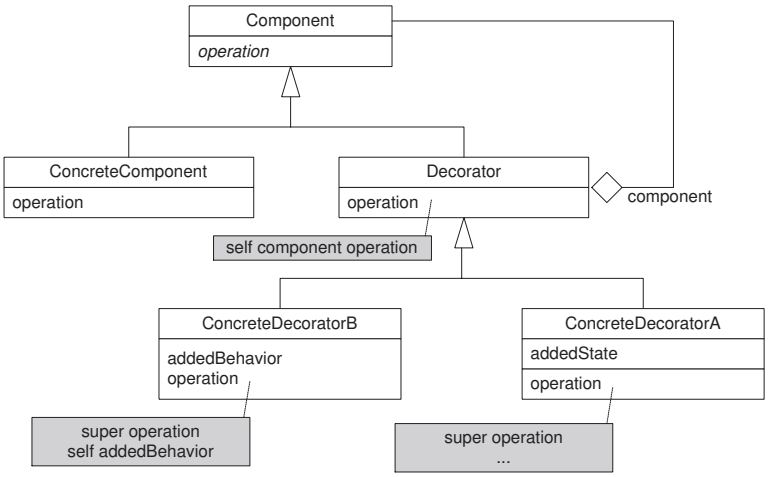


Figure A.6: Example Design Pattern for the *1:1 Recursive Connection* metapattern

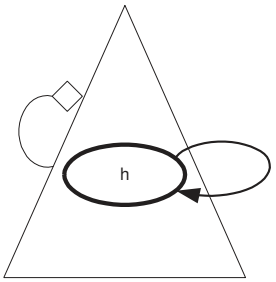


Figure A.7: The *1:1 Recursive Unification* metapattern

A.3.2 Example Design Pattern

An example of a design pattern that uses this structure is the *Decorator* design pattern, as depicted in Figure A.6. The role of the template class participant of the metapattern is played by the *Decorator* participant, while the hook hierarchy participant is represented by the `Component` hierarchy. The `operation` method defined on the `Decorator` class plays the role of template method participant, while the same method defined on the `Component` class, and overridden in the various leaf classes (such as the `ConcreteComponent` class), plays the role of hook method participant. The reference variable participant of the metapattern is represented by the `component` instance variable defined in the `Decorator` class.

This instance of the *Decorator* design pattern can be expressed formally as follows:

$$
\begin{aligned}
&hookhierarchy(hierarchy(Component))\\
&templatemethods(Component :: operation)\\
&templateclass(Decorator)\\
&hookmethods(hierarchy(Component) :: operation)\\
&referencevariable(component)
\end{aligned}
$$

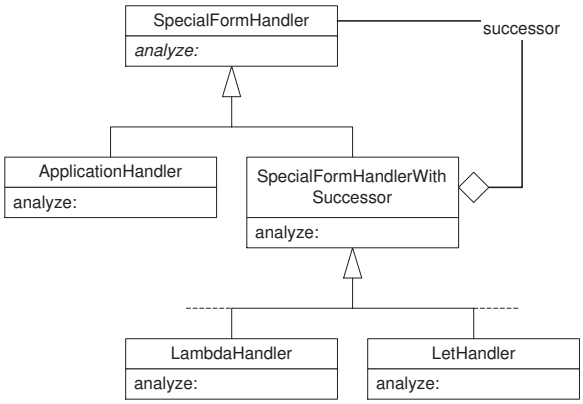


Figure A.8: Example Design Pattern for the *1:1 Recursive Unification* metapattern

A.4 The *1:1 Recursive Unification* Metapattern

Definition

In the *1:1 Recursive Unification* metapattern (see Figure A.7) the template class T and the root of the hierarchy $\mathcal{H}$ are unified into one class. This is reflected by the fact that this metapattern only has three participants: the hook hierarchy $\mathcal{H}$, a reference variable v and a set of hook methods $\mathcal{H} :: \mathcal{M}_h$. The root of this hook hierarchy contains a single reference to itself through the v variable, which is defined as an instance variable in the root of the hierarchy $\mathcal{H}$. Consequently, this pattern can be defined as follows formally:

$$oneToOneRecursiveUnificationMetapattern(\mathcal{H}, \mathcal{H} :: \mathcal{M}_h, V)$$
$$::=$$
$$recursionFundamentalMP(root(\mathcal{H}), \mathcal{H}, root(\mathcal{H}) :: \mathcal{M}_h, \mathcal{H} :: \mathcal{M}_h, V, single, variable)$$

A.4.1 Structure of the template methods

A template method in this metapattern has the following form:

```
h
   ...
   v isNil ifFalse: [ v h ]
   ...
```

A template method h thus calls itself recursively using the reference present in the root of the hook hierarchy, if this reference exists.

A.4.2 Example Design Pattern

The structure of this metapattern is used, for example, in the *Chain of Responsibility* design pattern. Consider the example instance of this design pattern as presented in Chapter 3 and depicted in Figure A.8. The `SpecialFormHandler` hierarchy represents the *hookHierarchy* participant of the metapattern. There is only one *hookMethod* participant in this design pattern instance, namely the `analyze:` method that is defined on this hierarchy. The `successor` instance variable of the `SpecialFormHandler` class corresponds to the *referenceVariable* participant.

Formally, we can express this instance as follows:

$$hookhierarchy(hierarchy(SpecialFormHandler))$$
$$hookmethods(hierarchy(SpecialFormHandler) :: analyze :)$$
$$referencevariable(successor)$$

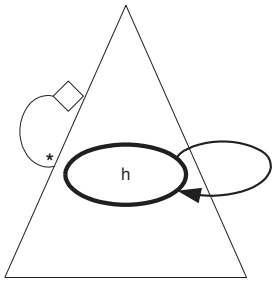


Figure A.9: The *1:N Recursive Unification* metapattern

A.5 The *1:N Recursive Unification* Metapattern

A.5.1 Definition

Just like the *1:1 Recursive Unification* pattern is a degenerate version of the *1:1 Recursive Connection* pattern, the *1:N Recursive Unification* pattern (see Figure A.9) is a degenerate version of the *1:N Recursive Connection* pattern. It unifies the template class T and the root of the hook hierarchy $\mathcal{H}$ into one class. This class holds a reference to a number of instances of classes of its own hierarchy through the reference variable v, that it defines as an instance variable. Formally:

$$oneToManyRecursiveUnificationMetapattern(\mathcal{H}, \mathcal{H} :: \mathcal{M}_h, V)$$
$$::=$$
$$recursionFundamentalMP(root(\mathcal{H}), \mathcal{H}, root(\mathcal{H}) :: \mathcal{M}_h, \mathcal{H} :: \mathcal{M}_h, V, multiple, variable)$$

A.5.2 Structure of the template methods

The template methods of this metapattern use the reference variable to call themselves recursively:

```
h
   ...
   v do: [ :hookobject | hookobject h ].
   ...
```

where h is a member of the set of hook methods, and the `v` object contains the collection of references to instances of the hook hierarchy.

A.5.3 Example Design Pattern

This metapattern can be used as the underlying structure for a variant of the *Observer* design pattern, where the *Subject* and the *Observer* classes are unified into one class. This is how many Smalltalk dialects implement the MVC paradigm, for example. The class `Object`, which is the superclass of all classes in the Smalltalk language, holds a reference to a number of objects in order to be able to update them when state changes occur. `Object` and all of its subclasses thus form the hook hierarchy participant of this metapattern. The `Object` class defines an instance variable `dependents`, that holds all observers and thus corresponds to the reference variable participant. The hook method participant is represented by the `notify:` method, also defined on the `Object` class, while the template method participants are all methods that change the state of an object and call the `notify:` method accordingly.

Appendix B

Discussion of the Remaining Merge Conflicts

As mentioned in Chapter 7, we can detect 12 possible merge conflicts due to the parallel application of metapattern-specific transformations and refactorings. We only discussed five of these conflicts in that chapter. Here, we discuss the remaining seven conflicts.

B.1 Naming Merge Conflicts

As was discussed in Section 7.3.2, naming conflicts arise when developers independently introduce a class or a method with the same name. Refactoring operations that rename a particular artifact, such as *renameClass* and *renameMethod* may also cause name clashes, and can thus also provoke a naming conflict.

B.1.1 Example Conflict

Figure B.1 shows an example of a *naming* merge conflict that is due to the parallel application of a *moveMethod* refactoring and an *addHookMethod* evolution operation. The refactoring moves a method `analyze:` from some class to the `CondHandler` class, while at the same time, an `analyze:` method is added as a *hookMethod* participant to the `SpecialFormHandler` hierarchy, and thus to the `CondHandler` class. In the merged result, the `CondHandler` class thus defines two `analyze:` methods.

B.1.2 Formal Definition

Table B.1 contains formal definitions of the conditions that give rise to a *naming* merge conflict involving classes. As can be seen, such a conflict occurs when two classes with the same name are introduced, or a class is renamed while at the same time a class with the same name is introduced.

Similarly, Table B.2 lists the conditions that lead to a *naming* merge conflict when introducing, renaming or moving methods. The *moveMethod* refactoring moves a method from one class to another, and may thus give rise to a *naming* merge conflict if the destination class already defines a method with the same name. Since a *moveClass* refactoring did not exist, this particular condition was not present for classes.

B.1.3 Discussion

As was already explained previously, *naming* merge conflicts can not be resolved in an automatic way, since it is difficult to know when similarly named artifacts actually represent the same concept.

$$addClass(className, superclass, subclasses) \parallel addLeaf_a(leafName) \Rightarrow namingConflict(className) \quad if \quad className = leafName$$

$$renameClass(class, newName) \parallel addLeaf_a(leafName) \Rightarrow namingConflict(leafName) \quad if \quad newName = leafName$$

Table B.1: Conditions giving rise to a *naming* merge conflict when adding or renaming classes

$$addMethod(class, m_1, body) \parallel addMethod_a(m_2) \Rightarrow namingConflict(m_1) \quad if \quad m_1 = m_2 \wedge (inherits_h^*(\{class\}, \mathcal{H}_a) \vee inherits_h^*(\mathcal{H}_a, \{class\})$$

$$renameMethod(class, m_1, m_2) \parallel addMethod_a(m_3) \Rightarrow namingConflict(m_3) \quad if \quad m_2 = m_3 \wedge (inherits_h^*(\{class\}, \mathcal{H}_a) \vee inherits_h^*(\mathcal{H}_a, \{class\})$$

$$moveMethod(class, m_1, destClass, m_2) \parallel addMethod_a(m_3) \Rightarrow namingConflict(m_3) \quad if \quad m_2 = m_3 \wedge (inherits_h^*(\{destClass\}, \mathcal{H}_a) \vee inherits_h^*(\mathcal{H}_a, \{destClass\})$$

Table B.2: Conditions giving rise to a *naming* merge conflict when adding, renaming or moving methods

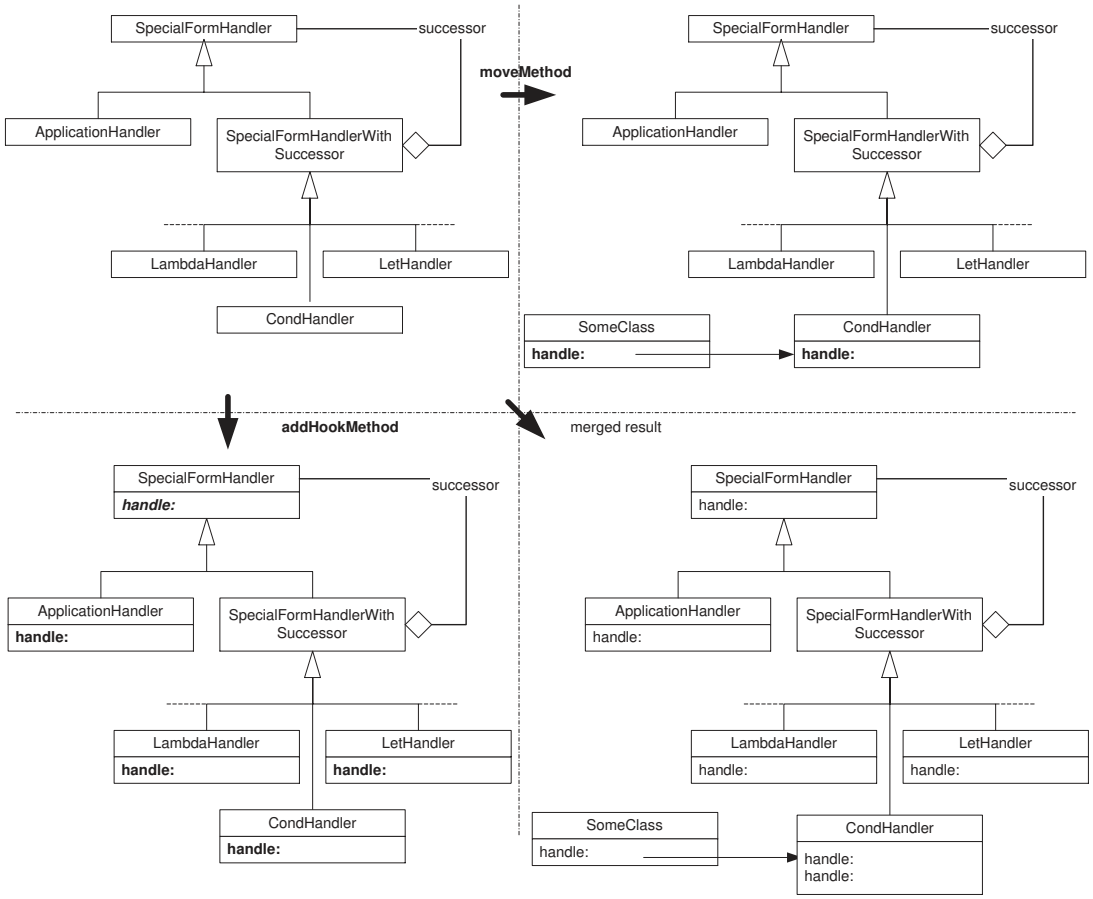


Figure B.1: A *naming* merge conflict

Therefore, the only way to solve such conflicts is by consulting the developers that performed the evolutions, who should look into the problem and resolve it appropriately.

B.2 Orphan Method/Variable Merge Conflicts

Example Conflict

Figure B.2 depicts a particular occurrence of an *orphan method* merge conflict. Again, an instance of the *Composite* design pattern in the Scheme framework is evolved in two different ways at the same time. One evolution consists of adding a particular new expression to the `ScExpression` hierarchy by applying an *addLeaf* evolution operation. This operation is specific to the *Composite* design pattern and adds a new leaf class participant (`ScQuoteExpression` in this case) to the instance. The other evolution consist of pulling up the `printOn:` method by means of a *pullUpMethod* refactoring. The rationale behind this refactoring is to remove the `printOn:` method from all subclasses, implement it as a concrete method in the `ScExpression` class, and use a *Visitor* design pattern to implement the appropriate behavior.

The end result of the merge process shows that the newly added `ScQuoteExpression` class does still contain an implementation for the `printOn:` method. Since it was the intent of the *pullUpMethod* refactoring to pull this method up in the class hierarchy, this situation is probably not satisfactory. Furthermore, since the `printOn:` method in class `ScExpression` is a generalization of all methods implemented previously in the subclasses, it should be checked whether it is

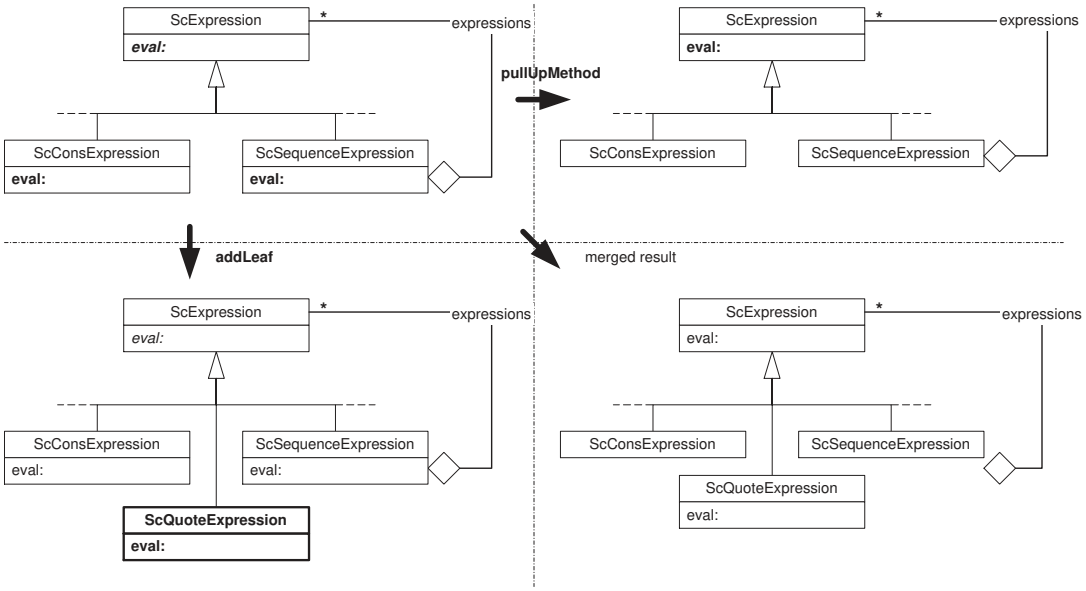


Figure B.2: An *orphan method* merge conflict

$$pullUpMethod(class, selector) \parallel addLeaf_a(leafName) \Rightarrow orphanMethodConflict(leafName, selector) \; if \; class = root(\mathcal{H}_a)$$

Table B.3: Conditions giving rise to an *orphan method* merge conflict

general enough to cover the new `ScQuoteExpression` class as well.

B.2.1 Formal Definition

The formal definition of the conditions which lead to an *orphan method* merge conflict are listed in Table B.3. Such a conflict arises when a *pullUpMethod* refactoring operation is performed on a particular class hierarchy, while at the same time, this hierarchy is extended with a new class, by means of an *addHookClass* or an *addTemplateClass* evolution operation. Formally, the condition specifies that the hierarchy to which the refactoring is applied should be a hierarchy participant in a design pattern (or metapattern) instance to which the evolution operation is applied. Under these conditions, the newly added class will provide an implementation for a method that it should actually inherit from its superclass.

Similar to the *orphan method* merge conflict, an *orphan variable* merge conflict arises when the hierarchy to which an *addHookClass* or *addTemplateClass* operation is performed is subject to a *pullUpVariable* refactoring. The conditions that need to be satisfied for a *orphan variable* merge conflict to occur are the same as those for the *orphan method* merge conflict, as can be seen from Table B.4.

$$pullUpVariable(class, variable) \parallel addLeaf_a(leafName) \Rightarrow orphanVariableConflict(leafName, variable) \; if \; class = root(\mathcal{H}_a)$$

Table B.4: Conditions giving rise to an *orphan variable* merge conflict

B.2.2 Discussion

Orphan method and *orphan variable* merge conflicts are classified as structural merge conflicts. Clearly, when merging the two versions of the framework, the resulting structure lacks uniformity, because one particular entity differs from all other entities. In the example presented above, the `ScQuoteExpression` class contains an implementation for the `eval:` method, while all other leaf class participants of the *Composite* design pattern instance inherit this implementation from the `ScExpression` superclass. Presumably, the `ScQuoteExpression` class should also inherit this behavior.

As all other structural conflicts that we encountered thus far, the *orphan method* and *orphan variable* merge conflicts can be resolved by enforcing an order on the application of the evolutions. Those operations that add participants should have precedence over those operations that merely change an already existing structure. Concretely, if in the presented example, the *addLeaf* evolution operation on the *Composite* design pattern instance is applied before the *pullUpMethod* refactoring, the conflict will not occur. Under these circumstances, the refactoring will pull up the implementation of the `eval:` method from the `ScQuoteExpression` class as well.

B.3 Missing Origin/Destination Merge Conflicts

Refactoring operations that move a method or a variable between classes may give rise to two different kinds of conflicts: *missing origin* and *missing destination* merge conflicts. This duality stems from the fact that moving a method involves two different classes, which can both be subject to changes by other evolution operations, as we will see.

B.3.1 Example Conflict

Figure B.3 shows the situation where one developer moves a method `method` to a `CondHandler` class, while another developer performs a *removeConcreteHandler* evolution operation on the instance of the *Chain of Responsibility* design pattern, and consequently removes the `CondHandler` class. Clearly, this results in a conflict in the merged version, since the destination class to which the method should have been moved is now missing from the implementation.

Figure B.4 contains an example of a *missing origin* merge conflict. In this figure, a developer manipulates an instance of the *Composite* design pattern, by applying a *removeCompositeMethod* operation to remove the `analyze` method. At the same time, another developer moves the `analyze` method of the `ScConsExpression` class to the `SpecialFormHandler` class by means of a *moveMethod* refactoring. This operation is performed while introducing an instance of the *Chain of Responsibility* design pattern, that is used in the `ScConsExpression` class, to avoid a large conditional statement that handles all possible special forms.

When merging both evolutions, the `analyze:` method of the `SpecialFormHandler` class is no longer called from within the `ScConsExpression`, which is not what was intended. This prevents the special forms of the Scheme language from being evaluated.

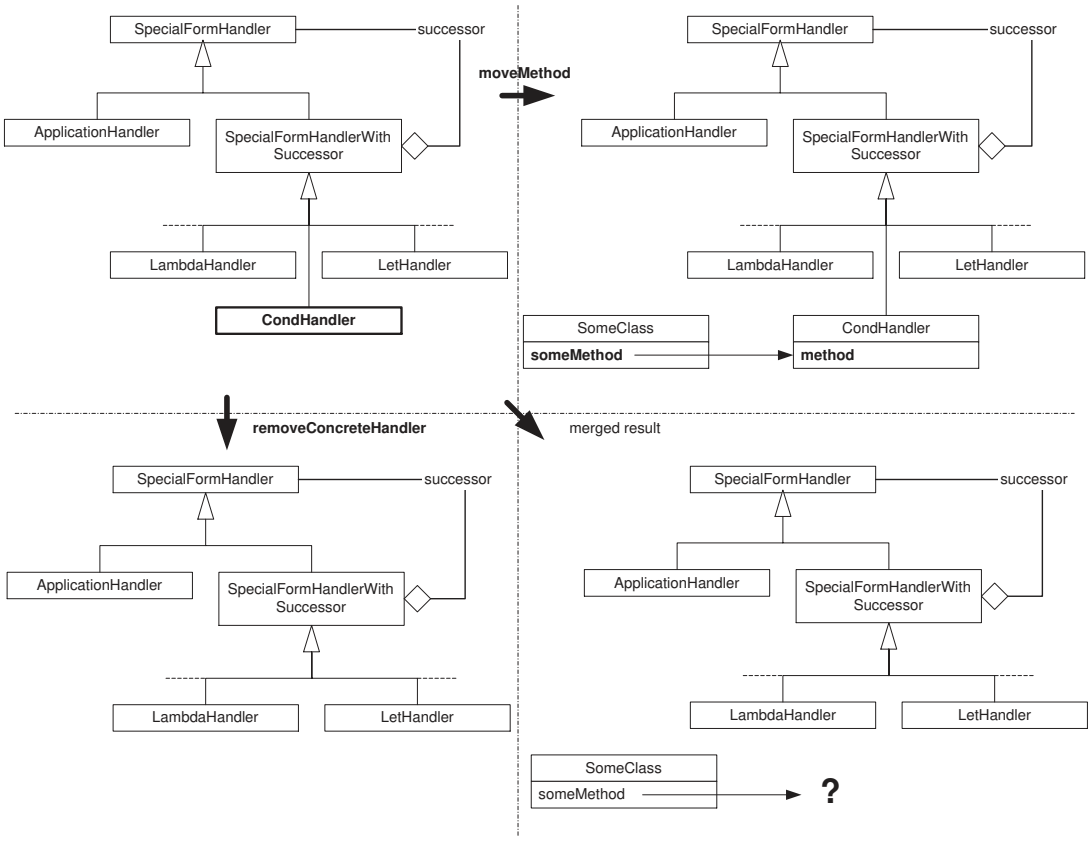


Figure B.3: A *missing destination* merge conflict

B.3.2 Formal Definition

Table B.5 defines the conditions that give rise to a *missing destination* merge conflict. Such conflicts are caused by a refactoring that moves a method or a variable, while at the same time, the class to which this method or variable is moved is removed (by means of a *removeHookClass* or a *removeTemplateClass* evolution operation).

In Table B.6, the conditions are specified that lead to a *missing origin* merge conflict. Such a conflict does not exist for variables, only for methods. It is caused by applying a *moveMethod* refactoring which moves a particular method to another class, and may provide the original method in the origin class with a default behavior that forwards the method to the destination class. When this original method is removed by means of a *removeHookMethod* or *removeTemplateMethod* evolution operation the conflict occurs. Formally, the conditions state that the conflict occurs whenever the origin class from which a method is moved is part of a class hierarchy to which a *removeMethod* evolution operation is applied, and the method that is moved is equal to the method that is removed.

Note that, strictly speaking, this combination of evolution operations will not always lead to a conflict in practice. If the *moveMethod* refactoring is applied and there is no need for a forwarding method, the original method may be removed by the refactoring itself. Therefore, when another evolution is applied in parallel that also removes this original method, the merged result will be correct, even though a warning will be issued.

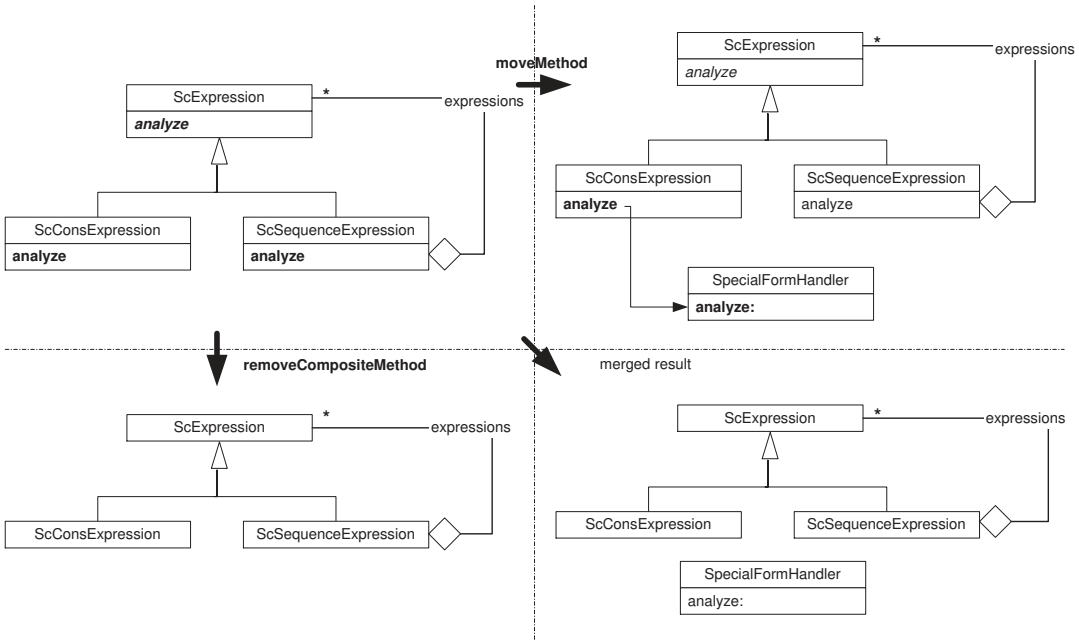


Figure B.4: A *missing origin* merge conflict

B.3.3 Discussion

Both *missing destination* and *missing origin* merge conflicts are classified as behavioral conflicts. In the latter case, this is obvious. The structure resulting from merging the two versions of the framework is correct with respect to the intended design structure, but does not exhibit the appropriate behavior: the method that is moved is no longer called, so the framework will probably not behave as intended. A *missing destination* merge conflict will most probably be detected by a compiler, since a method calls another method that does no longer exist. As such, it even becomes impossible to construct a correct version of the framework that incorporates both changes.

A *missing destination* merge conflict can not be automatically resolved. By imposing an order on the application of the evolutions, the conflict can be avoided however. When we first apply the *removeConcreteHandler*, the *moveMethod* refactoring will not be allowed, since its preconditions will detect that the `CondHandler` class no longer exists, and will report this as an error. Although

$$moveMethod(class, m_1, destClass, m_2) \parallel removeLeaf_a(leafName) \Rightarrow missingDestinationConflict(leafName, m_2) \quad if \quad class(leafName) = destClass$$

$$moveVariable(class, varName, destClass) \parallel removeLeaf_a(leafName) \Rightarrow missingDestinationConflict(leafName, varName) \quad if \quad class(leafName) = destClass$$

Table B.5: Conditions giving rise to a *missing destination* merge conflict

$$moveMethod(class, m_1, destClass, m_2) \parallel removeMethod_a(m_3) \Rightarrow$$
$$missingOriginConflict(destClass, m_2)$$
$$if$$
$$class \in \mathcal{H}_a \wedge m_1 = m_3$$

Table B.6: Conditions giving rise to a *missing origin* merge conflict

the conflict will be prevented in this way, it is not actually solved. The mere fact that the *moveMethod* refactoring was applied by one of the developers means that it was actually the intent to move this method to the desired class. The developer thus had some specific goal in mind when performing this operation. The fact that this class does no longer exist does not change this goal, so the situation should be reconsidered and the appropriate solution should be found by the developer.

A *missing origin* merge conflict can not be resolved by means of imposing an order on the application of the evolution either. The developers responsible for the evolutions should be warned and should look into the situation and come up with an appropriate solution.

B.4 Remove Merge Conflicts

B.4.1 Example Conflict

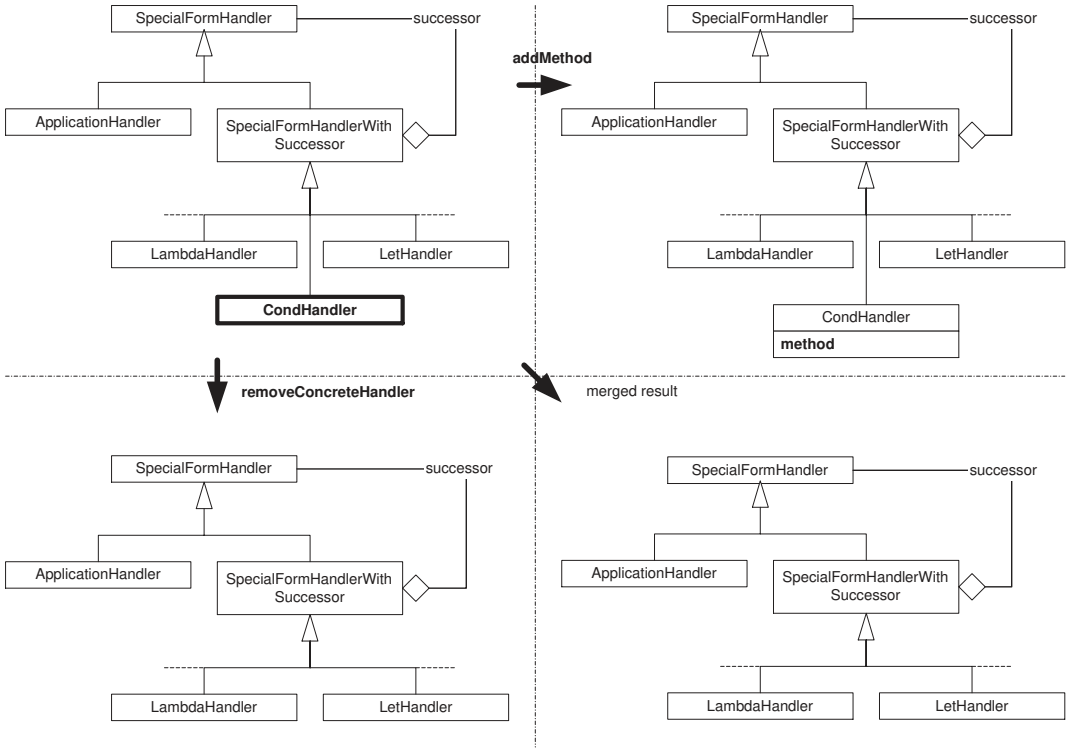


Figure B.5: A *remove* merge conflict

$$addMethod(class, selector, body) \parallel removeLeaf_a(leafName) \Rightarrow removeConflict(class, selector) \; if \; class = class(leafName)$$

Table B.7: Conditions giving rise to a *remove* merge conflict

$$addVariable(class, varName, initClass) \parallel removeLeaf_a(leafName) \Rightarrow removeConflict(class, varName) \; if \; class = class(leafName)$$

Table B.8: Conditions giving rise to a *remove* merge conflict

Consider the situation shown in Figure B.5. The first evolution that is applied is a *addMethod* refactoring operation that adds a method `method` to the `CondHandler` class. The second evolution removes the `CondHandler` class from the instance of the *Chain of Responsibility* design pattern by means of an *removeConcreteHandler* evolution operation. The version of this class hierarchy that results from merging both evolutions appears correct, except from the fact that the method that was added by the first evolution is not included. Since it was the specific intent of the developer to include this method, this may not be what is desired, so a conflict should be reported.

B.4.2 Formal Definition

The formal definitions of the conditions that give rise to a *remove* merge conflict are listed in Table B.7. As can be seen, such conflicts arise when a method is added to a particular class by means of an *addMethod* refactoring, while at the same time, this class is removed by a *removeHookClass* or *removeTemplateClass* evolution operation. Similarly, the same conditions can be defined for a *remove* merge conflict involving variables. In this case, an *addVariable* refactoring is applied instead of an *addMethod* refactoring, while the class to which the variable is added is removed by means of a *removeHookClass* or *removeTemplateClass* evolution operation. The conditions are in fact the same, as can be seen in Table B.8.

B.4.3 Discussion

Remove merge conflicts are another kind of behavioral conflicts. Clearly, the structure resulting from merging both evolutions is correct with respect to the design structure, and no design pattern (or metapattern) constraints are violated. It remains an open question whether the framework will behave as intended, however. Presumably, there are no references to the method, since it was just introduced. However, the developer who introduced the new method did so with a specific goal in mind. Even though the framework may behave as intended, we should still notify the developer to ensure that he takes appropriate action to achieve his specific goal in another way.

Once again, a *remove* merge conflict can be avoided, but not resolved, if the order of the evolution operations is taken into account. If the *removeConcreteHandler* operation is applied before the *addMethod* refactoring in the example, the latter's preconditions will not be satisfied, and the operation will thus not be allowed. However, since this is not what was intended, the

conflict is not actually resolved. Therefore, the developers responsible for the two evolutions should be consulted and should decide how the problem can be resolved in the appropriate way.

Bibliography

[ABW98] Sherman Alpert, Kyle Brown, and Bobby Wolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.

[AC98] Ellen Agerbo and Aino Cornils. How to preserve the benefits of Design Patterns. In *Proceedings of the OOPSLA'98 Conference*, pages 134–144. ACM Press, 1998.

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[AS85] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[BD99] Greg Butler and Pierre Dénommée. *Documenting Frameworks to Assist Application Developers*, chapter 7. John Wiley and Sons, 1999.

[Bec96] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1996.

[BFVY96] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic Code Generation from Design Patterns. *Object Technology*, 35(2), 1996.

[BGK98] Greg Butler, Peter Grogono, and Ferhat Khendek. A Reuse Case Perspective on Documenting Frameworks. In *Proceedings of the Asia Pacific Software Engineering Conference*, pages 94–101, 1998.

[BJ94] Kent Beck and Ralph Johnson. Patterns Generate Architectures. In *Proceedings of the European Conference on Object-Oriented Programming*, 1994.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley, 1996.

[Bos98] Jan Bosch. Design Patterns as Language Constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.

[Bos00] Jan Bosch. *Design & Use of Software Architectures – Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.

[Bra95] John M. Brant. HotDraw. Master's thesis, University of Illinois at Urbana Champaign, 1995.

[Bra98] Lars Bratthall. An Introduction to Software Architecture. Technical Report CODEN-LUTEDX (TETS7171)/1-60/(1998)&local28, Oslo University, 1998.

[Bri00] Johan Brichau. Declarative Composable Aspects. In *Proceedings of the Workshop on Advanced Separation of Concerns*, 2000.

[Bro90] R.J. Brockman. The Why, Where and How of Minimalism. In *Proceedings of the ACM SIGDOC90 Conference*, pages 111–119, 1990.

[CDHSV97] Wim Codenie, Koen De Hondt, Patrick Steyaert, and Arlette Vercammen. Evolving Custom-Made Applications into Domain-Specific Frameworks. *Communications of the ACM*, 40(10):71–77, 1997.

[CH00a] Aino Cornils and Görel Hedin. Statically Checked Documention with Design Patterns. In *Proceedings of TOOLS Europe '00*, 2000.

[CH00b] Aino Cornils and Görel Hedin. Tool Support for Design Patterns based on Reference Attribute Grammars. In *Proceedings of WAGA'00*, 2000.

[Cha92] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Computer Science Department, Stanford University, 1992.

[CMO97] Marcelo Campo, Claudia Marcos, and Alvaro Ortigosa. Framework Comprehension and Design Patterns: A Reverse Engineering Approach. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, 1997.

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

[CS95] James Coplien and Douglas C. Schmidt. *Pattern Languages of Program Design*. Addison-Wesley (Software Patterns Series), 1995.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard Reference Manual*. Springer-Verlag, 1996.

[DH98] Koen De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 1998.

[DMDGD01] Wolfgang De Meuter, Maja D'Hondt, Sofie Goderis, and Theo D'Hondt. Reasoning with Design Knowledge for Interactively Supporting Framework Reuse. In *Proceedings of the SCASE (Soft Computing applied to Software Engineering) Conference*, 2001.

[DV98] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 1998.

[DV99] Kris De Volder. Aspect Oriented Logic Meta Programming. In *Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, pages 250–272, 1999.

[DV01] Kris De Volder. Implementing Desing Patterns as Declarative Code Generators, 2001.

[Dyb96] R. Kent Dybvig. *The Scheme Programming Language*. Prentice Hall, 1996.

[Ede00] Amnon Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, Department of Computer Science, Tel Aviv University, 2000.

[Edw97] W. K. Edwards. Flexible Conflict Detection and Management in Collaborative Applications. In *Proc. Symp. User Interface Software and Technology*, ACM Press, 1997.

[EGHY99] Amnon Eden, Joseph Gil, Yoram Hirschfeld, and Amiram Yehudai. Towards a Mathematical Foundation For Design Paterns. Technical Report 1999-004, Department of Information Technology, Uppsala University, 1999.

[EGY96] Amnon Eden, Joseph Gil, and Amiram Yehudai. A Formal Language for Design Patterns. Technical Report WUCS-97-07, Washington University, 1996.

[EHY98] Amnon Eden, Yoram Hirschfeld, and Amiram Yehudai. LePUS - A Declarative Pattern Specification Language. Technical Report 326/98, Department of Computer Science, Tel Aviv University, 1998.

[Fea89] M. S. Feather. Detecting Interference when Merging Specification Evolutions. In *Proc. 5th Int'l Workshop Softw. Specification and Design*, ACM Press, 1989.

[Fis87] Gerhard Fischer. Cognitive View of Reuse and Redesign. *IEEE Software*, 4(4):60–72, 1987.

[Flo00] Gert Florijn. Anaja - tool support for architecture conformance checks and reviews. In *LAC (Landelijk Architectuur Congres)*, 2000.

[FMK96] Iain Ferguson, Edward Martin, and Burt Kaufman. *The Schemers Guide*. Schemers Inc., 1996.

[FMvW97] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool Support for Object-Oriented Patterns. In *Proceedings of ECOOP'97*, 1997.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FPR00] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. UML-F: A Modeling Language for Object-Oriented Frameworks. In E. Bertino, editor, *Proceedings of European Conference on Object-Oriented Programming*, pages 63–82. Springer-Verlag, 2000.

[FS97] Mohamed Fayad and Douglas C. Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10), 1997.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[GM95] Dipayan Gangopadhyay and Subrata Mitra. Understanding Frameworks by Exploration of Exemplars. In *Proceedings of 7th International Workshop on Computer Aided Software Engineering (CASE-95)*, pages 90–99. IEEE Computer Society Press, 1995.

[Gol00] Mitch Goldstein. *Hardcore JFC: Conquering the Swing Architecture*. Cambridge University Press, 2000.

[Gru97] Dennis Gruijs. A Framework of Concepts for Representing Object-Oriented Design and Design Patterns. Master's thesis, Utrecht University, 1997.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proceedings of the ECOOP/OOPSLA Conferences*, pages 169–180. Springer-Verlag, 1990.

[HHK+01a] Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paakki, Antti Viljamaa, and Jukka Viljamaa. Annotating Reusable Software Architectures with Specialization Patterns. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pages 171–180, 2001.

[HHK+01b] Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paakki, Antti Viljamaa, and Jukka Viljamaa. Generating application development environments for Java frameworks. In *Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering (GCSE'01)*, pages 163–176, 2001.

[Hol92] Ian M. Holland. Specifying Reusable Components using Contracts. In *Proceedings of the 6th European Conference on Object-Oriented Programming*, pages 287–308. Springer-Verlag, 1992.

[HPR89] S. Horwitz, J. Prins, and T. Reps. Integrating Non-Interfering Versions of Programs. *ACM Trans. Programming Languages and Systems*, 11(3):345–387, 1989.

[IEE95] IEEE. Revised Report on the Algorithmic Language Scheme, 1995.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future, The Story of Squeak, A Practical Smalltalk Written in Itself. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1997.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[JF88] Ralph Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1988.

[JO93] Ralph E. Johnson and William F. Opdyke. Refactoring and Aggregation. In *Proceedings of ISOTAS '93: International Symposium on Object Technologies for Advanced Software*, 1993.

[Joh92] Ralph Johnson. Documenting Frameworks Using Patterns. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1992.

[JR91] Ralph E. Johnson and Vincent F. Russo. Reusing Object-Oriented Designs. Technical Report UIUCDCS 91-1696, University of Illinois, 1991.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[Lea96] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison Wesley, 1996.

[LK94] Richard Lajoie and Rudolf K. Keller. Design and reuse in object-oriented frameworks: Patterns, contracts and motifs in concert. In *Proc. of the 62nd Congress of the Association Canadienne Francaise pour l'Avancement des Sciences (ACFAS)*, pages 94–105, 1994.

[Luc97] Carine Lucas. *Documenting Reuse and Evolution with Reuse Contracts.* PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 1997.

[MB99] Michael Mattsson and Jan Bosch. Observations on the Evolution of an Industrial OO Framework. In *Proceedings of the International Conference on Software Maintenance*, pages 139–145, 1999.

[MCK97] Matthias Meusel, Krysztif Czarnecki, and Wolfgang Köpf. A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and Hypertext. In *Proceedings of the European Conference on Object-Oriented Programming*, 1997.

[MDE97] Theo Dirk Meijler, Serge Demeyer, and Robert Engel. Making design patterns explicit in face. In *Proceedings of the Sixth European Software Engineering Conference*, pages 94–110. Springer–Verlag, 1997.

[Mei96] Marco Meijers. Tool Support for Object-Oriented Design Patterns. Master's thesis, Utrecht University, 1996.

[Men99] Tom Mens. *A Formal Foundation for Object-Oriented Software Evolution.* PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 1999.

[Men00] Kim Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming.* PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2000.

[Men02] Tom Mens. A State-of-the-Art Survey on Software Merging. *Transactions on Software Engineering*, 28(5), May 2002.

[Mic96] Sun Microsystems. JEEVES: Java-powered Internet server and framework, 1996.

[Min97] N. H. Minsky. Towards Architectural Invariants of Evolving Systems. Technical report, Rutgers University, 1997.

[MLS99] Tom Mens, Carine Lucas, and Patrick Steyaert. Supporting Disciplined Reuse and Evolution of UML Models. In *Proceedings of UML'98 International Workshop*, 1999.

[MM97] Thomas Mowbray and Raphael Malveau. *Corba Design Patterns.* Wiley Computer Publishing, 1997.

[MMW01] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting Software Development through Declaratively Codified Programming Patterns. In *Proc. Int. Conf. Software Engineering and Knowledge Engineering*, 2001.

[MMW02] Kim Mens, Tom Mens, and Michel Wermelinger. Supporting Unanticipated Software Evolution Through Intentional Software Views. In *Proceedings of the First International Workshop on Unanticipated Software Evolution*, 2002.

[MN96] Simon Moser and Oscar Nierstrasz. The Effect of Object-Oriented Frameworks on Developer Productivity. *IEEE Computer*, 29(9):45–51, 1996.

[MNS95] G. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-level Models. In *Proceedings of SIGSOFT 1995, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[MT01] Tom Mens and Tom Tourwé. A Declarative Evolution Framework for Object-Oriented Design Patterns. In *Proc. Int. Conf. Software Maintenance*, 2001.

[OC00] Alvaro Ortigosa and Marcelo Campo. Using Incremental Planning to Foster Application Frameworks Reuse. *International Journal of Software Engineering and Knowledge Engineering*, 10(4):433–448, 2000.

[O'C01] Mel O'Cinnéide. *Automated Application of Design Patterns: a Refactoring Approach.* PhD thesis, University of Dublin, Trinity College, 2001.

[OCN99] M. O Cinnéide and P. Nixon. A Methodology for the Automated Introduction of Design Patterns. In *Proceedings of the International Conference on Software Maintenance*, 1999.

[OCS99] Alvaro Oritgosa, Marcelo Campo, and Roberto Moriyon Salomon. Enhancing Framework Usability through Smart Documentation. In *Proc. of the Argentinian Symposium on Object Orientation*, pages 103–117, 1999.

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

[OQC97] Georg Odenthal and Klaus Quibeldey-Cirkel. Using Patterns for Design and Documentation. In *Proceedings of the European Conference on Object-Oriented Programming*, 1997.

[OR93] W.F. Opdyke and Johnson R.E. Creating Abstract Superclasses by Refactoring. In *In Proceedings of Computer Science Conference*, pages 66–73. ACM Press, 1993.

[PC95] M. Potel and S. Cotter. Inside Taligent Technology, 1995.

[PPSS95] Wolfgang Pree, Gustav Pomberger, Albert Schappert, and Peter Sommerlad. Active Guidance of Framework Development. *Software - Concepts and Tools*, 16(3), 1995.

[Pre94] Wolfgang Pree. Meta Patterns – A Means For Capturing the Essentials of Reusable Object-Oriented Design. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 150–162. Springer-Verlag, 1994.

[Pre95] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley/ACM Press, 1995.

[Pre97] Wolfgang Pree. Essential Framework Design Patterns. *Object Magazine*, 1997.

[PU98] Lutz Prechelt and Barbara Unger. A Series of Controlled Experiments on Design Patterns: Methodology and Results. In *Proc. Softwaretechnik'98, GI Conference, Paderborn*, pages 53–60, 1998.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 1997.

[Ret91] Mark Rettig. Nobody Reads Documentation. *Communications of the ACM*, 34(7):19–24, 1991.

[RJ96] Don Roberts and Ralph Johnson. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. In *Proceedings of Pattern Languages of Programs*, 1996.

[Rob99] Don Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois at Urbana Champaign, 1999.

[Sch95] H. A. Schmid. Creating the Architecture of a Manufactoring Framework by Design Patterns. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, ACM Sigplan Notices, pages 370–394, 1995.

[SLCM00] U. Schultz, J. Lawall, C. Consel, and G. Muller. Specialization Patterns. In *Proceedings of the 15 th IEEE International Conference on Automated Software Engineering*, pages 197–206, 2000.

[SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings of the OOPSLA Conference*, ACM Sigplan Notices, 1996.

[Som96] Ian Sommerville. *Software Engineering Fifth Edition.* Addison-Wesley, 1996.

[Szy97] Clemens Szyperski. *Component Oriented Programming.* Addison-Wesley, 1997.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[Tal94] Taligent. *Taligent's Guide to Designing Programs: Well Mannered Object-Oriented Design in C++.* Addison-Wesley, 1994.

[TB95] Lance Tokuda and Don Batory. Automated Software Evolution via Design Pattern Transformations. In *Proceedings of the 3rd International Symposium on Applied Corporate Computing*, 1995.

[TB99] Lance Tokuda and Don Batory. Automating Three Modes of Evolution for Object-Oriented Software Architecture. In *Proceedings of the COOTS'99*, 1999.

[TDM99] Tom Tourwé and Wolfgang De Meuter. Optimizing Object-Oriented Languages Through Architectural Transformations. In *Proceedings of the 8th International Conference on Compiler Construction*, pages 244–258, 1999.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring.* PhD thesis, University of Berne, 2001.

[TN01] Taibi Toufik and David Chek Ling Ngo. Why and How Should Patterns Be Formalized. *Journal of Object-Oriented Programming (JOOP)*, 14(4), 2001.

[vGB01] Jilles van Gurp and Jan Bosch. Design Erosion: Problems & Causes. *Journal of Systems & Software*, 61(2):105–119, 2001.

[vW96] Pieter van Winsen. (Re)engineering with Object-Oriented Design Patterns. Master's thesis, Utrecht University, 1996.

[WGM89] Andre Weinand, Erich Gamma, and Rudolph Marty. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2):63–87, 1989.

[WRS90] D. Wilson, L. Rosenstein, and D. Shafer. Programming with MacApp, 1990.

[Wuy98] R. Wuyts. Declarative Reasoning about the Structure of Object-Oriented Systems. In *Proceedings TOOLS USA'98, IEEE Computer Society Press*, pages 112–124, 1998.

[Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2001.

[YCM78] S. S. Yau, J. S. Collofello, and T. MacGregor. Ripple Effect Analysis of Software Maintenance. In *Proc. COMPSAC Conf.* IEEE Computer Society Press, 1978.