TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY
Julkaisu 521 • Publication 521

Juha Hautamäki

Pattern-Based Tool Support for Frameworks Towards Architecture-Oriented Software Development Environment

Tampere 2005

Tampereen teknillinen yliopisto. Julkaisu 521
Tampere University of Technology. Publication 521

Juha Hautamäki

Pattern-Based Tool Support for Frameworks Towards Architecture-Oriented Software Development Environment

Thesis for the degree of Doctor of Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB104, at Tampere University of Technology, on the 4th of February 2005, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2005

ISBN 952-15-1299-7 (printed)
ISBN 952-15-1508-2 (PDF)
ISSN 1459-2045

Abstract

Software engineering aims at techniques for producing better software products with less resources. A central concept for achieving this goal is a product line architecture. Frameworks are a popular object-oriented way to implement product line architectures. However, frameworks are often difficult to learn and their specializations consist of small and crosscutting logical entities that overlap with other design solutions of the software product. Implementation becomes fragmented, difficult to trace, and the original reasoning of the design is easily forgotten. Thus, the essential problems to be solved are the following:

- How to teach the software developer to understand different frameworks and design principles in the context of her software product?
- How to guide the software developer to use frameworks and product line architectures?
- How to maintain and document implemented design solutions and framework specializations?

In this dissertation it is argued that a practical pattern-based approach can be used to support the software developer to learn, implement, and sustain design solutions in her software project. Instructions, like how to use a particular framework, can be given as simple pattern specifications; a tool takes these specifications as input and generates tasks as output. The generated programming tasks will guide the software developer to gradually instantiate the patterns, repair possible violations, and in that way to adopt and maintain the design. Integrating the pattern-based tool support with a common software development environment makes that environment architecture-oriented in terms of the patterns used.

The main contributions of this dissertation are the following:

- Participation in the development of the Fred/JavaFrames tool concept.
- A description of a general pattern-based tool support that allows the use of the pattern concept in different software development environments, making the environment architecture-oriented.
- Integration of such a tool platform into the Eclipse environment.
- A goal-oriented process to use the pattern-based tool support to specialize frameworks.

- A specification of the extension interface of the tool platform using the tool itself.

- An evaluation of the pattern-based approach for framework engineering using case studies.

Keywords: framework, framework specialization, pattern, product line architecture, programming environment, separation of concerns, software architecture, software development environment, software engineering, software reuse, variability management.

Acknowledgements

First of all, I would like to thank my supervisor Professor Kai Koskimies for his support. Kai encouraged me to start writing this thesis and he gave me a lot of useful feedback and pieces of advice during the writing process. I am also grateful to Professors Tarja Systä and Tommi Mikkonen for reading the draft version of my dissertation. They gave me a number of good comments and proposals of improvements. I would also like to thank my thesis reviewers Professors Eila Niemelä and Marcelo Campo for their constructive comments and feedback.

In addition, I would like to thank the members of the research team. During the past few years, I have had a privilege to work with Markku Hakala, Esa Heino, Imed Hammouda, Miko Nieminen, Jukka Paakki, Mika Pussinen, Pekka Savolainen, Jyrki Tuomi, Antti Viljamaa, and Jukka Viljamaa. This dissertation is closely related to other theses and papers written by the research group. Developing our pattern methodology and tools has been intensive and interesting teamwork.

This thesis is a result of prolonged co-operation between academe and industry. It emerges from the research projects that created the JavaFrames system (formerly known as Fred). First versions of the system were released during the FRED project (FRamework EDitor for Java – Support for Framework Development and Use) 1997-1999. The project was carried out at University of Tampere and University of Helsinki. The project was financially supported by Finnish National Technology Agency (TEKES) and several industrial partners: Dycom, ICL Data, Major Blue Company, Necsom, Nokia Telecommunications, Nokia Research Center, Novo Group, Profit, Stonesoft, Sun Microsystems, Sonera, TietoEnator, and Valmet Automation. Since then, the methodology was further evolved and tested with the Nokia Networks case study in the JavaFrames project between Tampere University of Technology and University of Helsinki. The financiers of the JavaFrames project were TEKES, Nokia Networks, Necsom, Sensor SC, and SysOpen. I have also been financially supported by Tampere Graduate School in Information Science and Engineering (TISE).

Finally, I want to thank my parents for their support during my studies.

CONTENTS

CHAPTER 1

INTRODUCTION

Software engineering aims at techniques for producing better software products with less resources. This encourages the reuse of existing architectural solutions and software systems. A central concept for achieving this goal is product line architecture [Bosch 2000; Jazayeri et al. 2000; Clements and Northrop 2001], which describes a family of software products and captures the variability between them. Object-oriented frameworks [Fayad et al. 1999], in turn, are a popular way to implement this variability. With a framework, different products in the product family are created by implementing and configuring the application-specific parts of the product.

However, frameworks are often difficult to learn because their use consists of small and crosscutting logical entities that overlap with other design solutions of the software product. It is not always obvious which parts of the framework should be customized and how. The use of the framework may become fragmented, difficult to trace, and the original reasoning of the application-specific solutions is easily forgotten. For human it is hard to know and predict every detail of the complex use of the framework and to remember those details afterwards. Because of this, software product lines and frameworks need tools and techniques to manage the variability and traceability. Besides traditional interfaces and class hierarchy, such tools promote the crosscutting and fragmented design solutions and support their implementation, documentation, and maintenance during the software development process.

This dissertation presents a practical pattern-based [Alexander et al. 1977; Alexander 1979] approach to help the software developer to use frameworks in her software project. Instructions, like how to use a particular framework, are given as simple pattern specifications, which are precise pattern descriptions that can be used and manipulated by a tool. A tool takes these specifications as input and generates tasks as output. The generated programming tasks will guide the software developer to gradually instantiate the patterns, repair possible violations, and in that way to use the framework. Simultaneously, the software developer can concentrate on her software project as a whole, letting the tool to manage and generate much of the uninteresting implementation details. In addition, if creating a new framework, the software developer herself can use the tool to describe and document the intended usage of the framework, making it easier to learn and reuse by other software developers. Integrating the pattern-based tool support with a common software development environment makes that environment architecture-oriented in terms of the patterns used.

In this dissertation it is argued that a practical pattern-based approach can be used to support the software developer to learn, implement, and sustain design solutions in her software project. How could this kind of tool support be constructed and how it works? What are the true benefits and problems of such a system? Is it scalable for a wide range of application-domains? Can it be used from the highest level of architecture to the lowest level of implementation? In which stage of the software development process? These, and related questions are discussed in this dissertation.

The context of this dissertation is discussed in Section 1.1. The addressed problems and questions are given in Section 1.2. As a solution, the concept of patterns and a general pattern-based tool support are introduced in Section 1.3. Overview and contributions of this dissertation are given in Section 1.4.

1.1 Context

The context of this dissertation is the use of software architectures, product line architectures, and particularly the reuse of their concrete implementations in the form of object-oriented frameworks. One of the main goals of this dissertation is to describe a pattern-based approach to support this reuse. The context is introduced in the following subsections:

- *Software development process*. The aim of the software development process is to create software products. The ultimate goal of software engineering is to make this process faster, cheaper, more reliable, and improve the quality of the implemented software product. Subsection 1.1.1.

- *Software architecture*. Software architecture is a view of the whole design of the software product. It describes the high-level structure and behavior of the implementation. Subsection 1.1.2.

- *Product line architecture*. A product line architecture is a software architecture to create a family of software products. It promotes reuse of design solutions. Subsection 1.1.3.

- *Frameworks and middleware*. The software development process may utilize existing software, like object-oriented frameworks and middleware. They are used to implement product line architectures. Subsection 1.1.4.

1.1.1 Software Development Process

Designing and implementing a software system is hard. The software development process is a creative human activity, in which a group of people try to solve conceptual and computational problems in order to construct a working software product. Jacobson et al. [1999, p. 24] defines the software development process as follows:

> "A software development process is a definition of the complete set of activities needed to transform a user's requirements into a consistent set of artifacts that represent a software product and, later, to transform changes in those requirements into a new, consistent set of artifacts."

The software development process is an extensive concept. Besides implementation and coding, it includes also the used technology, project organization, and marketing issues. According to Fuggetta [2000], the software development process includes the following concepts:

- *Software development technology*. Technological support used in the process, including tools, infrastructures, and environments.

- *Software development methods and techniques*. Guidelines on how to use technology and accomplish software development activities.

- *Organizational behavior*. In general, software development is carried out by teams of people that have to be coordinated and managed within an effective organizational structure.

- *Marketing and economy*. Software must address real customer's needs in specific market settings.

Thus, the software development process is a complex effort with many aspects, from the applied tools, programming languages, and development environments to different design techniques, project management issues, and marketing strategies. From the viewpoint of the software developer concentrating on the design and implementation issues, the software development process includes all stages from finding and understanding the requirements to implementing, documenting, and maintaining the software product.

In this thesis, we will show that a pattern-based tool support can provide technological assistance to select, learn, program, document, and maintain design solutions on the implementation level.

1.1.2 Software Architecture

There are numerous definitions of software architecture, stressing that the architecture represents the design of a software product. For instance, Buschmann et al. [1996, p. 384] defines software architecture as follows:

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

IEEE [2000], in turn, gives the following definition:

> "Software architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution."

In practice, when planning a software product, software developers create and use different models (views) to figure out the system characteristics. Depending on the needs and skills of the software developer, modeling can be anything from informal discussions and superficial drafts to more precise and formal representations. For example, the Unified Software Development Process [Jacobson et al. 1999] includes business model, use-case model, analysis model, design model, deployment model, implementation model, and test model. Typically, models are related to each other; a single model may be based on other models and it describes a particular thing or feature belonging to the software architecture. Together the models describe the software architecture and define the structure and functionality of the software product.

According to Garlan [2000], software architecture plays an important role in at least six aspects of software development: understanding, reuse, construction, evolution, analysis, and management. It has also a key role as a bridge between requirements and implementation of the software system. The architecture may take different forms that are complementary to each other, like a class diagram and a statechart diagram. In an optimal situation, it provides sufficient and up to date information about the software product.

In this dissertation it is shown that a pattern-based tool support can be used to establish the connections between architectural solutions and the elements of the concrete software product. We believe that this kind of concrete and practical tool support is useful, as the design solutions tend to be overlapping and fragmented structures, making their implementation and maintenance difficult with traditional software development environments.

1.1.3 Product Line Architecture

Product line architecture [Bosch 2000; Jazayeri et al. 2000; Clements and Northrop 2001] is software architecture for a family of software products. In the software product family, individual products share common parts and functionality, but some parts are different and must be customized. Jazayeri et al. [2000, p. 27] defines a product line architecture as follows:

> "A product family software architecture (a product line architecture) defines the concepts, structure, and texture necessary to achieve variation in features of variant products while achieving maximum sharing parts in the implementation."

One of the key issues of software product lines is *variability management*. According to Gurp et al. [2001], the aim of variability management is to change, customize, or configure a software system for use in a particular context. Jacobson et al. [1997, pp. 440], in turn, defines the concepts of *variant* and *variation point* as following:

> "A variant is a type-like construct, typically use case or object type or class, intended to be inserted at an appropriate variation point to specialize an abstract type or class."

> "A variation point identifies one or more locations at which variation will occur within a class, type or use case."

A product line architecture emphasizes variability management, i.e., the use of variants and variation points. Frameworks, discussed in the next subsection, are an object-oriented way to implement this variability. In this dissertation it is proposed that the pattern-based tool support can help the variability management by guiding the software developer to use frameworks. Particularly, as will be shown, the tool can make the connections between the framework and the derived applications more explicit.

1.1.4 Frameworks and Middleware

The importance of good software architecture and the benefits of reusing approved architectural solutions are widely recognized in software engineering. Object-oriented frameworks [Fayad et al. 1999] are a way to improve the quality and effectiveness of the software development process by reusing and standardizing existing knowledge. In literature, using a framework is often called *framework specialization*, *framework adaptation*, or *framework instantiation*. Throughout this thesis the term framework specialization is used to mean the process to implement an application or a part of it with a framework.

Typically, a framework implements the crucial parts of a product line architecture and captures the programming expertise necessary to solve problems in a particular problem domain. Technically, Johnson and Foote [1988] define a framework as follows:

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

Another definition is given, for example, by Roberts and Johnson [1996]:

> "Frameworks are reusable designs of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate."

Thus, a framework is a kind of class library, but with the framework the flow of control is bi-directional between the application and the library. This feature is achieved by dynamic binding in object-oriented languages where an operation can be defined in a library class or interface but implemented in a subclass in the application. A framework can be used to implement a part of a system, such as the application's user interface, though application specific frameworks sometimes describe the domain of an entire application.

Middleware [Schmidt and Buschmann 2003], in turn, is a piece of software that increases reuse by providing usable, standard solutions to common problems, like how to implement a persistent storage. Examples of middleware standards are Common Object Request Broker Architecture (CORBA) [CORBA 2004] and Java 2 Enterprise Edition (J2EE) [J2EE 2004]. Typically, middleware system may include a number of frameworks, class libraries, pre-implemented components, and documentation.

We will show that the pattern-based tool support proposed in this dissertation can be used to map the desired use of frameworks and middleware systems. For example, a set of pattern specifications can describe how to use the framework's specialization interface to program applications. By using the given specifications, a tool can support the framework specialization and re-use in other software projects.

1.2 Problems

Frameworks are an object-oriented way to support product line architectures. However, working with frameworks is not always straightforward. The main problems to use frameworks are the following:

- *Understanding fragmented design solutions of frameworks*. How to teach the software developer to understand different frameworks and design principles in the context of her software product? Subsection 1.2.1.
- *Specializing complex frameworks*. How to guide the software developer to use frameworks and product line architectures? Subsection 1.2.2.
- *Preserving coherence of framework-based applications*. How to maintain and trace implemented design solutions and framework specializations? Subsection 1.2.3.

1.2.1 Understanding Fragmented Design Solutions of Frameworks

When using a framework the software developer often wants to design, implement, or learn a certain part or a subsystem of the wholeness and tries to figure out the required participants and their interactions. Typically, the solution constitutes a crosscutting structure inside the application and between the application and the framework. For example, Bosch [2003] discusses *design fragments*, which are logical units that capture a certain portion of design, its participants and their interactions. Here the design fragment is a synonym for the design solution, which emerges from the practical demands of the software developer during the use of the framework. Such a design solution consists of elements and relationships that are needed to construct a thing that is meaningful from the viewpoint of the application.

So, why it is difficult for the application developer to understand a particular design solution? As discussed by Tarr et al. [1999], all formalisms, including object-oriented languages, tend to have restricted sets of decomposition and composition mechanisms. This means that they support only a single, dominant view at a time. In the case of object-orientation, decomposition is based on objects, while in procedural programming languages it is based on functions. This “tyranny of the dominant decomposition” causes some problems:

- *Scattering*. A single design solution may affect multiple design and code modules. In the case of object-oriented languages, the solution is scattered across classes and their internals.

- *Tangling*. Multiple design solutions may affect the same implementation unit. That is, a particular block of code may be involved in a number of solutions.

Traditional software development environments and programming languages do not support the scattering and tangling problems directly. Instead, additional documentation is needed to map the fragmented design solutions. We will show that a pattern-based tool support can be used to highlight and maintain these solutions, thus reducing the scattering and tangling problems.

1.2.2 Specializing Complex Frameworks

Reusing approved design improves the software development process. However, reusing existing solutions requires knowledge and experience. The software developer must know which design to reuse and how it should be reused. For example, to specialize a framework may require that the application developer uses different components to configure and compose her application. In addition, she may have to derive new subclasses from the abstract base classes of the framework, implement required operations, and so on. This can be difficult, as the framework may contain a number of rules and conventions that cannot be determined directly from its class hierarchy and interfaces.

Thus, from the viewpoint of the application developer, a typical problem when specializing a framework is that the framework is often hard to learn and comprehend. However, she must know the purpose and invisible rules of the framework in order to use it in the context of her application. Due to the fragmented design solutions it may be very laborious to figure out how the framework should be used. In addition, after specializing the framework, it is difficult to ensure that the implemented solution is not violated later, when the application evolves.

We believe that a pattern-based tool support can be used to support the reuse of approved design solutions and framework specialization. A tool can guide the application's implementation and check that the specialization obeys the framework-specific rules. The system ensures that the framework is understood and correctly used in different software projects.

1.2.3 Preserving Coherence of Framework-Based Applications

One of the essential problems of the software development process is how to manage the evolution of the software product; how to keep different models, documentation, and implementation consistent? This is difficult, because the software development process is often iterative, making it necessary to modify existing models and implementation. In addition, an application is often based on multiple frameworks; how to keep the different framework specializations consistent when the application evolves during the software development process? More precisely, Gurp and Bosch [2002] enumerate several reasons for design erosion:

- *Traceability of design decisions*. Typically, connections between different models and the implementation have no direct and explicit form.
- *Increasing maintenance cost*. Design decisions may be suboptimal either because software developers do not understand the architecture or because a more optimal decision would be too effort demanding.
- *Accumulation of design decisions*. Whenever a decision needs to be revised, other design decisions may need to be reconsidered as well.
- *Iterative methods*. Software development is often iterative because of unforseen requirement changes and suboptimal design decisions.

Unfortunately, current software development environments do not usually support the demand for traceability between different design models, frameworks, and the final application. Ideally, besides the traceability between the application's design models and source code, the implementation should also be traced to the used frameworks and product line architectures. Change in the source code should be checked against the design models and framework's specialization rules.

Clearly, new kinds of tools are needed to solve the traceability problem. As suggested in this dissertation, this could be done by using patterns as a bridge between higher-level specifications and their implementation. In the case of product line architectures and framework specialization, patterns could be used to validate and point out the implemented variants in the application's source code, and in that way to reduce the overall design erosion. However, the software developer would still be responsible for finding the requirements and selecting the most suitable design decisions and frameworks.

1.3 Pattern-Based Tool Support

This dissertation oulines the characteristics of a general tool platform for patterns that supports framework specialization and the reuse of approved design solutions. The approach is introduced in the following subsections:

- *Patterns as a solution*. During the software development process, patterns can be used to document and apply approved design solutions. Subsection 1.3.1.
- *Towards an architecture-oriented environment*. If patterns are transformed into a more specific form, a tool can manipulate and support their use. This leads to a pattern-based architecture-oriented software development environment. Subsection 1.3.2.

1.3.1 Patterns as a Solution

In software engineering, patterns and pattern languages [Alexander et al. 1977; Alexander 1979] can be used to document the crosscutting structures of a software architecture and the way that those structures should be implemented. Originally, the pattern concept was introduced by the architect Christopher Alexander; he defines a pattern as follows:

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice." [Alexander et al. 1977, p. x]

> "The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it." [Alexander 1979, p. 247]

Though Alexander's idea of patterns was intended to create buildings and towns, it has been more successfully used in software industry. Together related patterns form a pattern language, in which patterns may refer to each other, until the solution has been implemented. Coplien [2004] defines a pattern language as follows:

> "A pattern language defines a collection of patterns and the rules to combine them into an architectural style. Pattern languages describe software frameworks or families of related systems."

Thus, a pattern is descriptive, generative, and informal documentation, which describes, on general level, a problem that occurs in a certain context, and a recipe-like solution to solve that problem. Like instantiating a class, a concrete manifestation of a pattern is called a *pattern instance*. It contains the elements and their interactions that participate in the solution described by the pattern.

Patterns are often confused with frameworks. A framework is a more concrete software product, implementing product line architecture, while a pattern is an abstract informal documentation. A framework may utilize several patterns in its design and contain several pattern instances in its implementation. Patterns can also be used to describe smaller architectural entities, like the use of a particular extension point (or hot spot) of the framework. In this dissertation it will be demonstrated how patterns can be used to document a framework's specialization interface; that is, how the customisable parts of the framework should be implemented in order to create a working application.

Patterns encapsulate logical steps to generate solutions for different problems. The scale of the problems varies; a problem may be how to specialize a framework, or how to iterate a data structure, or how to separate the user interface issues from the rest of the application logic. In any case, patterns offer common vocabulary and understanding, supporting the software development process by making it easier for software developers to communicate and implement complex design. For example:

- *Using patterns to design software architecture.* In general, designing good software architecture requires a substantial amount of skills and experience. The use of patterns does not remove the burden of design, but selecting and using a suitable pattern may help to solve some of the design problems. A good example of architectural patterns is the MVC (Model-View-Controller) pattern [Krasner and Pope 1988]; it makes a standardized separation between the graphical user interface and the rest of the application.

- *Using patterns to implement design solutions.* At some point the blueprint and models must be used to build the software system. Implementation may be error-prone and requires substantial amount of programming skills. Patterns are generative; recipe-like patterns can be used to guide the implementation.

- *Using patterns to document design solutions.* Original software developers may leave the company and new members of the development team must learn the software system as soon as possible. In the case of reuse, other software developers should learn typical use cases of the framework or middleware with reasonable efforts. Patterns are descriptive; they are especially useful for teaching and documenting purposes, to augment the more traditional documentation. As an example, Johnson [1992] has used patterns to document the design and the use of a framework.

- *Using patterns to maintain design solutions.* The software development process is often iterative as requirements change when the understanding of the problem domain grows. This causes the software system to evolve. However, there is a danger of design erosion if different models, the implementation, and the documentation are not continuously and carefully updated and reconsidered. Managing the consistency is a difficult and time-consuming task. Patterns can help, as they make fragmented design solutions more understandable. The used patterns emphasize crosscutting behavior and complex solutions that would otherwise be hard to comprehend.

1.3.2 Towards an Architecture-Oriented Environment

We believe that a tool that systematizes the use of patterns is useful in the software development process. Especially, if integrated with a real software development environment, such a pattern-based tool support can generate, document, and manage pattern instances, reducing the problems with design erosion and fragmented design solutions. If the software developer makes a mistake or wants to change or refine her solution, the system can utilize the applied patterns to keep track of the involved software elements. Also, if the user just wants to learn how a particular piece of design should be implemented and how it works, she can use the system as a mentor, experiment with real examples, and learn the design by instantiating and studying the underlying patterns.

JavaFrames (formerly known as Fred) [Hakala et al. 2001b, 2001c, 2001d; Hautamäki 2002; Viljamaa A. 2001, 2004; Viljamaa J. 2002, 2003, 2004; JavaFrames 2004] is an integrable system where patterns are used as simple but precise specifications. JavaFrames takes these pattern specifications as input and generates programming tasks as output to instantiate the selected patterns. Depending on the current stage of the pattern instantiation, each task may require some actions from the software developer; she may have to create or select a new subclass, implement a new operation, fix violations, like the operation's return type, and so on. Each task provides also some documentation and implementation hints, like the possibility to generate source code.

In JavaFrames, the list of pattern instantiation tasks is not linear or predetermined. Instead, normal programming actions, like changing or removing a portion of source code, have effects on the pattern instantiation, generating new tasks, documentation, and proposed default implementations. In this way, the pattern instantiation is carried out by adapting the selected patterns gradually to the current application. When integrated to a real software development environment, JavaFrames brings the idea of patterns into every day use and enables a progressive pattern instantiation that agrees with the idea of *piecemeal growth* [Alexander et al. 1975], in which the whole emerges from local acts.

Based on the gained experiences with JavaFrames and different case studies, this dissertation outlines the characteristics of a general tool platform for patterns. The idea is illustrated as an UML package diagram [Booch et al. 1999] in Figure 1. The core of the platform must provide algorithms, services, and concepts to create, instantiate, and manage pattern specifications. Various pattern semantics provide building blocks to model these pattern specifications, for example, to support solutions in different programming languages and application domains. Different specifications can then be constructed to guide the application developer how to achieve design solutions. As a platform, the system provides a common foundation to implement a rich set of pattern tools, for instance, to develop, instantiate, analyze, and document pattern specifications. For the software developer the use of patterns must be as easy as possible. Therefore the platform and the implemented pattern tools should be seamlessly integrated into some software development environment.

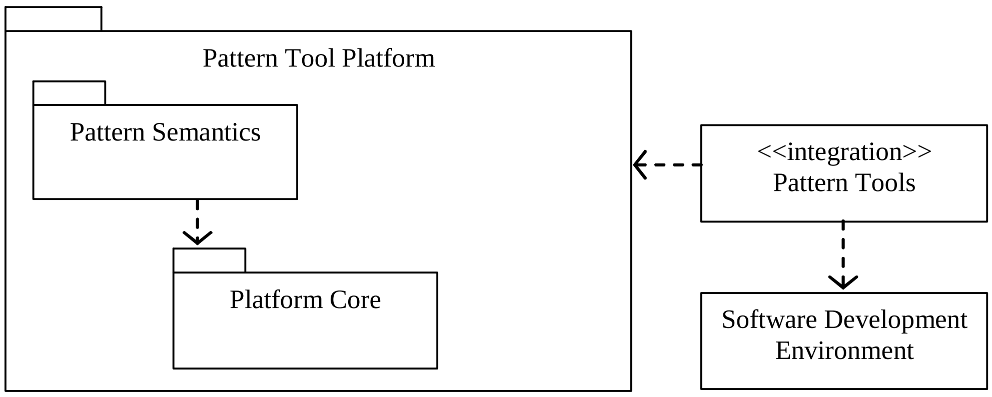


Figure 1. A general pattern-based tool support.

By integrating the pattern tool platform into a software development environment makes that environment architecture-oriented in terms of the pattern specifications used. Such an environment acts like an architecture-sensitive compiler or interpreter that reports conflicts with the architecture described by the underlying pattern specifications.

1.4 Contributions

The main contributions of this dissertation are the following:

- Participation in the development of the Fred/JavaFrames tool concept. As a member of the research group, the author has been responsible of developing the user interface framework for the pattern tools. The author has also carried out different case studies [Hautamäki 2002] to test the developed tools and methodology. The core of the Fred/JavaFrames pattern system, including an algorithm to instantiate patterns, has been implemented by other team members [Hakala et al. 2001d; Hakala 2002; Viljamaa A. 2004; Viljamaa J. 2004].

- A description of a general pattern-based tool support that allows the use of the pattern concept in different software development environments, making the environment architecture-oriented. The outlined platform is an integrable and universal approach that combines many of the best practices and experiences gained from the pattern literature and from the Fred/JavaFrames tool concept. The platform can be augmented with wide range of pattern tools and the ways of using patterns (Chapter 4).

- Integration of such a tool platform into the Eclipse environment. The author has integrated the JavaFrames system into the Eclipse software development environment (Chapter 5).

- A goal-oriented process to use the pattern-based tool support to specialize frameworks. During an industrial case study, the author has developed a goal-oriented approach to find, construct, and use pattern specifications with JavaFrames. This goal-oriented approach is a practical way to document the intended use of a framework (Chapter 6).

- A specification of the extension interface of the tool platform using the tool itself. JavaFrames can be extended with new pattern tools and semantics. The author has implemented a set of patterns to create new pattern semantics and to add these extensions to JavaFrames (Chapter 7).

- An evaluation of the pattern-based approach for framework engineering using case studies (Chapter 7 and Chapter 8).

As a background, patterns in software engineering are discussed in Chapter 2 and object-oriented frameworks are discussed in Chapter 3. Related work is discussed in Chapter 9. Conclusion is given in Chapter 10.

CHAPTER 2

PATTERNS IN SOFTWARE ENGINEERING

The concept of a pattern is central throughout this dissertation. It is based on the work of the architect Christopher Alexander [Alexander et al. 1977; Alexander 1979] and a number of authors that have used and developed the concept in the field of software engineering. Particularly, in the object-oriented community, patterns have been used as a methodology to document the best practices and experiences of object-oriented software systems. Perhaps the most well-known examples of this are the design patterns [Gamma et al. 1995] that describe solutions to common object-oriented design problems. As said by Gamma et al., none of the design patterns describe new or unproven design. Instead they record experience, the folklore of the object-oriented community, in a form that people can use effectively. Thus, instead of rediscovering a solution to a common problem, the software developer can use a suitable pattern to reproduce the solution in the application-specific context.

Patterns were briefly discussed in Section 1.3. A more detailed introduction to the pattern concept and its applicability in software engineering is given in Section 2.1. Using patterns to create new pattern instances and the process to find the applied patterns from existing software systems are discussed in Section 2.2. To enable pattern-based tool support, patterns must be presented precisely with some specification language or formalism; this is discussed in Section 2.3.

2.1 Introduction to Patterns

The original pattern concept and the ways it has been used in the field of software engineering are discussed in the following subsections:

- *Alexander's pattern concept*. The architect Christopher Alexander developed the pattern concept to create buildings and towns. However, the concept has been succesfully adopted in software engineering. Subsection 2.1.1.

- *Variety of patterns*. A number of authors have developed the idea of patterns and invented more sophisticated concepts and forms to use them. Subsection 2.1.2.

- *Elements of a pattern*. Despite of different forms and variations, patterns share a common set of features that can be found from their descriptions. Subsection 2.1.3.

- *Example*: *Abstract Factory pattern*. The well-known Abstract Factory pattern [Gamma et al. 1995, p. 87-95] is used as an example in this dissertation. It is employed to demonstrate the general pattern-based tool support discussed in Chapter 4. Subsection 2.1.4.

- *About writing patterns*. Writing patterns is a creative human activity for human audience. Subsection 2.1.5.

2.1.1 Alexander's Pattern Concept

Originally, the architect Christopher Alexander developed the idea of patterns to enable people to design their own homes and communities [Alexander 1979, 1981; Alexander et al. 1975, 1977, 1985, 1987]. His pattern language was a set of patterns, each describing how to solve a particular kind of a design problem. The pattern language starts at a very large scale, explaining how the world should be broken into nations and nations into smaller regions, and goes on to explain how to arrange roads, parking, shopping, places to work, homes, and so on. The patterns focus on finer and finer levels of detail where each pattern was written in a particular format, leading into the next one(s). The patterns in the pattern language were not only descriptive but also generative; besides describing the architecture they also described how to implement the architecture in practice.

Intuitively, a pattern describes recurring solution that has stood the test of time. Each pattern is an essay that describes a problem to be solved, a solution, and the context in which that solution works. A pattern describes costs and benefits of the solution and makes design trade-offs explicit. It names a technique and gives people a common vocabulary to discuss their designs. Patterns may refer to other patterns and constitute a pattern language. Patterns are descriptive, generative, and informal documentation to be used by humans, but if the core of that information could be precisely specified, it could be computationally manipulated and the use of patterns could be automatically supported and documented.

Beck and Cunningham [1987] applied the concept of patterns to the development of graphical user interfaces in Smalltalk. Since then, patterns have been succesfully used to describe and document object-oriented design. Obviously, as widely embraced in the object-oriented community [Lea 1994; Gamma et al. 1995; Coplien 1996; Harrison N. et al. 1999], the concept of a pattern, though exemplified with architectural artifacts, is suitable to describe software architectures. Lea [1994] enumerates a number of differences between building a house and building a software, but adds that most of the differences are matters of degree:

- Software entities engage in greater dynamic interaction (e.g., send messages to each other).
- Sometimes, describing software is the same as constructing it (as in programming).
- More of a software design is hidden from its users.
- Software generally has many fewer physical constraints.
- Some software requirements are allegedly more explicit and precise than "build a house here".

Lea continues that ideally a pattern has the following properties:

- *Encapsulation*. A pattern encapsulates a well-defined problem and solution.
- *Generativity*. A pattern describes how to construct its realizations (pattern instances).
- *Equilibrium*. By balancing forces and constraints, a pattern attempts to minimize the downsides of the solution and maximize the benefits.
- *Abstraction*. A pattern represents abstraction of empirical experience and everyday knowledge.
- *Openness*. Patterns have no top or bottom; a pattern can be extended down to arbitrarily fine levels of detail.

- *Composibility*. Patterns are hierarchically related. A pattern language expresses this layering.

Patterns can be seen as parts or building blocks of a software architecture, describing and encapsulating architectural rules and implementation instructions. Alexander [2004] himself discusses how patterns are used in software. According to Alexander, people have done the obvious thing and developed patterns that are a prescription of how to solve particular problems that come up in the software development process. A pattern language provides common vocabulary, common base of understanding what's important in programming, and a large corpus of solutions that make software developers effective.

A typical pattern encapsulates an approved and elegant object-oriented solution, which looses the coupling between the participating components in the solution. This lessens the need for class-refactoring and re-design during later implementation, when the software system evolves. For instance, Helm [1995] has noticed that designs based on patterns seem to be more reusable and more tolerant of requirement changes. In addition, patterns provide a valuable teaching resource to improve the productivity and knowledge of the software developers. Thus, in general, using patterns improves the understandability of the software system and makes it flexible.

Vlissides [1997] discusses some of the most common misconceptions about patterns. He says that patterns do not guarantee reusable software, higher productivity, or generate whole architectures; patterns do nothing to remove the human from the creative process. The software developer must understand the problem to select and use a right pattern. Vlissides also argues that the benefit from patterns comes mostly from applying them as they are, with no tool support of any kind.

However, as explained in Section 1.3, the author of this dissertation is convinced that patterns provide a good basis for a practical and easy-to-use tool support in software engineering. Nevertheless, to enable a tool support it is important to admit that one must make some compromises over the universal applicability of the original pattern documentation; as such, they are excellent for human but too abstract and ambiguous for a tool. This is discussed more precisely in Section 2.3.

2.1.2 Variety of Patterns

The idea of patterns is practical and its implications in software engineering are many. A number of authors have used the concept and developed more sophisticated pattern types. For example, *design patterns* [Gamma et al. 1995], *architectural patterns* [Buschmann et al. 1996], *analysis patterns* [Fowler 1997], *performance patterns* [Smith and Williams 2002], *domain-oriented patterns* [Harrison N. et al. 1999], *meta-patterns* [Pree 1994, 1995], *anti-patterns* [Akroyd 1996; Brown W. et al. 1998], *idioms* [Coplien 1992; Vlissides et al. 1996], *framework-specific patterns* [Johnson 1992; Hakala et al. 2001b, 2001c, 2001d], and *process patterns* [Coplien 1995; Ambler 1998] are used to describe the design and implementation problems of the object-oriented software in different circumstances and scale. As an example of other possible application areas, *organizational patterns* [Coplien 1995; Harrison N. and Coplien 2004] describe the best practices and general observations of human organizations.

Here the list of different pattern types gives the reader some background and various perspectives on the pattern concept, but this is not a comprehensive presentation of everything discussed in the pattern literature. To get more information about the history and variety of patterns, the reader is referred, for instance, to the article “Patterns and Software: Essential Concepts and Terminology” [Appleton 1997] and to visit the pattern WWW sites [Hillside 2004; Portland 2004].

Design patterns [Gamma et al. 1995] are descriptions of communicating objects and classes that are customized to solve a general object-oriented design problem within a particular context. A design pattern has four essential elements: the pattern name that unambiguously identifies the pattern, the problem describing when to apply the pattern, the solution explaining the elements and relationships that make up the design, and the consequences describing the results and trade-offs of applying the pattern. This form suits best for communicating generic design alternatives. As an example of a design pattern, see the Abstract Factory pattern discussed in Subsection 2.1.4.

Architectural patterns [Buschmann et al. 1996] are very high-level structural patterns. They are used to describe the general structure of software architectures, like how to organize components, handle distributed computation, keep the functional core independent of the user interface, and design adaptable systems. An architectural pattern may utilize several lower-level design patterns. A well-known example of architectural patterns is the Model-View-Controller pattern [Krasner and Pope 1988], which separates application logic (model) from its manipulation (controller) and user interface (view) issues. Architectural patterns are similar to *architectural styles* [Perry and Wolf 1992; Shaw and Garlan 1996], which describe composition and design rules to construct applications. Buschmann et al. have noticed that architectural styles can be expressed with architectural patterns, too.

Analysis patterns [Fowler 1997] are purposed for domain analysis, like modeling different business domains (trading, measurement, accounting, and so on). Thus, analysis patterns are focused on the requirements analysis and the acceptance and usability of the final system. They provide reusable analysis models with examples and they facilitate the transformation of the analysis model into a design model by suggesting design patterns and other reliable solutions.

Performance patterns [Smith and Williams 2002] address performance problems. They describe best practices for producing responsive and scalable software, in which the responsiveness is the ability of a system to meet its objectives for response time or throughput. Scalability, in turn, is the ability of a system to continue to meet its response time or throughput objectives as the demand for the software functions increases. Performance patterns are at a higher level of abstraction than design patterns; a design pattern may provide an implementation of a performance pattern.

Domain-oriented patterns [Harrison N. et al. 1999] cover the patterns that are used to address the problems of a specific domain, like patterns for creating database-reporting applications. They may use design patterns to guide the implementation and they may refine architectural patterns with domain-specific details and requirements. Thus, domain-oriented patterns are higher-level than design patterns, but they are not as domain independent as the architectural patterns.

Meta-patterns [Pree 1994, 1995] are patterns describing other patterns. Pree suggests that there is a limited set of meta-patterns that describe the basic ways to compose object structures. These meta-patterns can then be used to analyse design patterns; in fact, they can be seen as general building blocks of the design patterns. Each meta-pattern reflects a typical and essential combination of inheritance, aggregation and abstract operations.

Anti-patterns [Akroyd 1996; Brown W. et al. 1998] are used to describe recurring bad design and mistakes to common design problems. For example, Brown W. et al. present the Spaghetti Code anti-pattern using a design pattern like representation. Thus, the use of anti-patterns produces negative consequences, but they can be used to avoid and identify mistakes. They also support refactoring to fix the recognized problems.

Idioms [Coplien 1992; Vlissides et al. 1996] are low-level programming language dependent patterns that describe coding styles and implementation practices. For example, an idiom can be used to describe how to remove duplicates from a collection or how to iterate a list structure.

Framework-specific patterns [Johnson 1992; Hakala et al. 2001b, 2001c, 2001d] describe recurring actions and program structures in framework specialization or middleware solutions. For example, when specializing a particular framework, the process is repeated and can be documented with a set of patterns. Often the range of such a framework-specific pattern is very narrow; it describes a solution in the context of the used framework. Thus, a framework-specific pattern is more implementation-oriented and less general than a design pattern; it stores practical instructions and describes how to specialize a specific framework.

Process patterns [Coplien 1995; Ambler 1998] describe a collection of general techniques and actions for developing object-oriented software. They describe approved techniques for managing the complexities of large-scale, object-oriented software development projects. For example, a process pattern can describe how to review and validate the deliverables to ensure that they meet the needs of the users and the quality standards of the organization. Process patterns are closely related to organizational patterns that describe common management techniques and organizational structures.

Organizational patterns [Coplien 1995; Harrison N. and Coplien 2004] describe the structure and practices of human organizations. They can be used as inspiration to improve the organization and it's functioning. Because people are less predictable than code, the results of applying organizational patterns are less predictable than applying patterns in object-oriented software. Organizational patterns are used by groups of people.

The pattern types discussed in this subsection are normative but not absolute. In the pattern community the meaning and usefulness of a pattern may be argued, as well as if the pattern has or has not the characteristics of a particular pattern type or types. Table 1 illustrates the use of different pattern types in different phases of the software development process. Because of their special nature, anti-patterns and meta-patterns are not shown in the table. Also, process patterns and organizational patterns can be seen as patterns that are used by the organization throughout the software development process.

Since the release of the design pattern book [Gamma et al. 1995], numbers of pattern catalogs have become available (e.g., [Rising 2000]). Patterns have also been discussed in a number of conferences, especially in an annual conference Pattern Languages of Program Design (PLoP) [Coplien and Schmidt 1995; Vlissides et al. 1996; Martin et al. 1998; Harrison N. et al. 1999]. The latest PLoP proceedings and information about other pattern conferences can be found from the Hillside pattern WWW site [Hillside 2004]. All these patterns are based on informal documentation and examples; they are written for humans, not for machines.

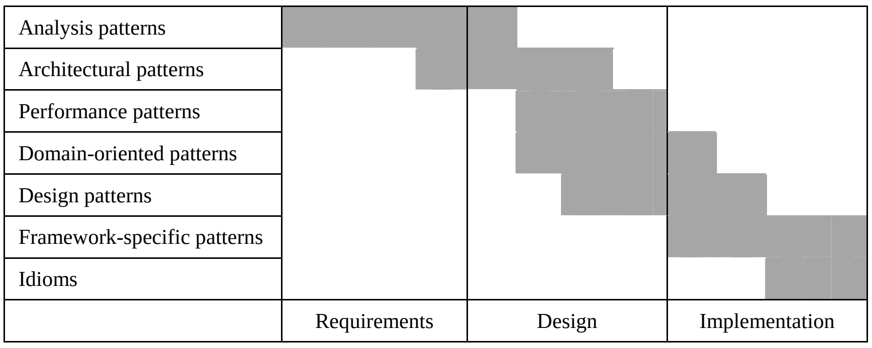


Table 1. A rough division of the pattern types.

At some point the number of patterns becomes a problem; finding the most suitable pattern may be difficult. The rough division of pattern types is not enough. To avoid the problem, various categorizations have been discussed. For example, Gamma et al. [1995] sort design patterns according to their purpose (creational, behavioral, and structural patterns) and scope (patterns based on static inheritance, and dynamic patterns based on collaborating objects). However, this categorization is not very intuitive for novice users. Other possibilities could be problem-based classification or grouping typical pattern combinations together. Perhaps the most ideal answer is still the one presented by Alexander et al. [1977]; to structure related patterns as a pattern language for a particular domain.

2.1.3 Elements of a Pattern

Though pattern documentation is informal, a pattern description is not a chunk of unstructured text. On the contrary, it often has a well-known form that promotes the different aspects of the pattern, like the problem, the context, and the solution. There are a number of formats with slight differences. Coplien [1996] summarizes some of the documentation styles that have been used for describing patterns. The format used in Alexander's work is called the *Alexandrian form*. The format used by Gamma et al. [1995] for design patterns is called the *GoF form* (see the example in the next subsection). Coplien also mentions *Portland form* and *Coplien form*. Appleton [1997] discusses the *canonical form*, which means the simplest, most basic or primordial form to document patterns. There are no strict rules for the form used; one rarely describes every possible detail for every pattern.

Common to all of the pattern forms is that they are informal text that is structured in some specific manner. Despite the differing pattern forms, both Coplien [1996] and Appleton [1997] discuss the essential elements that should be clearly recognizable upon reading a pattern. Appleton enumerates the following elements (the list of Coplien is almost identical, with some minor differences):

- *Name*. A pattern must have a meaningful name.

- *Problem*. The problem statement describes the goals and objectives for the pattern within the given context and forces; this can be seen as the intent of the pattern.

- *Context*. The context statement describes the applicability of the pattern; it describes preconditions under which the problem and its solution recur.

- *Forces*. The forces statement describes the forces and constraints that conflicts or interacts with the goals of the pattern. Forces reveal the complexity of a problem and define the trade-offs.

- *Solution*. The solution statement describes how to realize the desired outcome. The description may contain pictures, diagrams, and documentation that identify the structure, participants, and collaborations of the pattern. Both the static structure (the form and organization) and the dynamic behaviour of the pattern should be described.

- *Examples*. Examples can be used to illustrate the use and applicability of the pattern.

- *Resulting context*. The resulting context statement describes the state or configuration of the system after the pattern has been instantiated. This includes the consequences and side-effects of the pattern.

- *Rationale*. The rationale statement describes why the pattern resolves its forces in a particular way.

- *Related patterns*. This statement describes the relationships between this pattern and other patterns.

- *Known uses*. The known uses statement describes known occurrences of the pattern. These can be also used as examples of the use of the pattern.

2.1.4 Example: Abstract Factory Pattern

As an example, the well-known Abstract Factory pattern presented by Gamma et al. [1995, p. 87-95] is shown in Table 2. The example is given in the GoF form used in the book. The example is used later in Chapter 4, as a small case study to demonstrate the idea of a general pattern-based tool support.

Abstract Factory

Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

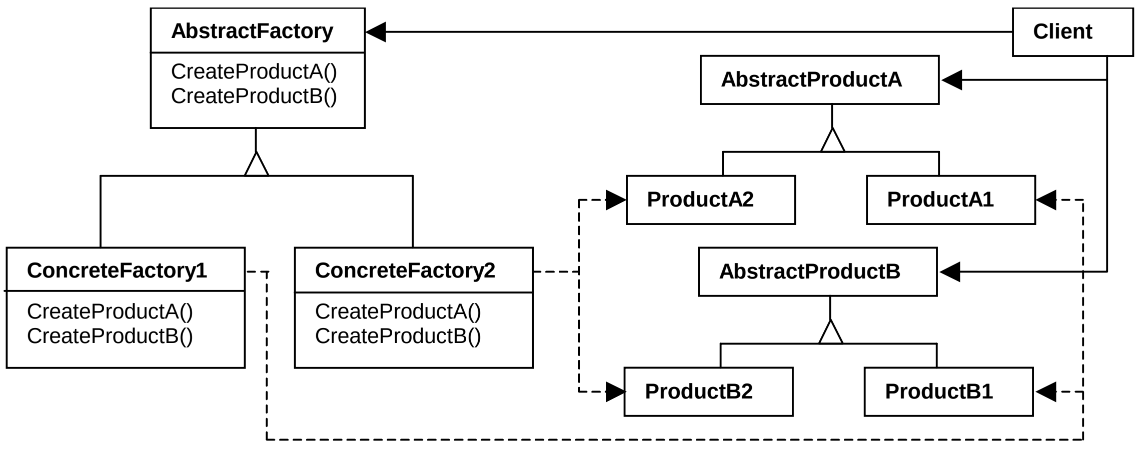
Also Known As: Kit

Motivation: Motivation to use the Abstact Factory pattern. See Gamma et al. [1995, p. 87-95]

Applicability:

- A system should be independent of how its products are created, composed, and represented.
- A system should be configured with one of multiple families of products.
- A family of related product objects is designed to be used together, and you need to enforce this constraint.
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Structure:



Participants:

- AbstractFactory. Declares an interface for operations that create abstract product objects.
- ConcreteFactory. Implements the operations to create concrete product objects.
- AbstractProduct. Declares an interface for a type of product object.
- ConcreteProduct. Defines a product object to be created by the corresponding concrete factory. Implements the AbstractProduct interface.
- Client. Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Collaborations:

- Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Consequences: The benefits and liabilities of the Abstract Factory pattern. See Gamma et al. [1995, p. 87-95]

Implementation: Useful techniques for implementing the Abstract Factory pattern. See Gamma et al. [1995, p. 87-95]

Sample Code: Code examples. See Gamma et al. [1995, p. 87-95]

Known Uses: The known uses of the Abstract Factory pattern. See Gamma et al. [1995, p. 87-95]

Related Patterns: AbstractFactory classes are often implemented with factory methods (Factory Method), but they can also be implemented using Prototype. A concrete factory is often a singleton (Singleton).

Table 2. The Abstract Factory pattern [Gamma et al. 1995, p. 87-95].

2.1.5 About Writing Patterns

Patterns are found rather than invented. A pattern documents an approved way to solve a problem, but it is more extensive than a simple recipe-like instruction. A pattern abstracts the problem and its solution, it makes explicit the trade-offs of the required steps to reach the solution, and describes the context in which the problem usually occurs. This makes the use of a pattern flexible; instead of exactly one fixed solution, a pattern can be applied to construct variety of solutions and to explain why they are constructed as they are.

On the other hand, flexibility is often achieved with ambiguous and indefinite instructions. The person who writes the pattern cannot write down its every possible occurrence, as the number of such occurrences can be infinite. As a consequence, a typical pattern does not precisely describe every detail of the application-specific solution. Instead, the usefulness of the pattern depends on how well the reader understands the pattern documentation and applies it in her application. The validity of patterns and pattern languages is testified by their use; the pattern must be revisited or rejected if it fails to explain the intended solution and how this solution should be achieved.

Thus, writing a good pattern is difficult and demands substantial amount of skills and experience. Patterns should not only capture the experience they are trying to convey but also explain how the design could be reused in different circumstances [Appleton 1997]. A pattern is never accurate enough unless the design has been carefully examined. Iteration is often necessary as trying to instantiate a pattern in practice motivates changes to that pattern; the ultimate test for a pattern is to carry out the design with it. Also, there is no single right way to write a pattern; like writing a novel or a poem it is a creative human activity for human audience.

Despite of the creative nature of the pattern writing process, some guidelines and criteria can be given. Meszaros and Doble [1998] present a pattern language to capture some of the best practices of pattern writing. In their meta-level pattern language they introduce patterns to define the concept of a pattern and a pattern language, patterns to set the desired content and structure of individual patterns, patterns to describe techniques for naming the patterns and for including references to other patterns, patterns to make the patterns easier to read, and patterns to define the desired content and structure of the pattern languages. They also enumerate the following forces of the pattern writing process:

- Keeping the solution hidden does not require any effort.
- Sharing the solution verbally helps only a few others.

- Writing down the understanding of the solution is difficult and requires much reflection on how to solve the problem.
- Transforming the solution into a more widely applicable form (pattern) is difficult.
- People are unlikely to use the solution if they do not know why it should be used.
- Writing down the solution may reduce the competitive advantage.

Buschmann et al. [1996], in turn, summarizes the following criteria that the pattern writing process should meet:

- *Focus on practicability*. Patterns should describe proven solutions to recurring problems.
- *Aggressive disregard of originality*. Pattern writers do not need to be the original inventor or discoverer of the solutions that they document.
- *Non-anonymous review*. Persons trying to use the patterns should contact the pattern authors and discuss with them how the patterns might be clarified or improved upon.
- *Writer's workshops instead of presentation*. Patterns should be discussed inside the development group and attending peoples to seek what is good about the patterns as well as the areas which they are lacking.
- *Careful editing*. The pattern authors have the opportunity to incorporate all the comments and insights during the user feedback and writer's workshop before presenting the patterns in their finished form.

Sometimes patterns – particularly the framework-specific ones (Subsection 2.1.2) – are not very general but try to describe a practical solution for a specific programming problem. Such patterns constitute a pattern language that documents the typical use of the framework. In that case, the participants of the writer's workshop would be framework experts and application developers who are using the framework. The discussion about the framework-specific patterns and the feedback would be very practical and implementation oriented. In addition, if a set of patterns is going to represent the specialization interface of a framework, they should be thoroughly tested and evaluated by specializing the framework with them.

2.2 From Patterns to Implementation and Back

A pattern is generative in the sense that its use creates solutions. Together the participating elements of the solution and their relationships constitute a concrete manifestation of the pattern. This pattern instance is organized as described in the pattern documentation. For legacy software systems, it may be beneficial to analyze if the system contains such pattern instances. Using patterns to create new pattern instances and the process to find the applied patterns from existing software systems are discussed in the following subsections:

- *Pattern instantiation*. A concrete manifestation of a pattern is called a pattern instance. A pattern is instantiated by creating and modifying the participating elements of the solution as described in the pattern documentation. Subsection 2.2.1.
- *Patterns after instantiation*. Connections between the patterns and the pattern instances are often lost. However, this is valuable information that should be documented. Subsection 2.2.2.
- *Pattern mining*. It may be useful to analyze a software system to find out if it applies well-known patterns. Subsection 2.2.3.

2.2.1 Pattern Instantiation

Alexander et al. [1975] say that the process to instantiate patterns is based on piecemeal growth. This piecemeal growth is evolutionary, dynamic, and continuous; it guides planning and construction and allows the whole to emerge gradually from local acts. Alexander [2004] mentions also the principle of organic order in which the order is achieved when there is a perfect balance between the needs of the parts (classes, operations, etc.) and the needs of the whole (a software system). Thus, instead of each act of design or construction being an isolated event that creates a perfect element, like a complete class or operation, every system is changing and growing all the time, in order to keep its use in balance. As said by Alexander et al. [1987, p. 32]:

> "Wholeness is too complicated to be built up in large lumps. The grain of development must be small enough, so that there is room, and time, for wholeness to develop."

Harrison N. and Coplien [2004] explain why the piecemeal-growth works. Firstly, the user does not do random things at random times. Instead, patterns encode experience and the instantiation follows steps that have repeatedly worked in the past. Secondly, patterns build structures which themselves offer a degree of resiliency under change. The structure is not rebuilt or reorganized after every change or increment. According to Harrison and Coplien, a pattern language is a graph, and there are many useful paths through it. The individual patterns tell about what patterns should come next in the path. Or there may be documented paths that have been successfully used in different circumstances.

Hence, from the software developer's standpoint, a pattern language with a meaningful set of patterns helps to implement complex design piece by piece. When a problem occurs, the software developer selects a suitable pattern and instantiates it, either by creating new elements or by modifying existing ones, as described in the solution statement of the pattern documentation. The path used through the pattern language - the sequence of applied patterns - results in a software system incrementally and gradually.

Florijn et al. [1997] discuss the pattern instantiation process. Firstly, a problem must be recognized to identify which pattern to use. Secondly, the selected pattern must be instantiated by mapping and integrating the design elements from the pattern description to the design elements in the software system; the software developer must decide which classes, operations, etc., in a program will play the roles defined in the pattern description. This can lead to situations where the elements in a program play multiple roles in different pattern instances. Florijn et al. enumerates three different ways of instantiating patterns:

- *Top-down*. The software developer creates an initial set of classes and their internals that follow the pattern.
- *Bottom-up*. The software developer selects a suitable set of existing code elements that follow the pattern. No new elements are created; instead, the bottom-up approach uses existing elements.
- *Mixed*. The software developer selects a suitable set of existing code elements that follow the pattern. However, the pattern instance is not completed and some new elements must be created to fulfill the pattern.

To summarize, the software developer must provide the individual building blocks that bring the pattern instance into existence. Rather than implementing the plan at once as an isolated action, this can be seen as a gradually proceeding work, where the software developer creates and modifies code elements. The acts to instantiate a pattern can be seen as a sequence of tasks or steps to select, create, and modify the required elements. The support of the piecemeal growth and the different ways to create pattern instances (top-down, bottom-up, and mixed) must be considered as important features of the general pattern-based tool support discussed in Chapter 4.

2.2.2 Patterns after Instantiation

Patterns act as a bridge between the design and its implementation. Many of the authors who work to promote the pattern concept stress that if the pattern instances cannot be traced back to the original patterns they become invisible in the final product. For example, Soukup [1995] has noticed that programmers tend to lose sight of the original patterns, as the pattern instances are not visible in the final source code. There may be some documentation and comments scattered throughout the code, but this is not enough as it is hard to see the overall design when working on the code-level. Soukup claims that this is a major source of maintenance problems; the documentation describing the patterns will eventually get lost or become obsolete. Also, documenting the applied patterns only at the code-level becomes difficult when the number of pattern instances increases. Due to the scattering and tangling problems discussed in Subsection 1.2.1, pattern instances may be partially overlapping and a single code element may be playing different roles in more than one pattern instance.

Soukup [1995] continues that when applying several patterns to the same set of classes, pattern behaviour is embedded in operations associated with those classes. Each pattern may require that such an operation calls the operations of several other classes making the entire design a big knot of interdependent classes and operations. For the software developer, code with these kinds of complex and embedded relations is difficult to understand, debug and test. Clearly, knowing the underlying patterns would help to understand the fragmented design solutions and the crosscuttings of the software architecture and framework specialization.

Florijn et al. [1997] discuss the difficulty to maintain pattern instances when code is changing. Modifying source code, like changing the interface of a class that is involved in some pattern instance, can cause situations where the pattern instance violates the structural or semantic constraints of the pattern. Such violations may be hard to find and they are easy to forget, as the code itself remains syntactically correct. To deal with such situations, they suggest that the software development environment should be able to check whether the pattern instance meets the requirements of the pattern. Obviously, the ability to check and locate pattern instances is crucial if patterns are going to be used as a development resource during the software development process.

As pointed out by Mikkonen and Pruuden [2001], evolving software gets more complex in each increment. Here the pattern instances can be seen as a valuable resource that helps the software developer to understand the required changes, their consequences, and how these changes could be implemented. However, without any tool support, this resource is hard to maintain and it is easy to forget and waste. Discovering the undocumented pattern instances manually from the source code is expensive and time-consuming. Even if the original state of the pattern instances was documented, future changes in the implementation level will make this documentation outdated.

2.2.3 Pattern Mining

It can be beneficial to analyze an existing software system to find out if it applies any well-known patterns. This is called *pattern mining* [Martin 1995]. Particularly, in the case of legacy software systems, finding the used patterns may help to understand, document, and reorganize the system. In addition, as mentioned by Lange and Nakamura [1995], a running object-oriented system produces huge amount of static and dynamic information. Patterns can be used to analyze and visualize this information. Hence, pattern mining supports reverse engineering. Chikofsky and Cross [1990] define reverse engineering as follows:

> ″Reverse engineering is the process of analyzing a subject system to (a) identify the system's components and their interrelationships and (b) create representations of a system in another form at a higher level of abstraction.″

There are two reasons why unrecognised pattern instances may exist in code. Firstly, as discussed in the previous subsection, if the use of patterns is not continuously and carefully documented, pattern instances will not stand out from the source code and they will be forgotten. Secondly, a software system may have structures and behavior that resembles some well-known patterns, even if the original software developers were not conscious of the patterns. A typical pattern represents existing, approved experience; the software developer may have this knowledge without being aware of the patterns.

Keller et al. [1999] discuss more about pattern-based reverse engineering. They argue that the patterns used are at the root of the main elements of software systems. As these patterns capture the rationale and trade-offs of design solutions, one can comprehend a software system by recovering the patterns used. An important observation is that effective pattern-based reverse engineering cannot be fully automatized; the human analyzer must direct the pattern analysis tool to separate the essential information from inessential. Same applies to reverse engineering in general.

As mentioned by Niere et al. [2002], one of the problems in pattern mining is that patterns and their instances typically have a number of variants. This is the main reason why pattern-based design recovery techniques have had only limited success. Another reason is that patterns may be structurally identical, though they have different behaviour or purpose. This makes the pattern mining even more complicated, as it is not enough to analyze only the structure of a software system, but also its behaviour.

Analysis and pattern mining tools are discussed in Subsection 4.5.3, Subsection 5.2.5, and Section 9.1. In any case, using a tool for pattern mining requires that informal patterns can be expressed in a form that is suitable for the tool.

2.3 Formal vs. Informal Pattern Descriptions

The evident problem of any tool support is that informal pattern documentation is too ambiguous for a tool. Patterns are written for humans and the documentation relies on the reader's ability to apply a pattern in the current situation. At the moment computers do not have this kind of intelligence or intuition. Instead, efficient tool support requires that patterns take a coherent and precise form. In this dissertation, as discussed in Chapter 1, such a tool-supported form of a pattern is called a pattern specification. It captures the core of the pattern documentation with some formalism or specification language so that a tool can be used to help the application developer to apply the pattern.

However, for humans the well-structured informal documentation with concrete examples is easier to read and comprehend. Thus, both informal and formal pattern documentation has their usages and advantages. The purpose of the precise pattern specifications should be to help the software developer by enabling tool support, not to displace or underestimate the more informal and readable documentation. In literature, pattern documentation has many slightly different forms. For example, the Alexandrian form, the GoF form, the Coplien form, and the Portland form that were discussed in Subsection 2.1.3. All these forms are informal descriptions about the pattern and its use. As said by Coplien [1996]:

> "Patterns guide humans, not machines. They will not generate code; they do not live inside CASE tools. They are literature that aids human decision-making processes. Patterns should not, cannot, and will not replace the human programmer."

On the other hand, also informal pattern documentation has its limitations. To use patterns the software developer must recognize the problem and select a suitable pattern that solves the problem. Then, to implement the solution, she must read the selected pattern documentation and apply the pattern to the context of her application. This can be problematic, as the pattern documentation cannot describe the desired application-specific solution precisely. Instead, the documentation uses abstract terms and fixed set of examples. As each software developer and programmer may use and understand this documentation differently, it can make the use of patterns slow, errorprone, and unsystematic. In addition, as discussed in Subsection 2.2.2, the connections between the pattern and its instances are difficult to maintain. Unlike sophisticated pattern tools using some pattern specifications, the informal and static pattern documentation does not provide any direct support to trace the existing pattern instances, to check their validity, or to document the participating code elements. In the source code, patterns tend to be used as a disposable resource without continuous maintenance.

Also Eden [2002b] criticizes informal pattern documentation, like verbal descriptions, class diagrams, and concrete examples. According to Eden, the informal and ultimately fuzzy descriptions puzzle the pattern users and cause confusion. Typically, it is debated what is the true meaning of a pattern, or if a particular implementation conforms to a certain pattern, or if one pattern is a special case of another. Moreover, as the number of patterns increases, they become an unstructured mass that lacks effective means of indexing.

Still, it must be admitted that there are some problems with more precise pattern specifications. Firstly, formalisms are often too heavy to be used in ordinary software projects [Lamsweerde 2000]. Secondly, typical application developers are not familiar with formalisms and exotic pattern specification languages; they are interested in applying a particular pattern to solve a specific and practical problem. With a formalism, the essential aspects of the original pattern documentation may become cluttered with many details making the pattern hard to understand. Thirdly, one can suppose that the informal pattern documentation uses richer and more ambiguous design vocabulary than the one offered by any formalism. A formal notation is rarely expressive enough to specify every not so clear detail of the original pattern documentation. Also, it seems that formalisms mainly concentrate on the solution part of the pattern documentation. However, as discussed in Subsection 2.1.3, pattern documentation describes also other statements, like context and consequences of the pattern. These are often difficult to formalize, though some approaches have been presented, for example, to specify the context of the pattern (see, e.g., [Mikkonen 1998; Kellomäki and Mikkonen 2000]).

A tool support itself has some trade-offs, too. Particularly, a complete and very fine-grained tool support may be computationally too expensive to be used in practice. On the other hand, if the tool support is too limited, it entrusts the user to look after laborious details, like tracing the pattern instances, which could be managed automatically. After all, the tool support is to be used by people; ordinary software developer or programmer must experience that the tool is easy to learn, easy to use, and it proves to be advantageous. To be successful, the adoption of a new technology should not hinder approved development methodologies, techniques, and project management.

To solve these problems, we suggest that the complexity of the used formalism is hidden from ordinary application developers. For instance, an experienced software architect could use some kind of pattern editor to create pattern specifications, while an application developer could utilize an easy-to-use deployment tool to instantiate them. In addition, we suggest a light-weight approach, in which the purpose of pattern specifications is to provide practical support for a portion of application development, not to express or proof the whole application and its requirements with patterns. The formalism could concentrate on describing only the essential participants and interactions needed in the solution. Altogether, this means that such a pattern specification may represent only a refined subset of the solution space described by the original pattern. These principles are applied in Chapter 4, when outlinig the general tool platform for patterns.

CHAPTER 3

FRAMEWORKS IN SOFTWARE ENGINEERING

Like any industry, also software industry has been motivated by the quest to improve its processes and reduce production costs. Time scale, quality, and the overall cost of the software development process should be optimized and the whole process should be more predictable. This goal has moved the software industry to embrace object-oriented programming because of its potential to significantly increase quality and productivity. Concepts like *abstract data types*, *encapsulation*, *inheritance*, *polymorphism* and *dynamic binding* [Sebesta 1999] make the object-oriented paradigm suitable to implement large software systems by reusing existing design and code. Object-oriented frameworks, introduced in Subsection 1.1.4, are one approach to use this technology in a most efficient way. Patterns, in turn, are a way to document the design and the use of a framework.

A framework is used by extending and refining its classes and operations and by using its components. In that way, the code and design of the framework becomes reused to create a new application or a significant portion of it. The programming is done faster and the code is more reliable. Altogether, this improves the quality of the created software product and reduces production costs. However, though frameworks have many advantages, their use is often problematic (Section 1.2). Using a framework to create an application requires expertise and knowledge that cannot be obtained directly from the framework's interfaces or class hierarchy. Instead, a framework requires good documentation and tool support to be used and understood. In Chapter 4 it will be shown that a pattern-based tool support can help and systematize the use of frameworks. This is also demonstrated with case studies in Chapter 7 and in Chapter 8.

As a background, a more detailed introduction to frameworks is given in Section 3.1. Designing and maintaining frameworks are discussed in Section 3.2. Framework specialization and how a pattern-based tool support could help are discussed in Section 3.3.

3.1 Introduction to Frameworks

A brief introduction to object-oriented frameworks is given in the following subsections:

- *Characteristics of a framework*. An object-oriented framework is a reusable software system that provides a skeleton to implement solutions for a particular application domain. Subsection 3.1.1.

- *Advantages of frameworks*. Frameworks have many benefits that make them attractive when creating applications. Subsection 3.1.2.

- *Problems with frameworks*. Frameworks have some drawbacks and trade-offs. Subsection 3.1.3.

- *Framework categories*. Frameworks can be classified by their specialization technique and scope. Subsection 3.1.4.

3.1.1 Characteristics of a Framework

There are a number of definitions for a framework. For example, Roberts and Johnson [1996] define frameworks as reusable designs of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate (Subsection 1.1.4). From the user's standpoint, Taligent [1994] defines a framework as a set of prefabricated software building blocks that programmers can use, extend, or customize for specific computing solutions. In any case, a framework captures the programming expertise necessary to solve problems in a particular problem domain; it hides the parts of the design that are common to all applications in that domain, and makes explicit the pieces that need to be customized. A typical framework usually provides the design for only a part of a system, such as its user interface, though application specific frameworks may describe an entire application.

An important characteristic of an object-oriented framework is that the operations defined by the application developer can be called from within the framework itself, rather than from the user's application code [Johnson and

Foote 1988]. This mechanism is often called ″the Hollywood principle″ or ″Don't call us, we'll call you″. Places where the framework should be extended by implementing these operations are often called *hot spots* [Pree 1995]. Hot spots are kinds of slots or extension points in the framework; it is the user's task to fill these slots with application-specific code. In practice, hot spots are typically indicated with abstract classes and unimplemented operations of the framework.

Hot spots and the Hollywood principle make frameworks flexible. A framework calls a code provided by an application-specific subclass, customizing the behavior of the framework. For instance, a GUI framework may have abstract classes and interfaces used by the GUI components. By implementing the required subclasses and operations, the application developer can create application-specific models to be viewed and managed by these GUI components.

Fayad et al. [1999] have summarized the four central properties of object-oriented frameworks:

- *Modularity.* Frameworks enhance modularity by encapsulating implementation details behind stable interfaces. This helps to improve software quality by localizing the impact of design and implementation changes.
- *Reusability.* With stable interfaces, frameworks provide generic building blocks to create new applications. Reuse of these framework components can enhance the quality, performance, reliability, and interoperability of software. In addition, using existing code and design solutions improves programmer's productivity.
- *Extensibility.* A framework provides explicit hook methods (hot spots [Pree 1995]) that allow applications to extend its stable interfaces.
- *Inversion of control.* Unlike normal class libraries, frameworks may call the operations of the derived application (the Hollywood principle). The role of a framework is to provide the general flow of control, while the application's code waits for a call from the framework

Taligent [1994] has noticed that to be successful, a framework should be:

- *Complete.* Frameworks must support features needed by users and they must provide default implementations and built-in functionality when possible. This makes it easier for the users to understand a framework and to allow them to focus on the areas that they need to customize.

- *Flexible*. Abstractions of a framework can be used in different contexts.

- *Extensible*. Users can easily add and modify functionality. The framework developers must provide hook methods so that clients can customize the behaviour of the framework by deriving new classes.

- *Understandable*. The framework should be well-documented. The framework developers should follow standard design and coding guidelines and provide sample applications that demonstrate the use the framework.

3.1.2 Advantages of Frameworks

Frameworks have significant benefits:

- *Reuse of code and design*. With frameworks, applications are built by reusing approved design and code. A framework is not just a collection of classes but it also defines a generic design and helps the user to apply the underlying product line architecture. Because of the bidirectional flow of control a framework can contain much more functionality than a traditional class library.

- *Stored experience*. Frameworks store experience; problems are solved once and the business rules and design are used consistently. By providing reusable design and code, the framework decreases the amount of architectural decisions and implementation efforts that the software developer has to make.

- *Coordination*. Frameworks can be used to coordinate distribution of work. For instance, a project can be divided into those improving, extending, or developing frameworks and those using them for a particular application [Johnson and Russo 1991].

- *Improved software development cycle*. Frameworks decrease the amount of code that the application developers have to program, test, and debug. By using frameworks the developers can focus their attention on more advanced design and implementation problems. This enables also rapid prototyping [Campbell et al. 1992].

- *Maintainability*. When implemented with a framework, the created applications share the same product line architecture with common structures and features. In the case of mature frameworks this makes the applications easier to maintain and understand. However, as discussed in the next subsection, the maintainability can also be a problem if the framework is not stabile enough.

3.1.3 Problems with Frameworks

Frameworks have many advantages making them attractive. However, they have some significant drawbacks and trade-offs, too. Fayad et al. [1999] have summarized the following often mentioned challenges dealing with frameworks:

- *Development effort*. It is hard to create a reusable framework for complex application domains. Unless the framework can be used in many projects, this investment may not be cost-effective.
- *Learning curve*. Frameworks are often complex and difficult to use.
- *Integratability*. Using multiple frameworks simultaneously can be difficult if the frameworks cannot cooperate.
- *Maintainability*. The requirements of applications and frameworks change frequently. As frameworks evolve, the applications that use them must evolve with them.
- *Validation and defect removal*. Validating and debugging applications built using frameworks can be difficult.
- *Efficiency*. Generality and flexibility of frameworks requires more processing time and memory resources.
- *Lack of standards*. There are no standards for designing, implementing, documenting, and using frameworks.

We argue that the framework-specific patterns (Subsection 2.1.2) and the pattern-based tool support proposed in this thesis could be used to reduce some of the problems. Particularly, learning and validating the use of frameworks could be supported by this kind of tool.

3.1.4 Framework Categories

Frameworks can be classified by their usage technique and scope. For example, depending on how a framework is specialized, it is said that it is a *white-box framework*, a *gray-box framework*, or a *black-box framework* [Fayad et al. 1999]. The use of a white-box framework is based on inheritance and dynamic binding. The user customizes a white-box framework by deriving new subclasses from the abstract base classes of the framework and by overriding and implementing required operations. The use of a black-box framework, in turn, is based on object composition; the user configures and composes objects (components) that implement the application or a part of it. Usually a framework is not purely white-box or black-box framework. Instead, the framework specialization may utilize both white-box and black-box techniques, where a part of the specialization is done by composing components and the missing functionality is implemented by deriving new subclasses. The use of a gray-box framework is a mix of white-box and black-box usage.

Johnson and Foote [1988] have noticed that there is no strict line between a white-box framework and a simple class hierarchy; every class hierarchy offers the possibility of becoming a white-box framework. In its simplest form, a white-box framework is a program skeleton, and the user-derived subclasses are additions to that skeleton. White-box frameworks may evolve into black-box ones. However, this process is not obvious; many frameworks will not complete the journey from white-box skeleton to black-box framework during their lifetime.

As an example of scope-based categorization, Fayad et al. [1999] suggest that the scope of frameworks may range from *system infrastructure frameworks* and *middleware integration frameworks* to *enterprise application frameworks*. System infrastructure frameworks simplify the development of portable and efficient system infrastructure. They are primarily used internally within a software organization and are not sold to customers directly. Middleware integration frameworks are used to integrate distributed applications and components. Enterprise application frameworks address broad application domains. They support the development of end-user applications and products directly, while system infrastructure and middleware integration frameworks focus on internal software development.

Another scope-based categorization is discussed by Taligent [1994], where the scope ranges from *support frameworks* to *domain frameworks* and further to *application frameworks*. Support frameworks provide system-level services, such as file access or device drivers. Domain frameworks encapsulate expertise in a particular problem domain, like multimedia or data access. Application frameworks, in turn, encapsulate expertise applicable to a wide variety of programs; for example, application frameworks to create graphical user interfaces.

The application domain can also be divided into different layers. Frameworks with different scope can then be used to implement these layers. Such a system consists of layered frameworks, where more specific frameworks are built on the more abstract ones [Fayad et al. 1999].

3.2 Designing and Maintaining Frameworks

Fayad et al. [1999] divides the framework-centered software development into the following phases: *framework development phase* (to create the framework), *framework usage phase* (to derive applications), and *evolution and maintenance phase* (to maintain the framework and the derived applications), where each phase is affected by other ones making the process iterative. People have different viewpoints to the framework depending on the phase they are involved in and if they are actually developing the framework or using it to derive applications. The usage phase is discussed in Section 3.3. Issues to design, maintain, and document frameworks are discussed in the following subsections:

- *Development phase*. Developing a framework is often difficult; it is an iterative process which needs both domain and design experience. Subsection 3.2.1.
- *Evolution with iteration*. Defects and shortcomings are discovered when the framework is used and the understanding of the problem domain increases. Subsection 3.2.2.
- *Documentation with patterns*. Patterns can be used to document the framework. Subsection 3.2.3.

3.2.1 Development Phase

Though there are numerous design techniques for object-oriented programming, developing object-oriented frameworks is significantly more difficult than developing individual applications. Developers have noticed that framework designing is an iterative process which requires both domain and design expertise [Johnson and Foote 1988; Johnson and Russo 1991; Johnson 1997; Taligent 1994; Fayad et al. 1999]. Firstly, a framework must be simple enough to be learned, or at least it must have good documentation and tool support. Booch [1994] says that a framework will never be reused unless the cost of understanding it and then using its abstractions is lower than the programmer's perceived cost of writing the application from scratch. Secondly, a framework must embody a theory of its application domain; it must provide good abstractions and enough functionality to be useful.

The framework development usually starts with domain analysis trying to find the reusable design and the extension points (hot spots), making the framework flexible. Identifying the key abstractions and extension points may be difficult if the framework developers are not familiar with the problem domain. Therefore, the framework developers should examine applications written by others and consider writing application in the domain. These examples are then generalized to find out similarities.

Johnson and Russo [1991] say that the framework's range of applicability depends heavily on the examples on which it is based. Each example that is considered makes the framework or abstract class more general and reusable. Abstract classes are small, so it is easy to generate lots of examples on paper and reduce the chance of iteration. Frameworks are large, so it is too expensive to look at many examples, and paper designs are not sufficiently detailed to evaluate the framework.

Johnson [1993] has found that a typical way to develop a framework is the following. At first, an application in a particular problem domain is developed in an object-oriented language. Then the application is divided into reusable and nonreusable parts. Then a second application is develop reusing as much software of the first application as possible. It will be noticed that the framework obtained from the first application is not very reusable, so it must be fixed. Then a third application is developed reusing as much software as possible. Again it is usually noticed that the framework is still not completely reusable. The framework is improved and the iteration continues.

The first version of the framework is usually a white-box framework meaning that it is mainly specialized by deriving new subclasses and overriding operations. Derived applications point out faults in the framework and experience leads to improvements, making the development process iterative. Usually the improvements make the framework more black-box meaning that it can be customized by using different combinations of classes in the framework's component library [Roberts and Johnson 1996; Aksit et al. 1999]. This is due to the fact that the design of a particular framework system gradually becomes better understood, which leads to components with higher functionality.

Many authors have noticed that there are hardly any formal techniques for the design of frameworks. Many experts even believe that frameworks cannot be the result of systematic design, but that they rather evolve in a bottom-up fashion, where new features are added to the framework as the framework is used and the developer's experience increases. However, some systematic approaches have been proposed to support the framework development process.

For example, Koskimies and Mössenböck [1995] suggest that the design of a framework shall proceed in two phases. The first phase is called problem generalization. It starts from a single example problem, which is then generalized in a sequence of steps into the most general (sensible) form. The second phase is called framework design. During this phase the generalization levels of the original example are considered in reverse order; a framework is created for each generalization level so that the created framework can be used to implement the corresponding generalization. In other words, a framework for the most generalized level of the original example is implemented first. Then the most general framework can be used to derive a framework for the more concrete generalization level and so on. The result is a hierarchy of more and more refined frameworks. The second phase makes use of general design experience and domain knowledge to find the hot spots in each framework.

Roberts and Johnson [1996] suggest a pattern language which can be used during the framework's development process. The language contains patterns related to each other. In their pattern language, they suggest that the development starts by creating three example applications. After that, the shared functionality and common features of the applications are separated as a white-box framework. The development continues towards a black-box framework: new components are added to the framework's component library and the library evolves to provide more and more functionality. Finally, applications can be built entirely by object composition.

3.2.2 Evolution with Iteration

Because of its importance to applications and software projects developing them, a framework requires routine maintenance to fix errors, assist users, and respond to their problems and requests. Typically, defects and shortcomings are discovered when the framework is used and the understanding of the problem domain increases. The framework is then changed to address the identified problems and trying to use this new version points out new problems and improvement requirements. Even if the framework itself is robust and accurate, the requirements of the application domain may change affecting the product line architecture and the rules of the framework.

A major drawback is that these modifications may cause serious problems for the existing applications based on the framework. Thus, after releasing the framework it should be as stable as possible. Taligent [1994] suggests that framework developers should fix simple bugs immediately, add new features occasionally, and change public interfaces as infrequently as possible. On the other hand, it is hard to know if the framework is stable enough without use cases that verifies it.

One reason for iteration is that framework designers are not skilled enough; they don't have enough experience in the problem domain or software development. However, lack of experience is not the only reason for iteration. Johnson and Russo [1991] say that the main reason that framework design iterates is because frameworks are supposed to be reusable; all software requires iteration before it becomes reusable. This follows from the general observation that software never has a desirable property unless it has been carefully examined and tested in terms of the property. The ultimate test for whether a framework is complete is to specialize it. This is also known as a problem of verifying abstract behavior [Fayad et al. 1999], in which the framework cannot be tested before the framework user provides the application-specific implementations. Without application-specific parts a framework is too abstract to be completely tested.

Johnson and Russo [1991] continue that since frameworks require iteration and deep understanding of the application domain, it is hard to create them on schedule. A common mistake is to start using a framework for important projects while its design is still iterating. It is better to first use the framework for some small pilot projects to make sure that it is sufficiently flexible and general. If not, these projects will be good test cases for the framework developers. A framework should not be used widely until it has proven itself; the more widely a framework is used, the more expensive it is to change it later. Thus, framework design should never be on the critical path of an important project. On the other hand, framework design must be closely associated with the application developers; building applications with a framework shows which parts of the framework need to be improved.

3.2.3 Documentation with Patterns

Documenting frameworks is difficult because they are more abstract than most software. Johnson [1992] suggests that patterns could be used to describe frameworks and their usage. According to Johnson, framework documentation must meet several requirements; it must describe the purpose of the framework, how to use the framework (framework-specific patterns, Subsection 2.1.2), and the detailed design of the framework. He continues that these requirements can all be met by structuring the documentation as a pattern language:

- *The purpose of the framework*. The first pattern of the pattern language should describe the framework's application domain. It is usually hard to specify the problem domain precisely, but a pattern with small set of examples can make the purpose clear. These examples are not intended to show how to use the framework to build applications, nor to explain the design of the framework, but rather to show what the framework is good for. In addition, the first pattern introduces the rest of the patterns in the pattern language, and it tells which patterns should be studied next. Thus, it acts both as a catalog entry for the framework and as a road map to the patterns.

- *Using the framework*. The framework documentation should show how the framework is used to build applications. The framework users are not usually interested to know exactly how the framework works, but how it could be used to solve a particular problem. This means that they want a kind of cookbook that gives detailed instructions to specialize the framework. This cookbook can be done by structuring the documentation as a pattern language, in which each pattern describes a specific specialization problem of the framework.

- *The detailed design of the framework*. The framework's technical documentation includes the different classes in the framework and the way that instances of these classes collaborate. The detailed design information should be hidden from ordinary framework users, because they are not interested in seeing it. This information is most important to the framework developers maintaining the framework.

Since Johnson, patterns have been used for describing the rationale behind design decisions for a framework [Beck and Johnson 1994] and to provide higher-level descriptions of frameworks [Hüni et al. 1995; Schmidt 1997]. Many authors (e.g., [Florijn et al. 1997; Riehle 2000]) have also recognized the close relationship between patterns and the framework's hot spots. More about patterns and how they can be used to provide tool support for framework speacialization is discussed in the next section.

3.3 Using Frameworks

The framework specialization and tool support are discussed in the following subsections:

- *Framework specialization*. The framework specialization is the process to implement an application or a part of it with the framework. Subsection 3.3.1.

- *Pattern-based tool support for frameworks*. Framework-specific patterns can be used to support framework specialization. Subsection 3.3.2.

3.3.1 Framework Specialization

Using a framework requires substantial amount of knowledge. The framework user must know which framework to use and how it should be specialized. In addition, application development is seldom based on a single framework. Instead, the use of frameworks is increasingly based on the integration with other frameworks, together with class libraries, legacy systems, and existing components [Fayad et al. 1999]. This integration process is not always straightforward because the architectural styles [Perry and Wolf 1992; Buschmann et al. 1996; Shaw and Garlan 1996] of two or more frameworks can be too different. These integration problems arise at several levels of abstraction, ranging from documentation issues [Hamu and Fayad 1998] to the event dispatching model and other framework-specific decisions. This is sometimes referred to as architectural mismatch [Garlan et al. 1995].

In the case of a single framework, the framework user specializes the framework by extending its functionality (white-box reuse) and by composing its components (black-box reuse). Figure 2 illustrates how the framework is used to derive an application. After selecting a suitable framework, the framework user writes glue code to configure and compose the framework components to form the application. If there are no suitable components available, she derives new subclasses to implement the desired functionality.

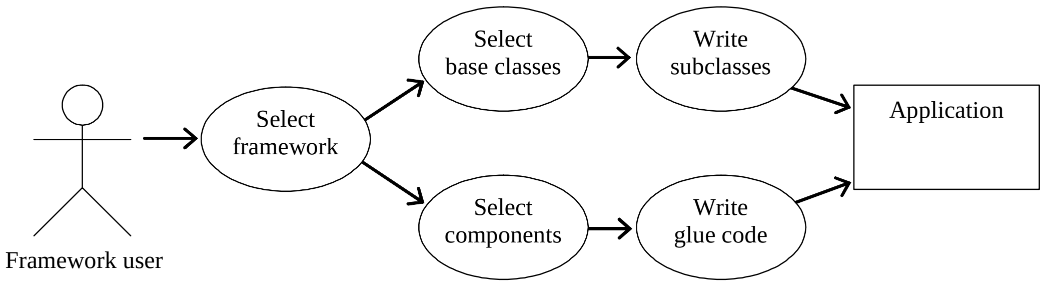


Figure 2. Specializing a framework.

White-box frameworks, based on inheritance, require the framework user to create many new subclasses with a substantial amount of code. While most of these new subclasses may be simple, their number and interactions can make the task difficult for an inexperienced programmer [Johnson and Foote 1988]. Black-box frameworks based on object composition require less coding, but still the rules behind the component composition may be hard to understand. Ideally, the framework user should be able to create the application just by assembling existing components. However, in practice the set of available components is seldom rich enough. Reuse by inheritance is often necessary to build the missing components and functionality that cannot be expressed with the current set of components.

3.3.2 Pattern-Based Tool Support for Frameworks

One of the reasons why learning and using a framework is hard, is that comprehending an object-oriented design as such is difficult. For instance, Demeyer et al. [1997] has noticed that the inheritance hierarchy of a software system tells only little about its architecture; inheritance describes relationships between classes, not objects. Although the features of the implementation language can be used to state some aspects of the software architecture in the interface-level (e.g., abstract and final methods in Java) they can express only a fraction of the rules associated with the architecture. Clearly, as discussed in Section 1.2, conventional object-oriented language structures consisting of class declarations and operation signatures is not enough to explicitly describe the flow of control and rules between the framework and its specializations.

To make its use easier, the framework may have tools to help its specialization. Traditionally, black-box frameworks have been considered to be better at serving as the foundation of a supporting tool. Such a black-box supporting tool can let the user to choose framework components and connect them together. Finally, based on the selected components, the tool can generate the specialization.

However, as discussed in Subsection 3.3.1, the framework's component library is usually insufficient. The white-box reuse is often needed in order to implement more advanced features and design solutions with the framework. Thus, a supporting tool should also guide how to write new subclasses. In addition, it should check that the application-specific implementation obeys the – often invisible – specialization rules of the framework.

As discussed in Subsection 2.1.2, framework-specific patterns can describe how a framework should be used. This is illustrated in Figure 3; for a particular specialization problem, a pattern could describe which component or base class should be selected and how it should be used.

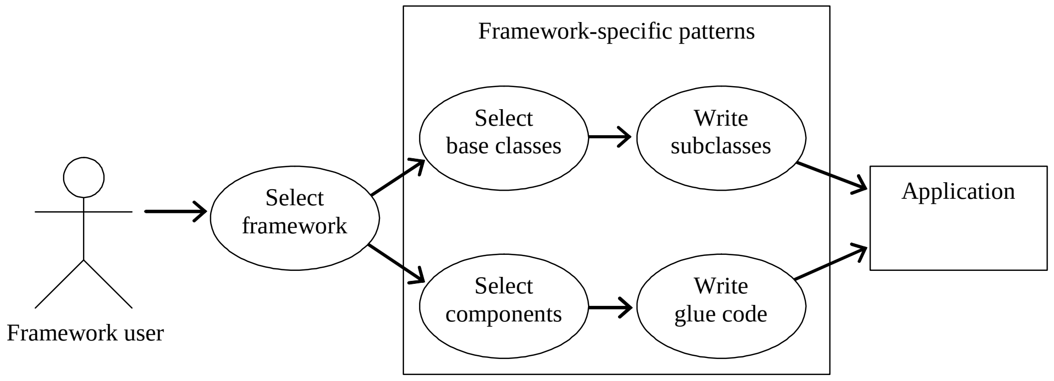


Figure 3. Specializing a framework with framework-specific patterns.

If an advanced pattern tool could use these framework-specific patterns as input, it could help the application developer to specialize both black-box and white-box frameworks. Such a pattern-based tool support could partially automatize the pattern instantiation process and maintain the bindings between the pattern specifications and the pattern instances. The tool could help the user to write the required specialization code and to check that the specialization obeys the rules of the framework. In addition, the use of patterns could be systematically documented. Particularly, as will be shown in this dissertation, the pattern-based tool support could help the framework user in the following tasks:

- A tool could guide the user to specialize the framework.
- A tool could trace the involved code elements and verify the correctness of the specialization against the rules of the underlying patterns.
- A tool could help the user to learn and understand the framework.

CHAPTER 4

OUTLINING A TOOL PLATFORM FOR PATTERNS

Patterns are a universal and practical approach to document approved solutions and how to apply those solutions in different circumstances. The problem is that patterns are informal documentation, suitable for human readers, but too ambiguous and indefinite for efficient tool support. As reading and applying patterns and documenting pattern instances is not systematically supported, this hinders the large-scale industrial use of the pattern concept. Thus, a general and integrable tool support to create and exploit patterns sounds promising. For example, framework developers and middleware providers could annotate their software with patterns describing how the products should be used. Then, the client who is working with a common software development environment could apply this reusable software by instantiating the associated patterns. Further, the use of the patterns could be tracked by the system, making it possible to check violations and to generate pattern-based documentation and tutorials for learning and maintenance purposes. Such pattern-based environment could speed up the software development process and improve the quality of the software product.

This chapter outlines a tool platform to apply patterns during software development. The platform can be integrated into existing software development environments and it can be used to instantiate patterns and to maintain and document their instantiations. The described tool platform assumes that the key to provide a successful and practical pattern-based tool support is to accept the fact that software development and programming is creative and gradually proceeding work carried out by humans with different skills and understanding. The use of patterns should be transparent, respect human cognition, and utilize the idea of piecemeal growth discussed by Alexander (Subsection 2.2.1). The pattern-based tool support should smoothly and almost inconspicuously guide ordinary programmers and software developers to reuse approved design solutions and conventions, even if the users are not conversant with the theory behind the pattern concept itself.

The outlined pattern tool platform has some characteristics that make it universal, practical, and easy to use; these are discussed in Section 4.1. An overview of the platform is given in Section 4.2. Tool supported pattern specifications are discussed in Section 4.3. The hierarchical system of these pattern specifications is discussed in Section 4.4. Different tools to utilize pattern specifications are discussed in Section 4.5. The possibility to integrate the tool platform into a real software development environment is discussed in Section 4.6.

4.1 Requirements

The main advantage of the pattern tool platform is that it provides common services and functionality for a family of pattern tools. The platform can, for example, trace the pattern instances automatically. The following list summarizes the general requirements for such a pattern tool platform. These requirements emerge from the nature of the pattern concept and from the practical experiences gained with JavaFrames [JavaFrames 2004]. The JavaFrames system itself is compared to these requirements in Section 10.1:

- *Integrable.* To be used, the platform must be integrable into other development environments. For instance, the system should receive notifications and parse information from the software development environment to check the pattern instances.

- *Extensible.* As patterns are widely applicable in different programming languages, tools, and development environments the platform must be extensible. The platform cannot provide a predetermined and fixed set of different pattern types and tools; instead it must allow the user to extend the system with new kinds of tools and pattern specifications.

- *Cohesive.* Patterns are a bridge between the design and its implementation. The platform must support this unity. The platform is cohesive because it manages the unity of patterns and their instances. It must be able to trace individual program elements to the corresponding pattern specifications and to check that those program elements are not violating the patterns they are involved in. The architectural rules that must be followed during and after the pattern instantiation process can be seen like a higher level typing system. In the same sense as the code must conform to the typing rules of the implementation language, it must conform to the architectural rules encapsulated by the used pattern specifications.

- *Scalable*. The range of patterns varies from architectural patterns and design patterns to idioms and coding conventions. The scale of software products varies from components to full-scale applications and large software systems. The platform must be scalable to allow the pattern modeler to create both architectural pattern specifications and idiom-level pattern specifications for different domains.

- *Precise and explicit*. With pattern specifications, it must be able to describe the precise structure of the intended solution. The platform must provide the required building blocks (because the platform is extensible, new building blocks can be created if necessary) that cover the problem domain as completely as possible and allow the user to create detailed pattern specifications. Clients must be able to apply these pattern specifications in full-scale software projects.

- *Incremental*. As discussed in Subsection 2.2.1, it seems that for humans the best way to instantiate patterns is to instantiate them gradually, step by step, where each step may have effects to the steps to come. Thus, it sounds natural that the platform is incremental and supports the pattern instantiation as a gradually proceeding work. To do this, the platform should provide a dynamically adjusted list of fine-grained instantiation tasks. The user should be able to execute these tasks in small portions, see their effects in practice, and go back to undo tasks or to change the involved elements. This kind of working is inherent to software engineering, and the platform should support it. In this way, the user has better control and understanding of the pattern instantiation and of the constructed software product.

- *Generative*. Patterns are used to create new elements and to modify existing ones; they are generative. Also the platform must support this generativity to create and adjust the elements to conform patterns. During the pattern instantiation the platform can gather application-specific information, which can then be used to generate the required elements and documentation, so that the result is perfectly customized to the current application. The incremental nature of the platform makes it natural to propose these new elements with appropriate explanations immediately to the pattern user. The platform can then generate most of the regularly repetitive parts of the solution. Using the pattern instantiation tasks to generate source code makes the platform an incremental code generator.

- *Open-ended.* There is no strict end point for the use of patterns. The platform must allow the user to add new pattern specifications and to modify existing ones. Clients must be able to start instantiating new patterns and to revisit old instantiations, even for an already completed software product. The platform must be able to save and restore the associations between the pattern specifications and the concrete software elements. This makes the use of the platform open-ended.

4.2 Overview

With a proper tool support the use of the pattern concept discussed in Chapter 2 could be made more effective and standardized. The concept could then be used, for instance, to specialize frameworks as discussed in Chapter 3. A general insight of the outlined pattern tool platform is given in the following subsections:

- *Problems with patterns.* Patterns are informal documentation. Due to their informal nature and the large number of available patterns, the use of patterns has some drawbacks. Subsection 4.2.1.

- *Use and users of the tool platform.* With the platform, the pattern-based tool support resembles a human tutor, which, rather than giving a lecture beforehand with abstract terms, guides the user with practical advices during everyday programming activities. Subsection 4.2.2.

- *Pattern specifications and the core of the tool platform.* The core of the tool platform extracts instantiation tasks from the pattern specifications and verifies the pattern instances. Subsection 4.2.3.

- *Pattern tools and integration into other environments.* Pattern specifications are managed and used with various pattern tools. To be generally applicable, the platform and the tools must be integrated into a real software development environment. Subsection 4.2.4.

4.2.1 Problems with Patterns

Clearly, patterns have many benefits. However, patterns cannot resolve all problems and they certainly do not guarantee that the software development process will be a success. As such, patterns have some drawbacks that hinder their use. A tool support could solve some of the following problems and improve the use of patterns:

- *How to write a pattern?* Like with any documentation, it demands time and skills to write a good pattern. This is creative human activity that cannot be fully automatized.

- *How to find a suitable pattern that solves a specific problem?* There exists a large number of patterns, like the design patterns presented by Gamma et al. [1995], and each pattern may have a number of variations. Selecting the right pattern requires experience. It is not always obvious what is the correct pattern, or even if there exists a suitable pattern that is applicable for the current problem. Thus, for inexperienced software developer it can be difficult to find a set of pattern candidates and to select the optimum one, or to note that any of the patterns cannot be used to solve the problem. If the patterns are formally specified, a tool could be used to compare and find them with some criteria. However, selecting patterns automatically is a difficult problem. The final responsibility of the used patterns belongs to the software developer.

- *How to ensure that a pattern becomes correctly and efficiently instantiated in the current context?* Patterns are abstract and informal descriptions; following ambiguous instructions and small set of examples may be difficult and time-consuming. If the original patterns are formally specified, a tool could help in their instantiaton; the tool could provide instantiation hints and verify the implementation against the given pattern specifications.

- *How to document the use of a pattern after it has been instantiated?* A pattern describes the problem-solution pair in an abstract and common way that can be used multiple times in different contexts. Its instantiation, in turn, consists of certain software elements in a very specific context. The original pattern documentation is too abstract, missing the bindings to the concrete instantiation and the problem specific reasoning why the pattern was selected and used. If the original patterns are formally specified, a tool could trace their instances. As the bindings between the software elements and the used pattern specifications are known, a tool could generate documentation about the instantiated patterns.

- *How to trace the instantiated solution back to its pattern(s)?* Typically, a pattern is forgotten once it has been instantiated. Related to the documentation problem, it is hard to see directly from the source code or design models what roles a software element plays in different pattern instances. This makes it difficult to maintain and validate the solution later on. If the original patterns are formally specified, a tool could trace the software elements that are involved in the pattern instance.

- *How to find existing pattern instances from a software system?* Identifying patterns from existing software systems may help to document and maintain those systems, making their internals more structured, flexible, and easier to understand. Again, it is cumbersome to trace patterns manually, based on abstract and informal descriptions. If a tool knows the general structure of the searched patterns, it could help to find their instances.

- *How to remove or replace obsolete or counterproductive patterns from a software system?* Patterns can be misunderstood and misused. When the software system evolves, it may be necessary to remove or replace old patterns. If the original patterns are formally specified, a tool could automatically trace their instances and help to remove or replace these instances.

4.2.2 Use and Users of the Tool Platform

A general pattern-based tool support has four kinds of users shown in the UML use case diagram in Figure 4. The *pattern modeler* creates precise pattern specifications. Note that the pattern modeler should not be confused with the pattern writer discussed in Subsection 2.1.5. Instead, the pattern modeler may use informal pattern descriptions as blueprints, when creating more precise specifications. The *pattern user* applies and instantiates the created pattern specifications. She can also use the pattern-based documentation to learn and compare the applied design solutions. The *documentation producer* creates documentation, reports, and tutorials by utilizing the pattern specifications. Finally, the *platform developer* is an expert who can create advanced pattern tools (Section 4.5) and new pattern semantics (Section 4.3). For instance, a typical pattern semantics could cover the elements and structures of some programming language, like Java or C++. The pattern modeler could then utilize this pre-made semantics to create, for example, Java-specific pattern specifications to describe the intended specialization of a Java framework.

To illustrate the use and advantages of the platform, consider typical instructions for framework specialization. The problem with traditional documentation and examples is that they have to be written before the specialization takes place. Therefore, the documentation has to be given by using the abstract concepts of the framework, not with the concrete concepts of the specialization. This static and informal documentation cannot generate any of the required software elements or verify that the specialization really obeys the rules of the framework. Meanwhile, a sophisticated pattern-based tool could be used to guide the pattern user to specialize the framework with illustrative and practical programming tasks. The underlying tool platform then gathers information about the current stage of the specialization and customizes the pattern-based documentation and default implementations with more specific terms, reflecting the choices and source code the pattern user has already made. At the same time, the user sees step by step how the framework specialization proceeds making it easier to understand the architectural implications of the framework.

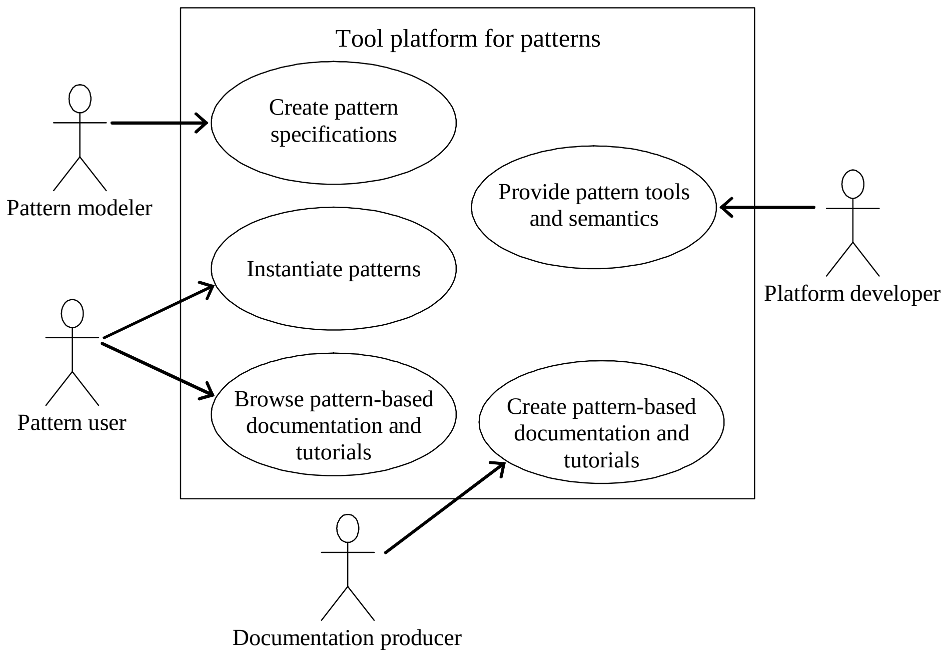

Tool platform for patterns
Create pattern specifications
Pattern modeler
Provide pattern tools and semantics
Instantiate patterns
Platform developer
Pattern user
Browse pattern-based documentation and tutorials
Create pattern-based documentation and tutorials
Documentation producer

Figure 4. Use and users of the pattern-based tool support.

We argue that the incremental task-driven pattern instantiation process supports learning-by-doing. For instance, in a software company, new employees could start by going through the generated documentation and by instantiating some example patterns. As the use of patterns is supported by generating guidance to provide the missing elements and to fix errors, the user could experiment with different aspects of the solution. This kind of activity speeds up the learning process and could make novice software developers productive more quickly. Hence, besides making the use of patterns more efficient with advanced code generation facilities, the platform could be used as a training aid in a company, complementing more traditional documentation.

Besides novice users, also the experts could be served. They could utilize the platform by letting it automatically generate a lot of essential and strictly regulated, but uninteresting code. Unlike with ordinary wizards and software development environments, the code is not generated as a large and static block. Instead, with a proper tool support, the code generation can proceed step by step, so that the pattern user is not overwhelmed by the generated code but can become convinced of its rationale.

In addition, to ensure quality and robustness of the software product, it is often important that programmers obey some company related rules and conventions. By defining these rules as a set of pattern specifications, the pattern-based tools could be used to restrict and remind programmers. Though this may sound limiting, it assures that the programmers will perceive the vital aspects of the used software architecture and framework.

4.2.3 Pattern Specifications and the Core of the Tool Platform

As discussed in Section 2.3, informal pattern descriptions are too ambiguous to enable efficient tool support. Instead, more precise pattern specifications are required. The pattern tool platform provides a common mechanism to construct these pattern specifications (Section 4.3). In addition, the platform provides a general mechanism to instantiate patterns, where the instantiation process can be seen as a sequence of tasks that are carried out by the pattern user. The process hides the complex details of the underlying pattern specifications. Instead, the pattern user sees the instantiation as a sequence of simple programming tasks that can be performed during normal programming activities.

The core of the tool platform is the *pattern engine* discussed in Subsection 4.3.5. It extracts tasks from pattern specifications and verifies pattern instances. Advanced pattern tools can then utilize the services of the pattern engine to show the required tasks. For the pattern engine this is not a trivial problem, as tasks cannot be given as a fixed predetermined list. Patterns describe a family of solutions and each solution may require a slightly different set of tasks. The challenge is to provide meaningful and simple programming tasks dynamically during the pattern instantiation so that they are focused on the current application-specific problem. This *task automaton* is discussed in Subsection 4.3.6.

To illustrate the task mechanism, consider a pattern specification that describes the solution in terms of roles. Typically, a role represents some code element required in the solution. In a simplified manner, if the pattern specification contains a role that represents a class, the pattern engine can ask the pattern user to provide the required class. Further, if the role has more information about the class, this information can be utilized to generate, for example, a default implementation for the class.

4.2.4 Pattern Tools and Integration into other Environments

The pattern tool platform constitutes a foundation that provides a common mechanism to create and use pattern specifications and advanced pattern tools. Various pattern tools are discussed in Section 4.5. For instance, the pattern modeler can use pattern development tools to create new pattern specifications. For the pattern modeler it is not necessary to know how the pattern engine or the platform actually works; she can rely on that the platform will eventually calculate and present the required instantiation tasks to the pattern user. The pattern user, in turn, can use deployment tools to instantiate pattern specifications. For the pattern user it is not necessary to know the details of the underlying pattern specifications or the used formalism. Instead, she can proceed by performing simple and meaningful programming tasks. Other possible tools are, for intance, tools to generate pattern-based documentation and tools for pattern mining and analysis.

To be practical and truly useful, the platform must be integrated into a real software development environment. This is discussed in Section 4.6. Such an environment could then utilize the services of the pattern tool platform and offer a useful set of pattern tools to the software developer.

4.3 Concepts of a Pattern Structure

In this dissertation the exact syntax of pattern specifications is omitted. Instead, to discuss about the pattern tool platform, it is enough to explain the abstract concepts of such a pattern specification. Here the author has selected the concepts that were used and evolved during the implementation of JavaFrames [Hakala et al. 2001b, 2001c, 2001d; Hautamäki 2002; Viljamaa A. 2001, 2004; Viljamaa J. 2002, 2003, 2004; JavaFrames 2004]. The basic concepts are explained in the following subsections:

- *Roles, dependencies, multiplicities, and bindings*. A pattern specification consists of roles. Typically, a role represents a concrete element or set of elements in the solution. When a pattern specification is instantiated, its roles are bound to those elements. Subsection 4.3.1.

- *Role properties and pattern semantics*. To enable different role types and to make the roles flexible, each role may have a different set of properties that can be evaluated when the pattern is instantiated. Subsection 4.3.2.

- *Constraints*. Each role may have a set of constraints. Subsection 4.3.3.

- *Example: specification for the Abstract Factory pattern*. The Abstract Factory pattern is used to demonstrate pattern specifications. Subsection 4.3.4.

- *Tasks and the pattern engine*. A task is a small logical step that guides the pattern user to instantiate the pattern specification. A typical task asks the pattern user to create a particular code element. The pattern engine extracts tasks from the given pattern specifications. Subsection 4.3.5.

- *Task automaton*. The list of tasks is not fixed and static; instead it is continuously checked and re-evaluated. Subsection 4.3.6.

4.3.1 Roles, Dependencies, Multiplicities, and Bindings

In literature, the basic building blocks of a pattern are often called *roles*. Therefore, it is natural to think that also a formal pattern specification consists of roles. A role describes a concrete implementation elements or some other aspect of the solution. Roles are organized hierarchically so that each role may have a number of child roles. The structure of roles corresponds to the structure of elements in the solution. Instantiating a pattern specification means that the pattern user creates, modifies, or locates elements so that the roles become *fulfilled*.

For example, in the case of a Java program, roles are typically used to represent code elements like classes, methods, fields, and constructors. Correspondingly, one can talk about class roles, method roles, field roles, constructor roles, and so on. A class role may have a constructor role and some method roles and field roles. These roles, in turn, may have more fine-grained roles to represent the internals of the actual code elements.

Intuitively, there is a *dependency* between two roles if the first one cannot be fulfilled before the second one. Thus, there is a dependency from role R_2 to role R_1 if the definition of R_2 refers to R_1. For example, as mentioned before, roles are organized hierarchically so that each role may have a number of child roles. A class role may have some method roles; such a method role depends on the class role because the required method cannot be implemented before the enclosing class has been defined. Besides containment relations, the described solution may require a number of other relations and interactions between the involved elements. For example, a method may require a particular return type or it must override another method. All these requirements have implications for the corresponding roles in the pattern specification. The dependencies between roles set up the order in which they can be fulfilled.

A role represents an element or a set of elements in the intended solution. Thus, a role may be fulfilled more than once during the instantiation. The *multiplicity* defines the minimum and maximum number of fulfilments for the role in respect to its dependencies. Typically, the multiplicity is the number of elements that should play the role in respect to other elements in the solution. For example, role R_1 may represent a base class while role R_2 represents its subclasses. It is natural to think that the multiplicity of R_1 is 1 (exactly one) while the second role has multiplicity [1..*] (one or more) indicating that there can be multiple subclasses. However, if R_1, in turn, depends on role R_3 which has multiplicity [1..*], R_1 must be fulfilled once for each fulfilment of R_3 (for example, to create new product families with different base classes).

A common problem when using patterns is that the pattern instances are lost after instantiation (Subsection 2.2.2). Afterwards it is difficult to find and verify the participating elements of the solution. To solve this traceability problem, the pattern tool platform maintains the associations between the roles and the elements. Correspondingly, the commitment of an element, like a class or an operation, to play a particular role is called a *binding*. When an element plays a role, the role is bound to that element and vice versa. If the element does not exist, it can be created as instructed by the role. In addition, roles can be used to check the validity of the bound elements.

To summarize, in the pattern specification each role is like a small abstraction of a thing or a relationship that is required in a solution. The more extensive and fine-grained the set of available role types is the more expressive and powerful the pattern specifications can be.

4.3.2 Role Properties and Pattern Semantics

A pattern specification consists of roles describing the elements in the solution. This hierarchical structure of roles is like the syntax of the pattern specification. Roles must be flexible and adjust themselves to the current situation of the pattern user. When fulfilled, a role is applied like a template that can be used to produce a family of elements. To do this, each role may have a different set of *properties* that are evaluated at run time during the instantiation. The way these properties and the role bindings are evaluated and used defines the semantics, i.e., the behavior of the role.

Hence, the role semantics defines how the properties are evaluated and used and what is required to fulfil the role. By using properties to denote the generic parts of the represented element the role can be used to produce variations of that element. A role-specific property can refer to information (like another role and its bindings) that is solved at run time by the pattern tool platform. Because the previously bound roles and elements are known by the platform, it is possible to use this knowledge when the properties are evaluated. In that way, the changes in the source code and in the role bindings affect the property values, which can be utilized when creating and checking the pattern instance. That makes it possible to create context-sensitive documentation and source code that adapts seamlessly to the terms and structures of the created software product.

For the platform developer it should be possible to add new role semantics, i.e., role types to model different kinds of elements and constraints. A collection of related role types is called *pattern semantics*. For example, different role types are needed to represent Java classes, methods, fields, and so on. The platform developer has implemented these role types (Java pattern semantics) to model Java elements. The pattern modeler can then use the available role types when constructing her pattern specifications. Finally, the pattern specifications are instantiated by the pattern user, for instance, when specializing a Java framework.

4.3.3 Constraints

Each role may have a set of constraints. These constraints can be used to force naming conventions, exceptions, method parameters, method calls, and so on. For instance, in the case of object-oriented languages, concepts like inheritance, overriding, and return type can be seen as constraints set to the element in the solution. When a role is fulfilled by creating an element, these constraints can then be checked to ensure that the element has the required relationships and features.

Technically, constraints can be seen as special kinds of roles. The main reason why constraints should be implemented as separate role types is that it simplifies their use and makes the system flexible. Instead of implementing the logic of every possible constraint inside every possible role type, the platform developer creates a set of constraint types which can then be attached to a role only if necessary. For instance, the pattern modeler may create a Java method role with a number of constraints as child roles. Each of these constraints is fulfilled separately, after the method itself has been created.

Of course, it is up to the platform developer how different role types and constraints are actually implemented. The platform provides a common basis to implement different kinds of pattern semantics. This will be demonstrated in Chapter 7.

4.3.4 Example: Specification for the Abstract Factory Pattern

To illustrate the discussed pattern specification, a more precise specification for the Abstract Factory pattern (Subsection 2.1.4) is shown in Figure 5. Details of the specification are given in Appendix A. In the figure, roles are indicated as UML stereotypes (e.g., <<classRole>>), where the stereotype indicates the type of the role. Dependencies between roles are shown as dashed arrows and by nesting roles. The role's multiplicity is shown after the role name; by default the multiplicity is 1. To avoid confusing role's multiplicity with the multiplicity concept used in UML, special multiplicity symbols are used: "?" for [0..1], "*" for [0..*], and "+" for [1..*].

As shown in Figure 5, the specification of the Abstract Factory pattern has implementation details that were not precisely specified in the original pattern shown in Table 2. For instance, besides the roles to represent the required classes and operations, there is a specific role to add code fragments inside the createProduct method. In addition, there are some inheritance and overriding constraints that were only superficially or indirectly discussed in the original pattern documentation. For a tool, these kinds of constraints must be defined precisely. ProductFamily role, in turn, is a special role to start creating a new product family. Instantiation of the Abstract Factory pattern specification is demonstrated in Subsection 4.3.6.

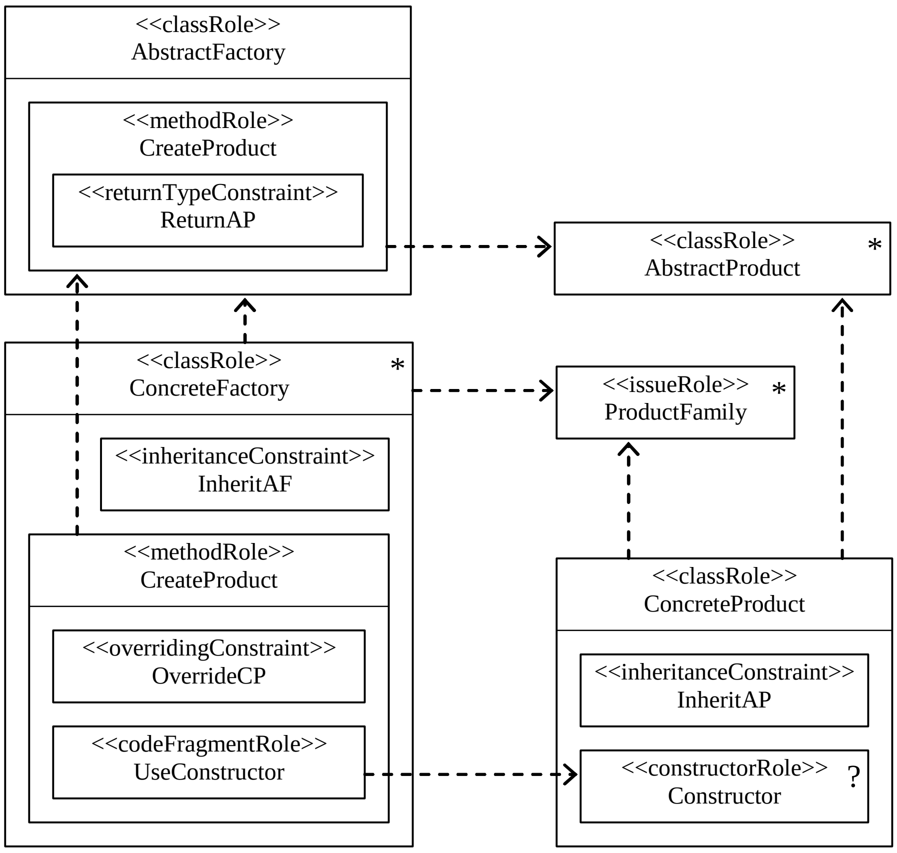


Figure 5. Roles and dependencies of the Abstract Factory pattern specification.

4.3.5 Tasks and the Pattern Engine

The pattern tool platform provides a pattern engine, which is like an interpreter that takes a pattern specification as input and generates tasks as output. A task is always based on a specific role and it utilizes the semantics and properties of that role. The meaning of the tasks is to help the pattern user to instantiate the pattern by creating and modifying the required elements and to bind the elements to the corresponding roles. A typical task is a simple programming task that can be performed with the default tools of the software development environment, like the source code editor, but it could also be performed with more advanced role-specific tools. Depending on the underlying role, a task may have one of the following purposes:

- *Binding task*. A binding task is used to bind an element to a particular role. To perform a binding task, the pattern user must create or select an element as specified by the underlying role. For example, when doing a binding task, a dialog may show the available elements or, if a suitable element does not exist, the system may generate a default implementation for it. Typical binding tasks could be: "Provide class C_1", "Provide method M_1 in class C_1", and "Provide a call of method M_2 inside method M_1". The performed binding tasks can also be seen as a link between the role and the element.

- *Notification task*. A notification task is used to advise or remind the pattern user. Unlike with a binding task, performing a notification task does not create or modify any concrete elements. A notification task could be: "Read the licensing agreement".

- *Grouping task*. A grouping task is used to organize other tasks, and in that way the pattern instantiation. In the case of the Abstract Factory pattern, a typical grouping task could be: "Provide a new product family". This would start the process to create that product family.

- *Repair task*. A repair task is used to repair an element that violates a constraint. If the pattern user violates the rules implied by the underlying constraint, a repair task is generated to fix the situation. Examples of repair tasks could be: "The class C_1 must inherit the class C_0" and "The method M_1 must override the method M_0".

In addition, depending on the current stage of the instantiation process, a task can be in one of the following states:

- *Done*. A task is done if it has been performed. Performing a binding task means that the pattern user binds an element to its role. Performing a notification task means that the pattern user acknowledges the task. Performing a grouping task means that the pattern user wants to start performing the subtasks. Performing a repair task means that the pattern user repairs the violation that created the task.

- *Pending*. A task is pending if it is not done but cannot yet be performed. As discussed in Subsection 4.3.1, a role cannot be fulfilled if it depends on another role, which hasn't yet been fulfilled. Tasks extracted from such a dependent role cannot be performed until the tasks to fulfil the other role have been performed.

- *Mandatory*. A task is mandatory if it can be performed and it is required in order to fully instantiate the pattern specification.
- *Optional*. A task is optional if it can be performed but it is not required in order to fully instantiate the pattern specification.

The fact whether a task is mandatory or optional can be determined from the role's multiplicity settings, dependencies, and the number of the current bindings of the role. Performing a mandatory or an optional task may cause new tasks, as some pending ones may become available.

4.3.6 Task Automaton

As discussed in Subsection 2.2.1, patterns should be instantiated gradually, step by step. To imitate this piecemeal growth, the pattern engine extracts tasks to instantiate pattern specifications; concrete implementations of this *task automaton* has been explained by Hakala et al. [2001d] and Viljamaa A. [2004]. Roughly speaking, the pattern engine generates a task for any role that can be fulfilled at that point of the instantiation process or if the role is bound to an element and the element violates some of the constraints. For example, in the case of Java pattern semantics, the overriding constraint analyzes method signatures and determines if the overriding relation between two methods is true. If an error is detected, the constraint causes a mandatory repair task to fix the method signature. Similarly, based on the role's multiplicity settings, a binding task can be generated to point out or create a suitable element, either compulsorily or optionally.

Task automaton is possible because the pattern engine knows the existing roles, code elements, and bindings between them. If the system is properly integrated into the software development environment, it also gets notifications if the elements are modified. The mechanism ensures that when starting the instantiation process, the first mandatory or optional tasks are based on the roles that are not depending on any other role. After performing these first tasks, other tasks to fulfil the depending roles become available. By using the dependency information and multiplicity settings, the pattern engine can determine the order in which the tasks can be performed and if the task is optional or mandatory.

A sophisticated pattern deployment tool (Subsection 4.5.2) can utilize the services of the pattern engine and represent the generated tasks to the pattern user as a dynamically updated task list. Violated constraints and missing elements will cause new tasks until the whole pattern specification is correctly instantiated. In addition to the explicit user interactions, different role semantics may include rules and heuristics to determine if a task can be performed automatically. However, often the instantiation process cannot rely on such heuristics alone. By asking explicit commitment from the pattern user, the pattern engine utilizes human's intuition and wider knowledge about the problem.

Undoing a performed task means that the pattern engine reverses to the previous possible state before the task was performed. In the case of a binding task, the binding between the role and the element is cancelled. However, it may be problematic to change the elements and to destroy the ones created with the tasks. The same element may be bound to multiple roles in different pattern specifications. In addition, the element may have responsibilities outside the pattern specifications that cannot be determined by the pattern engine.

Figures 6 - 9 illustrate the instantiation of the Abstract Factory pattern specification, discussed in Subsection 4.3.4. As mentioned before, a task is always based on a role. In a sense, roles can be seen as meta-classes that are used to create task objects. For this reason, the UML object diagram is used to represent the task automaton.

In Figure 6, the relationship between a task and a role is indicated with the stereotype <<basedOn>>. In the figure, a task object has also some attributes. *Prompt* is the title of the task; the pattern engine generates this string value by using the corresponding property of the underlying role. *State* represents various task states discussed in Subsection 4.3.5. *Role* is the underlying role this task is based on. The same role can have multiple tasks; *id* is used to identify different tasks of the same role. In the initial state, the pattern engine has generated three tasks; one to create the abstract factory class, one to provide a new product type, and one to create a new product family. These tasks are based on roles that are not depending on any other roles and therefore they can be fulfilled first. Of course, when creating the pattern specification, the pattern modeler could define the order of these roles differently, by adding new dependencies between them.

In Figure 7, the pattern user has performed the task to provide a new abstract factory class. She has also performed another task to provide a new product interface. The created code elements (WidgetFactory and ScrollBar) have been associated with the tasks, and in that way, they have been bound to the underlying roles. This is indicated with the <<boundTo>> stereotype. To make the figure more illustrative, the performed tasks are shown as gray rectangles. Also the states of the performed tasks have been changed. In the figure, the dependencies between the task objects are corresponding to the dependencies between the underlying roles. For instance, the CreateProduct role is depending on AbstractFactory and AbstractProduct roles that have just been fulfilled.

At this time, the pattern engine can present the corresponding mandatory task to provide a method to create ScrollBar objects. Also, a new optional task to provide another product type is shown. The multiplicity of the Abstract-Product role indicates that there can be more than one product types. After the pattern user has provided the first one, the pattern engine generates a new optional task to provide the next product type, and so on.

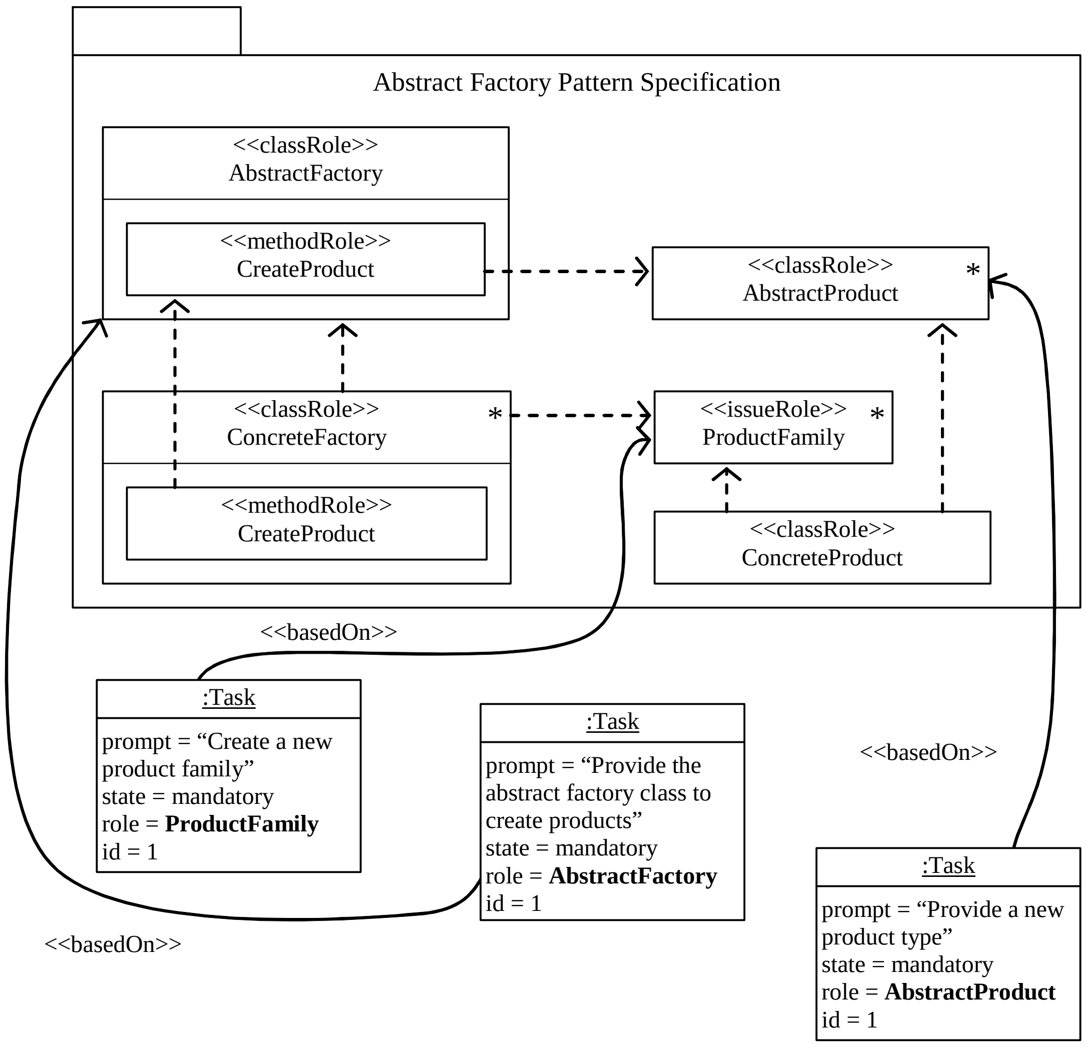


Figure 6. Roles and tasks of the initial state.

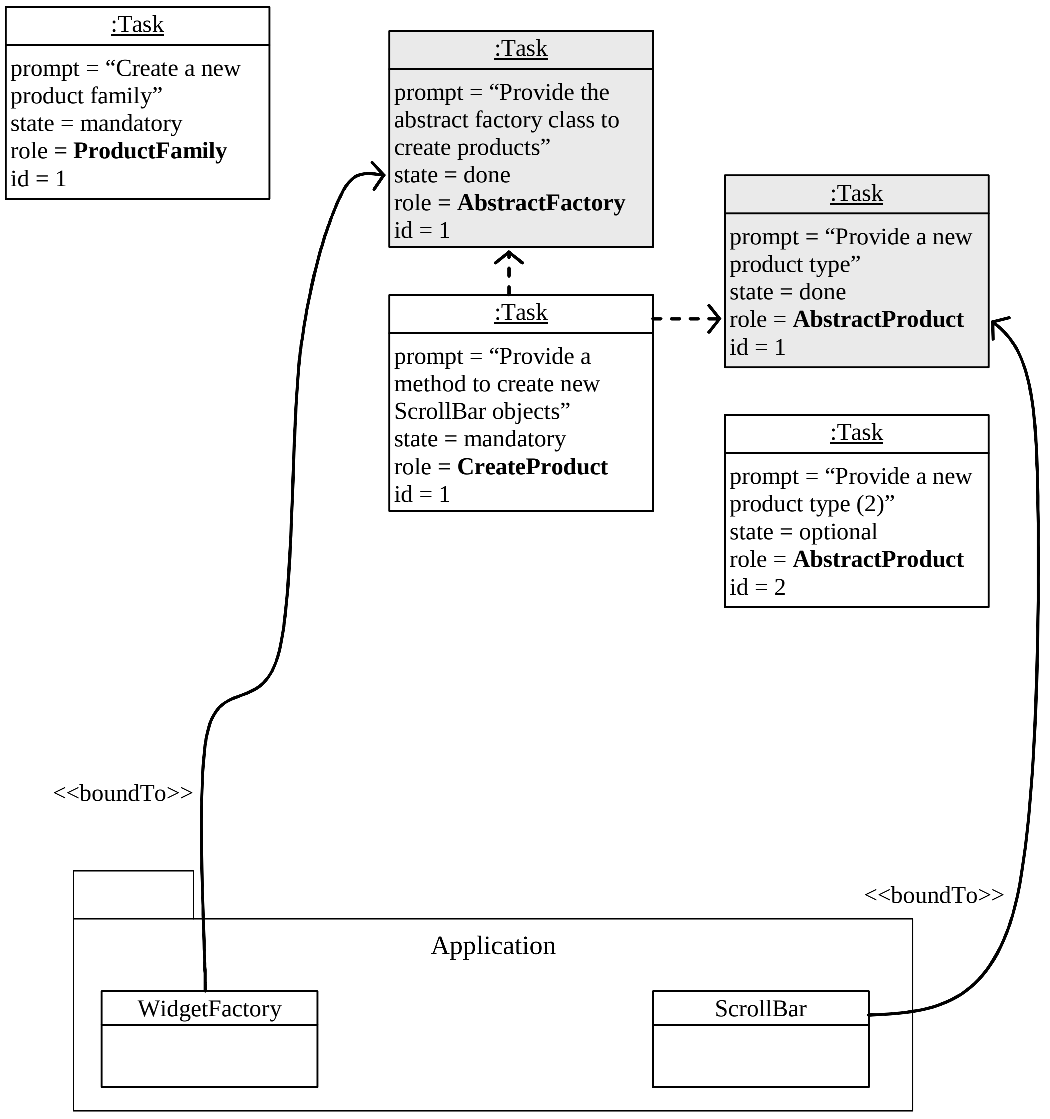

:Task
prompt = "Create a new product family"
state = mandatory
role = **ProductFamily**
id = 1
:Task
prompt = "Provide the abstract factory class to create products"
state = done
role = **AbstractFactory**
id = 1
:Task
prompt = "Provide a method to create new ScrollBar objects"
state = mandatory
role = **CreateProduct**
id = 1
:Task
prompt = "Provide a new product type"
state = done
role = **AbstractProduct**
id = 1
:Task
prompt = "Provide a new product type (2)"
state = optional
role = **AbstractProduct**
id = 2
<<boundTo>>
<<boundTo>>
Application
WidgetFactory
ScrollBar

Figure 7. Tasks after creating the WidgetFactory class and the ScrollBar interface.

In Figure 8, the pattern user has provided the abstract method to create ScrollBar objects. In addition, she has created a new product type called Window. Again, the pattern engine checks the dependencies and multiplicity settings and generates more tasks to continue the pattern instantiation.

Figure 9, in turn, shows the situation after the pattern user has performed the task to create a new product family. This means that she has to create concrete factory implementations to create the defined product types. In that way, the pattern instantiation continues until the Abstract Factory pattern has been applied to the current application-specific context.

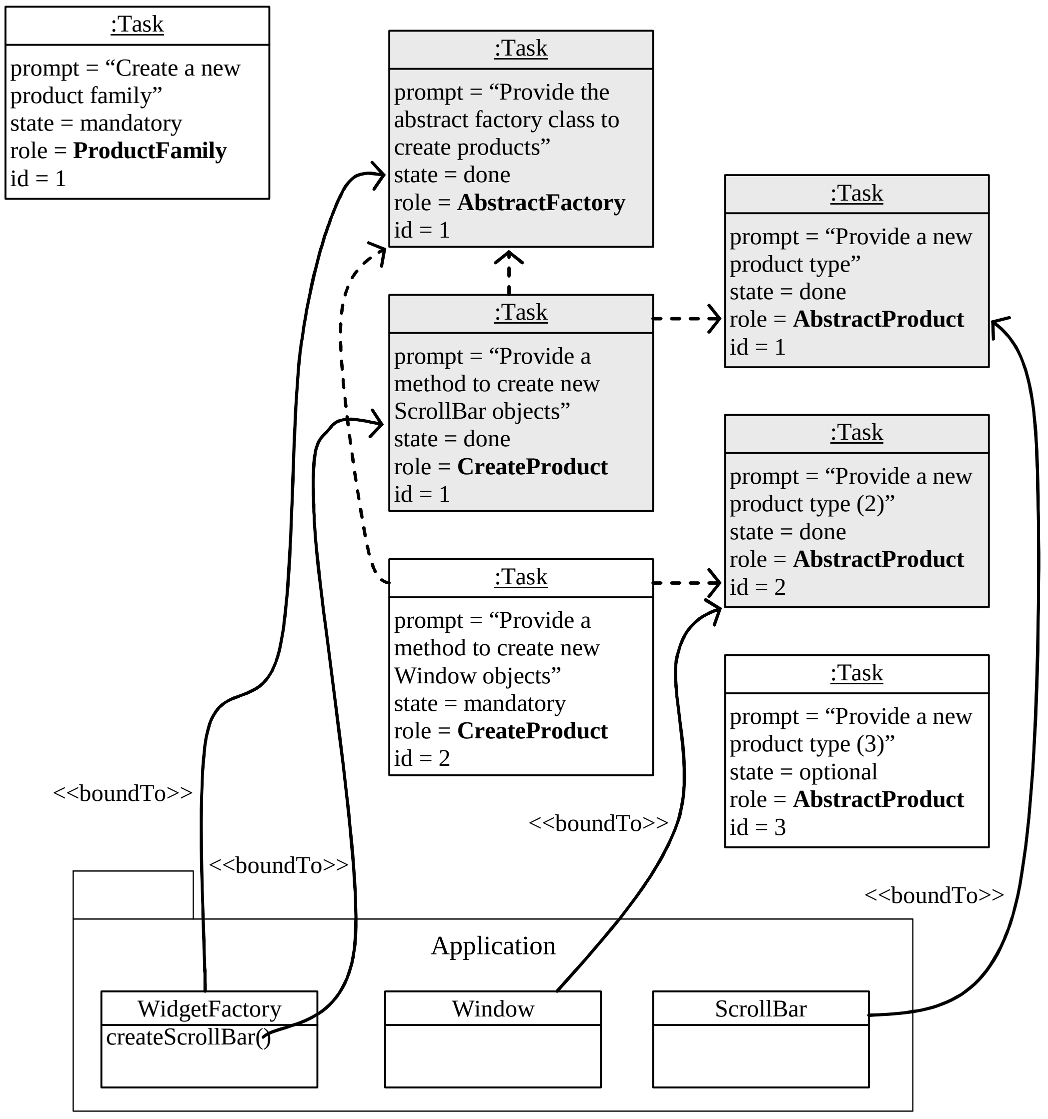

:Task
prompt = "Create a new product family"
state = mandatory
role = **ProductFamily**
id = 1
:Task
prompt = "Provide the abstract factory class to create products"
state = done
role = **AbstractFactory**
id = 1
:Task
prompt = "Provide a method to create new ScrollBar objects"
state = done
role = **CreateProduct**
id = 1
:Task
prompt = "Provide a method to create new Window objects"
state = mandatory
role = **CreateProduct**
id = 2
:Task
prompt = "Provide a new product type"
state = done
role = **AbstractProduct**
id = 1
:Task
prompt = "Provide a new product type (2)"
state = done
role = **AbstractProduct**
id = 2
:Task
prompt = "Provide a new product type (3)"
state = optional
role = **AbstractProduct**
id = 3
<<boundTo>>
<<boundTo>>
<<boundTo>>
<<boundTo>>
Application
WidgetFactory
createScrollBar()
Window
ScrollBar

Figure 8. Tasks after providing the createScrollBar method and the Window interface.

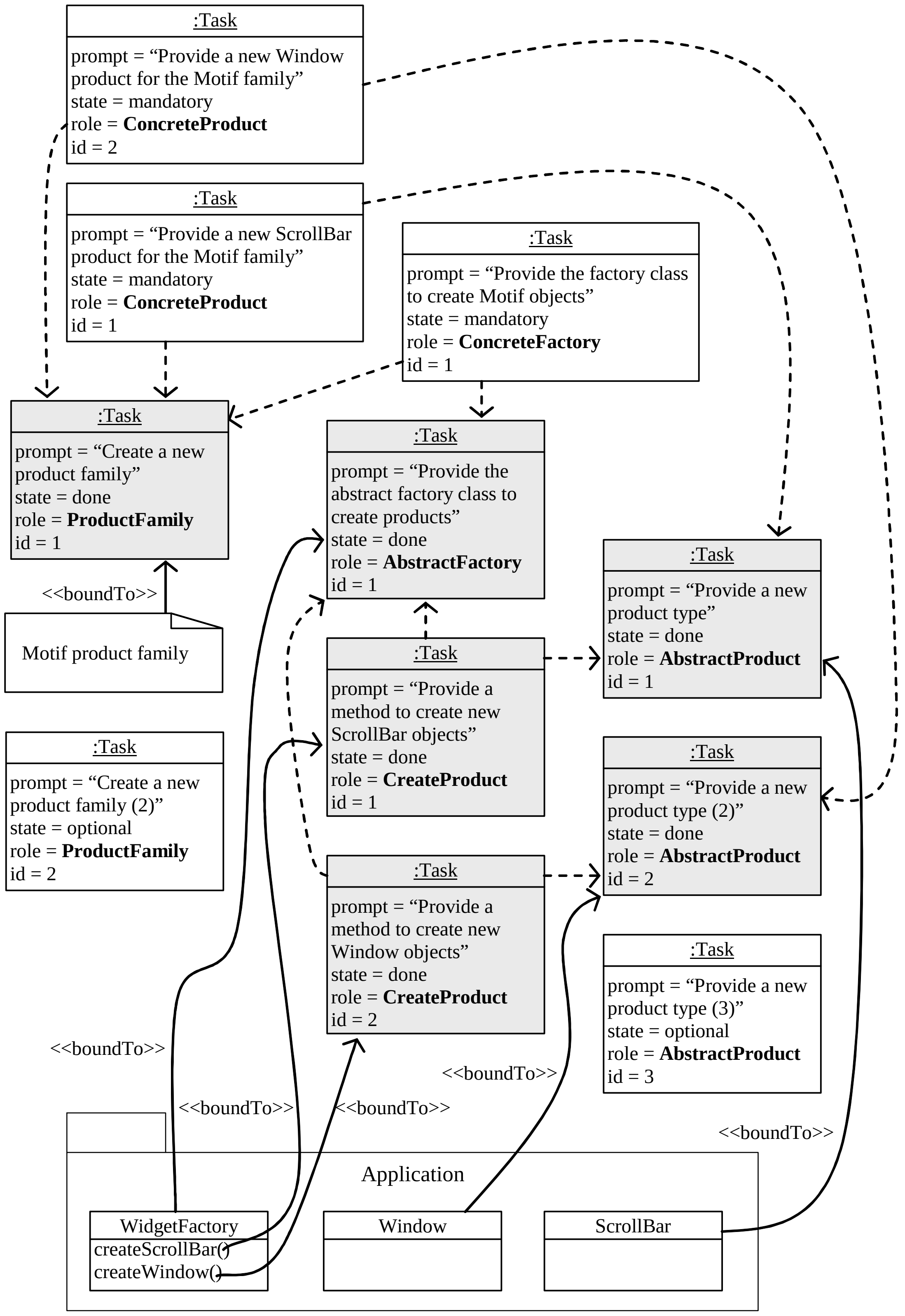


Figure 9. Tasks to create concrete products after establishing the Motif product family.

4.4 System of Pattern Specifications

Each software project has its own system of pattern specifications and their instances maintained by the pattern tool platform. Some of the specifications may represent general design solutions, like the design patterns and architectural patterns discussed in Subsection 2.1.2. Other specifications, in turn, are representing more specific implementation details, like how to specialize a particular framework. A collection of pattern specifications and their instances constitute a hierarchical structure, where more detailed pattern specifications can be derived from the more general ones. This is discussed in the following subsections:

- *Pattern instances*. The concrete result of the pattern instantiation process is the pattern instance. Subsection 4.4.1.
- *Pattern interface*. A pattern interface is a collection of pattern specifications that describe how to use a particular design, for example, how to specialize a framework. Subsection 4.4.2.
- *Pattern layers as a chain of refinements*. Together pattern specifications constitute a hierarchical structure where one specification refines another. Subsection 4.4.3.
- *Pattern composition*. Pattern specifications can be composed of existing pattern specifications to get synergetic advantage. Subsection 4.4.4.

4.4.1 Pattern Instances

Instantiating an informal pattern description manually is illustrated in Figure 10. The pattern user reads the pattern documentation and applies it in the current problem (Subsection 2.2.1). The result is a pattern instance with a set of interacting elements that have no direct connection to the original pattern. This traceability problem makes it difficult to maintain the pattern instance (Subsection 2.2.2).

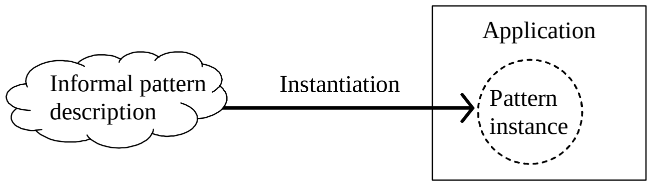


Figure 10. Pattern and its instance.

Precise pattern specifications, in turn, are instantiated with a tool by performing the generated tasks (Subsection 4.3.6). Performing tasks fulfils the corresponding roles and creates bindings between the roles and the elements in the solution. These bindings are stored and utilized by the task automaton to produce more tasks and to check the validity of the pattern instance. This is illustrated in Figure 11.

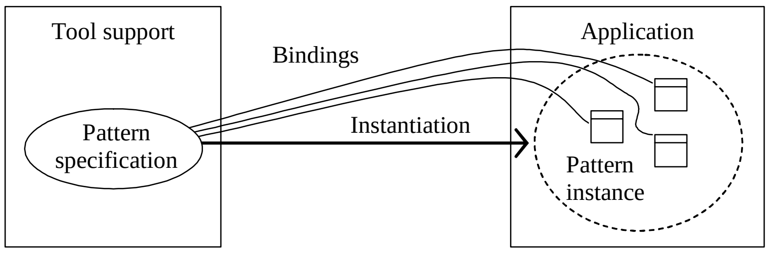


Figure 11. Pattern specification and its instance.

Particularly, the instantiation process of a pattern specification can be in one of the following phases:

- *Uninstantiated*. Any of the mandatory instantiation tasks have not been performed. Usually, this kind of pattern specification represents abstract design solution, like a design pattern or an architectural pattern.
- *Partially instantiated*. Some of the mandatory instantiation tasks have been performed. Typically, this kind of pattern specification represents a framework-specific pattern, in which the pattern modeler has fulfilled specific roles and left the other roles to be fulfilled by the pattern user.
- *Fully instantiated*. All the required tasks have been performed so that the roles of the pattern specification are fulfilled.

4.4.2 Pattern Interface

By creating a set of pattern specifications the pattern modeler creates a *pattern interface* [Hakala et al. 2003] that describes how to use a particular design or a framework. The principle is illustrated in Figure 12. In the figure, framework F provides patterns F_1, F_2, and F_3. The pattern user specializes the framework by instantiating these pattern specifications with some pattern instantiation tool.

Constructing a pattern interface requires the creation and grouping of pattern specifications and determining the order in which they should be instantiated. This imitates the concept of a pattern language; the outcome is a tool supported pattern language with tightly interrelated pattern specifications. To create a pattern interface, the pattern modeler must understand the underlying principles and subsystems of the framework. It may also be helpful to define different pattern interface alternatives for different users. For example, novice framework users may want to learn the basics of the framework, while advanced users just want to boost their coding. A goal-oriented approach to create pattern interfaces is proposed in Chapter 6.

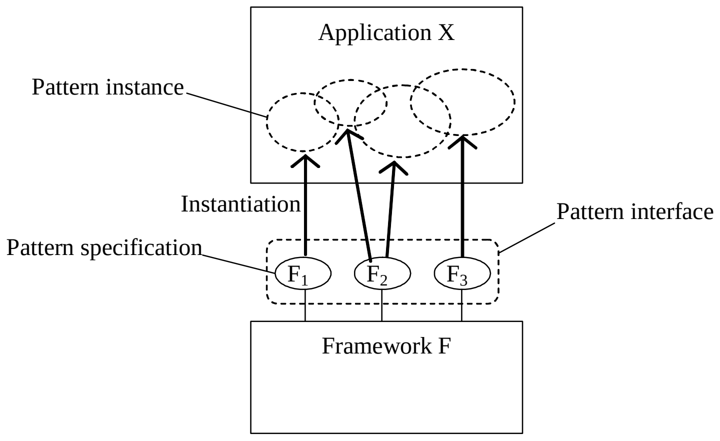


Figure 12. Specializing a framework with a pattern interface.

4.4.3 Pattern Layers as a Chain of Refinements

In a pattern language, more abstract and general patterns may refer to other patterns to produce the final solution. These patterns can be organized hierarchically so that the most abstract layer defines the general outline of the solution, while other layers refine this solution. With tool-supported pattern specifications this can be done by creating a new pattern specification P_2 that *refines* the original pattern specification P_1. This new pattern specification can be seen as a refined copy of P_1. The roles of P_2 can be modified to contain more specific constraints and documentation. The pattern specification can also be augmented with new roles and constraints to describe elements that were not represented in the more general solution. Thus, the pattern modeler can create a new pattern interface that is based on the more general pattern interface.

The principle is illustrated in Figure 13. The pattern catalog contains design pattern specifications that can be used to instantiate design patterns. The pattern modeler refines these pattern specifications in order to create framework-specific patterns that obey some design pattern conventions. She may bound some of the roles to the framework's base classes, add new framework-specific constraints, and so on. Such a system could also support the use of layered frameworks discussed in Subsection 3.1.4. The pattern interface of the more abstract layer could be used to implement other layers and their pattern interfaces.

The refinements can be further refined by other pattern specifications. In Figure 13, the pattern F_5 refines D_3 and it is itself refined by even more specific pattern specifications F_3 and F_4. This leads to a chain of pattern specifications, from the most abstract one to the more specific ones and finally to the pattern instances. Refinements are made by the pattern modeler who creates the pattern interface. For the pattern user, the refined pattern interface offers more specific guidance to specialize the framework than using the higher level pattern specifications directly.

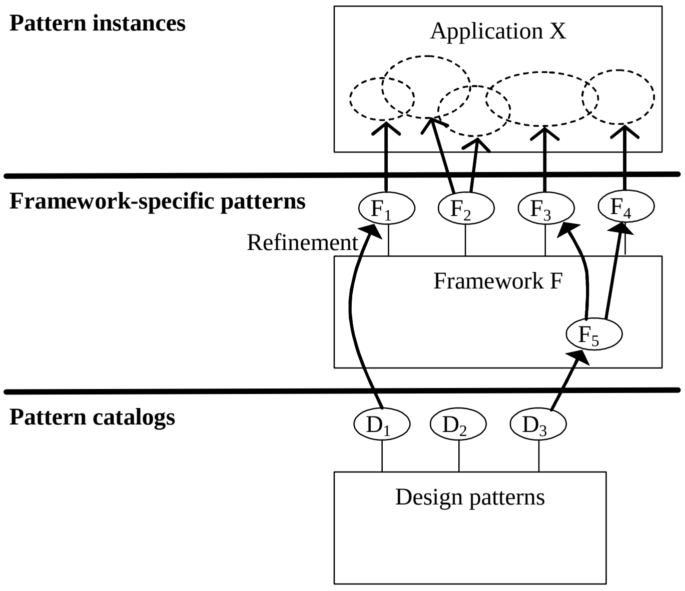


Figure 13. Pattern layers as a chain of refinements.

4.4.4 Pattern Composition

A solution of a specific problem may utilize a number of patterns; it may be beneficial to compose a new pattern that describes the whole solution rather than using the original patterns separately. For instance, Zimmer [1995] has noticed the following relationships between design patterns: "X uses Y in its solution", "X is similar to Y", and "X can be combined with Y". According to Zimmer, the latter one creates larger building blocks in design and raises the abstraction level. Riehle [1997], in turn, suggests that a pattern is a composite pattern, if it can be best explained as the composition of further atomic or composite patterns. Here an atomic pattern is a pattern that cannot be described as the composition of further patterns. Riehle continues that a composite pattern's synergy emerges from the interaction of different elements from different patterns.

Figure 14 illustrates the above described *static pattern composition* from the viewpoint of the pattern tool platform. In the figure, pattern F_4 is refined from the design patterns D_2 and D_3 so that the overlapping roles of D_2 and D_3 are unified. Pattern F_1, in turn, is refined from D_1 and D_2.

Hammouda [2005] discusses the possibility to merge tool-supported patterns on the fly in a single pattern. He also outlines a mechanism for *dynamic pattern composition* and shows how the composition can be applied using the role-based pattern concept of the tool. Thus, instead of being known in advance, the references to other patterns define a generic pattern type. As said by Hakala [2002], to enable this kind of dynamic pattern composition, one approach is that the pattern modeler leaves some parts of the composing pattern specification open by introducing a placeholder role so that the pattern user herself can select different pattern specifications to augment it at a later time. For example, in Figure 14, pattern F_2 may have been defined so that at some point during the pattern instantiation, the pattern user can select one of the available design pattern specifications, supposing that they can be plugged into the dynamic placeholder role.

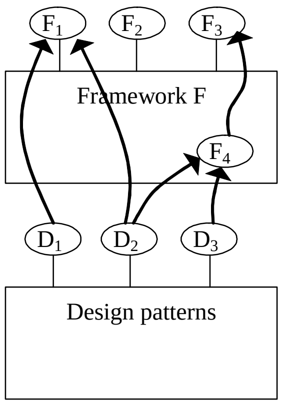


Figure 14. Pattern composition with overlapping refinements.

4.5 Pattern Tools

A tool platform with different pattern tools systematizes the use of the pattern concept. For the person who creates the pattern specifications, the writing and testing process resembles programming, making the pattern development more like the software development process itself. For the pattern user who applies patterns, the complexity of the underlying formalism and instantiation algorithms is hidden so that a pattern specification can be easily used and instantiated. At least the following tools are possible:

- *Development tools*. Development tools are used to create pattern specifications. Subsection 4.5.1.
- *Deployment tools*. Deployment tools are used to instantiate pattern specifications. Subsection 4.5.2.
- *Analysis tools*. Analysis tools are used to pattern mining and to analyze pattern specifications and their use. Subsection 4.5.3.

- *Documentation tools*. Documentation tools are used to generate documentation and pattern-based tutorials. Subsection 4.5.4.

4.5.1 Development Tools

Development tools are needed to construct and edit pattern specifications. They are like programming tools or editors to create and edit pattern specifications in terms of the used formalism or specification language. In addition, a sophisticated development tool could assist the pattern modeler by verifying the syntax and semantics of the created pattern specification. This resembles type checking or debugging in more traditional programming environments.

For example, pre-defined role components (Subsection 4.3.1) make it possible to create a *visual builder* [Roberts and Johnson 1996; Fayad et al. 1999] to compose new pattern specifications. In such a builder, the values of the role-specific properties could be edited and the dependencies could be drawn between the roles. In the case of framework-specific patterns, the initial bindings could also be made between the roles and the framework elements, for example, if a role represents a particular abstract base class.

With pre-defined role types the development tool can let the pattern modeler to think the pattern writing process in terms of simple role components and their interactions, instead of showing the underlying formalism. Only the platform developer must know the internals of the pattern specifications, in order to create new pattern semantics (Subsection 4.3.2). A concrete example of a semi-graphical pattern development tool is discussed in Subsection 5.2.3.

4.5.2 Deployment Tools

Pattern deployment tools are used to instantiate pattern specifications. Deployment tools are like small semi-automatic programming systems to manage the instantiation process. Based on the selected pattern specification, a deployment tool utilizes the general services of the pattern tool platform and shows the task list (Subsection 4.3.6) to ask the required input (e.g., the participating code elements) from the pattern user.

According to Florijn et al. [1997], a deployment tool can assist the software developer to apply patterns in the following ways:

- The tool generates program elements, like classes, to be part of a pattern instance (the top-down approach).

- The tool binds existing program elements to be part of a pattern instance (the bottom-up approach).
- The tool checks if pattern instances obey the original pattern definitions and tries to repair the program in case problems arise.

Deployment tools can be categorized as shown in Figure 15. The *pattern instantiation* category describes how well the deployment tool supports the pattern instantiation process. The *pattern verification* category describes the capability of the deployment tool to check pattern instances during the software development process. The *pattern selection* category describes how the pattern user selects suitable patterns. Thus, the ultimate deployment tool could select and instantiate right patterns automatically, without human interactions. In addition, it would check the pattern instances automatically and repair violations when the software system evolves.

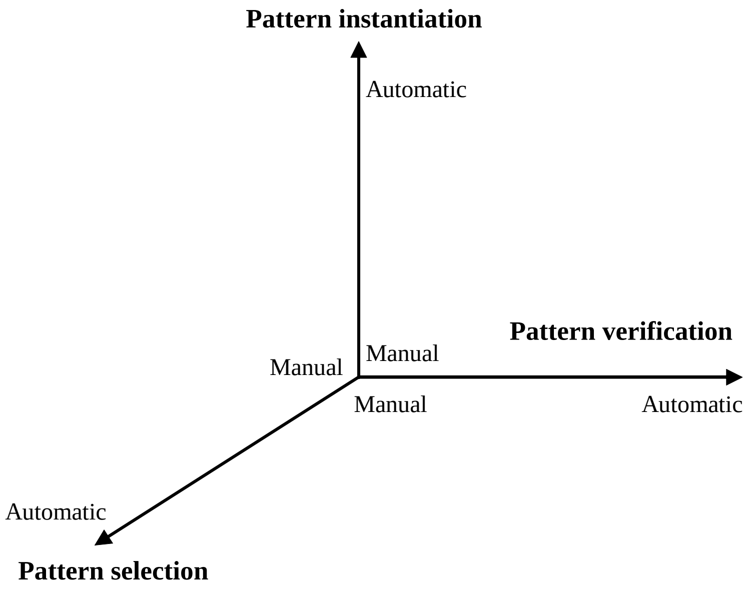


Figure 15. Categorization of pattern deployment tools.

For instance, the current deployment tool used in JavaFrames Eclipse Integration (this will be discussed in Subsection 5.2.2) supports automatic pattern instantiation and verification. However, the tool is not fully automatic. Patterns are selected manually and the pattern user still has to make some decisions during the pattern instantiation. Also the verification of the pattern instances works within the limits of the used pattern semantics. Despite of these limitations, even if not providing full automation in code generation and pattern instantiation, such a task-driven deployment tool can be useful when instantiating patterns in domains where full automation is impossible or inefficient, such as in specialization of a white-box framework.

Many of the deployment tools discussed in literature (Section 9.1) act as simple code generators and just generate a skeletal implementation to instantiate the selected pattern. With such a tool, the pattern selection is manual, the deployment tool provides a block of source code to implement the pattern instantiation, and the pattern instance is not verified after instantiation. Cornils [2001] has noticed that the focus of the pattern deployment tools is usually on code reuse and rule checking. Tools that support pattern-based code reuse contain pattern implementations which are then glued into the application's source code. Then, if the deployment tool has some rule checking capabilities, it can help the pattern user to instantiate the patterns correctly by checking these rules, making pattern verification more automatic.

4.5.3 Analysis Tools

Analysis tools are used to pattern mining (Subsection 2.2.3) and to measure different pattern specifications and their use. Such an analysis tool could explore large amounts of code in short time to visualize the applied patterns and to improve the system by suggesting desirable patterns. Also, it could be interesting to analyze different pattern specifications and their instances to compare their usability and quality factors, or to estimate the required instantiation and maintenance efforts.

For instance, Viljamaa J. [1997] suggests a pattern-based reverse engineering tool that could be used to gather statistics about the use of patterns and to estimate the required amount of work to instantiate patterns. In addition, such a tool could analyze the quality of object-oriented systems based on loose coupling, shallow inheritance hierarchies, high use of composition, and other similar positive measures of object-oriented software. However, Atkinson and Griswold [1996] warn that for an analysis tool both the running time and the required memory space may be too expensive, particularly in an interactive (and real time) context.

Examples of analysis and pattern mining tools are given in Subsection 5.2.5. and in Section 9.1.

4.5.4 Documentation Tools

Pattern specifications and their instances constitute a structure that can be utilized to automatically generate different pattern-based documentation. Such documentation can describe the crosscutting design fragments of the software system and help the reader to understand the system. For instance, a pattern-based documentation tool could generate hypertext [Conklin 1987] that emphasizes various aspects of the instantiated pattern specifications and the software architecture. With hypertext the reader could follow links to related patterns, read examples of pattern usage without technical implementation details, and study the most interesting parts of the pattern instances. A pattern specification itself could also be presented more informally, for example, by generating a hypertext document that describes the roles and properties of the pattern. Such documentation resembles more like the traditional pattern documentation discussed in Chapter 2.

Also Odenthal and Quibeldey-Cirkel [1997] have noticed that if patterns have been used during the software development process, their descriptive nature helps to document the implemented system as the pattern instances lay the foundation for documenting the design. Particularly the pattern based documentation explains the rationale of design decisions: why, in which context, and how a pattern has been instantiated. They stress that documents are living products that should be allowed to evolve together with the iterative and incremental design cycle of a software product. If patterns are used during the design, it becomes documented while it is still evolving and the design knowledge is at its highest. However, as discussed in Subsection 2.2.2, this kind of documentation is difficult without a sophisticated documentation tool.

Besides technical documentation, also concrete examples and tutorials are important. The problem is that creating and maintaining a compact and comprehensive set of useful examples is time-consuming and difficult. An interesting possibility is to create examples semi-automatically by recording pattern instantiations [Hakala et al. 2003]. That is, if a pattern specification describes an example solution for a particular problem, the pattern could be instantiated and the instantiation process could be recorded. Such recordings are interactive and easy to follow tutorials about how to solve the addressed problem. Different pattern instantiations can be documented as use cases or mini-tutorials for learning purposes. Pattern users could then browse through these recordings, step by step, like an interactive tutorial to learn the solution. The recording tool has been implemented in the JavaFrames environment and it is discussed in Subsection 5.2.4.

The learning of a software system is a persistent process where humans proceed iteratively and with intuition to find answers to their questions about the system. Ruhe [2000] has enumerated typical training problems and noticed that because the learning needs of the software developers are changing continuously it is important that one can add new topics (pattern-based documentation and tutorials) to the used learning environment. Ruhe says that the learning environment should also help the software developers to know the people within the company who are working on similar problems and they should be able to communicate about the training material with each other. For the users of the learning environment, it should be able to easily add new insights and experience.

The general tool platform for patterns and the integrated documentation tools could be one approach to create an efficient and easy-to-use learning environment. For instance, Hammouda et al. [2004b] argue that learning complex software libraries can be supported with a pattern-driven learning environment. In the environment, pattern specifications could be used to describe various learning concerns and the order in which those concerns should be introduced. The environment could then support the learning process gradually by generating meaningful learning tasks.

4.6 Integration

According to Harrison W. et al. [2000] *programming support environments* (software development environments) are collections of tools that support coding activities. They are powerful environments used by majority of software developers today. However, their major limitation is that they support only rather low-level programming and compiling issues, but ignore other major activities, like requirements engineering, specification, design, testing, and analysis. The identification of the need for integrated support for these other activities has motivated innovations toward *software engineering environments* (architecture-oriented software development environments). These environments support software engineering across the software lifecycle and traceability across the artifacts of the software product and design. Integrating the general pattern-based tool support into a real software development environment can be seen as one step towards such software engineering environments.

The integration and its requirements are further discussed in the following subsections:

- *Requirements for the environment*. To integrate the pattern-based tool support into a software development environment has some requirements. Subsection 4.6.1.

- *Outlining the integration*. There must be a clear separation between the third party development environment and the pattern tool platform. Subsection 4.6.2.

- *Real time vs. turn-based pattern support*. Ideally, the consistency of the pattern specifications and their instances should be verified simultaneously with the normal software development and programming activities. Subsection 4.6.3.

4.6.1 Requirements for the Environment

To integrate the pattern-based tool support, the software development environment has to provide the following facilities:

- *Integration API*. There must be an application programming interface to create and integrate new tools and to utilize the common services of the environment.

- *Parse information*. To create role bindings and to check pattern instances it is necessary to locate the participating elements. This requires access to the parse information maintained by the used software development environment. Further, it may be necessary to have access to the internals of the element, like the actual source code of a Java method or a field. For Java programs, the environment obtains parse information from the source code files of the project and from the compiled files outside the project. Because the used parser is usually more forgiving than the compiler, some information may be incomplete or inaccurate. For instance, when writing a method in the source code editor, its parameter types or return type may not be fully qualified. Thus, the reflected classes need not to be compiled or in a compilable state.

- *Code generation*. To create and edit software elements during pattern instantiation, the pattern tools may utilize the code generation facilities of the underlying software development environment. This helps the pattern user to perform tasks and makes it possible to partially automate the pattern instantiation. For example, in the case of source code files that are maintained by the software development environment, the pattern deployment tool could create new files and insert new code lines in a specific location.

- *Event notifications*. To validate pattern specifications and their instances and to update the task list, the pattern engine must be notified if elements are created, deleted, or modified. In addition, the platform needs notifications about opening and closing projects to restore and save the state of the pattern specifications and their instances.

4.6.2 Outlining the Integration

There must be a clear separation between the third party software development environment and the pattern tool platform. This is necessary because each environment tends to have its own internal parse information (for example, the Java syntax tree representation), event mechanism, extension interface, and so on. The separation ensures that the platform can be integrated into different environments. Figure 16 gives a general idea of how the pattern tool platform might be integrated and used with a third party software development environment.

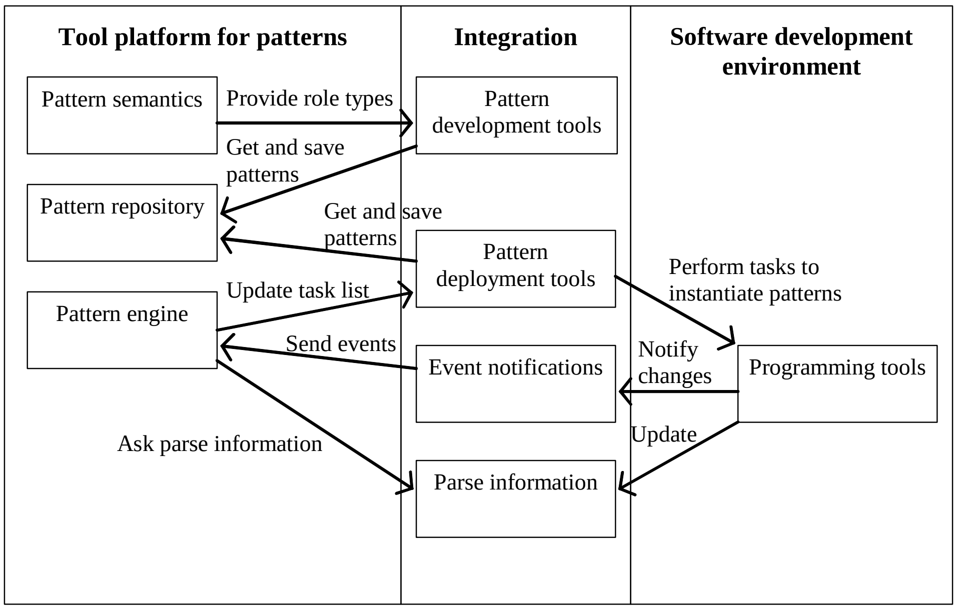


Figure 16. Integration of the pattern tool platform.

To make it possible to create and use the pattern specifications, the integration must provide at least the basic pattern development and deployment tools. As a pattern development tool, the pattern modeler needs some kind of editor or programming tool to create new pattern specifications. Basically, the pattern development tool can be an independent editor that hides the complexity of the underlying pattern formalism.

The pattern user needs pattern deployment tools to select and instantiate pattern specifications. A deployment tools must be tightly integrated into the development environment. It must be aware of the current project(s) and show the associated pattern specifications and their instances. It also shows the task list to guide the user to instantiate patterns. Performing a task typically opens a tool or a wizard to fulfil the underlying role, unless the task can be performed automatically. Such a task-specific tool can be the default source editor to write the required code, or it can be more advanced tool, like a graphical editor to draw UML diagrams.

The current state of the pattern specifications and their instances is stored for each project. The information can then be restored whenever the project is opened in the development environment. The pattern repository and the stored project-specific information should be independent from the used development environment. Information could be stored, for example, as XML (eXtensible Markup Language) [XML 2004] files to the pattern repository.

4.6.3 Real Time vs. Turn-Based Pattern Support

Ideally, the software development environment uses incremental parsing techniques to constantly maintain the syntax tree of the source code. If the parse information is continuously updated and the pattern engine is instantly notified for the changes, whenever the software developer edits the source code, the task automaton can generate tasks simultaneously. However, it may be necessary to provide also more limited versions of the task automaton. As discussed by Egyed and Balzer [2004], the problem is that the environment may lack the ability to interact with external integrations. Also, interactions may be too expensive, making it impractical to update the task list continuously.

Thus, there are two different approaches to implement the task automaton:

- *Real time pattern support*. A real time pattern support means that pattern specifications and their instances are instantly and continuously evaluated and verified. If the software development environment provides suitable parse information and efficient event notifications the pattern engine is able to generate tasks simultaneously with other software development activities.

- *Turn-based pattern support*. A turn-based pattern support means that the pattern user starts the pattern engine only when required to verify the pattern specifications and their instances. Thus, the pattern engine is invoked like a compiler that verifies the selected pattern specifications against the current situation and updates the task list.

JavaFrames Eclipse Integration, a concrete example of a real time pattern support, is discussed in Chapter 5.

Chapter 5

JAVAFRAMES ECLIPSE INTEGRATION

JavaFrames Eclipse Integration [JavaFrames 2004] is a prototype of a task-driven architecture-oriented environment that utilizes the pattern concept to model and implement design solutions. JavaFrames (formerly known as Fred) provides the core of the system, including the pattern engine and task automaton discussed in Chapter 4. Currently, Java [Viljamaa A. 2001, 2004; Viljamaa J. 2002, 2004] and UML [Hammouda et al. 2004c] pattern semantics have been built on top of the JavaFrames system.

First versions of JavaFrames were released during the FRED project (FRamework Editor for Java – Support for Framework Development and Use) 1997-1999. Since then, the methodology has been further evolved in different projects [Hakala et al. 2001b, 2001c, 2001d]. One of the constituting ideas of these projects was to develop a tool that instructs the application developer to specialize object-oriented frameworks. In JavaFrames, based on the given pattern specifications, the application developer is guided with small and context-sensitive programming tasks to gradually specialize the framework. JavaFrames keeps track of the progress of the tasks, verifying that the requirements of the framework specialization are followed as required in the patterns. In that way, JavaFrames advises the framework user to create an application that obeys the rules of the framework.

The author has participated in the development of JavaFrames, particularly in the development of the user interface framework to implement various pattern tools. The author has also been responsible to integrate the JavaFrames system into the Eclipse environment [Eclipse 2004]. The integration utilizes the services of JavaFrames and Eclipse, providing the basic pattern tools to create and instantiate pattern specifications. However, the idea of general and extensible tool support for patterns is not restricted to any particular programming language or environment. Other team members have implemented the core of the pattern system and the task automaton used in JavaFrames.

An overview of the integration is given in Section 5.1. The implemented pattern tools are discussed in Section 5.2.

5.1 The Integration

A brief introduction to Eclipse, JavaFrames, and their integration is given in the following subsections:

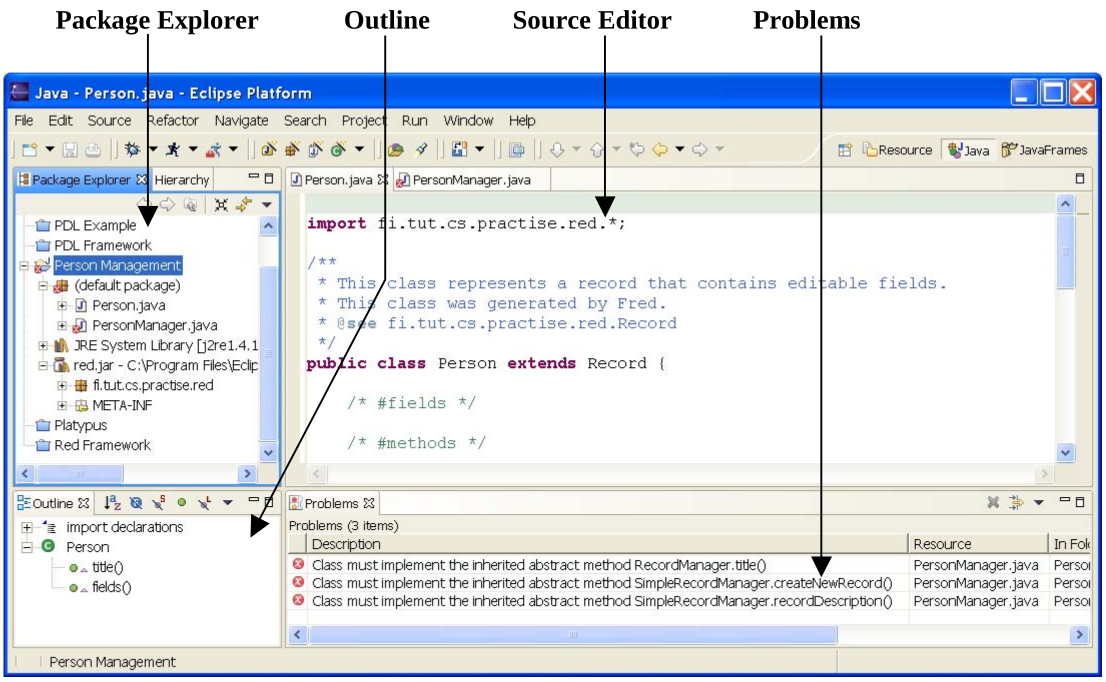
- *Eclipse*. Eclipse is designed for building integrated development environments. Subsection 5.1.1.
- *Integrating JavaFrames*. JavaFrames Eclipse Integration is a practical experiment to integrate the pattern tool platform into an existing software development environment. Subsection 5.1.2.

5.1.1 Eclipse

Eclipse is a platform to create software development environments. It is an IDE (Integrated Development Environment) for anything and for nothing in particular. Eclipse is extended with extensions called plugins, making it easy to integrate various software development tools. For example, to develop Java applications, JDT (Java Development Tooling) provides plugins to add the Java programming capability to the Eclipse platform.

As a Java development environment, Eclipse contains a number of tools, like tools to manage Java projects, source code editing, and compilation. Figure 17 shows a typical Eclipse Java development user interface. The Package Explorer shows the hierarchical structure of Java projects, the Outline tool shows the contents of a Java class, the Problems view shows compilation errors and warnings, and the Source Editor is a typical text editor for programming. In addition, the environment has other tools and views that are not shown in the figure.

In Eclipse, JDT uses in-memory object model to represent the structure of a Java program. This model is hierarchical meaning that the elements of a program can be decomposed into child elements. For example, the Java package "org.eclipse.jdt.core" defines interfaces like IJavaElement, IMember, IType, IMethod, and IField. For a plugin developer, these interfaces provide methods to get the current Java project, appropriate child elements, the parent element, the associated source code resources, and so on. Also JavaFrames uses these interfaces to obtain the required parse information.



```
import fi.tut.cs.practise.red.*;

/**
 * This class represents a record that contains editable fields.
 * This class was generated by Fred.
 * @see fi.tut.cs.practise.red.Record
 */
public class Person extends Record {

    /* #fields */

    /* #methods */
```

Problems (3 items)

Description	Resource	In Fol
Class must implement the inherited abstract method RecordManager.title()	PersonManager.java	Perso
Class must implement the inherited abstract method SimpleRecordManager.createNewRecord()	PersonManager.java	Perso
Class must implement the inherited abstract method SimpleRecordManager.recordDescription()	PersonManager.java	Perso

Figure 17. Eclipse as a Java development environment.

Besides parse information and event notifications, Eclipse and JDT provide facilities to create and manipulate Java code elements and the file system. For example, one of the main plugins provided by Eclipse is the Resource Plugin that provides services for accessing files, folders, and projects. The following example code illustrates how to get a project object that can be used to access the folders and files under the project, as well as to create new files and folders:

```
IWorkspaceRoot root = ResourcePlugin.getWorkspace().getRoot();
IProject project = root.getProject("MyProject");
if (project.exists() && !project.isOpen()) {
      project.open(null);
}
```

To summarize, the Eclipse Platform is very flexible and extensible system to build application development environments. It provides enough support to integrate JavaFrames as separate plugins and allows the real time pattern support discussed in Subsection 4.6.3.

5.1.2 Integrating JavaFrames

As explained in Section 4.6, the integration of JavaFrames to any environment requires that the target environment provides an integration API, enough parse information, code generation facilities, and suitable event notifications. In Subsection 5.1.1, it was demonstrated that Eclipse fulfils these requirements. Figure 18 illustrates the details of the integration. JavaFrames includes the pattern engine and a framework to create new pattern semantics. In addition, it provides interfaces and services to implement pattern tools and utilize the task automaton. JavaFrames has been registered to Eclipse as a new Eclipse plugin. This plugin provides also the Eclipse-specific implementations of the pattern tools discussed in Section 5.2.

Also different pattern semantics are registered as Eclipse plugins. These semantics extend the JavaFrames pattern system and provide role types to create different kinds of pattern specifications. The Java pattern semantics and the UML pattern semantics are the current extensions of the JavaFrames pattern system. The first provides role types to construct pattern specifications to describe Java structures, like how to specialize a Java framework. The latter provides role types that are used to represent UML structures. More detailed description about the JavaFrames extension framework is given in Chapter 7.

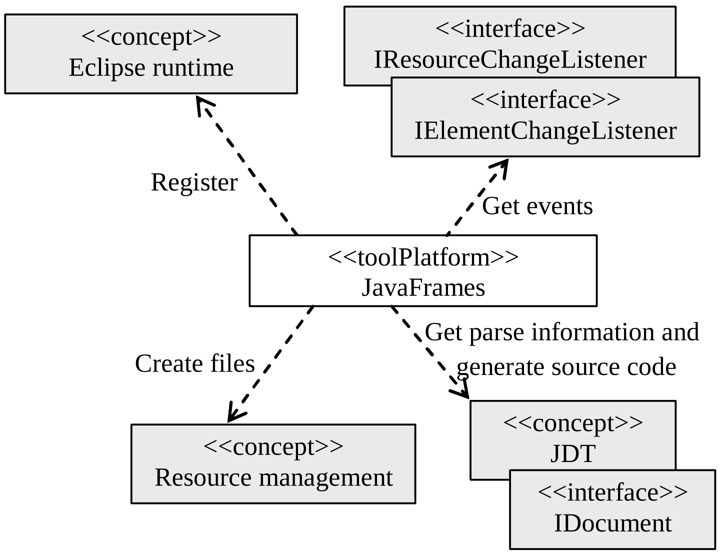
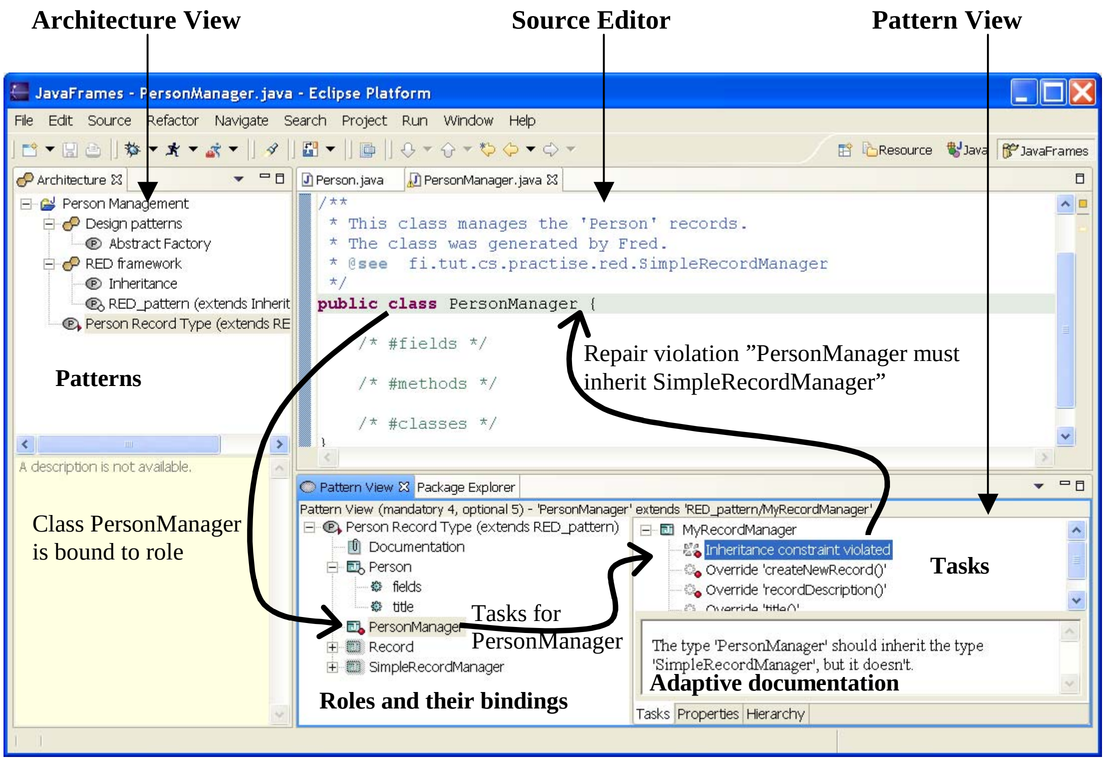


Figure 18. Integrating JavaFrames into Eclipse.

Here the Person Management project is used to illustrate the use of JavaFrames Eclipse Integration. Person Management is a simple Java application to store personal data. At some point, the application developer notices that a framework could be used to provide the user interface of this application. She imports the pattern interface of the RED framework [Hakala et al. 2001a; Pree and Koskimies 1999], which is a small Java framework that is used as a tutorial in the JavaFrames documentation. This particular framework is specialized by deriving a new Record subclass and a suitable manager class; they represent and manage the objects that need the user interface facilities. Once these classes have been created, the framework can provide a dialog to manage the data objects. In the case of personal data, the dialog is used to set the person's name, age, and portrait. The content of the dialog view is constructed and maintained by the framework and depends on the attributes of the current data object. The imported framework-specific pattern interface guides the application developer to specialize the framework and to utilize its services when implementing her application.

A typical user interface of the JavaFrames Eclipse Integration is shown in Figure 19. In the figure, the application developer has opened the JavaFrames perspective for the Person Management project. Pattern specifications of the Person Management project are shown in Architecture View (Subsection 5.2.1). The application developer has imported the pattern interface of the RED framework and instantiated RED_pattern to create a new person record type. Pattern View is the deployment tool used to instantiate the selected pattern specifications (Subsection 5.2.2). By performing tasks in the task list, the application developer instantiates these specifications and specializes the framework. Simultaneously JavaFrames notices the application developer if she violates the rules of the underlying pattern specifications. For instance, the created PersonManager class should inherit one of the abstract base classes provided by the framework. Because this constraint is violated, a repair task is generated to fix the situation.



```
/**
 * This class manages the 'Person' records.
 * The class was generated by Fred.
 * @see  fi.tut.cs.practise.red.SimpleRecordManager
 */
public class PersonManager {

    /* #fields */

    /* #methods */

    /* #classes */
}
```

A description is not available.

Pattern View (mandatory 4, optional 5) - 'PersonManager' extends 'RED_pattern/MyRecordManager'

The type 'PersonManager' should inherit the type 'SimpleRecordManager', but it doesn't.

Figure 19. The JavaFrames perspective in Eclipse.

Note that the user can customize the Eclipse user interface by opening and closing tools and dragging them around. Using the integrated pattern tools together with other Java development tools (see Figure 17) promotes an architecture-sensitive typing system in which violations against the architectural rules are instantly notified to the software developer.

5.2 JavaFrames Pattern Tools

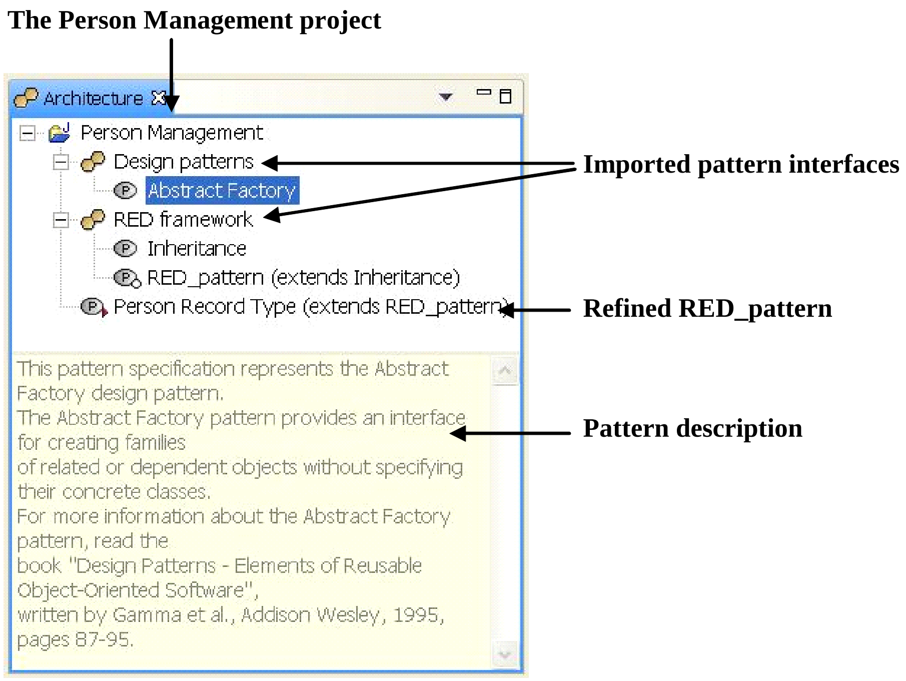
The JavaFrames Eclipse Integration includes the essential pattern tools to create and manage pattern specifications. As a pattern tool platform, JavaFrames can also be extended with new pattern tools. The current tools are discussed in the following subsections:

- *Architecture View*. The Architecture View is a pattern deployment tool to import and manage pattern interfaces. Subsection 5.2.1.
- *Pattern View: instantiating patterns*. The Pattern View is a pattern deployment tool to instantiate single pattern specifications. Subsection 5.2.2.
- *Pattern View: developing patterns*. The Pattern View is also a pattern development tool to create and edit pattern specifications. Subsection 5.2.3.
- *Pattern Recorder*. The Pattern Recorder is a documentation tool to create pattern-based tutorials. Subsection 5.2.4.
- *Other tools*. As examples of other possible tools, JavaFrames includes various tools for document generation, pattern mining, and UML modeling. Subsection 5.2.5.

5.2.1 Architecture View

Each project has its own pattern interface (Subsection 4.4.2) containing pattern specifications and imported pattern interfaces. The Architecture View, shown in Figure 20, manages this project-specific pattern interface. The upper pane shows the hierarchical structure of the pattern interface. This tree-like structure has the project as a root node. If a new project is created or opened a corresponding pattern interface is shown in the tree. If a project is closed its pattern interface is stored and removed from the tree. The lower pane shows the documentation associated with the selected pattern specification. In the Architecture View, the pattern user can import existing pattern interfaces, refine patterns, and open the Pattern View (Subsection 5.2.2) to instantiate the selected pattern specification. The pattern modeler uses the Architecture View to manage the overall structure of her pattern interface.

In Figure 20, the pattern user has imported the pattern interface of the RED framework. In addition, she has imported the pattern interface that contains specifications for design patterns (currently, it contains only the Abstract Factory pattern specification). She has instantiated the RED_pattern by refining and instantiating it as a new Person Record Type pattern specification. With a small arrowhead symbol the tool indicates that the Person Record Type pattern has some mandatory tasks and the pattern user should open the Pattern View tool to perform these tasks. The reason why the original RED_pattern is not directly instantiated is that now the pattern user can apply it to produce multiple slightly different instantiations. In JavaFrames this is handled by refining the pattern specification and then instantiating the refinement.

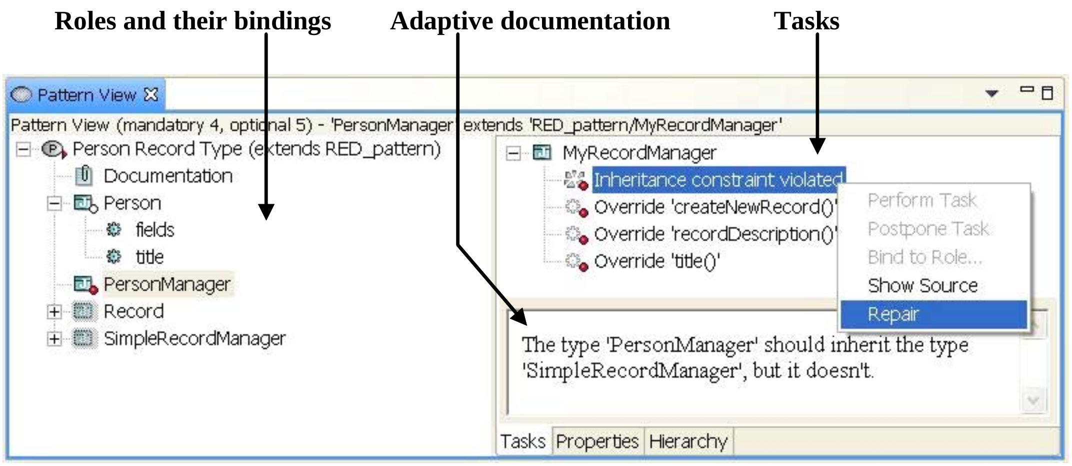

The Person Management project
Architecture
Person Management
Design patterns
Abstract Factory
RED framework
Inheritance
RED_pattern (extends Inheritance)
Person Record Type (extends RED_pattern)
Imported pattern interfaces
Refined RED_pattern
Pattern description

This pattern specification represents the Abstract Factory design pattern.
The Abstract Factory pattern provides an interface for creating families
of related or dependent objects without specifying their concrete classes.
For more information about the Abstract Factory pattern, read the
book "Design Patterns - Elements of Reusable Object-Oriented Software",
written by Gamma et al., Addison Wesley, 1995, pages 87-95.

Figure 20. Architecture View.

5.2.2 Pattern View: Instantiating Patterns

The Pattern View has two different purposes. Firstly, it is used to perform the role-based tasks to instantiate the selected pattern specification. Secondly, it is used to create and edit pattern specifications. Figure 21 shows how the Pattern View is used to instantiate patterns. It represents the selected pattern specification in terms of dynamically updated task list, adaptive role-specific documentation, and program elements, like Java classes, methods, fields, and constructors that has been bound to the roles so far. Different visual indicators are used to illustrate optional and mandatory tasks, as well as to even more clarify the instantiation process. For instance, a small red (or dark) circle in the left pane indicates that there are still some tasks to do with the created PersonManager class. In the figure, the pattern user is performing a mandatory repair task to fix her PersonManager class. Other mandatory tasks in the task list show that PersonManager must also override some methods to specialize the RED framework. Note that the pattern user can let JavaFrames to perform these tasks automatically, if the system has enough information to generate the required method implementations and corrections.



Pattern View (mandatory 4, optional 5) - 'PersonManager' extends 'RED_pattern/MyRecordManager'

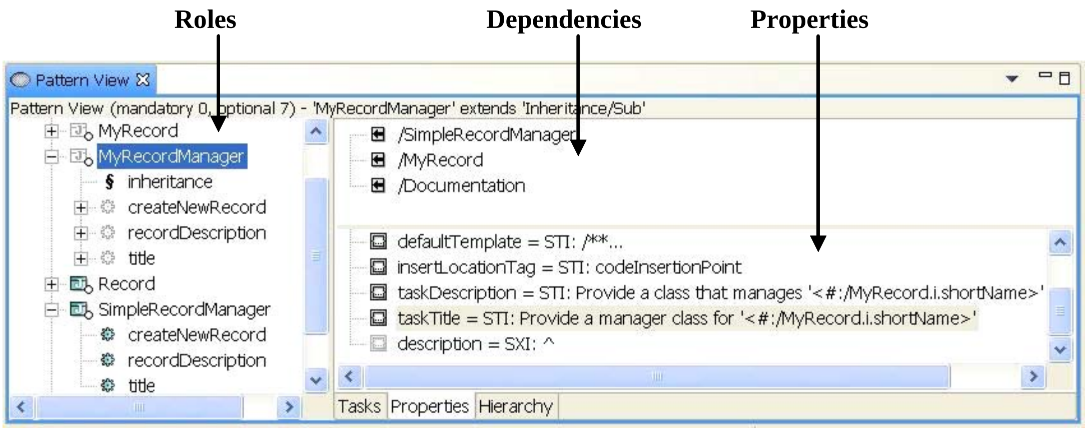
The type 'PersonManager' should inherit the type 'SimpleRecordManager', but it doesn't.

Figure 21. Pattern View: Instantiating a pattern specification.

As explained in Subsection 4.3.6, the pattern engine updates the task list repeatedly. The Eclipse Java development environment is tightly integrated with this pattern instantiation process and the consistency of the source code is verified during programming. Violated constraints and missing code elements cause new tasks until the whole pattern specification is instantiated to the application-specific context. Thus, the pattern user instantiates the pattern by doing these meaningful and relatively small programming tasks.

5.2.3 Pattern View: Developing Patterns

Concepts of a pattern specification were discussed in Section 4.3. Figure 22 shows how the Pattern View is used to develop a new pattern specification. Roles are shown as a tree-like structure on the left pane; most of the roles are not bound (light-colored symbols) because the pattern modeler wants that the pattern user will fulfil them when specializing the framework. Some of the roles, in turn, are already bound to the base classes of the framework (dark symbols). The dependencies of the selected role are listed in the upper right pane and its properties are shown in the lower right pane. A popup menu is used to create new roles, dependencies between roles, and to change the role's multiplicity settings. Creating a role under another role implicitly establishes a containment relationship between elements; for instance, a class role (MyRecordManager) that contains a method role (createNewRecord) means that the described Java class should contain the described method. This particular pattern specification is part of the RED framework's pattern interface; it describes how the application developer should specialize the framework to introduce a new record type. In the figure, the pattern modeler is just finishing the MyRecordManager role. The modeler has decided that the role depends on SimpleRecordManager, MyRecord, and Documentation roles. With the property fields, she has also created some documentation and default implementation that can be used when the role will be fulfilled.



```
Pattern View (mandatory 0, optional 7) - 'MyRecordManager' extends 'Inheritance/Sub'
+ MyRecord
- MyRecordManager
    § inheritance
  + createNewRecord
  + recordDescription
  + title
+ Record
- SimpleRecordManager
    createNewRecord
    recordDescription
    title
```

```
/SimpleRecordManager
/MyRecord
/Documentation
```

```
defaultTemplate = STI: /**...
insertLocationTag = STI: codeInsertionPoint
taskDescription = STI: Provide a class that manages '<#:/MyRecord.i.shortName>'
taskTitle = STI: Provide a manager class for '<#:/MyRecord.i.shortName>'
description = SXI: ^
```

Figure 22. Pattern View: Developing a new pattern specification.

Editing a property field sets the property values. For instance, the *taskTitle* property is a template that is used to generate a short title for the role-based binding task. Writing property templates (or scripts) is simple; besides ordinary text, a template can contain macro tags that are expanded at run time with a suitable interpreter. The expression inside these macro tags may contain references to roles and their functions. For example, the task title template "Provide a manager class for '<#:/MyRecord.i.shortName>'" evaluates a title that uses the name of the Java class that is bound to the MyRecord role. In the cases where there may be multiple elements playing the same role, the pattern engine determines the relevant one. The available functions are defined by the used pattern semantics. Each role type may have special functions, like "signature" that returns the signature of a Java method. In the example, the "i" function just returns the element that plays the MyRecord role. This element is a Java class and the "shortName" function returns its name.

A macro tag becomes expanded when the expression inside the tag is evaluated. This enables JavaFrames to provide flexible documentation and source code that adapts to the current task-specific problem of the software developer. For instance, unlike with static documentation and straightforward source code generation, roles can be utilized to produce sophisticated implementation elements during the pattern instantiation process. This can be seen as an advantage when the framework user is trying to write the required piece of source code to specialize a complex framework.

5.2.4 Pattern Recorder

It is widely accepted that framework documentation benefits from concrete specialization examples. In JavaFrames a framework is specialized by instantiating framework-specific pattern specifications. When the pattern user specializes a framework by performing pattern instantiation tasks she simultaneously creates valuable information about the use of the framework. Hence, it would be beneficial to store the actions made during a typical specialization process and let the future framework users browse this information afterwards as part of the framework documentation. Though these pattern recordings could not explicitly teach the framework's architecture, they could help to illustrate the specialization process with concrete examples.

The Pattern Recorder [Hakala et al. 2003; Savolainen 2003] is a documentation tool to record pattern instantiations. It saves the instantiation steps so that this information can be reused as a concrete specialization example that repeats the actions made by the pattern user. The gathered information allows the replay of the whole specialization process. A special browser tool is then able to show the execution of programming tasks, one by one. Associated with each task, the browser can show the generated code, and how the developer modified it to meet the needs of the particular example. The recorded pattern instantiation steps give the rationale of each action just like they provide instructions for the developer when the specialization process was being recorded. Person who is trying to learn the framework can use these example specializations as illustrative tutorials. She can browse through recorded framework specializations step by step, and see how different problems are solved in practice.

Figure 23 illustrates the principle of Pattern Recorder. When the pattern user has performed a suitable set of tasks (this is decided by the person who makes the recording), those tasks can be recorded as a step, which is a kind of snapshot of the system at the current moment. The amount of information that needs to be recorded is relatively small. It is not necessary to store the complete snapshots of the system after each step; instead, it will suffice to store only the hand-made changes and the bindings that result from the pattern instantiation. Steps are recorded in order and each recorded step knows the bindings, associated code elements, and source code modifications made during the step. It can also be augmented with documentation that explains the step.

The use of the Pattern Recorder is not limited to framework-specific pattern specifications. In Figure 24, the use of the Abstract Factory pattern specification (Subsection 4.3.4) has been recorded as a small tutorial and browsed with a dedicated browser tool. The recorded steps constitute a linear structure. The name of the current step is shown in the browser's title bar. There are also buttons to browse the recording forward and backward. On the left pane there is the hierarchical structure of involved code elements belonging to the currently shown step. Small black arrow indicates that the element is new and it has been provided during the step. On the right pane are the associated documentation and source code. The source code view highlights the piece of code that was generated or modified when the current step was recorded. With each step, the documentation grows incrementally, allowing the reader to inspect all the previous steps. Similarly, the reader is able to browse all the source code that has been produced at that point of pattern instantiation.

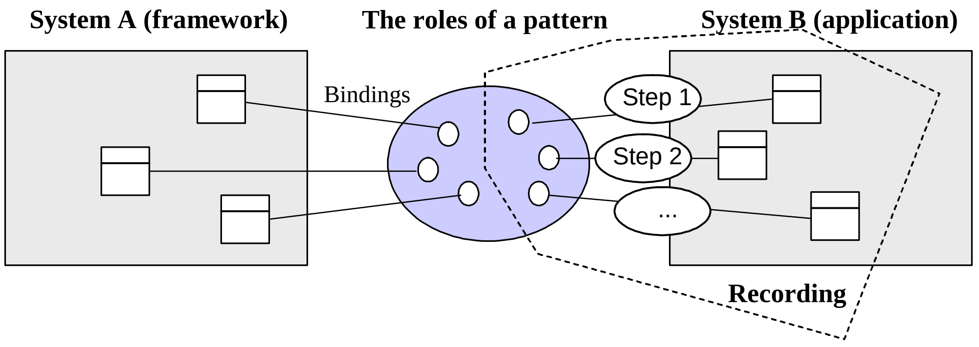
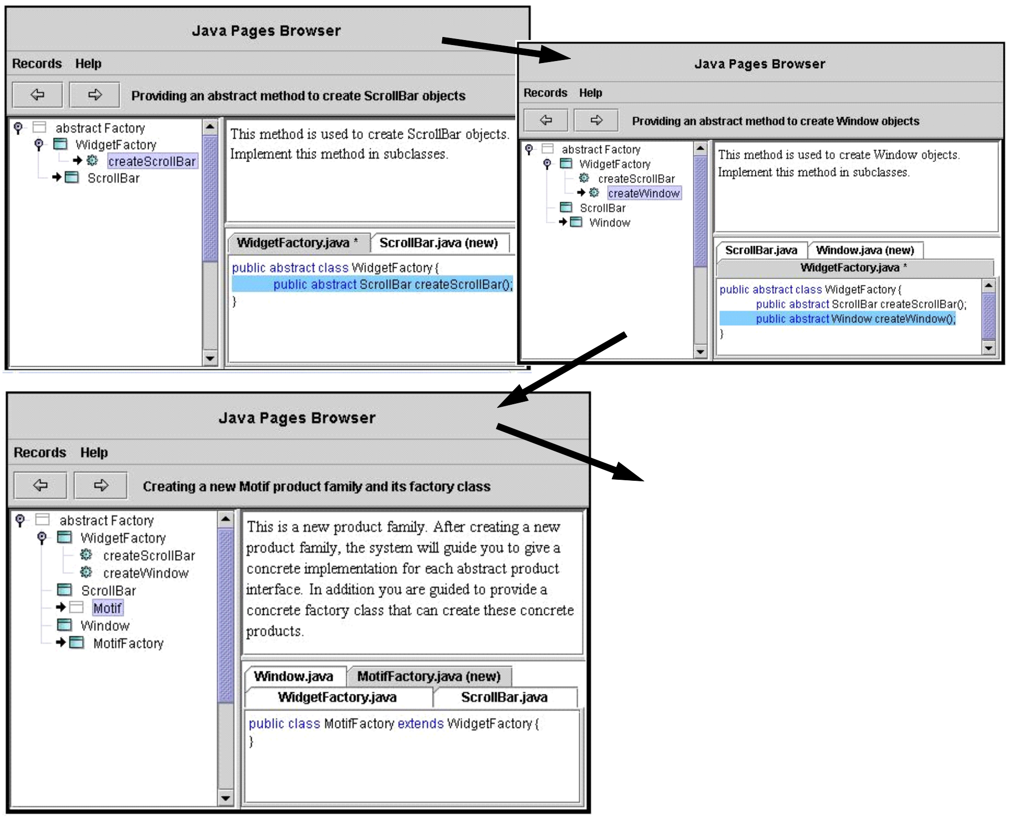


Figure 23. Recording the use of a pattern specification.



Java Pages Browser

Records Help

Providing an abstract method to create ScrollBar objects

- abstract Factory
  - WidgetFactory
    - createScrollBar
  - ScrollBar

This method is used to create ScrollBar objects. Implement this method in subclasses.

WidgetFactory.java * | ScrollBar.java (new)

```
public abstract class WidgetFactory {
        public abstract ScrollBar createScrollBar();
}
```

Java Pages Browser

Records Help

Providing an abstract method to create Window objects

- abstract Factory
  - WidgetFactory
    - createScrollBar
    - createWindow
  - ScrollBar
  - Window

This method is used to create Window objects. Implement this method in subclasses.

ScrollBar.java | Window.java (new) | WidgetFactory.java *

```
public abstract class WidgetFactory {
        public abstract ScrollBar createScrollBar();
        public abstract Window createWindow();
}
```

Java Pages Browser

Records Help

Creating a new Motif product family and its factory class

- abstract Factory
  - WidgetFactory
    - createScrollBar
    - createWindow
  - ScrollBar
  - Motif
  - Window
  - MotifFactory

This is a new product family. After creating a new product family, the system will guide you to give a concrete implementation for each abstract product interface. In addition you are guided to provide a concrete factory class that can create these concrete products.

Window.java | MotifFactory.java (new) | WidgetFactory.java | ScrollBar.java

```
public class MotifFactory extends WidgetFactory {
}
```

Figure 24. Browsing the recorded Abstract Factory pattern instantiation.

5.2.5 Other Tools

JavaFrames lays the foundation for creating a variety of pattern tools, like tools for pattern mining and tools to generate different pattern-based documentation, reports, and so on. For example, a simple documentation tool that converts JavaFrames pattern specifications into more readable form has been used to generate HTML documentation for the patterns. Tools to visualize pattern specifications and to draw them graphically could also be possible.

Pattern Extractor [Viljamaa J. 2002, 2003, 2004], in turn, is an example of combined analysis- and pattern development tool. It utilizes a general mathematical method called *formal concept analysis* (FCA) [Tonella and Antoniol 1999] to extract framework-specific patterns from the available source code. FCA can be used to find different groups of code elements and to analyze the static properties and relationships of those elements. The pattern modeler gives the relevant parts of the framework's source code and example applications as input and the tool generates skeletons of pattern specifications as output. As a result, the approach produces a set of pattern specifications that documents the framework's specialization interface. The pattern modeler can then complete these initial versions to get a complete pattern interface for the framework.

As a pattern tool platform, JavaFrames can be augmented with advanced pattern semantics and tools to enable design capabilities. For instance, MADE [Hammouda et al. 2004c] is an experimental platform for pattern-driven UML modeling. It is based on the JavaFrames pattern tool platform and the Rational Rose UML development environment. The communication between the JavaFrames UML pattern semantics and Rational Rose is achieved through a UML model processing platform, xUMLi [Peltonen and Selonen 2004], providing a tool independent API for accessing the UML models. With JavaFrames, MADE promotes Rational Rose as a pattern-based UML modeling system. MADE has also been used as a tool support for feature-oriented design and implementation of software systems [Hammouda et al. 2005].

The aspect-oriented programming [Kiczales et al. 1997] resembles the pattern concept in that both aspects and patterns can be used to manage and document crosscutting and fragmented design solutions. Hammouda et al. [2004a] suggest an aspectual pattern, which captures a generic aspect. They have noticed that both aspects and patterns capture a crosscutting slice of a system and weaving of an aspect corresponds to the pattern instantiation. The unified concept of an aspect and a pattern could be used to provide tool support, like JavaFrames, to weave the aspects interactively and incrementally, which makes the weaving process open and allows customizable weaving. In addition, with the pattern tool, the used aspects can be viewed and traced, so that if the system is later changed, the tool can show possible violations. Hammouda et al. have built a prototype tool environment which supports aspectual patterns in UML modeling, using the JavaFrames pattern tool platform and MADE as the core of the environment.

CHAPTER 6

A GOAL-ORIENTED APPROACH TO SPECIALIZE FRAMEWORKS

When solving a problem, working with small tasks to achieve meaningful goals seems to be the strategy that people adapt spontaneously. For example, when starting to specialize a framework, the application developer usually has some objectives in her mind or at least a hint of the desired outcome. When solving these application-specific problems, the developer struggles with the framework's architecture to implement the required configurations and code elements. This is often difficult, as specializing a framework requires that the specializer knows also the rules and implementation details that cannot be determined directly from the framework's interfaces and class hierarchy. Often, the framework's application programming interface (API), informal documentation, and the use case examples do not describe the required steps to achieve a specific goal precisely, but have more general or technical character.

This chapter presents a goal-oriented approach [Hautamäki 2002; Hakala et al. 2001c] that can be used to instruct the framework user on how to specialize a framework. The approach is based on the analysis of the expected behavior of the framework user. It assumes that the user tries to reuse a framework or some other reusable design by setting meaningful goals in the context of her application. The goals are then achieved by doing a sequence of programming tasks. Thus, to enable pattern-based tool support, the pattern modeler must find the goals pursued by the framework user. Then she must write a set of patterns to document how to achieve these identified goals. Finally, the obtained patterns can be transformed into more precise pattern specifications that constitute a tool supported pattern interface to achieve the goals. As a result, the framework user can use both the original patterns as informal documentation and the tool supported specifications to specialize the framework.

The goal-oriented approach is further explained in Section 6.1. To demonstrate the approach, a small example is given in Section 6.2. The goal-oriented approach is used in Chapter 7 to define a pattern interface for the JavaFrames extension framework and in Chapter 8 to obtain a pattern interface for an industrial case framework.

6.1 Goal-Oriented Approach

In this dissertation, the goal-oriented process is an approach to find, specify, and use the specialization interface of a framework as a set of framework-specific patterns. The process is shown as an UML activity diagram in Figure 25.

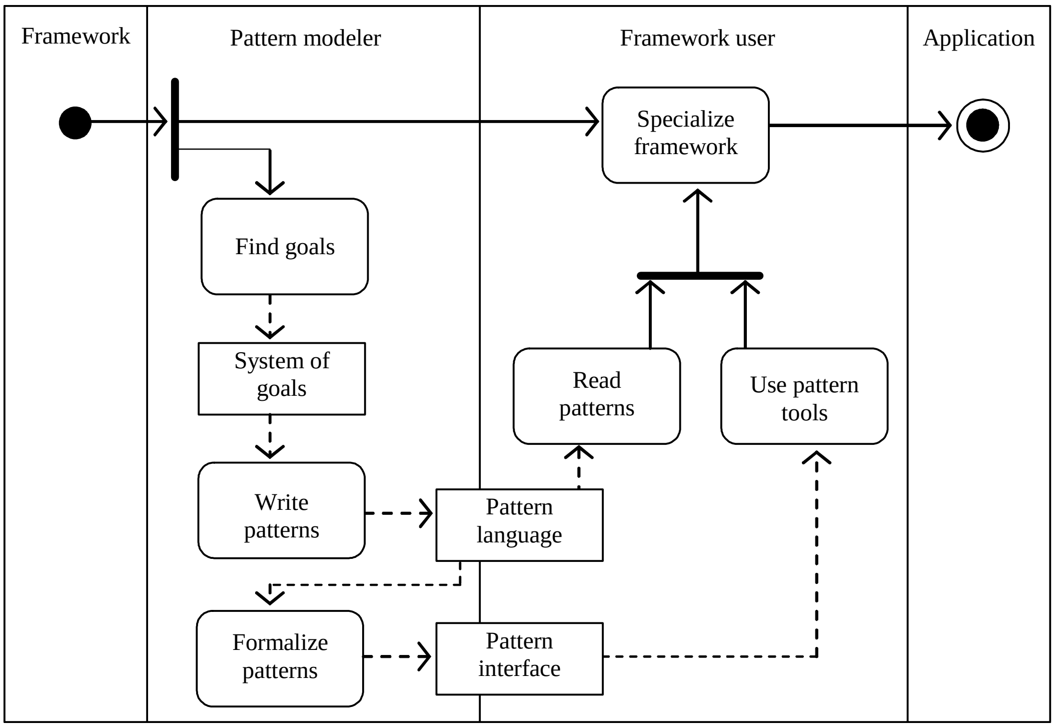


Figure 25. The goal-oriented process to create and use patterns to specialize a framework.

The process is discussed in the following subsections:

- *Find goals.* The goal-oriented approach assumes that the framework user is directed by a structure of goals. Subsection 6.1.1.

- *Write patterns*. If the pattern modeler can find the goals she can write patterns to document how to achieve these goals. Subsection 6.1.2.
- *Tool-supported pattern interface*. Formalizing the obtained goal-oriented patterns creates a tool supported pattern interface. With a pattern deployment tool, this pattern interface can then be used to specialize the framework. Subsection 6.1.3.

6.1.1 Find Goals

The application developer has a motivation to use the framework to implement a part of her application. The feeling of application-specific goals directs the way in which she derives new subclasses, overrides and implements operations, and deploys and configures components. If the application developer does not know if the framework supports a goal or how the goal could be achieved she tries to learn the framework. This slows down the application development and is the main reason why the framework specialization is so difficult and tedious.

To study how the framework user recognizes and reaches her objectives resembles the problem solving issues studied in cognitive psychology. As summarized by Anderson [1990], cognitive psychology attempts to understand the nature of human intelligence and how people think. When talking about problem solving, the problem space consists of physical states or knowledge states that are achievable by the problem solver (framework user). To solve a problem involves finding a sequence of operators to transform the initial state (of the application) into a goal state, in which the goal is achieved. Anderson [1990, p. 221] continues: "The basic argument (in cognitive psychology) is that human cognition is always purposeful, directed to achieving goals and to removing obstacles to those goals." He also enumerates three essential features of problem solving:

- *Goal directedness*. The behavior is organized toward a goal.
- *Subgoal decomposition*. The original goal is decomposed into subgoals.
- *Operator selection*. Operators are actions that will transform the problem state into another problem state. The solution of the overall problem is a sequence of these operators.

In the case of framework specialization, goals constitute a structure where achieving one goal leads to another; here this is called *system of goals*. As an example, Figure 26 presents some goals to specialize a framework that is used to derive MVC (Model-View-Controller) applications. The MVC paradigm was first used in Smalltalk environment, and it aims at making a standardized separation between the graphical user interface and the rest of the application [Krasner and Pope 1988]. It divides the user interface into three kinds of components: models, views, and controllers. A view manages a region of the display and keeps it consistent with the state of the model. A controller converts user actions into operations between the view and the associated model. In the figure, the MVC framework provides a skeleton to create such a system. The framework user may identify new goals or divide the goals into more specific subgoals. The order, in which the goals are pursued, is not fixed; for example, the user may want to first implement some views and after that she may create controllers and the factory class to launch her application. Operators to achieve the goals are typical programming activities, like creating a class and implementing an operation.

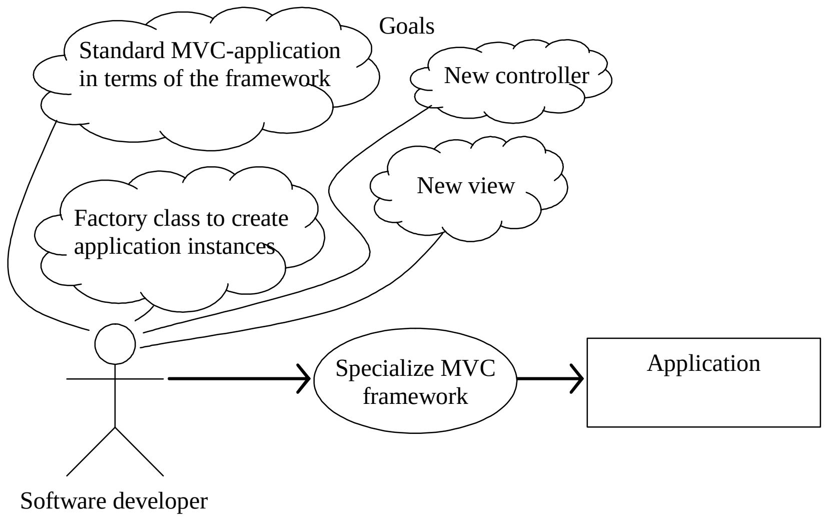

Goals
Standard MVC-application in terms of the framework
New controller
New view
Factory class to create application instances
Specialize MVC framework
Application
Software developer

Figure 26. Goals to use the MVC framework.

The system of goals is not necessarily complete; instead, new problems may arise during the specialization process, forcing the framework user to invent new ways to use the framework. In addition, it must be emphasized that the system of goals is subjective; it depends on the skills and experiences of the framework user. An experienced programmer may concentrate on finding the most efficient solution while a novice just tries to understand the problem in the first place. It is a well-known fact that the human information perception is associative and new information must be included into an existing associative net of known information [Anderson 1990]. Besides, people have only a restricted capacity to comprehend information at a time [Miller 1956]. Here the experienced framework user has, of course, advantage over the novice one.

To support the framework user, the pattern modeler has to find the goals. She must think how the application developer will use the framework. For instance, Carroll [1990] says that rather than reading a manual and trying to understand the whole, people are more interested in trying to work on real programming tasks to solve meaningful problems. It seems that when learning and using a framework, users often skip over crucial material if it does not address their current goal-oriented concern or they try to read several manuals, composing their own instructional procedure on the fly.

To find the goals, the pattern modeler must think the best and most typical ways to specialize the framework and how the framework has been used in different circumstances. She can use her past experience and available documentation. She may also study example solutions and interview framework users. The result is a document that outlines the system of goals; each goal has a short description and references to the next goals or subgoals. The purpose of the document is not to describe solutions, but to get the general idea of the required patterns.

To summarize, a goal is a situation that must be reached in order to solve the problem. A goal can be divided into subgoals and it arises from the current needs of the framework user. Together goals constitute a structure that directs the work of the framework user. Writing patterns to achieve goals is discussed in the next subsection.

6.1.2 Write Patterns

Writing patterns to describe how to achieve the most typical goals makes the goals explicit and standardizes the way the framework should be specialized. However, as discussed in Subsection 2.1.4, writing good patterns is not an easy task. The pattern modeler must be an expert with the framework and pattern writing. During the pattern mining process she must decide how the map of goals could be achieved with patterns and how these patterns constitute a goal-oriented pattern language to specialize the framework.

Goal-oriented patterns can be used to document the framework's specialization interface. The purpose of such a pattern is to help the framework user to achieve her application-specific goals by specializing suitable parts of the framework. Primarily, a goal-oriented pattern is not a new pattern type; it can be described with some of the existing documentation styles discussed in Subsection 2.1.3 and it can be categorized as idiom, design pattern, architectural pattern, and so on. But, as the goals tend to be very practical and application-specific, a goal-oriented pattern usually concentrates on practical details, like how to use a small portion of the framework's specialization interface.

One method to write the solution part of a goal-oriented pattern is to implement an example solution that reaches the goal or goals. The example solution helps the pattern writer to identify the required program elements and their interactions. This process is similar to the object-oriented analysis on the architecture level: central concepts of the example solution are identified and abstracted as the roles and constraints of the pattern. Essential roles to represent object-oriented concepts like classes and operations are easy to find. However, the solution usually includes interactions and other elements, like required method calls and other constraints, which may not be seen directly from the example solution.

As discussed in Subsection 3.2.3, using patterns to document frameworks is not a new idea. However, goal-oriented patterns are not trying to describe every detail of the framework's architecture, but how to solve small and practical application-specific problems with the framework. Audience of goal-oriented patterns are ordinary framework users rather than pattern experts or framework developers. For the application developer the goal-oriented patterns are only a practical documentation to achieve her goals; she is not changing or developing the framework itself.

Ultimately, the obtained goal-oriented patterns can be used as a blueprint when constructing more precise pattern specifications. This is discussed in the next subsection.

6.1.3 Tool-Supported Pattern Interface

Though patterns are useful as informal documentation, efficient tool support requires that they take a coherent and precise form (Section 2.3). The pattern modeler can use the goal-oriented patterns as a scheme to construct these precise pattern specifications. If documented with some of the common documentation styles, the informal pattern documentation already describes the context, the problem, and the required participants of the solution. By using this documentation as a start point, it should be possible to create the corresponding pattern specifications with pattern development tools, like the one discussed in Subsection 5.2.3. After that, various pattern deployment tools can be used to help the framework user to specialize the framework.

The pattern-based tool support for framework specialization was discussed in Subsection 3.3.2. By creating a set of tool-supported pattern specifications the pattern modeler establishes a pattern interface that describes how to use the framework. With a goal-oriented pattern interface the use of the framework becomes more systematized. For example, a novice programmer can recognize her goals faster and use the pattern specifications to start productive working, even if she is not very familiar with the framework. Expert users, in turn, can use the system to generate the desired outcome faster, as most of the implementation details are already outlined by the pattern specifications. In addition, the system can check the current state of the specialization against the used pattern specifications. This helps the application developer to see possible violations and missing elements of the specialization.

However, the process of creating good pattern specifications and a working pattern interface is very demanding. It is an iterative process in which the pattern modeler is continuously noticed that some parts of the goal-oriented pattern interface should be modified. The process to create pattern specifications resembles programming, in which the programming language is the formalism or specification language used to describe the elements and interactions of the solution.

As discussed in Section 4.3, the pattern modeler should think the design of the pattern specification in terms of roles representing the required program elements. But, as precise specifications and available role types tend to be more limited than the informal pattern documentation, the mapping between the original patterns and their formal counterparts is not always straightforward. The pattern specification is not necessarily as general and flexible as the informal pattern. Instead, the specification is targeted to a specific programming language or environment and its every implementation detail and constraint must be precise and unambiguous.

To prove the usability of the goal-oriented pattern specifications they must be tested by specializing the framework with a pattern deployment tool. The problem of changing the pattern interface later resembles the problem that occurs if the specialization interface of a framework is changed so that the existing specializations become outdated. Therefore, like frameworks, the pattern interface should be adequately tested before it is utilized in large scale.

Finally, it must be emphasized that the purpose of the pattern specifications is not to replace the more informal documentation. On the contrary, when the application developer needs to specialize a framework, she can use the original patterns as informal documentation or she can apply the pattern interface. In the latter pattern case, a deployment tool takes the pattern specifications as input and helps the framework user to specialize the framework by instantiating the patterns.

6.2 Example: Pattern Interface for a Framework

To concretize the goal-oriented approach to document and specialize frameworks, it is demonstrated with a simple example framework:

- *About the framework*. The example framework is a small toy framework to derive drawing applications. Subsection 6.2.1.
- *Find goals*. The pattern modeler must find the goals to use the framework. Subsection 6.2.2.
- *Write patterns*. The process to achieve the goals is documented with goal-oriented patterns. Subsection 6.2.3.
- *Tool-supported pattern interface*. The goal-oriented patterns are formalized to construct a tool-supported pattern interface for the framework. Subsection 6.2.4.

6.2.1 About the Framework

Figure 27 presents a toy framework that is used to derive drawing applications. The framework is written in Java and it has only three classes: MainFrame, Figure, and FigureManager. The MainFrame class implements a window for drawings; it contains a canvas and two buttons to select available figure types and to draw them to the canvas. By itself, the framework does not provide any figure types that could be drawn. Instead, each application must provide its own drawable figures by subclassing the Figure base class and by registering these new figure types to the framework.

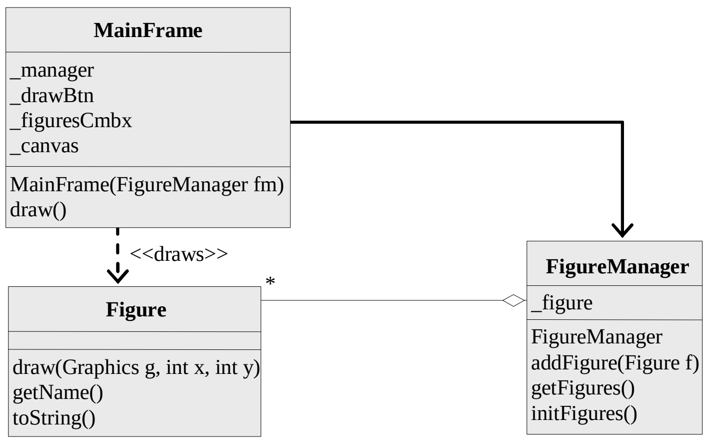


Figure 27. The example framework.

The example framework is intentionally very simple white-box framework. A real industrial framework has much more classes and possible extension points that make it flexible. The purpose of this example is to demonstrate the goal-oriented approach to construct pattern support for frameworks. More realistic and complicated use cases are given in Chapter 7 and in Chapter 8. The goals to use the example framework are discussed in the next subsection.

6.2.2 Find Goals

To find the specialization goals, the pattern modeler has created an example specialization shown in Figure 28. She has derived new figure classes MyCircle and MyRectangle. These new figure types are registered with the application-specific manager class MyManager. The Test class provides the main method to start the application. A screenshot of the final application is shown on the upper right corner; the application allows the client to draw circles and rectangles.

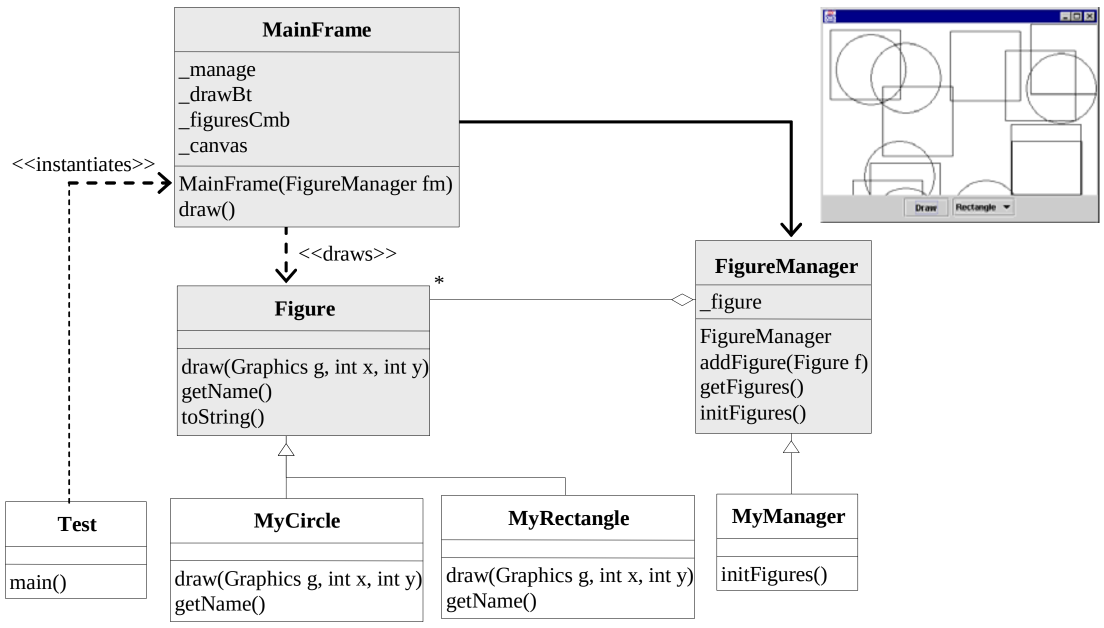


Figure 28. Specializing the example framework.

From the application developer's viewpoint, creating the application-specific manager class can be seen as one of the main goals. The second goal is to create and register new figure types. As a third goal, the framework user most probably wants to test her application to see the figures. Thus, for this particular framework, the pattern modeler presumes that the main specialization goals are the following:

- *Create a manager class*. For each application, there should be a manager class for the application-specific figures.
- *Create new figure types*. New figure types are created by subclassing the Figure base class. Each new figure type must also be registered with the application-specific manager class.
- *Test the application*. A main class is needed to configure the framework with the new application-specific manager class.

6.2.3 Write Patterns

The pattern writing process of goal-oriented patterns was discussed in Subsection 6.1.2. In this example, the pattern modeler starts to write patterns to document how to achieve the specialization goals enumerated in the previous subsection. Note that there are different ways to describe the goals with a set of patterns. Each goal may have a pattern of its own, or a group of goals may be represented by a single pattern. In this particular case, by using her experience and the example specialization, the pattern modeler has created two patterns:

- *Figure pattern*. The Figure pattern describes how the framework user can create a new figure type and how it should be registered to the framework. This pattern combines solutions to achieve the goals "create a manager class" and "create new figure types".

- *Test pattern*. The Test pattern describes how to create a runnable application to test figure drawings.

To analyze how the goals can be reached, the pattern modeler analyzes the example specialization that was shown in Figure 28 and identifies the code elements that are involved in the solution. This includes both the existing base classes (Figure and FigureManager) and the program elements that must be provided by the application developer when specializing the framework. The application-specific part must be abstracted to support a variety of possible solutions. For example, in the Figure pattern, the MyCircle and MyRectangle classes obviously should be represented by a common MyFigure role. The roles representing Java classes can be further divided into subroles representing methods and other code structures and constraints that should play a role in the solution.

After figuring out the roles of the pattern, its skeleton can be outlined like the one in Figure 29. This diagram represents the roles, constraints, and their dependencies as explained in Subsection 4.3.1. To make it more illustrative, the roles played by the framework elements are indicated as gray boxes, in contrast to the roles played by the application elements. Roles in the diagram are based on the observations made from the example specialization. If a code element in the example solution needs another element, a dependency is drawn between the corresponding roles. Similarly, if the role is representing a single program element or a set of elements, like MyFigure (the framework user must provide at least one new figure type), its multiplicity must be set accordingly. By default the multiplicity is 1. In this way, the abstract structure of the solution emerges from the given example specialization.

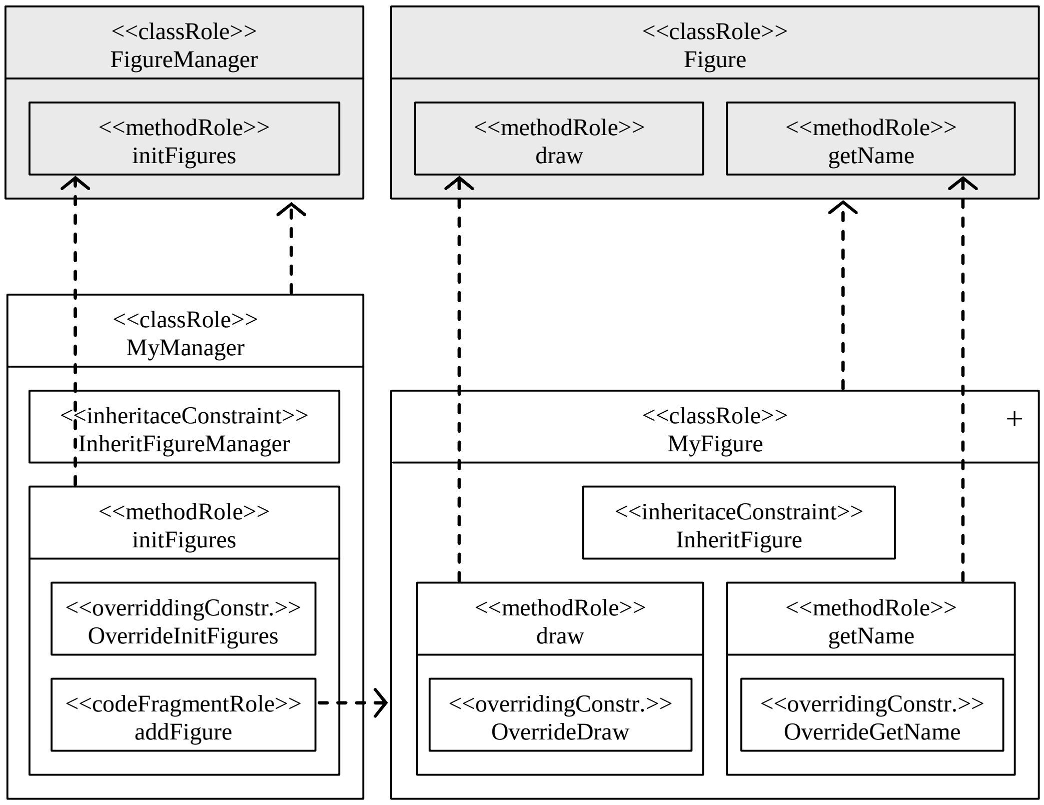


Figure 29. The Figure pattern.

6.2.4 Tool-Supported Pattern Interface

Finally, after outlining the framework-specific pattern, a corresponding pattern specification is created with the development tool provided by the pattern tool platform. In JavaFrames Eclipse Integration, roles, their properties, dependencies, and multiplicity settings are constructed as explained in Subsection 5.2.3. A detailed description of the constructed Figure pattern specification is given in Appendix B.

The created pattern interface describes the most typical use of the example framework. It emphasizes the goals and guides the framework user to create new figure types. With the JavaFrames pattern deployment tool discussed in Subsection 5.2.2, the specialization process is guided with meaningful programming tasks and much of the required code can be generated automatically. For instance, a task is generated to indicate that the framework user should register new figure type. To perform the task, the required registration code can then be generated automatically inside the initFigures method. Otherwise this small detail could be forgotten and it would cause problems, as a new figure type is not available in the user interface until it is properly registered.

The use of the obtained pattern interface is illustrated in Figure 30. The application developer has instantiated the Figure pattern twice, to provide circle and rectangle figure types. The Test pattern has been instantiated to generate the required main class to run the application. The instantiation process is based on the pattern engine and task automaton discussed in Subsection 4.3.5 and in Subsection 4.3.6.

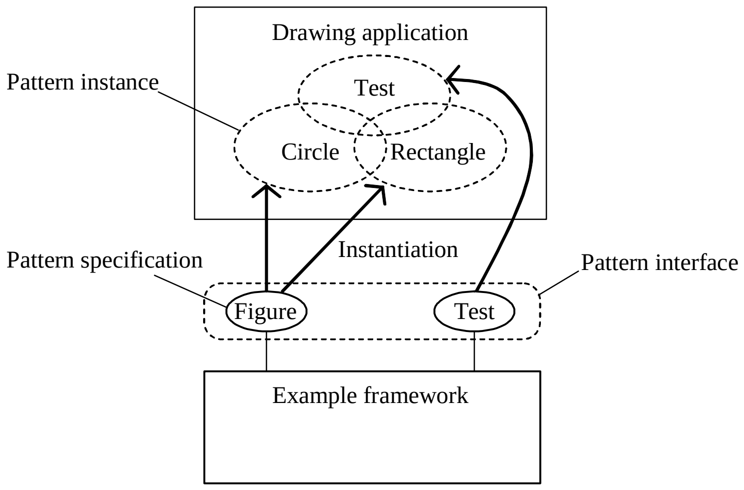


Figure 30. Using the goal-oriented pattern interface.

CHAPTER 7

SPECIALIZING PATTERN SEMANTICS

In JavaFrames the mechanism to construct pattern specifications is a special Java framework. Applications, namely the pattern specifications, are created by composing the available role types with a semi-graphical pattern development tool. Here the framework is called the Pattern Specification Framework (PSF). With PSF, it is possible to precisely represent the pattern structures and concepts discussed in Section 4.3. Though used as a black-box framework, PSF itself can be extended with new role types by deriving new subclasses from the base classes of the framework. For example, different role types can be created to represent Java classes, Java methods, inheritance constraints, UML class nodes, and so on. The pattern modeler can then use these role types when composing pattern specifications with the pattern development tool.

Currently, PSF has role types to represent Java code structures, so it is natural to think that with a suitable pattern interface JavaFrames could be used to extend PSF itself. The platform developer could then apply this pattern interface with pattern deployment tools to create new role types, making PSF to guide its own extension. Here the goal-oriented approach discussed in Chapter 6 is applied to create a pattern interface to extend the framework.

Overview of PSF is given in Section 7.1. The goals and patterns to extend the framework with new role types are discussed in Section 7.2. An example specialization to create two new role types and their user interface is done in Section 7.3. Experiences are discussed in Section 7.4.

7.1 Pattern Specification Framework

The main concepts of PSF are discussed in the following subsections:

- *Role types*. New role types are derived from PSF's base classes. The pattern modeler can then use these role types to compose pattern specifications. Subsection 7.1.1.

- *Properties*. Each role type may have properties like functions and scripts. They are evaluated during the pattern instantiation process, making it possible to provide role-specific documentation, code generation, and so on. Subsection 7.1.2.

- *User interface*. In JavaFrames, each role type defines its own user interface. A sophisticated wizard system is used when the pattern modeler creates a new role and when the pattern user performs a task that is based on that role. Subsection 7.1.3.

- *Eclipse plugin*. JavaFrames is integrated into the Eclipse environment. New extensions must be introduced to the system as Eclipse plugins. Subsection 7.1.4.

7.1.1 Role Types

PSF and one of its extensions, the role types implementing Java pattern semantics, are shown in Figure 31. Each role is associated with a semantic object that defines the behavior of the role. These semantic objects are divided into instance semantics and constraint semantics. A role that uses instance semantic can represent a concrete element while the instance semantic manages the binding between the role and the element. A role that uses constraint semantic, in turn, can perform various calculations to check that the constraint is not violated. For example, an inheritance constraint is typically associated between two class roles and it checks that there is an inheritance relation between the Java classes that are bound to those roles.

In Figure 31, roles to represent Java classes, constructors, methods, and fields are quite evident. However, because pattern specifications must be able to express also complex interactions, implementation details, and constraints precisely, some more fine-grained roles are needed. For example, the code fragment role is used to express a piece of arbitrary code, typically inside a Java method or constructor. Constraints (roles with constraint semantics), like the parameter, exception, and inheritance are needed to express context-sensitive method parameters, exception types, inheritance relations, and so on.

Figures 32 and 33 give more detailed picture of creating a new role type that utilize instance semantics or constraint semantics. In the case of instance semantics, the platform developer must implement the parseReference method to locate the actual element that is bound to the role. For constraint semantics, the platform developer must implement the checkConstraint method that validates the constraint. Each semantic class has a special inner class called Type that acts as a factory to create semantic objects during pattern instantiation. A semantic class has also a Handler class that is used by the pattern engine to generate role-specific tasks. A pattern to create new instance semantics is presented in Subsection 7.2.2. A pattern to create new constraint semantics is presented in Subsection 7.2.3.

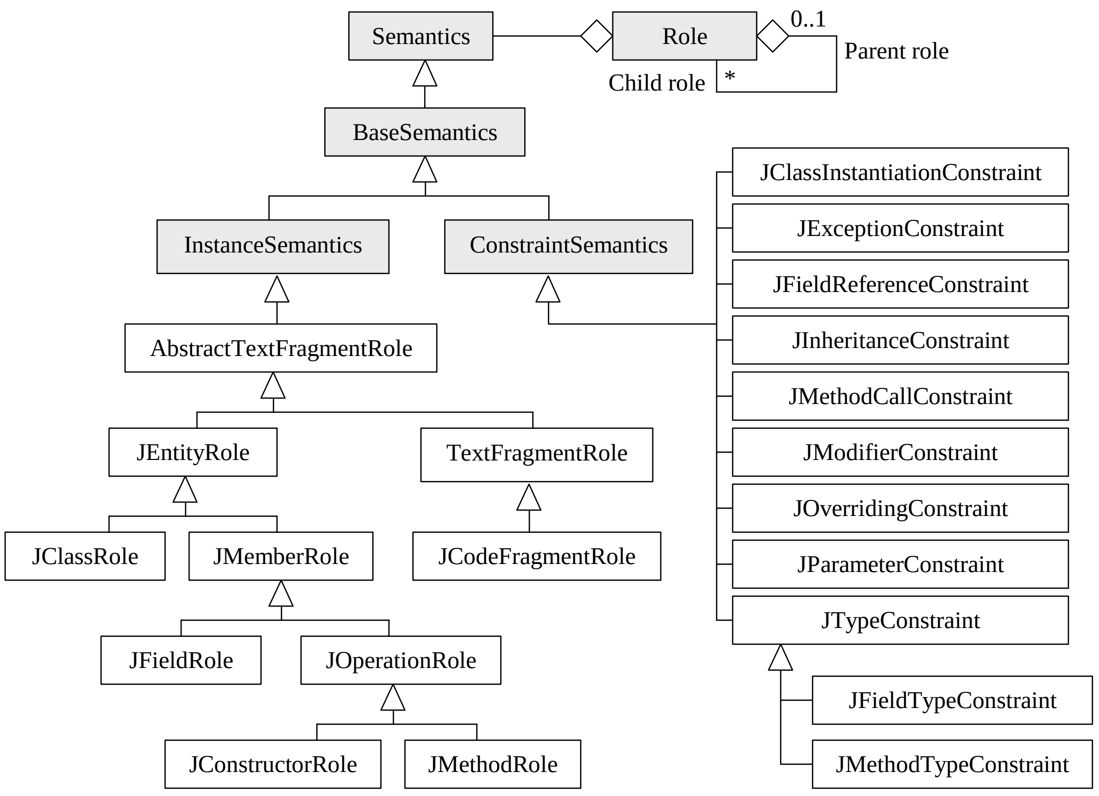


Figure 31. PSF and Java pattern semantics as its example extension.

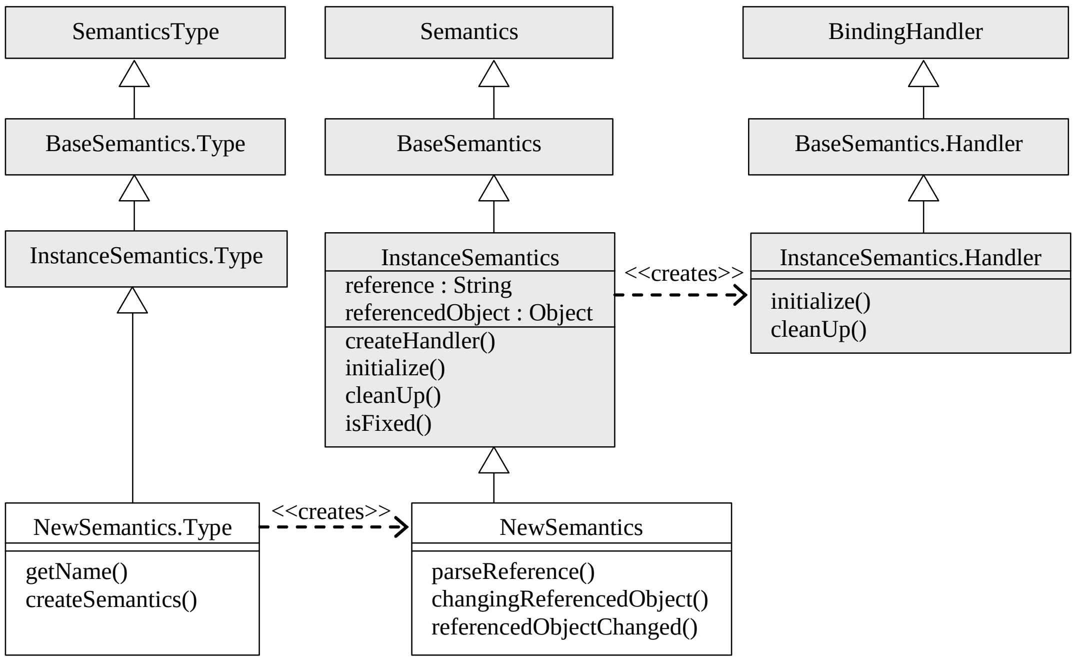


Figure 32. Deriving new instance semantics.

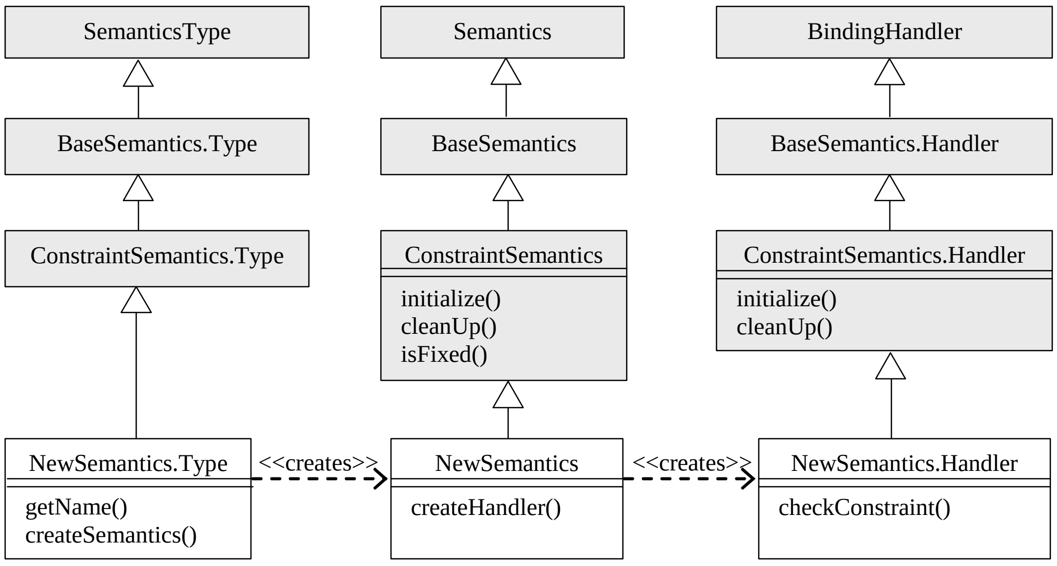


Figure 33. Deriving new constraint semantics.

7.1.2 Properties

In PSF each role type may have a set of properties. As shown in Figure 34, these properties are divided into functions and scripts. Scripts are shown to the pattern modeler as editable fields in the JavaFrames pattern development tool (Subsection 5.2.3). Depending on the script, it can be used to generate adaptive documentation, source code suggestions, task titles, and so on. For example, a specific script is used to generate a title for the role-based task. A typical function, in turn, is the instance function that returns the element that is bound to the role. The value of a function is determined by the evaluate method. For scripts the value is a text that results when the used script interpreter evaluates the source string of the script. This source string is an expression that may contain function calls, references to other roles, and so on.

When extending PSF with new role types, the platform developer may want to create new functions and script interpreters to be used with these role types. The problem is that the implementation of the evaluate method and the interpret method cannot be predicted; the implementation is too domain-specific for a general pattern specification. In addition, the current Java pattern semantics has only limited support to model the complex implementation details of method bodies. Therefore the pattern specification to create a new function or script interpreter is omitted here.

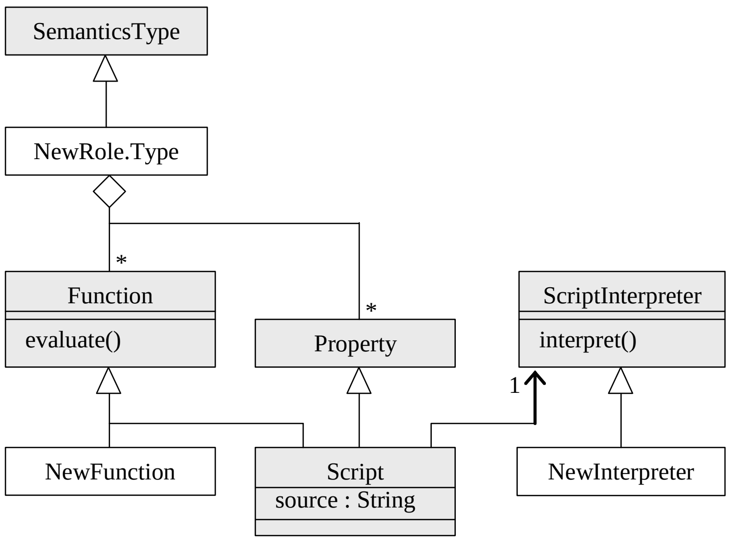


Figure 34. Deriving new functions and script interpreters.

7.1.3 User Interface

In the JavaFrames Eclipse Integration, the user interface to perform tasks and create roles is based on a sophisticated wizard system. Each role type provides its own wizard pages to be used in the wizard system. For example, performing a binding task means that an element is created or selected and then bound to a specific role. When the pattern user selects the task, the Perform Task wizard is opened. The first page of this common wizard shows the selection of more specific alternatives to perform the task; it is the task's role type who provides these more specific alternatives. The pattern user then selects one of the available alternatives to continue. Typically, in the case of a binding task, she may want to point out the desired element or ask the system to generate a new element for the role. In the case of constraint semantics, the pattern user may also need a wizard page to repair possible constraint violations.

The pattern modeler who creates pattern specifications uses another wizard to create roles. The first page of this New Role wizard shows available role types. After selecting a suitable role type, like a Java class role, the second page of the wizard shows alternatives to create a new role of that role type. The most typical alternatives are to create a new unbound role (to be fulfilled later by the pattern user) or a role that is bound to a specific element (for example, to a particular framework base class) by default. In the case of constraint semantics, the pattern modeler needs a wizard page to initialize the constraint.

Figure 35 illustrates how the user interface is specialized. For each new role type the platform developer creates a suitable set of wizard pages by deriving subclasses from the wizard base classes. As mentioned before, in the case of a role type that uses instance semantics, there should be wizard pages to create and locate elements that can be bound to the role. In addition, a wizard page is also needed to create an unbound role. This is needed when the pattern modeler creates new pattern specifications. A pattern to create role-specific wizard pages is presented in Subsection 7.2.4.

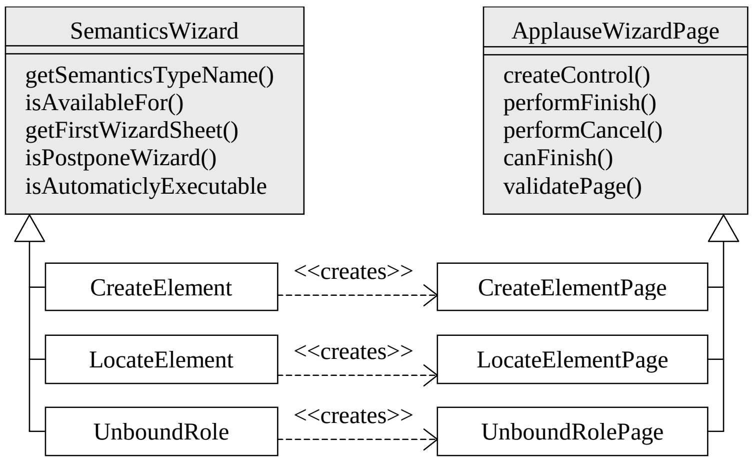


Figure 35. Creating wizard pages for a role type.

7.1.4 Eclipse Plugin

Eclipse is structured around the concept of extension points [Eclipse 2004]. Extension points are places in where other tools (called plugins) can contribute functionality. Eclipse provides the run time engine that starts the platform base and dynamically discovers plugins. Each plugin is a structured component that describes itself to the system using a manifest (plugin.xml) file. Eclipse maintains a registry of installed plugins.

In fact, JavaFrames itself has been integrated into Eclipse as a set of Eclipse plugins. Also the platform developer should introduce new role types and their wizard-based user interface to the system as a new plugin. Figure 36 illustrates the required plugin classes and the configuration file. A pattern to create new Eclipse plugins is presented in Subsection 7.2.5.

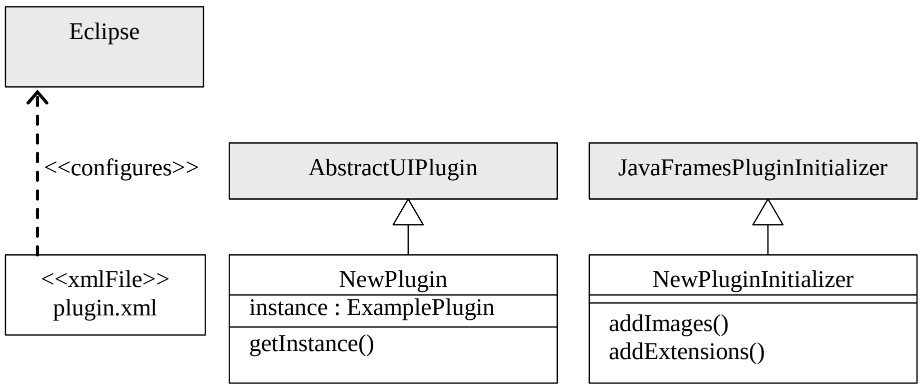


Figure 36. Creating a new Eclipse plugin.

7.2 Pattern Interface to Achieve Goals

The framework-specific specialization goals and the obtained pattern interface to achieve them are discussed in the following subsections:

- *Goals to extend the pattern specification framework*. The goal-oriented approach discussed in Chapter 6 was used to create a pattern interface to extend PSF with new role types. Subsection 7.2.1.
- *Instance Semantics pattern*. The Instance Semantics pattern is used to create new role types that are suitable to represent concrete implementation elements in the solution. Subsection 7.2.2. Appendix C.
- *Constraint Semantics pattern*. The Constraint Semantics pattern is used to create new role types to represent constraints, like the inheritance relation between Java classes. Subsection 7.2.3. Appendix D.
- *Semantics Wizard pattern*. The Semantics Wizard pattern is used to create wizard-based user interface for a role type. Subsection 7.2.4. Appendix E.
- *Eclipse Plugin pattern*. The Eclipse Plugin pattern is used to create a new Eclipse plugin. Subsection 7.2.5. Appendix F.

7.2.1 Goals to Extend the Pattern Specification Framework

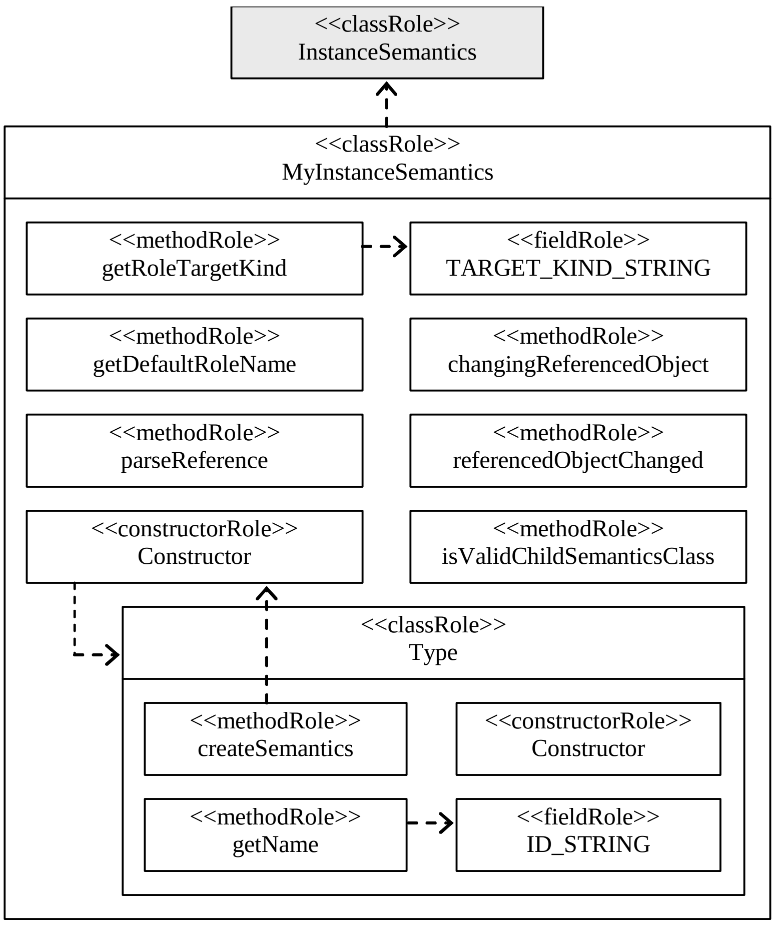
When starting the case study, the author was familiar with the concepts of PSF, but not with the implementation details of its extensions. The process to find the most typical goals was carried out by studing the framework's source code, existing role types, and by reading the available documentation. During this goal mining process, observations were made how to use the framework in order to create a working PSF extension. In fact, writing Section 7.1 and drawing the class diagrams of the example extensions were an important part of the goal mining process. The found goals are the following:

- *Derive new instance semantics*. Instance semantics represent roles that can be bound to elements, like Java classes or methods. Figure 32 illustrated how to create new instance semantics.
- *Derive new constraint semantics*. Constraint semantics represent roles that are used to guard specific requirements and commitments; like that a class must inherit a particular base class. Figure 33 illustrated how to create new constraint semantics.
- *Create new function*. In role types functions are used to retrieve role-specific values, like the name of the role or the reference to the element that is bound to the role. Figure 34 illustrated how to create new functions.
- *Create new script interpreter*. Various script interpreters can be used to interpret the expressions of the script properties. Figure 34 illustrated how to create new script interpreters.
- *Create wizards*. Each role type may introduce its own wizard pages to the JavaFrames wizard system. Figure 35 illustrated how to create new wizard-based user interface for a role type.
- *Create Eclipse plugin*. Extensions, like new role types and wizards, must be introduced as Eclipse plugins. Figure 36 illustrated how to create a new Eclipse plugin.

After finding the goals, the author started to outline patterns as described in Subsection 6.1.2. For each goal, the author studied how the goal was achieved in existing use cases and then he implemented his own example extensions to test and study the details of the solution. Pattern specifications were constructed with the JavaFrames pattern development tool and tested by reproducing these example extensions. As an estimation of the amount of work, learning the framework and finding goals was carried out in one week and the pattern writing, construction of pattern specifications, and testing took also one week. The obtained patterns are presented in the next subsections.

7.2.2 Instance Semantics Pattern

The Instance Semantics pattern guides the platform developer to extend PSF in order to create new instance semantics. An overview of the pattern is shown in Figure 37. The MyInstanceSemantics role represents the Java class that should inherit the InstanceSemantics base class. The base class will handle the bindings between the role and elements bound to it. The new instance semantics should have an inner class called Type that is used to create instances of this role type. In addition, MyInstanceSemantics has a number of method roles to override and implement the abstract methods of the InstanceSemantics base class. The platform developer can apply the Instance Semantics pattern specification to generate most of the required classes and methods, except the implementation of the parseReference method and possible domain-specific event listeners. A more detailed specification is given in Appendix C.

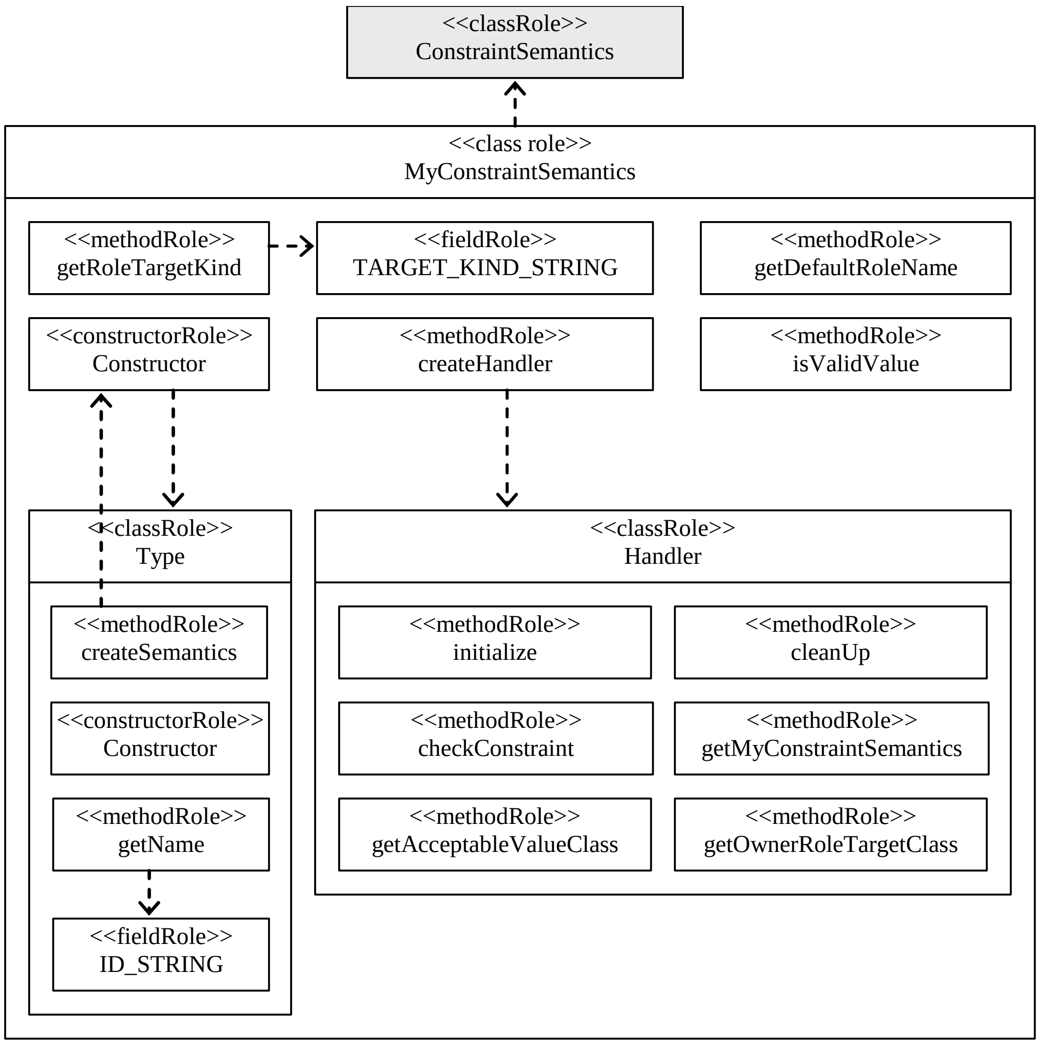


<<classRole>> InstanceSemantics
<<classRole>> MyInstanceSemantics
<<methodRole>> getRoleTargetKind
<<fieldRole>> TARGET_KIND_STRING
<<methodRole>> getDefaultRoleName
<<methodRole>> changingReferencedObject
<<methodRole>> parseReference
<<methodRole>> referencedObjectChanged
<<constructorRole>> Constructor
<<methodRole>> isValidChildSemanticsClass
<<classRole>> Type
<<methodRole>> createSemantics
<<constructorRole>> Constructor
<<methodRole>> getName
<<fieldRole>> ID_STRING

Figure 37. Instance Semantics pattern.

7.2.3 Constraint Semantics Pattern

The Constraint Semantics pattern guides the platform developer to extend PSF in order to create new constraints. An overview of the pattern is shown in Figure 38. The MyConstraintSemantics role represents the Java class that should inherit the ConstraintSemantics base class. The class should have an inner class called Type that is used to instantiate the constraint objects. Another inner class, called Handler, is used to check the constraint itself. The platform developer can apply the Constraint Semantics pattern specification to generate most of the required classes and methods, except the implementation of the checkConstraint method and possible domain-specific event listeners. A more detailed specification is given in Appendix D.



<<classRole>> ConstraintSemantics
<<class role>> MyConstraintSemantics
<<methodRole>> getRoleTargetKind
<<fieldRole>> TARGET_KIND_STRING
<<methodRole>> getDefaultRoleName
<<constructorRole>> Constructor
<<methodRole>> createHandler
<<methodRole>> isValidValue
<<classRole>> Type
<<methodRole>> createSemantics
<<constructorRole>> Constructor
<<methodRole>> getName
<<fieldRole>> ID_STRING
<<classRole>> Handler
<<methodRole>> initialize
<<methodRole>> cleanUp
<<methodRole>> checkConstraint
<<methodRole>> getMyConstraintSemantics
<<methodRole>> getAcceptableValueClass
<<methodRole>> getOwnerRoleTargetClass

Figure 38. Constraint Semantics pattern.

7.2.4 Semantics Wizard Pattern

The Semantics Wizard pattern guides the platform developer to create wizard pages to be used with her role types. An overview of the pattern is shown in Figure 39. MySemantics represents the semantics class that requires the wizard user interface. MyWizard represents the wizard class and MyWizardPage represents the actual user interface of that wizard. Note that there can be multiple wizards for the same role type. The wizard has some simple methods to be overridden. The wizard page is more problematic, as the user interface can be very complicated. The pattern specification provides a default user interface with text fields to set the role name and value expression. In addition, the performFinish method role contains code fragment alternatives to provide the most typical implementation; in the figure, only the dependencies of the createElement code fragment role are shown. With these default implementations, the platform developer can create a working user interface to quickly test her role types. A more detailed specification is given in Appendix E.

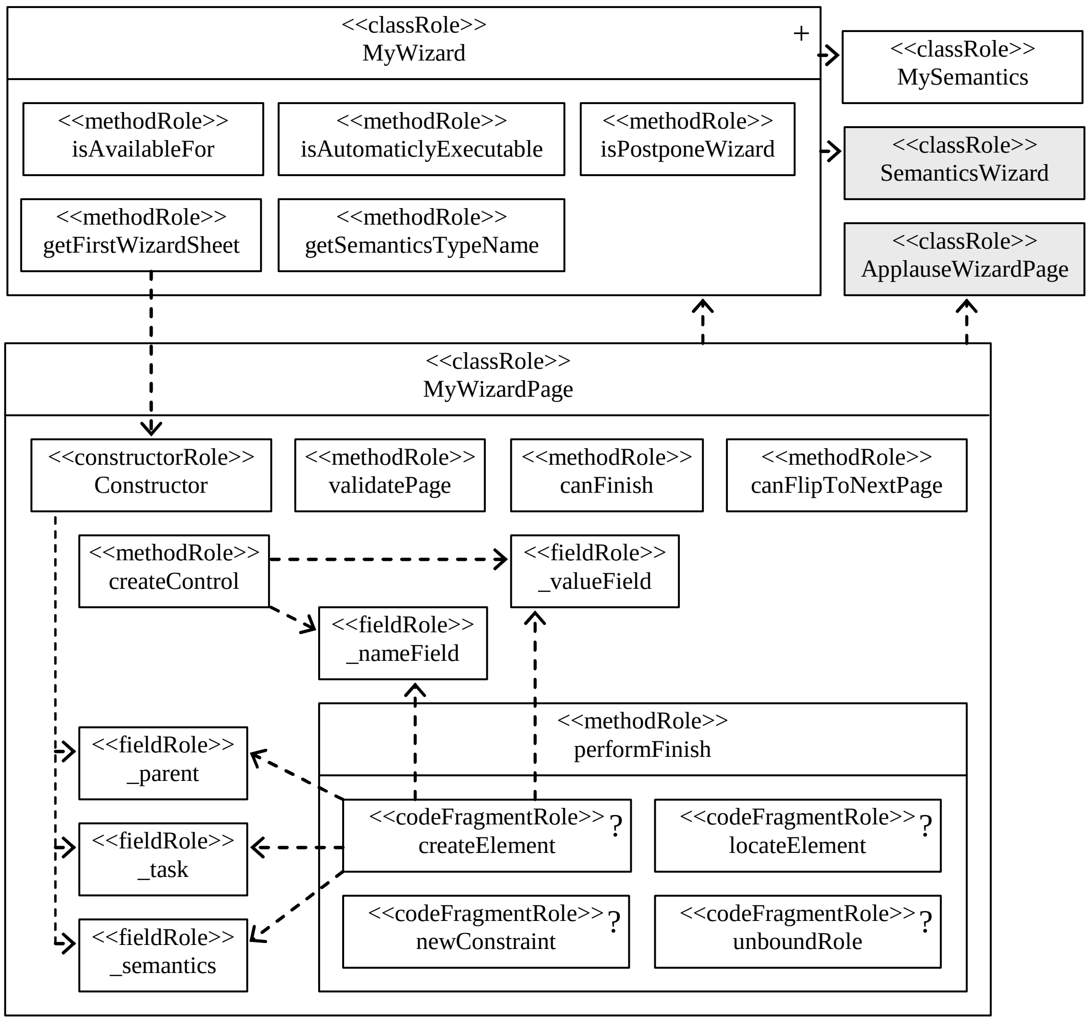

<<classRole>> MyWizard
+
<<classRole>> MySemantics
<<methodRole>> isAvailableFor
<<methodRole>> isAutomaticlyExecutable
<<methodRole>> isPostponeWizard
<<classRole>> SemanticsWizard
<<methodRole>> getFirstWizardSheet
<<methodRole>> getSemanticsTypeName
<<classRole>> ApplauseWizardPage
<<classRole>> MyWizardPage
<<constructorRole>> Constructor
<<methodRole>> validatePage
<<methodRole>> canFinish
<<methodRole>> canFlipToNextPage
<<methodRole>> createControl
<<fieldRole>> _valueField
<<fieldRole>> _nameField
<<methodRole>> performFinish
<<fieldRole>> _parent
<<codeFragmentRole>> createElement ?
<<codeFragmentRole>> locateElement ?
<<fieldRole>> _task
<<codeFragmentRole>> newConstraint ?
<<codeFragmentRole>> unboundRole ?
<<fieldRole>> _semantics

Figure 39. Semantics Wizard pattern.

7.2.5 Eclipse Plugin Pattern

The Eclipse Plugin pattern guides the platform developer to create a new Eclipse plugin. An overview of the pattern is shown in Figure 40. The platform developer must provide the plugin class and the initializer class that adds the extensions to the JavaFrames system. In addition, Eclipse requires that the plugin is deployed under the Eclipse's plugins folder and that there is a specific XML file to introduce the plugin. The implementation of the plugin class is very simple and it can be easily generated. The implementation of the plugin initializer class is also rather straightforward; the platform developer must introduce her role types and user interface wizards inside the addExtensions method. The pattern specification provides two code fragment roles to introduce the selected extensions. A more detailed specification is given in Appendix F.

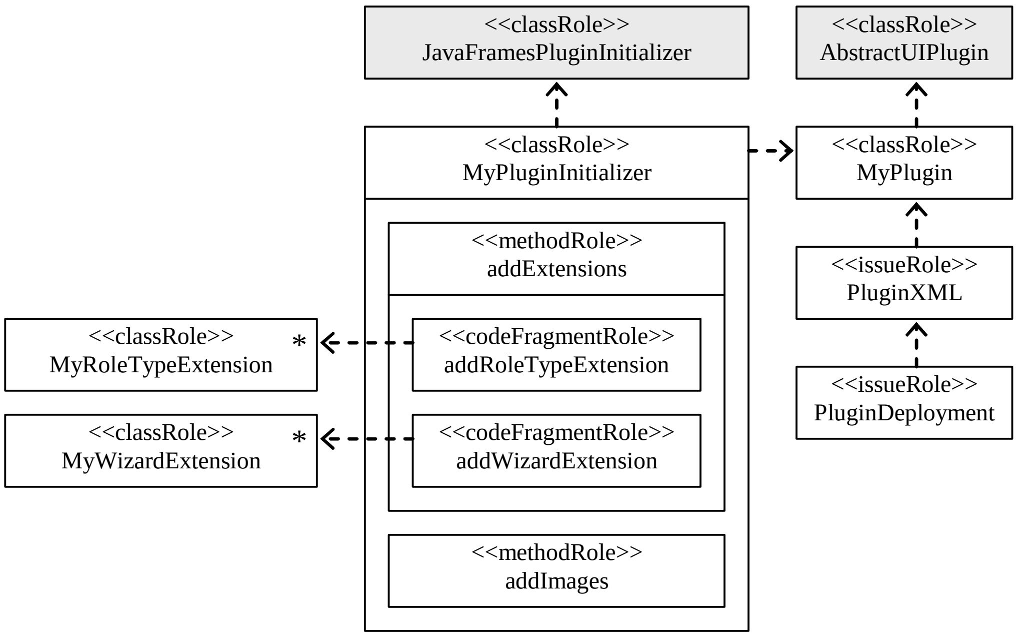


Figure 40. Eclipse Plugin pattern.

7.3 Using the Pattern Interface

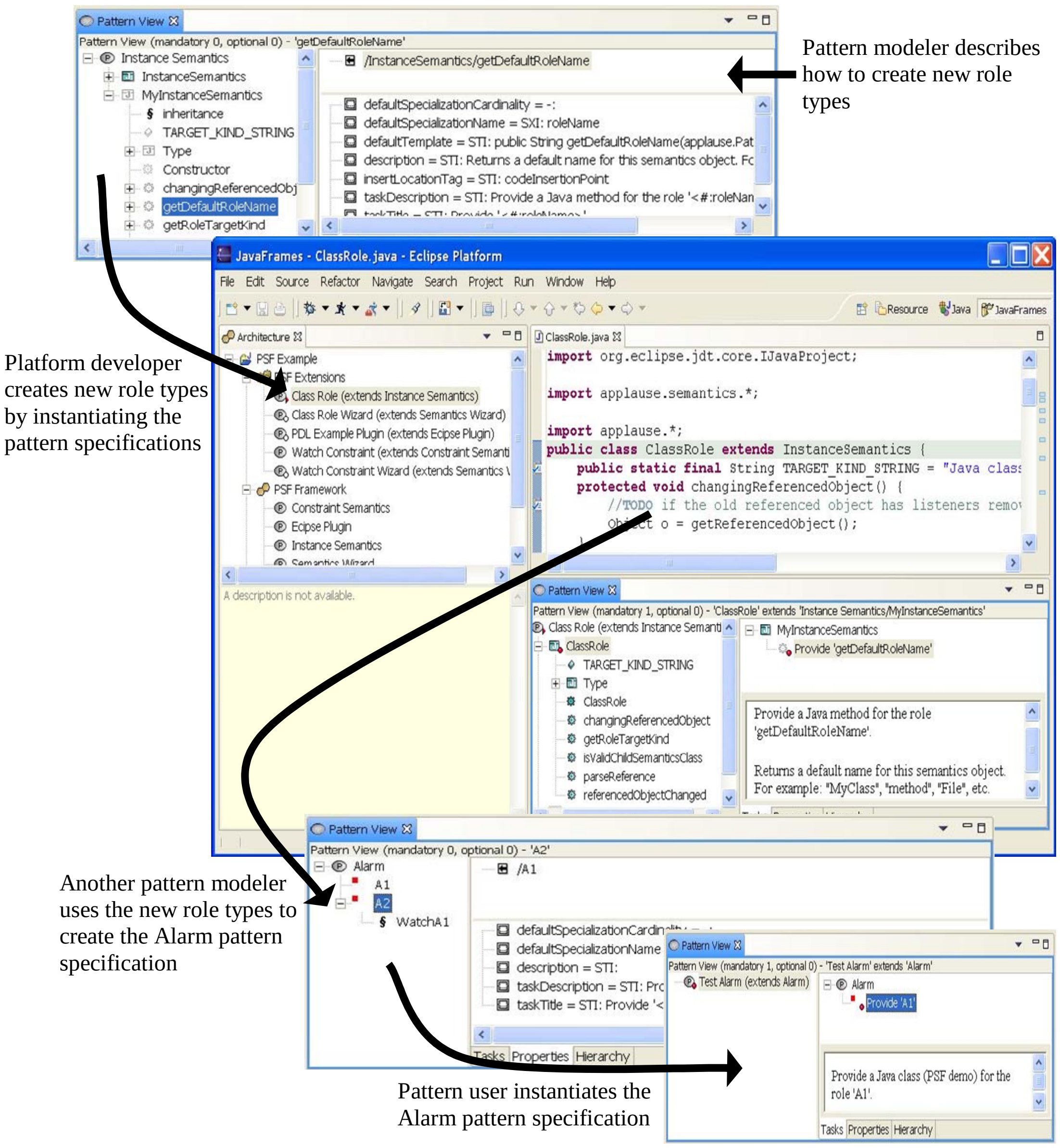
To demonstrate the use of the pattern interface, it is applied to produce two new role types, their user interface, and the required Eclipse plugin. The specialization is discussed in the following subsections:

- *Demonstration*. To better understand the process, a simple example illustrates the creation and use of the new role types. Subsection 7.3.1.

- *Creating two new role types*. The demonstration uses role types that are implemented as extensions of PSF. Subsection 7.3.2.

7.3.1 Demonstration

The process is briefly illustrated in Figure 41. The screenshots are taken from JavaFrames Eclipse Integration discussed in Chapter 5. Firstly, the pattern modeler creates pattern specifications to guide how to create PSF extensions. The platform developer then uses this pattern interface – as a pattern user – to create a new extension. Another pattern modeler can now use the created role types to compose new kinds of pattern specifications. Finally, an ordinary pattern user applies these new pattern specifications in her application.



Pattern modeler describes how to create new role types
Platform developer creates new role types by instantiating the pattern specifications
Another pattern modeler uses the new role types to create the Alarm pattern specification
Pattern user instantiates the Alarm pattern specification
Pattern View
Instance Semantics
InstanceSemantics
MyInstanceSemantics
inheritance
TARGET_KIND_STRING
Type
Constructor
changingReferencedObj
getDefaultRoleName
getRoleTargetKind
JavaFrames - ClassRole.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Resource
Java
JavaFrames
Architecture
PSF Example
PSF Extensions
Class Role (extends Instance Semantics)
Class Role Wizard (extends Semantics Wizard)
PDL Example Plugin (extends Eclipse Plugin)
Watch Constraint (extends Constraint Semanti
Watch Constraint Wizard (extends Semantics \
PSF Framework
Constraint Semantics
Eclipse Plugin
Instance Semantics
A description is not available.
ClassRole.java
Class Role (extends Instance Semanti
ClassRole
TARGET_KIND_STRING
Type
ClassRole
changingReferencedObject
getRoleTargetKind
isValidChildSemanticsClass
parseReference
referencedObjectChanged
MyInstanceSemantics
Provide 'getDefaultRoleName'
Alarm
A1
A2
WatchA1
/A1
Test Alarm (extends Alarm)
Alarm
Provide 'A1'
Tasks Properties Hierarchy

Pattern View (mandatory 0, optional 0) - 'getDefaultRoleName'

/InstanceSemantics/getDefaultRoleName

defaultSpecializationCardinality = -:
defaultSpecializationName = SXI: roleName
defaultTemplate = STI: public String getDefaultRoleName(applause.Pat
description = STI: Returns a default name for this semantics object. Fc
insertLocationTag = STI: codeInsertionPoint
taskDescription = STI: Provide a Java method for the role '<#:roleNan

```
import org.eclipse.jdt.core.IJavaProject;

import applause.semantics.*;

import applause.*;
public class ClassRole extends InstanceSemantics {
    public static final String TARGET_KIND_STRING = "Java class
    protected void changingReferencedObject() {
        //TODO if the old referenced object has listeners remo
        Object o = getReferencedObject();
```

Pattern View (mandatory 1, optional 0) - 'ClassRole' extends 'Instance Semantics/MyInstanceSemantics'

Provide a Java method for the role 'getDefaultRoleName'.

Returns a default name for this semantics object. For example: "MyClass", "method", "File", etc.

Pattern View (mandatory 0, optional 0) - 'A2'

defaultSpecializationCardin
defaultSpecializationName
description = STI:
taskDescription = STI: Pro
taskTitle = STI: Provide '<

Pattern View (mandatory 1, optional 0) - 'Test Alarm' extends 'Alarm'

Provide a Java class (PSF demo) for the role 'A1'.

Figure 41. Pattern support to create new role types and using those role types.

Figure 42, in turn, shows more details of the Alarm example pattern specification. The specification utilizes the new role types <<classRole>> and

<<watchConstraint>>. By using the JavaFrames pattern development tool, the pattern modeler has created the Watcher and Watched class roles and the watch constraint that will observe the Java class bound to the Watched role. Thus, when this pattern is instantiated, the watcher class is associated with a task to check changes whenever a change occurs in the watched Java class. The pattern user instantiates the Alarm pattern by performing the generated tasks; she selects a Java class to be bound to the Watcher role and a Java class to be bound to the Watched role.

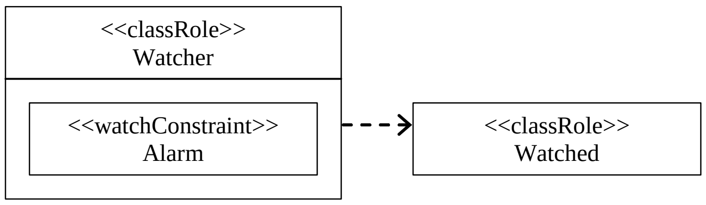


Figure 42. The Alarm pattern specification demonstrating the use of the new role types.

7.3.2 Creating Two New Role Types

The platform developer must implement the new role types before the pattern modeler can use them to compose her pattern specifications. Here PSF's pattern interface is used to create two new role types. The first role type uses instance semantics and it represents a Java class; it is a simplified version of the existing Java class role (see Figure 31) without any code generation facilities or event listeners. The second role type uses constraint semantics. It represents a watch constraint that can be set to observe a particular Java class and warn the pattern user about any changes of that class.

Figure 43 shows how the platform developer uses the pattern interface. To create the new role types, she instantiates Instance Semantics and Constraint Semantics pattern specifications. The wizard-based user interface for these new role types is created by instantiating the Semantics Wizard pattern specification. In addition, the pattern interface guides the platform developer to create the Eclipse plugin that introduces these extensions to the JavaFrames Eclipse Integration.

Figure 44 presents the final application. The platform developer has created the ClassRole and WatchConstraint classes to implement the new role types. She has also created wizard classes to provide the user interface for these role types. The PSFExamplePlugin and PSFExamplePluginInitializer are needed to establish a new Eclipse plugin. The EclipseJavaUtils class contains some utility methods to work with the Eclipse development environment.

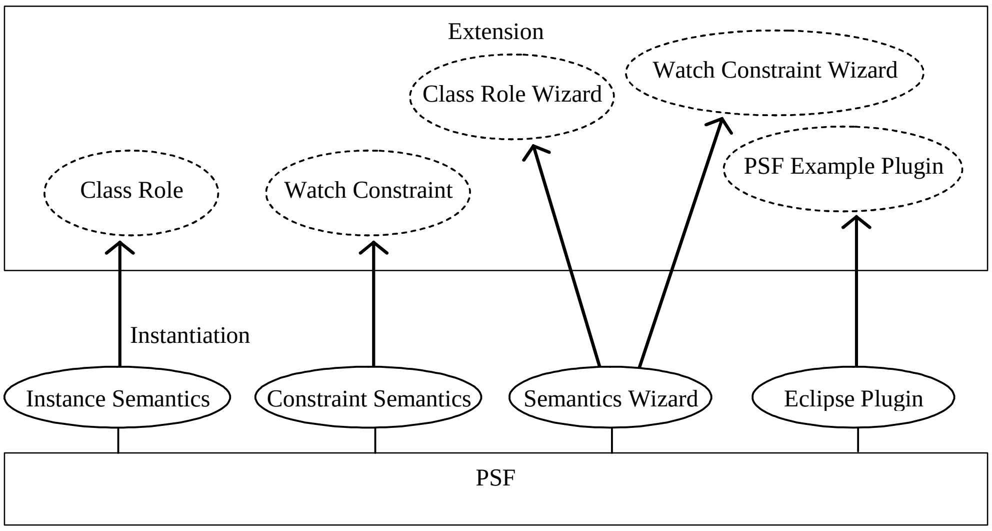


Figure 43. Extending PSF by using its pattern interface.

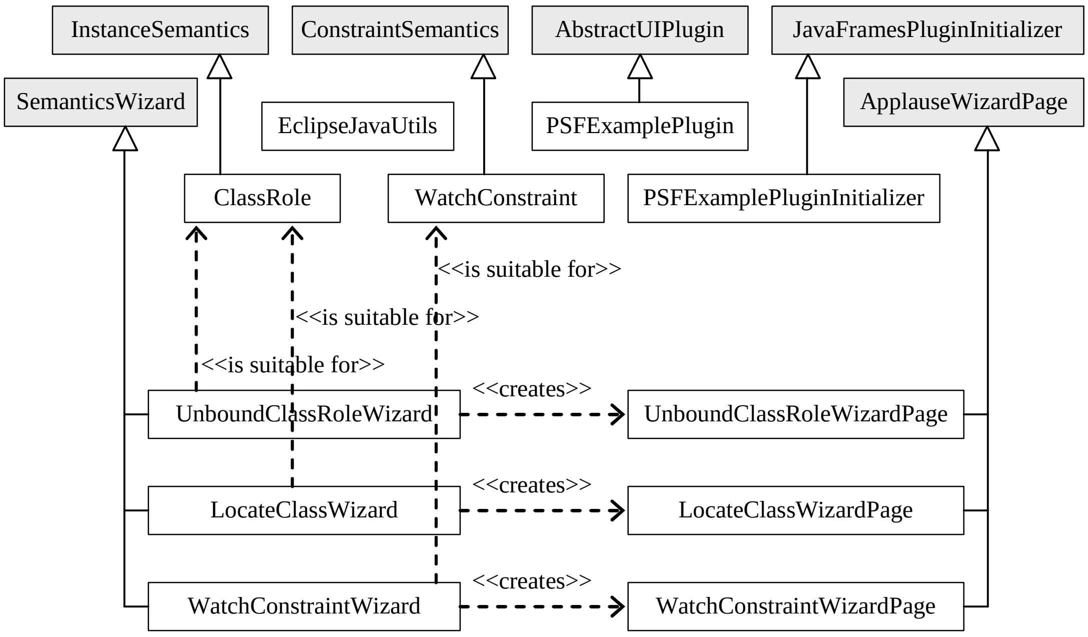


Figure 44. Class diagram of the final PSF example extension.

7.4 Experiences

Analysis and statistics of the results are given in the following subsections:

- *Support for code generation*. Subsection 7.4.1.
- *Support for fragmented design solutions*. Subsection 7.4.2.

7.4.1 Support for Code Generation

Table 3 presents more statistics about the created PSF extension and the use of the pattern interface. The final application has 11 classes and 400 code lines (Java source code lines inside the class declaration). The platform developer had to manually create or edit 85 lines while the pattern interface generated 315 lines, 79% of the code lines. Though the example application is small, the experiment shows that with a suitable pattern interface much of the required specialization code can be generated automatically. The amount of the generated code depends on how detailed the pattern specifications are and how well they capture the problem domain, in this case the extension of PSF. In addition, JavaFrames keeps track of the generated pattern instances, which makes it different from traditional code generators or wizards.

Class	Code lines	Manually edited lines	Generated code ratio
ClassRole	48	8	83%
EclipseJavaUtils	18	18	0%
LocateClassWizard	17	0	100%
LocateClassWizardPage	54	2	96%
PSFExamplePlugin	10	0	100%
PSFExamplePluginInitializer	12	0	100%
UnboundClassRoleWizard	17	1	94%
UnboundClassRoleWizardPage	51	2	96%
WatchConstraint	102	52	49%
WatchConstraintWizard	17	0	100%
WatchConstraintWizardPage	54	2	96%
	400	85	79%

Table 3. The amount of generated code in the PSF example.

To analyse the results, the classes are put into four categories according to the generated code ratio:

- *Weak generated code ratio (0%-25%).* The lowest generated code ratio, 0% for the EclipseJavaUtils class, is because the pattern interface has no patterns to work with the services and interfaces of the Eclipse Java development environment; they are not in the scope of the pattern interface. Another reason is that creating pattern support for an arbitrary utility class is not very cost-effective, unless the form of the class is predictable and similar utility classes are needed repeatedly.

- *Fair generated code ratio (26%-50%).* Similarly, the next lowest generated code ratio, 49% for the WatchConstraint class, is because the general Constraint Semantics pattern specification cannot predict the domain of the constraint. The WatchConstraint class must provide Eclipse-specific listeners and implement the checkConstraint method by utilizing the Eclipse's internal Java syntax tree representation. What could be done is to create patterns for narrower domains. The tradeoff is the complexity and general applicability of the pattern specifications. However, the Constraint Semantics pattern speed up the implementation of the WatchConstraint class by supporting the extension rules of PSF and letting the platform developer to concentrate on the non-supported domain-specific implementation details.

- *Good generated code ratio (51%-75%)*. Currently there are no classes in this category. However, if the ClassRole class is further developed, its generated code ratio will decrease similarly to the generated code ratio of the WatchConstraint class. Also, the generated code ratios of the wizard page classes will decrease if the pattern user is not satisfied with the default user interface and wants to implement more complicated wizard pages. Nevertheless, the pattern interface can maintain the overall architecture and specialization rules that affect these classes, which can be seen as a significant advantage.

- *Excellent generated code ratio (76%-100%)*. The ClassRole, LocateClassWizard, LocateClassWizardPage, PSFExamplePlugin, PSFExamplePluginInitializer, UnboundClassRoleWizard, UnboundClassRoleWizardPage, WatchConstraintWizard, and WatchConstraintWizardPage classes have the highest generated code ratio. Most of them are wizard classes (with a default user interface) and Eclipse plugin classes; their structure is very stable and variations are predictable. This makes it easy for the pattern modeler to create pattern specification that can almost automatically create the required code elements.

7.4.2 Support for Fragmented Design Solutions

Table 4 shows which classes are involved in different pattern instances. For simplicity, the table presents the pattern instances only in the level of classes, though the instances may also include methods, fields, constructors, and code fragments. The letter “M” indicates that the class has been modified during the pattern instantiation. The letter “S” means that the class has been selected to play a role in the pattern instance, but it has not been modified. The letter “F” indicates the framework classes that are part of the pattern instances. The pattern modeler has selected these classes when creating the pattern specifications. Subscripts are used to identify the pattern instances (see Figure 43). For example, the Semantics Wizard pattern has been instantiated twice, to produce user interface for the class role and for the watch constraint.

As demonstrated in Table 4, patterns can be used to group elements in different logical entities and combinations. If an element is modified, the system can check that the element still obeys the pattern-specific rules. If not, a repair task can be generated. Further, also other elements that play roles in the same pattern instance(s) can be checked. A change in one element may require some modifications in other elements. In this particular example, there are no overlapping class modifications across the pattern instances. Instead, some classes, like the WatchConstraint class, are created with one pattern (Constraint Semantics) and then referenced in other pattern instances (Semantics Wizard, Eclipse Plugin). For instance, based on the selected class, new code lines are generated to PSFExamplePluginInitializer. This is necessary to register the new extension classes to the Eclipse system.

Class	Instance Semantics	Constraint Semantics	Semantics Wizard	Eclipse Plugin
AbstractUIPlugin				F_5
ApplauseWizardPage			$F_{3,4}$	
ConstraintSemantics		F_2		
InstanceSemantics	F_1			
JavaFramesPluginInitializer				F_5
SemanticsWizard			$F_{3,4}$	
ClassRole	M_1		S_3	S_5
EclipseJavaUtils				
LocateClassWizard			M_3	S_5
LocateClassWizardPage			M_3	
PSFExamplePlugin				M_5
PSFExamplePluginInitializer				M_5
UnboundClassRoleWizard			M_3	S_5
UnboundClassRoleWizardPage			M_3	
WatchConstraint		M_2	S_4	S_5
WatchConstraintWizard			M_4	S_5
WatchConstraintWizardPage			M_4	

Table 4. The use of the patterns across classes in the PSF example.

CHAPTER 8

CASE STUDY: INDUSTRIAL FRAMEWORK

Nokia produces a family of NMS (Network Management System) and EM (Element Manager) applications that are used to manage the network or network elements. The company has a Java GUI (Graphical User Interface) platform developed to support the implementation of the graphical user interface parts for the variants of this product family [Bonnet 1999]. The purpose of this case study is to annotate the GUI framework with a goal-oriented pattern interface so that JavaFrames can be used as a specialization wizard when creating user interfaces with the Nokia platform. The main work of the case study consists of becoming familiar with the framework, identifying its goals, writing patterns as informal documentation, constructing the pattern interface as a set of formal pattern specifications, testing the installation, and reporting the work.

The case framework is based on the MVC paradigm [Krasner and Pope 1988] and it has about 300 classes. The aim of the MVC paradigm is to provide clear separation between the graphical user interface and the rest of the application. As explained in the framework's documentation, the fundamental principle of the case framework is that every view object is managed by exactly one controller object and every controller (except the main controller) is managed by a parent controller. In addition, the framework provides some useful services for GUI applications, like the clipboard and internationalization facilities. After analyzing the specialization goals of the framework, a collection of patterns was defined to cover a major part of its specialization interface. Due to the confidential nature of the case framework, detailed descriptions about its architecture and implementation are omitted.

The goals to specialize the framework and the obtained pattern interface are discussed in Section 8.1. An example specialization that uses the pattern interface is given in Section 8.2. Experiences are discussed in Section 8.3.

8.1 Pattern Interface to Achieve Goals

The case framework has several extension points in which the framework user must provide her application-specific implementation. Fulfilling these extension points can be seen as a set of specialization goals. The framework-specific specialization goals and the obtained pattern interface to achieve them are discussed in the following subsections:

- *Goals to specialize the framework*. The goal-oriented approach discussed in Chapter 6 was used to create a pattern interface to specialize the most important parts of the case framework. Subsection 8.1.1.
- *Application patterns*. The application patterns are used to build up a basic MVC application with a main view and controller. Subsection 8.1.2.
- *MVC patterns*. The MVC patterns provide support for adding new views, like dialogs and internal frames. Subsection 8.1.3.
- *Service patterns*. The service patterns support topics like the use of the clipboard and internationalization services. Subsection 8.1.4.

8.1.1 Goals to Specialize the Framework

When starting the case study, the author was not familiar with the Nokia platform. The first task was to learn the case framework and to find its specialization goals. The framework's source code or any realistic use cases were not available. However, the framework was annotated with a good documentation that carefully explained the different ways of using it; the goals to specialize the framework were found rather straighforwardly by reading this cookbook-like documentation. The author learned the essentials of the framework in a couple of weeks and was able to create a draft of the required patterns. During the pattern modeling process, iteration was required as each pattern specification was created and tested with JavaFrames Eclipse Integration. The first version of the case study was carried out and reported when evaluating Fred – the former version of JavaFrames [Hautamäki 2002]. Since then, during the development of the JavaFrames Eclipse Integration and writing this dissertation, the case study was iterated and updated once more, and the obtained framework-specific patterns were slightly modified.

An overview of the main specialization goals is shown in Figure 45. In the figure, the goals are grouped into three subsets. Firstly, in the Application goals subset it is supposed that the framework user starts by providing a specific factory class to launch the application, provides a main controller that makes the application compatible with the framework system, and implements the main window. Secondly, in the MVC goals subset it is supposed that the framework user implements additional view- and controller classes as described by the used MVC paradigm. Thirdly, in the Service goals subset the framework user may utilize features like the internationalization service or the clipboard.

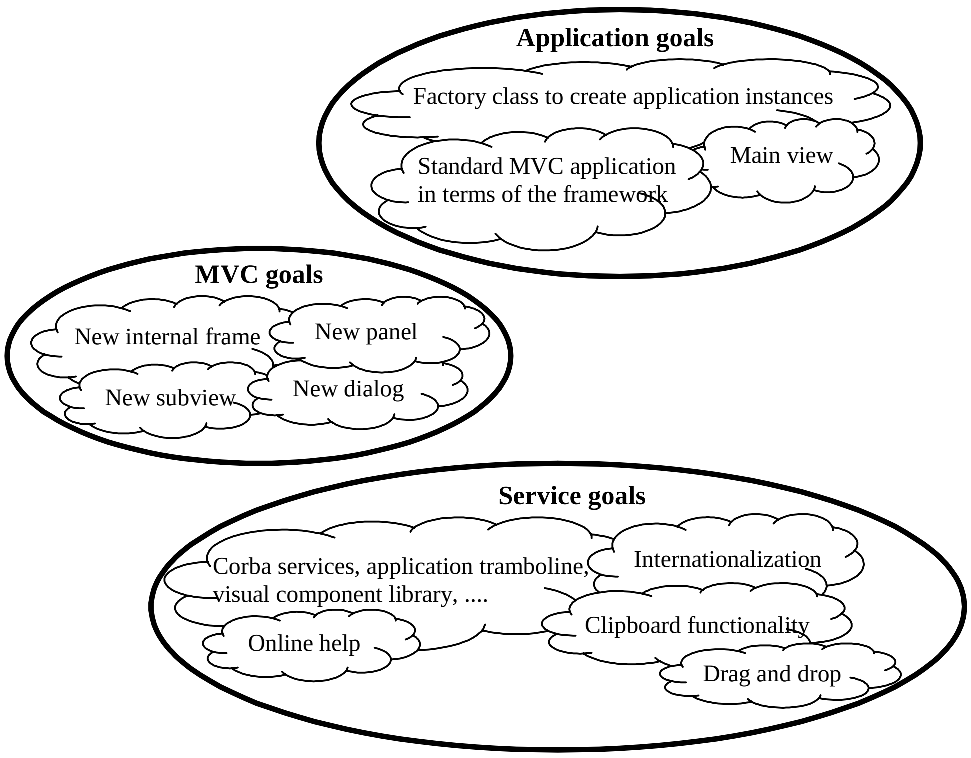

Application goals
Factory class to create application instances
Standard MVC application in terms of the framework
Main view
MVC goals
New internal frame
New panel
New subview
New dialog
Service goals
Corba services, application tramboline, visual component library, ...
Internationalization
Clipboard functionality
Online help
Drag and drop

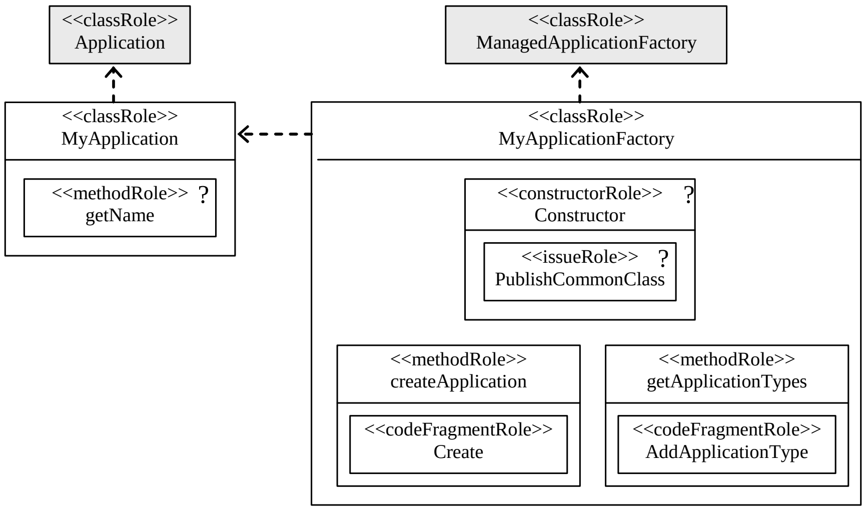
Figure 45. Specialization goals to use the case framework.

Correspondingly, the patterns to achieve these specialization goals can be divided into three categories. Application patterns are used to build up a basic MVC application with a main view and its controller. MVC patterns provide support for adding new views, like dialogs and internal frames. Service patterns support miscellaneous services and features of the framework.

8.1.2 Application Patterns

The application patterns are used to associate the derived application with the framework system and to create the main controller and main window for it. After using these patterns, the framework user has a working skeleton application compatible with the Nokia platform. The application can be launched with a specific factory class and it has the main controller and the corresponding main view. The application patterns are the following:

- *Application Factory*. The Application Factory pattern defines the constructional relationship between an application and the factory class used to create this application. The Application Factory pattern is outlined in Figure 46.
- *MVC Application*. Each application created with the Application Factory pattern should be a standard MVC application in terms of the framework. The MVC Application pattern helps to implement such an application. This includes the creation of the application's main controller and main view and the interactions between the UI and the main controller. Before using this pattern, one should create the application class with the Application Factory pattern. After creating the main user interface, the framework user should apply the MVC patterns (Subsection 8.1.3) to create other views, like windows, dialogs, internal frames, and panels. The MVC Application pattern is outlined in Figure 47.



<<classRole>> Application
<<classRole>> ManagedApplicationFactory
<<classRole>> MyApplication
<<methodRole>> getName ?
<<classRole>> MyApplicationFactory
<<constructorRole>> Constructor ?
<<issueRole>> PublishCommonClass ?
<<methodRole>> createApplication
<<codeFragmentRole>> Create
<<methodRole>> getApplicationTypes
<<codeFragmentRole>> AddApplicationType

Figure 46. Application Factory pattern.

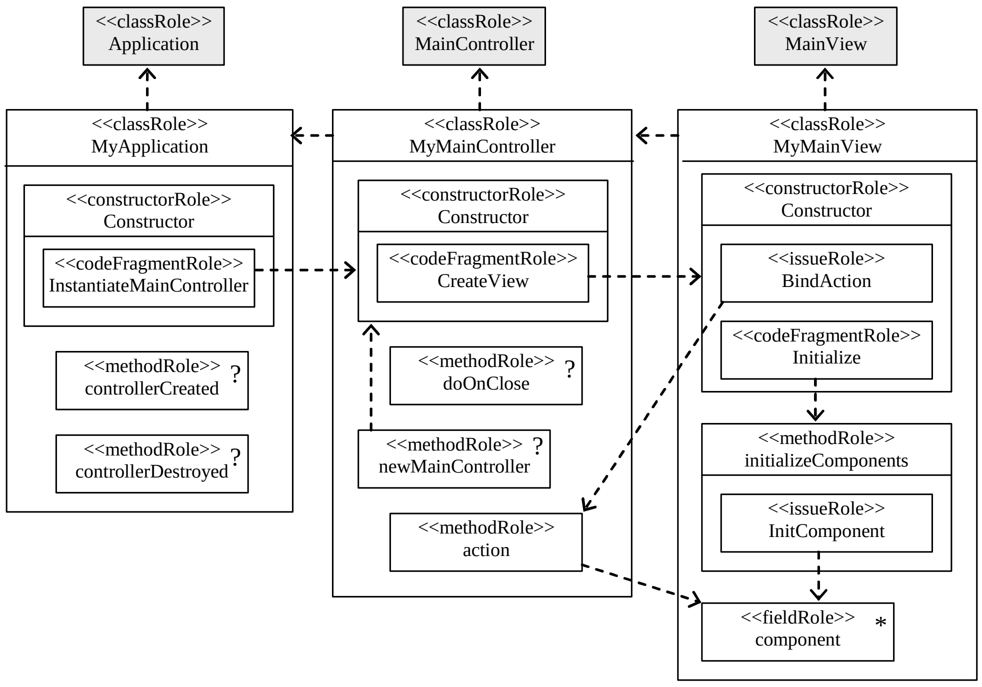

<<classRole>> Application
<<classRole>> MainController
<<classRole>> MainView
<<classRole>> MyApplication
<<constructorRole>> Constructor
<<codeFragmentRole>> InstantiateMainController
<<methodRole>> controllerCreated ?
<<methodRole>> controllerDestroyed ?
<<classRole>> MyMainController
<<constructorRole>> Constructor
<<codeFragmentRole>> CreateView
<<methodRole>> doOnClose ?
<<methodRole>> newMainController ?
<<methodRole>> action
<<classRole>> MyMainView
<<constructorRole>> Constructor
<<issueRole>> BindAction
<<codeFragmentRole>> Initialize
<<methodRole>> initializeComponents
<<issueRole>> InitComponent
<<fieldRole>> component *

Figure 47. MVC Application pattern.

8.1.3 MVC Patterns

The fundamental principle of the case framework is that every view object is managed by exactly one controller object and every controller (except the main controller) is managed by a parent controller. There are different kinds of controllers that handle different kinds of views. The view-controller interaction includes callbacks from the view to the controller, triggered by user actions. The controller, in turn, gives orders to its view. To create subviews, dialogs, panels, and internal frames the framework user applies the corresponding MVC patterns. As an example, the Subview pattern is shown in Figure 48. The MyView role represents the view and the MyController role represents the corresponding controller. MyParentController represents the controller's parent controller. The event mechanism between the controller and its parent is represented with MyControllerEvent and MyControllerListener roles.

Other MVC patterns are almost identical. They establish the required view-controller pairs, in which each child controller may send event notifications to its parent controller. The MVC patterns are the following:

- *MVC Subview*. The MVC Subview pattern is used to create controllers and views for new frame windows. It describes a parent-child relation between a parent controller (e.g., the application's main controller) and a subcontroller handling the frame window.

- *MVC Dialog*. The MVC Dialog pattern is used to create controllers and views for dialogs. It describes a parent-child relation between a parent controller and a dialog controller handling the dialog.

- *MVC Internal Frame*. The MVC Internal Frame pattern is used to create controllers and views for internal frames (windows opened in the desktop area of the parent window). It describes a parent-child relation between a parent controller (must be the main controller or other frame controller) and an internal frame controller handling the internal frame.

- *MVC Panel*. The MVC Panel pattern is used to create controllers and views for panel components. It describes a parent-child relation between a parent controller and a panel controller handling the panel.

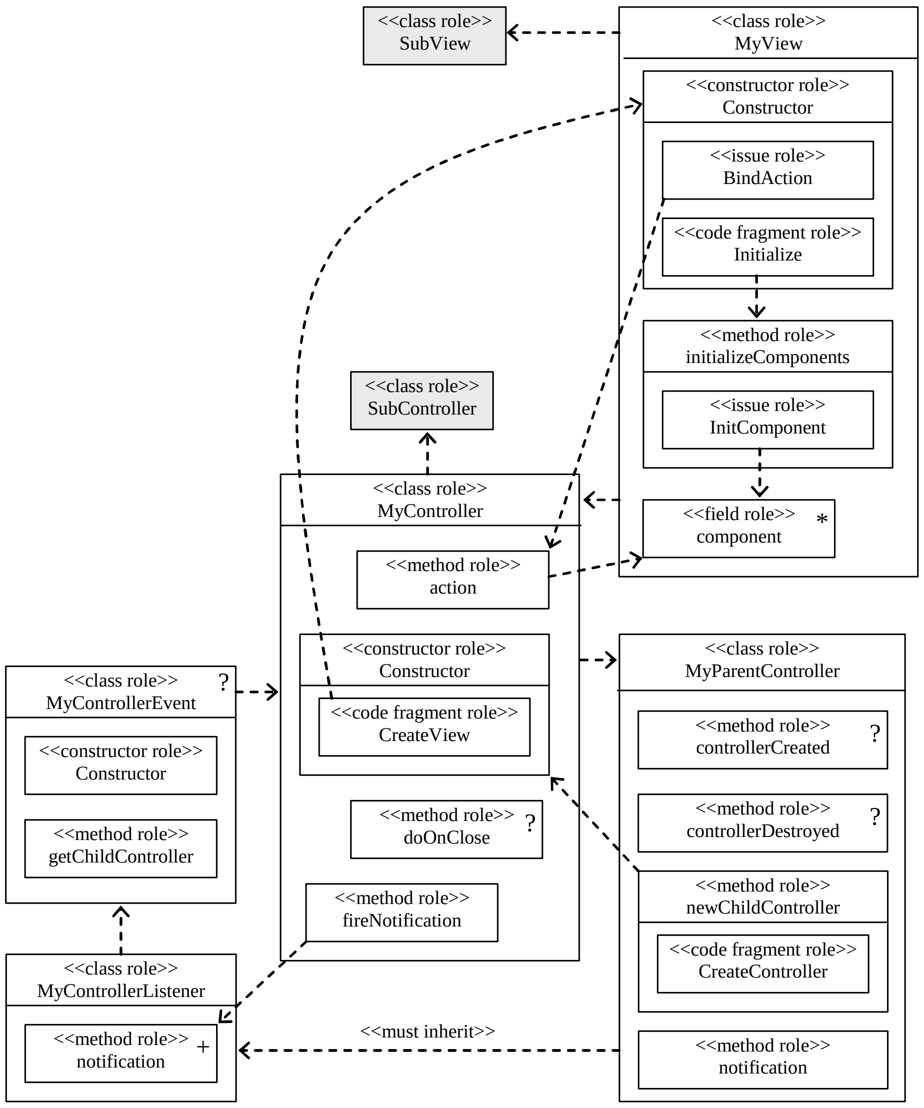

<<class role>> SubView
<<class role>> MyView
<<constructor role>> Constructor
<<issue role>> BindAction
<<code fragment role>> Initialize
<<method role>> initializeComponents
<<issue role>> InitComponent
<<field role>> component *
<<class role>> SubController
<<class role>> MyController
<<method role>> action
<<constructor role>> Constructor
<<code fragment role>> CreateView
<<method role>> doOnClose ?
<<method role>> fireNotification
<<class role>> MyControllerEvent ?
<<constructor role>> Constructor
<<method role>> getChildController
<<class role>> MyControllerListener
<<method role>> notification +
<<class role>> MyParentController
<<method role>> controllerCreated ?
<<method role>> controllerDestroyed ?
<<method role>> newChildController
<<code fragment role>> CreateController
<<method role>> notification
<<must inherit>>

Figure 48. The MVC Subview pattern specification.

8.1.4 Service Patterns

Besides the MVC paradigm, the case framework provides services and components that are useful when building GUI applications, like internationalization, clipboard, and various GUI components. Unlike with the MVC system, their use is typically splattered inside methods of the view- and controller classes. The service patterns have not been implemented with the latest JavaFrames version, but some of them have been reported in the earlier case study [Hautamäki 2002]:

- *I18n*. The case framework contains classes to be used when creating global applications. The internationalization handles strings, colors, fonts, and icons. The data for these is stored in locale specific property files. The handling of the files is normally done by the framework system. The I18n pattern defines how to use the internationalization service.

- *Clipboard*. The Clipboard pattern defines how to transfer data between components and the clipboard. Each component type should have a specific adapter that handles the data transfer. However, note that the standard Java UI components already support the clipboard functionality.

- *Drag And Drop*. The drag and drop metaphor means the graphical operation in which items are dragged from one UI component to another. To implement this feature, there are several supporting classes in the case framework that can be used to reduce the amount of code needed in normal drag and drop cases. However, the framework user must implement number of classes dealing with the drag source and the drop target. The Drag And Drop pattern supports the use of this complex mechanism.

8.2 Using the Pattern Interface

To demonstrate the use of the obtained pattern interface, it is applied to reproduce the Bank example application discussed by Bonnet [1999]. He specifies the Bank application as following. A bank consists of a set of clients who are characterized by their name and password. An account has exactly one owner, which is the bank client. There can be withdrawal and deposit transactions; the account maintains a history of all the transactions made by its owner. The bank has an authentication system where several clients can be logged in concurrently. The bank also keeps a list of the current connections.

When a client is successfully logged in a session is created and her account is opened. During a session, a client can perform transactions and view her transaction history.

The user interface of the Bank application is shown in Figure 49. The main window shows the list of opened sessions. The client can open a session with the authentication dialog, which asks the client's name and password. If the authentication is successful, a new session window is opened. In the session window, the client can use different views to perform deposit and withdrawal transactions. She can also view the history of her transactions.

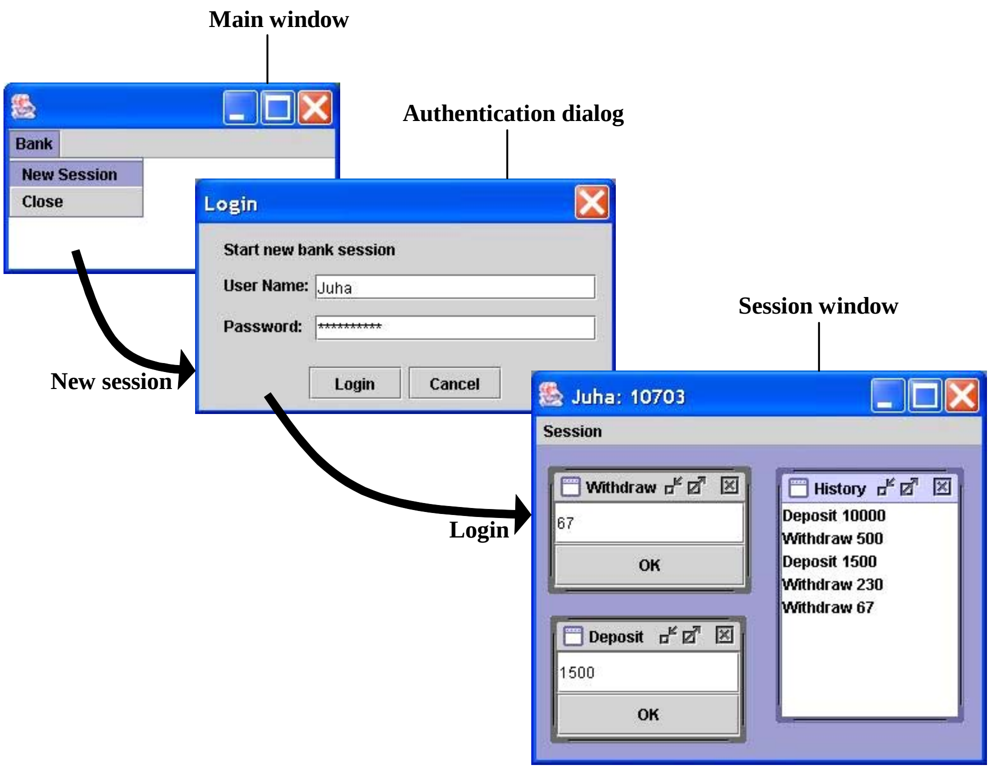


Figure 49. User interface of the Bank application.

The final application consists over 500 code lines and 24 classes. Of course, the application is not a complete banking system; for example, it does not store account information or check the validity of authentications and transactions. Implementing these features would increase the number of classes and code lines, but it is irrelevant from the viewpoint of this example, as the current pattern interface has no patterns to support their implementation. Instead, the pattern interface is able to produce the required MVC classes and guide the framework user through their implementation.

Figure 50 shows how the pattern interface was used to instantiate pattern specifications to create the user interface of the Bank application. First, the Application Factory and MVC Application pattern specifications were instantiated to create the required factory class and the main window and controller of the Bank application. Then, the authentication dialog was implemented by instantiating MVC Dialog. The session window and its internal frames to perform transactions and to view history were implemented similarly. The created classes and their statistics are discussed in the next section.

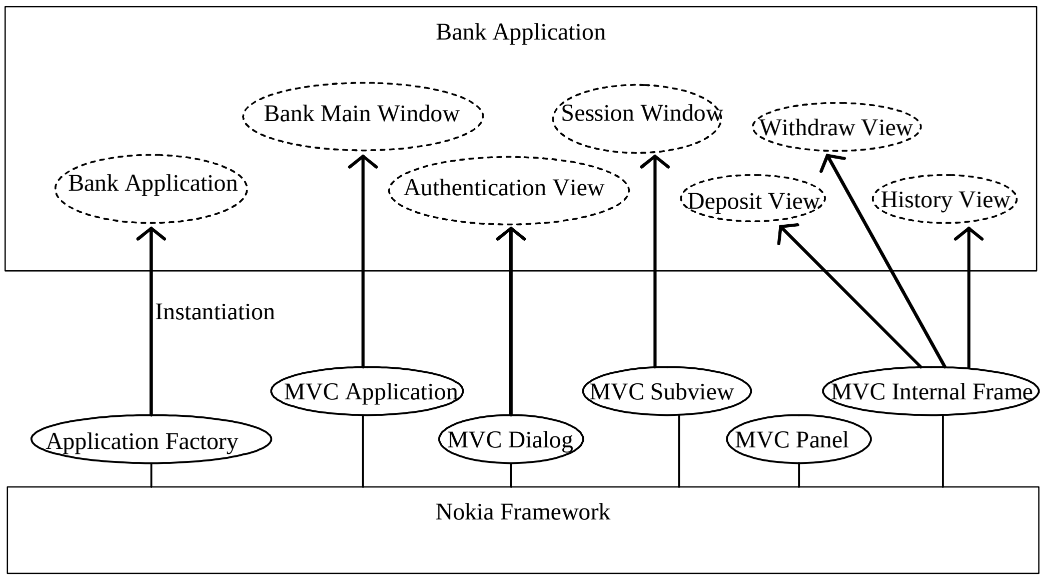


Figure 50. Using the pattern interface to specialize the case framework.

8.3 Experiences

The experiences gained from the industrial case study were encouraging, showing that the goal-oriented approach discussed in Chapter 6 and the JavaFrames Eclipse Integration discussed in Chapter 5 are sufficiently powerful to instruct the specialization interface of a real framework. Especially, the incremental specialization process with context sensitive specialization instructions facilitates the understanding of the framework by supporting learning-by-doing. Analysis and statistics of the results are given in the following subsections:

- *Support for code generation*. Subsection 8.3.1.
- *Support for fragmented design solutions*. Subsection 8.3.2.

8.3.1 Support for Code Generation

In the case of a small example application, the system worked satisfactorily. For instance, as illustrated in Table 5, when making the Bank application compatible with the framework's MVC system, lot of the required source code was generated automatically. The final application has 24 classes and 551 code lines (Java source code lines inside the class declaration). The framework user had to manually create or edit 293 lines while the pattern interface generated 258 lines, 47% of the code lines. The generated code ratio is lower than the ratio of the PSF example (79%, Chapter 7), but still the pattern interface accelerated programming and helped the framework user to implement non-trivial and crosscutting design fragments.

Class	Code lines	Manually edited lines	Generated code ratio
Account	51	51	0%
AccountEvent	9	9	0%
AccountListener	3	3	0%
AuthenticationController	18	1	94%
AuthenticationControllerListener	3	0	100%
AuthenticationEvent	14	7	50%
AuthenticationView	12	3	75%
BankApplication	14	0	100%
BankApplicationController	39	11	72%
BankApplicationFactory	25	0	100%
BankApplicationView	42	32	24%
DepositController	21	4	81%
DepositControllerEvent	8	0	100%
DepositControllerListener	3	0	100%
DepositView	35	21	40%
HistoryController	12	3	75%
HistoryView	30	16	47%
SessionController	70	42	40%
SessionView	61	51	16%
Transaction	14	14	0%
WithdrawController	21	4	81%
WithdrawControllerEvent	8	0	100%
WithdrawControllerListener	3	0	100%
WithdrawView	35	21	40%
	551	293	47%

Table 5. The amount of generated code in the Bank application.

Like with the PSF example in Subsection 7.4.1, the classes are put into four categories according to the generated code ratio:

- *Weak generated code ratio (0%-25%)*. The lowest generated code ratio, 0% for the Account, AccountEvent, AccountListener, and Transaction classes, is because the pattern interface has no patterns to implement the application model of the banking system. The BankApplication-View class (24%) and the SessionView class (16%), in turn, utilize the pattern interface but they also contain lot of UI components and event listeners that cannot be predicted in the pattern interface. Using the goal-oriented pattern interface to support the usage of individual components, like buttons and scroll bars, can be tedious. In addition, there exist RAD (Rapid Application Development) tools that can be used to compose the user interface with these components. However, if the usage of the components is part of a more complex and predictable framework specialization, a pattern could be used at some point to teach and guide the framework user.

- *Fair generated code ratio (26%-50%)*. Similarly, the AuthenticationEvent, DepositView, HistoryView, SessionController, and WithdrawView classes must provide some implementations that could not be predicted by the pattern interface. Typically, these are some user interface components and their event listeners. However, the pattern interface significantly speeded up the overall implementation and helped to make these classes compatible with the used MVC paradigm.

- *Good generated code ratio (51%-75%)*. Over half of the source code of the AuthenticationView, BankApplicationController, and HistoryController classes was generated automatically. Again, there were some issues that could not be predicted by the pattern interface. For example, the BankApplicationController class has to implement logic to add and remove bank clients. The HistoryController class need a method to update the history view. The AuthenticationView utilized a special dialog base class provided by the framework to ask the user name and password. Clearly, the current version of the pattern interface is more suitable to express structural solutions, like how to create a skeleton for a new MVC application, than small, unpredictable behavioral aspects. From the standpoint of the pattern interface, the application-specific logic or the place where this kind of behavioral aspect is created cannot be specified beforehand, unless the application domain is very limited.

- *Excellent generated code ratio (76%-100%)*. The AuthenticationController, AuthenticationControllerListener, BankApplication, BankApplicationFactory, DepositController, DepositControllerEvent, DepositControllerListener, WithdrawController, WithdrawControllerEvent, and WithdrawControllerListener classes have the highest generated code ratio. The structure of the controller classes and their event classes is very stable from application to application. Though there are some variations between different applications, the implementation of these classes is rather predictable, which makes it possible to provide a strong pattern-based tool support for them.

8.3.2 Support for Fragmented Design Solutions

As with the PSF example in Subsection 7.4.2, Table 6 shows how the classes are involved in different pattern instances. The letter "M" indicates that the class has been modified during the pattern instantiation while "F" indicates the framework classes. Subscripts are used to identify pattern instances (see Figure 50).

In Table 6, some of the classes are involved and modified in multiple pattern instances. For instance, the SessionController class is a parent controller for the internal frames used in the Bank application. It is mainly created with the MVC Subview pattern, but it is also involved as a parent controller when instantiating the MVC Internal Frame pattern to create the transaction and history windows. In this way the pattern interface helps to control implementation details that scatter across classes.

Class	App. Factory	MVC App.	MVC Dialog	MVC Subview	MVC Internal Frame
Application	F_1	F_2			
DialogController			F_3		
DialogView			F_3		
InternalFrameController					$F_{5,6,7}$
InternalFrameView					$F_{5,6,7}$
MainController		F_2			
MainView		F_2			
ManagedApplicationFactory	F_1				
SubController				F_4	
SubView				F_4	
Account					
AccountEvent					
AccountListener					
AuthenticationController			M_3		
AuthenticationControllerListener			M_3		
AuthenticationEvent			M_3		
AuthenticationView			M_3		
BankApplication	M_1	M_2			
BankApplicationController		M_2	M_3	M_4	
BankApplicationFactory	M_1				
BankApplicationView		M_2			
DepositController					M_5
DepositControllerEvent					M_5
DepositControllerListener					M_5
DepositView					M_5
HistoryController					M_6
HistoryView					M_6
SessionController				M_4	$M_{5,6,7}$
SessionView				M_4	
Transaction					
WithdrawController					M_7
WithdrawControllerEvent					M_7
WithdrawControllerListener					M_7
WithdrawView					M_7

Table 6. The use of patterns across classes in the Bank example.

One issues that this case study can be criticized is that ordinary software developers creating real industrial applications have not used the constructed pattern specifications; such use cases would most probably point out some weaknesses and missing patterns in the used pattern interface.

As a conclusion, we believe that for a novice user, working with the tool supported pattern interface is illustrative, making it easier to learn the framework and to start working with the framework. One of the most important benefits of the goal-oriented pattern interface could be its applicability to teach programmers as they are simultaneously creating meaningful applications with the Nokia platform. An experienced user, in turn, can use the system to automatically produce a lot of essential and strictly regulated, but uninteresting code.

CHAPTER 9

RELATED WORK

This chapter is a brief survey of tools and techniques dealing with patterns, frameworks, and product line architectures. Examples of various pattern tools are discussed in Section 9.1. Concentrated on framework specialization, different approaches and tools are discussed in Section 9.2. From a wider perspective, the support for framework specialization is only one step towards architecture-oriented software development environment; this is discussed in Section 9.3. Different techniques to create tool supported pattern interfaces are discussed in Section 9.4.

9.1 Pattern Tools

A number of authors have suggested various formalisms, specification languages, and tools to use patterns and pattern-like concepts. A thorough survey of these tools goes beyond this dissertation. Instead, some of the pattern tools are discussed here to demonstrate the different approaches and ideas to use patterns in software engineering. A good catalogue of various pattern tools is given also by Viljamaa J. [1997]. More information about patterns and supporting tools can be found from the pattern WWW sites [Hillside 2004; Portland 2004].

In Section 4.5, pattern tools were roughly categorized in development tools, deployment tools, analysis tools, and documentation tools. In literature, most of the suggested pattern tools seem to be either deployment tools to instantiate patterns, or analysis tools for pattern mining and visualization. Usually, the pattern specification is given directly with some formalism or specification language without any sophisticated pattern development tool. Pattern-based documentation tools, like the Pattern Recorder discussed in Subsection 5.2.4, seem to be rare. When compared to different pattern tools, the advantage of JavaFrames is that it is an extensible tool platform for a number of pattern tools and pattern semantics, not just to deploy or analyze, say, a restricted set of design patterns.

The design patterns presented by Gamma et al. [1995] have inspired a number of pattern deployment and analysis tools. For example, Budinsky et al. [1996] discuss the COGENT interpreter that uses template-based macro expansion mechanism to generate instances of design patterns. Wild [1996], in turn, discuss a tool called SNIP that uses template-based code generation rules and structural object models for source code generation. Typically, these kinds of tools are used as code generators, in which the pattern instance is not maintained after it has been implemented.

To maintain pattern instances also after instantiation, Kim and Benner [1996] present a C++ tool called POE for creating, deleting, and verifying pattern instances. POE includes components to represent classes, relations, and operations. These components have attributes like name, parent link, optionality, links to other components with cardinalities (multiplicity), and bindings to the user's implementation classes. When a pattern is instantiated the tool makes bindings between the pattern classes and the application classes. The tool implements also validation algorithms to ensure that different pattern instances and role bindings are used properly. In practice, the algorithms check components and their attributes for possible conflicts. The idea of validation algorithm (pattern engine) and bindings between pattern classes and application classes resembles the approach used in JavaFrames.

Florijn et al. [1997] discuss a pattern deployment tool (OMT-tool) that can be used for Smalltalk applications. An important point is that they have abandoned the approach in which the software developer only works on the level of design patterns and a tool generates skeleton code for the application. They have noticed that this approach is impractical as there cannot be a pattern for every possible design problem. In addition, generating design patterns as one-shot atomic actions ignores the iterative nature of the software development process. Instead, they have selected an approach in which design patterns, software architecture, and code are represented as different levels of abstraction within the same development environment. On the design pattern level the software developer can instantiate design patterns, on the software architecture level she can split classes into a hierarchy, and on the code level she can modify the source code. All the time, the software developer can use suitable patterns, but the lack of patterns does not prevent her work.

Another useful point discussed by Florijn et al. [1997] is that a pattern could be associated with a set of pattern specific operations to perform tasks that are particular to that pattern. That is, the creation and modification of the participating program elements could be managed with tasks. In addition, it should be possible to add new patterns to the environment and to save, export, and reload programs without losing the pattern information. Moreover, the tool should be independent of a particular programming language. All these features have been implemented in JavaFrames Eclipse Integration. However, the OMT-tool itself concentrates on instantiation and analysis of design patterns in Smalltalk applications, while JavaFrames provides extensible and scalable system to use variety of pattern semantics and pattern tools, for example, to specialize Java frameworks. The OMT-tool was mainly developed to support the identification of design patterns in existing software and to reorganize these programs.

Cornils [2001] present the DPDOC tool that uses reference attribute grammars to represent design patterns. A pattern is expressed by a grammar with syntactic and context sensitive rules which specify the roles and rules of the pattern. The tool supports both visibility and rule checking of design patterns and it also uses reference attributes to connect the design pattern instances with the program code. The connection between the pattern specification and the actual program code is made by entering class names, method names, and so on, in the pattern application code. In JavaFrames, the use of patterns is more transparent as the pattern instantiation is made by performing simple programming tasks derived from the pattern specification.

Eden [2001, 2002a] presents LePUS (LanguagE for Patterns Uniform Specification), which is a formal specification language for object-oriented design and architecture. It is a visual formalism that is based on small number of well defined building blocks that can be found from object-oriented design. LePUS allows also formal specifications of generic formations, like design patterns and the intended usage of framework's specialization interface. LePUS formulae are purely symbolic logic statements and the supporting tool is implemented in Prolog. With the tool, LePUS statements are translated into Prolog predicates and the tool can validate and recognize them in the source code. The tool can also modify a program so that it satisfies the formula. This sounds similar approach than the one used in JavaFrames. However, in JavaFrames Eclipse Integration pattern specifications have no textual formalism. Instead, they are implemented by composing roles with a semi-graphical development tool. Also, new role types can be created to represent other structures than just the most typical object-oriented concepts.

Besides at the level of source code, patterns can be utilized also at the higher levels of software architecture and design. Yacoub et al. [2000] present the POD tool, which is a design environment for visual composition of design patterns for the purpose of developing pattern-oriented designs. Patterns are integrated at the architecture level and traced to lower design lewels; the user can identify the patterns of the participating application classes. The tool has a repository of design patterns that are available for the user as reusable components. However, unlike JavaFrames, the POD tool does not support code generation or traceability between patterns and the source code. Instead, the output is a refined class diagram that can be given to some UML tool to further develop a detailed design. As a tool platform, also JavaFrames can be augmented with higher level design tools (the MADE environment, Subsection 5.2.5).

As an example of a pattern analysis tool, Lange and Nakamura [1995] have implemented a program visualization tool called Program Explorer. The tool provides class- and object-centered visualization of design patterns in C++ programs. Lange and Nakamura suggest that the tool can be used to help the framework user to understand the underlying software architecture and framework. Another example is Pattern-Lint [Sefika et al. 1996], which is used to check that implementation corresponds to the expected set of design patterns. Each pattern is associated with a set of rules and the implementation is verified against these rules. Pattern-Lint represents information as hyperlinked diagrams and graphs that visualize the analyzed program. However, the tool does not provide a methodology to create applications using patterns. Brown K. [1996], in turn, discusses the KT tool that is an automatic reverse engineering tool for detecting design patterns in Smalltalk applications. When compared to JavaFrames, the main difference is that JavaFrames is a platform that provides a solid basis for these kinds of analysis tools. For instance, JavaFrames has a tool to construct framework-specific pattern specifications by analyzing the specialization interface of a Java framework (Pattern Extractor, Subsection 5.2.5).

This dissertation has discussed pattern-based tool support, in which the pattern specifications are given separately from the target programming language. A different approach is to integrate the pattern concept directly into the programming language. For instance, Bosch [1998] suggests that the design patterns are part of the software engineer's paradigm and the programming language should represent the concepts in the paradigm as accurately as possible. Bosch presents LayOM, which is an extended object-oriented language for the explicit representation of design patterns. Eventually, LayOM code is compiled into C++ and the generated code can then be used to construct applications. Bünnig et al. [1999] discuss another design pattern oriented programming model and programming language called PaL. The syntax of PaL is close to Eiffel and the underlying patterns take the form of a class structure. When compared to JavaFrames, the main difference is that separate pattern specifications can be used to document existing software systems, like Java frameworks, without modifying the original source code. Also, it is possible to create different pattern specifications for different users. For instance, novice users could have a simple tutorial-like pattern interface while other users could utilize a more advanced and technical pattern interface to speed up the framework specialization.

9.2 Tool Support for Framework Specialiation

A framework without any supporting tools or documentation is hard to use. Therefore it is tempting to create framework-specific tools to help framework-based application development. Different approaches are discussed in the following subsections:

- *Visual builders*. Subsection 9.2.1.
- *Expert systems*. Subsection 9.2.2.
- *Feature-based framework specialization*. Subsection 9.2.3.
- *Tool supported cookbooks*. Subsection 9.2.4.
- *Using pattern tools in framework specialization*. Subsection 9.2.5.

9.2.1 Visual Builders

Johnson and Foote [1988] define a toolkit as a collection of high level tools that allow the framework user to interact with the framework to configure and construct new applications. Typically, a black-box framework that is specialized by object composition provides a good basis for *visual builder* [Roberts and Johnson 1996; Fayad et al. 1999]. With a sophisticated tool, like a visual builder, the framework user can construct an application almost without programming, for instance, by selecting icons representing standard components and application structures, connecting them graphically and letting the system generate an executable program. Typically, visual builders are used to create the application's user interface, which consists of well-defined user interface components and their interactions.

When compared to JavaFrames, the shortcoming of visual builders is that they do not support white-box reuse, in which the specialization is done by subclassing the abstract base classes of the framework, for example, to create a new component type. Instead, they expect a rich set of ready-to-use components. Particularly, as discussed by Goebl [1999], in the case of visual builders the graphical programming paradigm helps the creation of the application's user interface but it typically fails to ease the implementation of the logic of the application; it is easier to implement the behavior of an application using the underlying implementation language.

9.2.2 Expert Systems

Giarratano and Riley [1989] describe an expert system as a computer system that emulates the decision-making ability of a human expert. It consists of the *user interface* that allows the user and the expert system to communicate, *explanation facility* that explains the reasoning to the user, *knowledge base* containing rules, *working memory* of facts used by the rules, *inference engine* that manages and executes the rules, *agenda* that contains a prioritized list of instantiated rules created by the inference engine, and *knowledge acquisition facility* that allows the user to add knowledge to the system. In a rule-based system, the inference engine determines which rules, if any, are satisfied by the facts. The rules can be driven forward or backward. *Forward chaining* is reasoning from facts to the conclusion. *Backward chaining* is reasoning from the conclusion to be proved, to the facts supporting the conclusion.

Meuter et al. [2001] discuss a rule-based reasoning engine (KAN) to support framework specialization. They suggest that an expert system whose knowledge base consists of the design and reuse knowledge of a specific framework, can interactively guide the framework user and enhance the quality of the reuse process. If the framework user asks help, the system takes control and offers possible specialization recipe alternatives to the user, which then reacts by taking steps to perform one of the recipes. The system uses forward chaining; when certain steps are performed, the system gives further instructions and adapts its advice to the current situation. The system is embedded to a Smalltalk environment (Squeak) and it has been evaluated by designing small expert systems. In the case of a black-box reuse, the specialization is done by answering yes-no questions and multi-valued questions, in which the question seems to be rather low-level and implementation-oriented, like "What is the value of store for theCollection? (objects, characters, integers)". During white-box reuse the system maintains a task-pool (agenda) containing tasks the programmer still has to do. Each time the programmer writes a method or a class the task pool is consulted; performed tasks are removed and possible new tasks are added to the pool.

In principle, expert systems and JavaFrames have much in common. For instance, in JavaFrames the knowledge base is expressed as pattern specifications, inference engine equals pattern engine, agenda corresponds the task list, and the knowledge acquisition facility can be achieved with advanced pattern development tools. The main difference is that JavaFrames is focused on implementing the prepared decisions made by the pattern modeler. Expert systems emulate the decision-making ability of a human expert, while Java-

Frames concentrates on implementing these decisions and tries to minimize the amount of decisions the pattern user has to make.

9.2.3 Feature-Based Framework Specialization

One approach to represent variability is to express variations in terms of *feature models* [Kang et al. 1990; Czarnecki and Eisenecker 2000]. In a feature model, each of the alternative design decisions is represented by a feature and selections of these features are reflected in the design and implementation of the software product. Many authors have suggested that if variability is described with feature models, a software product could be created by selecting and refining customer-specific features for the product [Deursen et al. 2002; Batory et al. 2003; Lago et al. 2004].

Based on feature models, Lago et al. [2004] suggest a tool that traces the required product features through architecture and components to the level of source code. Such a tool must support top-down and bottom-up traceability, from the feature graph down to implementation files and back. They have demonstrated the approach by extending the Together ControlCenter environment [Together 2004] with a tool that uses three abstract levels. Firstly, the product family is modeled as a *product family feature map*, which captures the design time decisions and includes all the features and variation points in the application domain. Secondly, a single product is described at the product level in terms of the subset of features it supports. In this level, the *product feature map* captures these product-specific features and they are translated into the *product component map*, representing the design decisions (or a framework) to implement the features. Thirdly, reusable implementation aspects are used to concretize the features, i.e., to specialize the framework. Traceability is achieved by managing links between these different maps.

9.2.4 Tool Supported Cookbooks

A document containing instructions that describe typical ways to use a framework is often called a *cookbook* [Krasner and Pope 1988]. Correspondingly, individual instructions in a cookbook are called *recipes*. Hence, a cookbook contains numerous recipes which describe in an informal way how to use a framework in order to solve specific problems. The framework's internal design and implementation details are usually omitted in these kinds of instructions.

Another, similar approach is *implementation case* [Pasetti 2002] that describes how functionality for an application in the framework domain can be implemented using the constructs offered by the framework. Besides acting as cookbook recipes, implementation cases can be used as test cases when developing a framework. They have no any specific formalism; rather, they are a mixture of informal language, pseudo-code, and illustrative diagrams.

Pree et al. [1995] outlines the general principles of *active cookbook*, a cookbook-like tool to provide guidance for framework specialization. The tool includes a *knowledge base*, a *rule interpreter*, and a *working memory*. The knowledge base contains the recipes to specialize the framework. The rule interpreter allows the selection of a particular recipe, presents the recipes as hypertext, maintains temporary information accumulated during the interpretation of a recipe in the working memory, and generates the source code of additional or modified framework classes.

The SmartBooks [Ortigosa and Campo 1999; Ortigosa et al. 2000; Pace et al. 2003] system is an agent-based task-oriented system that supports framework instantiation based on the active cookbook concept. With SmartBooks, the framework user expresses her objective by selecting items from a list of options. Based on the selected options, a planning agent then derives a sequence of tasks to specialize the framework in order to implement the selected functionality. The process of selecting functional requirements, generating the specialization plan and executing the corresponding tasks follows a spiral model, in which the framework user can refine or change the required functionality. The planning process itself is based on a predefined format of rules, in which a rule can be specified graphically using a special UML extension. Finally, a generator translates the created rule into a Prolog representation. The SmartBooks method reminds the approach used in JavaFrames; selecting functionality from a list of alternatives in SmartBooks can be seen as selecting suitable pattern specification to be instantiated in JavaFrames. After selecting the alternatives, both approaches use tasks to guide the framework user to create the resulting framework specialization. In fact, besides Fred/JavaFrames, the SmartBooks system was among the first that introduced the task model for framework specialization.

Pace et al. [2003] present the Smartweaver tool based on the SmartBooks method to assist the software developer to use AOP (aspect-oriented programming) frameworks [Constantinides et al. 2000]. In an AOP framework each aspect is expressed as a class in the framework and the tool can be used to specialize these frameworks. Smartweaver provides an UML-based environment where developers can define classes, aspects, and crosscutting relationships among them.

9.2.5 Using Pattern Tools in Framework Specialization

Pattern tools were discussed in Section 9.1. Many of the approaches seem to stress the selection and instantiation of the most suitable design patterns early in the application's implementation phase, rather than helping in the framework specialization. From the viewpoint of the framework user, the specialization problems are typically very application-specific and implementation-oriented. We argue that solutions to these specialization problems should be described as framework-specific patterns (Subsection 2.1.2), which are not general enough to be published in pattern catalogs. Based on these patterns, the pattern modeler creates the pattern specifications to describe the intended framework specialization. By using the pattern specifications and advanced pattern deployment tools, the framework user can utilize the encapsulated instructions when programming her application.

An example of tool-supported framework specialization with a pattern-like concept is discussed by Froelich et al. [1997, 1998]. They present *hooks*, which are points in the framework to be specialized, for instance, by filling in parameters or creating subclasses. For more complex problems hooks can be grouped so that each hook focuses on a smaller subproblem. Hooks are written in a specific format, which includes the description of participating components and the required steps to use the hook. In practice, the framework expert defines a hook as a recipe-like algorithm. This algorithm is intended to be read, interpreted, and carried out by the framework user. Froelich et al. have noticed that the hook concept enables a general tool that can be adapted to different frameworks, instead of developing a custom tool for each framework.

Meijler et al. [1997] discusses the FACE tool (Framework Adaptive Composition Environment), which is used for code generation and pattern instantiation. The tool starts with a primal-schema, containing the abstract classes of a pattern and their associations, then proceeds to a meta-schema for concrete classes, operations, and associations. Thus, the tool has two levels; composing a set of objects and composing a schema that descripes possible structures and cooperations of run-time objects in the target application. A basic element of a schema is a class-component. The class-component is basically an object that has a certain composition interface defining how this particular component can be connected to other components. All FACE components are black-box entities, which are used through parametrization and composition by the application developer. FACE includes also a typechecking mechanism, which prevents incorrect component compositions. Thus, FACE can be seen as a pattern-based visual builder.

Riehle [2000] presents an approach for specifying object-oriented frameworks and their specialization using *role models*. Riehle's role models do not try to replace the class-based object-oriented modeling but rather to refine and improve it. A role model describes object collaborations, in which objects play roles that are described by role types. A class model composes all relevant role models to describe how instances of its classes collaborate. According to

Riehle, describing classes as role type compositions and class models as role model compositions reduce the complexity of class and object collaborations. A framework can define how it should be specialized with the help of role types that describe the roles that clients of a framework have to play to make proper use of the framework. Thus, like framework-specific patterns, role models can be used to represent variability in the level of the framework. However, there seem to be no tool support for role models.

9.3 Architecture-Oriented Software Development Environments

Harrison W. et al. [2000] outlines that besides supporting coding activities, software development environments should also support other major activities, like requirements engineering, specification, design, testing, and analysis. An architecture-oriented software development environment should support software engineering across the software lifecycle and traceability across the artifacts of the software product and design. Various reverse engineering and forward engineering techniques can be used to analyze and construct the software and to represent it in an abstract form. Ideally the software development environment supports traceability and both reverse engineering and forward engineering to keep the design models and implementation consistent.

Different approaches to provide an architecture-oriented software development environment are discussed in the following subsections:

- *UML CASE tools*. Subsection 9.3.1.
- *High level specifications and tools*. Subsection 9.3.2.

9.3.1 UML CASE Tools

UML [Booch et al. 1999; Rumbaugh et al. 1999; UML 2004] is a standard modeling language to design and document object-oriented software systems. Many of the current CASE tools support the use of UML. For instance, Rational Rose [Rational 2004] is one of the market-leading CASE tools to create UML models. It also transforms UML notations into source code in Java and C++. Rational XDE Developer, in turn, is an Eclipse-based UML modeling tool from the same vendor. Clearly, using a common modeling language helps software developer to design, implement, and document software products.

The shortcoming of traditional UML CASE tools is that they offer only limited support for product line architectures and framework specialization. Though they can describe the framework's specialization interface in terms of well-known object-oriented structures, like abstract classes and method signatures, it does not capture the variability of the required specialization. Riva et al. [2004] enumerates the problems of UML CASE tools: they offer no support for specifying various architectural rules, for checking the conformance of a design against architectural conventions, for constructing new designs according to given architectural rules, for managing variability supported by a product line architecture, for creating architectural views from design-level models, and for establishing tracing capabilities between implementation and architectural models. In addition, traditional CASE tools are not intended to be incremental, iterative, and interactive programming environments. Instead, high-level UML models are transformed into source code as a one-shot activity.

To make UML CASE tools to support framework specialization, Fontoura et al. [2000] present UML-F, which is a UML extension to explicitly describe the framework's variation points. They use a UML tagged value (a name-value-pair that can be attached to a modeling element to extend its properties) to identify and document the extension points of the framework. The UML-F descriptions can be executed with a wizard-like tool, in which the framework specialization is considered a straightforward process if only the framework's extension points are clearly designed. Pree et al. [2002], in turn, demonstrate how UML-F can be used as a notation to represent design patterns and product lines. The idea of using a wizard tool to specialize frameworks is close to the approach used in JavaFrames, but in JavaFrames the framework specialization is considered as more fine-grained, gradually proceeding, evolutionary, and interactive process.

To help modeling software architectures in general, Robbins [1999] suggests cognitive support for the software developer in terms of advanced decision making features, like checklists, non-modal wizards, and the dynamic "to do" list. The suggested features are evaluated in the context of the Argo/UML tool [Robbins et al. 1997], which is a design tool using UML notations. The design environment uses *design critics* to give feedback, in which a critic is an agent that watches for a specic condition in the design under construction and advises the designer of potential errors or needed improvements. Thus, designers receive feedback while they are considering individual design decisions and modifying the architecture. The idea of dynamic "to do" list and design critics is close to the approach used in JavaFrames. The main difference is that JavaFrames Eclipse Integration is used to implement prepared design solutions rather than to create new software architecture.

9.3.2 High Level Specifications and Tools

In higher abstraction levels, software architecture can be specified by using an *architecture description language* (ADL) [Bass et al. 1998; Garlan 2000]. These languages provide notations to explicitly describe architectural structures and they support early analysis and feasibility testing of architectural design decisions. Hoek et al. [1999] have noticed that ADLs do not usually support handling of variations within a product family. However, representing the variability at an architectural level rather than at the program code level would be desirable. An ADL should provide means to describe both the common architecture and the variable parts of each product. Examples of such ADLs are Koala [Ommering 1998], Menage [Hoek 2000], and ADLARS [Brown T. et al. 2004].

Harrison W. et al. [2000], in turn, suggest that XML (eXtensible Markup Language) [XML 2004] could be used as a mid-level standard that can enable the creation of more flexibly-deployed and reusable software tools. The main advantage of this approach is that XML is a standard language for representing information having a number of available tools, like viewers, editors, and translators. For instance, Cleaveland [2001] uses XML to represent program specifications.

Eden et al. [2003] outlines two-tier programming to keep architecture and its implementation consistent. They stress that software development is an open-ended, step-wise process of adaptation to a stream of changes. To solve the traceability and design erosion problems, they suggest that a program is defined in two levels of abstraction. Statements in the lower level are made in a traditional programming language (e.g., Java or C++). The second layer consists of design constraints specified in an architectural specification language (e.g., LePUS [Eden 2001, 2002a]). The system includes also association mapping to maintain consistency between the two tiers. The approach resembles JavaFrames. In JavaFrames pattern specifications can be seen as a higher level specification language, in which the pattern engine maintains the assosiation mapping between pattern roles and target elements.

9.4 Approaches to Obtain Pattern Interfaces

So far, our empirical experiences with JavaFrames and large software systems have produced three different approaches to create tool supported pattern interfaces. The goal-oriented approach was explained in Chapter 6. Other methodologies are briefly discussed in the following subsections:

- *Systematic approach*. Formal pattern specifications and pattern interfaces can be constructed by using different reverse engineering techniques.

Based on this idea, the systematic approach enables systematic and semi-automatic mapping of the framework's specialization interface. Subsection 9.4.1.

- *Concern-oriented approach.* The software developer has number of concerns when implementing a software system [Parnas 1972; Hürsch and Lopes 1995; Tarr et al. 1999]. The concern-oriented approach creates a pattern interface to work with those concerns. Subsection 9.4.2.

9.4.1 Systematic Approach

The *systematic approach* [Viljamaa A. 2001; Hakala et al. 2001c; Viljamaa J. 2004] assumes that the framework has a layered structure and its basic concepts are implemented on the highest layer as abstract interfaces; this kind of white-box framework uses inheritance and method overriding as a means to provide extensibility. For such a framework, the systematic approach enables systematic mapping of the specialization interface and it also allows automated support to create pattern specifications, as some heuristics can be used to identify and specify the usage of these interfaces.

For instance, Viljamaa J. [2004] presents an approach that utilizes formal concept analysis (FCA) [Tonella and Antoniol 1999] to produce role-based pattern specifications for defining the extension points of a framework. Pattern Extractor, the tool discussed in Subsection 5.2.5, is a concrete implementation of the FCA-based analysis tool.

When compared to the goal-oriented approach, it has been noticed that the systematic approach is more suitable for automatic pattern construction and framework documentation, upon the condition that the target framework has well-defined layered structure. The goal-oriented approach, in turn, is applicable also in situations where the specialization interface is not clear or the pattern modeler is not throughout familiar with the framework. Rather than semi-automatically mapping the framework's whole specialization interface with some reverse engineering technique, the goal-oriented approach provides solutions to practical specialization problems that address the current goals of the framework user. Thus, the goal-oriented approach is less systematic and more based on intuition and practical experiences.

9.4.2 Concern-Oriented Approach

When planning and programming a software system, the development team and individual software developers has to deal with number of concerns. For example, Hürsch and Lopes [1995] enumerate the following concerns: class organization, synchronization, location control for distributed computation, real-time constraints, and failure recovery. Some of the concerns are dealing with the fundamental algorithm and the basic functionality of the software system, while others can be seen as reasoning to implement more advanced features. Altogether, concerns are abstract things that affect the implementation of the software system; software developers should be able to point out the code and design that deals with each of those concerns.

Hürsch and Lopes [1995] stress that by abstracting concerns out and separating them, programming individual concerns becomes less complex and code can be effectively reused. They call this approach *separation of concerns* (SoC). SoC follows the well-established principle in software engineering to hide complexity by abstraction [Parnas 1972]. The concerns are mapped into the implementation level through programming language constructs. Ideally, at the implementation level, SoC promotes the blocks of code which address the different concerns. In short, the goal of SoC is to decompose and organize a software system so that it becomes easier to create and understand, which, in turn, improves its reuse and maintainability.

However, because of the problems caused by crosscuttings concerns, the object-oriented programming typically suffers from strong coupling between classes. This makes the software system hard to understand and change, making it more difficult to maintain and eventually reducing its reusability and adaptability. As discussed by Sutton and Rouvellou [2002], most programming and modeling formalisms enforce a dominant decomposition that allows only a few concerns to be separated, whereas software in reality is subject to multiple simultaneous, overlapping, and crosscutting concerns.

Many SoC approaches have been proposed, particularly to produce less tangled and less scattered source code [Harrison W. and Ossher 1993; Kiczales et al. 1997; Tarr et al. 1999; Pace and Campo 2001]. Though managing the crosscuttings in various decomposition hierarchies tends to have a central position in these approaches, their ultimate goal is to make the software system more understandable with better separation of concerns in the level of a programming language.

The *concern-oriented approach* [Hammouda et al. 2004c] is based on the idea that with a proper tool support, formal pattern specifications can be used to make explicit the ways in which the concerns can cooperate or interface with each other at the levels of design and implementation. The approach tackles the complexity resulting from scattering and tangling by providing a one-to-one match between concerns and the corresponding patterns; each identified requirement is encapsulated by a separate concern and then the patterns treating the concerns are identified. Because each pattern has roles bound to the concrete implementation elements they simultaneously indicate the elements that are implementing a particular concern.

The goal-oriented approach resembles the concern-oriented approach. However, the concern-oriented approach is intended to describe solutions for rather high level problems. A goal, instead, can be seen as a more practical and atomic issue inside a larger concern. Goal-oriented patterns try to answer very practical specialization problems of the framework user. When using a goal-oriented pattern, the application developer is typically working with some specific implementation detail and wants precise instructions and support to create a particular piece of code.

CHAPTER 10

CONCLUSION

This dissertation has outlined a general tool platform for patterns and concentrated on a tool support to specialize object-oriented frameworks. Requirements of the pattern tool platform and how they have materialized in JavaFrames Eclipse Integration are further discussed in Section 10.1. Contributions of this dissertation are reviewed in Section 10.2. Future work is discussed in Section 10.3. Concluding remarks are given in Section 10.4.

10.1 JavaFrames as a Pattern Tool Platform

JavaFrames Eclipse Integration is a concrete implementation of the pattern tool platform discussed in Chapter 4. Instead of creating separate tool support for each type of pattern and application-domain, such a tool platform provides an universal mechanism to model and instantiate patterns with a common software development environment during the software development process. The platform is integrable and open for extensions; new kinds of pattern semantics and tools can be added to support the more advanced use of patterns.

The requirements of the platform were discussed in Section 4.1. Based on the experiences gained from different use cases and examples, the following compares JavaFrames Eclipse Integration to these requirements:

- *Integrable*. Section 4.6 explained how the pattern tool platform can be integrated into an existing software development environment. As discussed in Section 5.1, JavaFrames is integrable; currently it is integrated into the Eclipse environment.

- *Extensible*. JavaFrames Eclipse Integration can be extended with new pattern tools. Examples of such tools were discussed in Section 5.2. JavaFrames can also be extended with new pattern semantics; this was presented as a case study in Chapter 7.

- *Cohesive*. JavaFrames maintains the connection between the pattern specifications and their instantiations. However, to maintain bindings between very fine-grained roles and code fragments requires detailed parsing of the source code. For example, the current Java pattern semantics does not provide techniques to verify the semantics of a method body, that is, to check the behaviour of the software system. Such a support would require a lot of system resources and advanced parsing techniques. Defining the abstract semantics of a method (e.g., by pre- and post-conditions) and checking the implementation against such specifications is a difficult problem. Also, in the current version of JavaFrames, the binding between a code fragment role and the corresponding block of code is not maintained. One possible solution is to augment the Java pattern semantics with a richer set of statically verifiable constraints.

- *Scalable*. The original motivation of Fred/JavaFrames was to support the framework specialization process in Java. Besides small example frameworks, it has been successfully evaluated with the JHotDraw framework [Viljamaa A. 2001; JHotDraw 2004], Enterprise JavaBeans (EJB) [Hammouda and Koskimies 2002; EJB 2004], and with an industrial use case (see Chapter 8). It has also turned out that the pattern-driven approach has much wider scope than just Java frameworks. For instance, pattern specifications can be created and applied even if the framework is thin if non-existent and the architecture relies on just a set of architectural conventions, like design patterns.

- *Precise and explicit*. So far, the experiences gained from various case studies have been encouraging, showing that the pattern-based approach is sufficiently powerful to define the specialization interface of a real framework, and that the JavaFrames Eclipse Integration – though still a prototype - scales up for industry-sized frameworks. However, it depends on the used pattern semantics how detailed the pattern specifications can be. For instance, atomic implementation details inside method bodies are not currently supported in the Java pattern semantics. This makes it difficult to model solutions that are mainly based on atomic changes in some algorithm or method body.

- *Incremental*. JavaFrames Eclipse Integration supports the use of the pattern specifications as incremental, iterative, and interactive process. As illustrated in Subsection 5.2.2, it provides the pattern user a pattern deployment tool that shows a constantly updated list of instantiation tasks. This task list evolves dynamically during the instantiation and helps the pattern user to apply the pattern specification. Besides providing wizard-like code generation facilities and instructions, the task list shows warnings if JavaFrames detects any violations against the underlying pattern specification. All the time, the instructions, task titles, and code suggestions adapt to the terms and structures of the current application. This piecemeal approach facilitates the understanding of the framework by supporting learning-by-doing.

- *Generative*. JavaFrames Eclipse Integration guides the pattern user to create and modify program elements. Created artifacts and documentation conform to the pattern specifications making the process adaptive. Unlike with other wizards and development environments, the code is not generated as large and static lump; instead, the code generation proceeds piecemeal and adapts to the terms and names selected by the pattern user. Because the code generation proceeds piecemeal, step by step, the pattern user is not overwhelmed by the generated code but can reason the rationale of it. For advanced users, allowing JavaFrames to perform some of the tasks automatically, the system acts like an advanced code generator. The amount of automatically generated code depends on the used pattern specifications and the underlying pattern semantics. The code generation capabilities of the current Java pattern semantics were evaluated in Chapter 7 and Chapter 8.

- *Open-ended*. JavaFrames Eclipse Integration allows the pattern modeler to create a project-specific pattern interface. This pattern interface can be stored and imported to other projects. In JavaFrames, like explained in Section 4.4, the pattern specifications can be refined and partially instantiated so that the instantiation will continue at later time and across different software projects.

10.2 Summary of Contributions

Contributions of this dissertation were enumerated in Section 1.4. The main contributions are the following:

- Participation in the development of the Fred/JavaFrames tool concept.

- A description of a general pattern-based tool support that allows the use of the pattern concept in different software development environments, making the environment architecture-oriented.

- Integration of such a tool platform into the Eclipse environment.
- A goal-oriented process to use the pattern-based tool support to specialize frameworks.
- A specification of the extension interface of the tool platform using the tool itself.
- An evaluation of the pattern-based approach for framework engineering using case studies.

The following summarizes how these contributions answer the problems discussed in Section 1.2:

- *How to teach the software developer to understand different frameworks and design principles in the context of her software product?* As demonstrated with JavaFrames Eclipse Integration, a framework can be annotated with a goal-oriented pattern interface that describes the intended specialization. By using this tool supported pattern interface, the software developer can experiment with the framework. Going through typical specialization tasks illustrates the use of the framework and supports learning-by-doing. In addition, to support the learning process, the platform can be extended with advanced documentation tools, like the Pattern Recorder discussed in Subsection 5.2.4.
- *How to guide the software developer to use frameworks and product line architectures?* A goal-oriented pattern interface can be constructed to help the software developer to achieve her goals with the framework. Various pattern deployment tools can then be used to guide the framework specialization process and to check that the created application obeys the framework-specific rules. The pattern instantiation is done with small and practical programming tasks that are adjusted to the created application. The approach supports both black-box and white-box frameworks.
- *How to maintain and document implemented design solutions and framework specializations?* The pattern tool platform solves the traceability problem by using pattern specifications as a bridge between a framework and its specializations. The occurred pattern instances promote the scattered design solutions inside the application and between the application and the framework. For instance, if an element is changed, it can be checked against the pattern instances it is participating. Further, other elements in those pattern instances can be verified, to ensure that the rules of the implemented design solution are not violated. In addition, because the pattern instances are continuously traced, it is possi-

ble to generate documentation that describes the use of the framework-specific pattern specifications in different applications.

10.3 Future Work

JavaFrames provides a platform to experiment with the pattern concept. JavaFrames Eclipse Integration is also a prototype of a task-driven pattern-based architecture-oriented software development environment. Here are some directions for future work:

- *Different pattern semantics*. So far pattern semantics have been created for Java and UML. However, as explained in Chapter 7, the JavaFrames system can be augmented with new kinds of pattern semantics. One possible direction for future work could be to create different pattern semantics and to study how they could be combined and used in different application domains.

- *Different pattern tools*. As a pattern tool platform, JavaFrames provides a basis for new kinds of pattern tools. Various tools could be created to experiment with the pattern concept and to study different ways to develop, deploy, analyze, and document patterns.

- *Different integrations*. Currently the core of JavaFrames has been implemented in Java and it is integrated into the Eclipse environment. However, the core of the pattern tool platform could be integrated into some other development environment, too.

- *Different case studies*. So far JavaFrames has been mainly used to specialize Java frameworks. New case studies with new kinds of pattern semantics and tools are needed to test the applicability of the pattern tool platform in different application domains and in different phases of the software development process.

10.4 Concluding Remarks

This thesis reaches its goal if the reader gets the idea of patterns and how their use could be supported with a practical, extensible, integrable, and easy-to-use system. The point is that the use of patterns should be as easy as possible, like using a compiler or an interpreter that supervises architectural rules and framework specialization. To enable this pattern-based tool support, the presented system provides the basic infrastructure, like pattern semantics, task automaton, and different pattern tools. By guiding the use of patterns, the system helps the software development team and individual application developers to carry out and speed up the software development process, even if they are not familiar with the pattern concept itself. The pattern modeler, in turn, can use the system, for example, to improve the documentation and usability of a framework.

References

[Akroyd 1996] Akroyd M.: AntiPatterns – Vaccinations Against Object Misuse. In: *Session Notes of Object World West Conference*, San Francisco, August 1996.

[Aksit et al. 1999] Aksit M., Tekinerdogan B., Marcelloni F.: Deriving Frameworks from Domain Knowledge. In: Fayad M., Schmidt D., Johnson R. (eds.): *Building Application Frameworks – Object-Oriented Foundations of Framework Design*. Wiley, 1999.

[Alexander 1979] Alexander C.: *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[Alexander 1981] Alexander C.: *The Linz Café*. Oxford University Press, New York, 1981.

[Alexander 2004] Alexander C.: *The Search for Beauty*. Available at http://hillside.net/patterns/papersbibliographys.htm. August 2004.

[Alexander et al. 1975] Alexander C., Silverstein M., Angel S., Ishikawa S., Abrams D.: *The Oregon Experiment*. Oxford University Press, 1975.

[Alexander et al. 1977] Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I., Angel S.: *A Pattern Language – Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

[Alexander et al. 1985] Alexander C., Davis H., Martinez J., Corner D.: *The Production of Houses*. Oxford University Press, New York, 1985.

[Alexander et al. 1987] Alexander C., Neis H., Anninou A., King I.: *A New Theory of Urban Design*. Oxford University Press, New York, 1987.

[Ambler 1998] Ambler S.: *Process Patterns - Building Large-Scale Systems Using Object Technology*. Cambridge University Press/SIGS Books, 1998.

[Anderson 1990] Anderson J.: *Cognitive Psychology and Its Implications: Third Edition*. W. H. Freeman and Company, New York, 1990.

[Appleton 1997] Appleton B.: Patterns and Software – Essential Concepts and Terminology. *Object Magazine Online*, Vol. 3, No. 5, 1997. Available at http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html. August 2004.

[Atkinson and Griswold 1996] Atkinson D., Griswold W.: The Design of Whole-Program Analysis Tools. In: *Proc. of the 18th International Conference on Software Engineering (ICSE 1996)*, Berlin, Germany, March 1996. IEEE Computer Society, 1996, 16-27.

[Bass et al. 1998] Bass L., Clements P., Kazman R.: *Software Architecture in Practice*. Addison-Wesley, 1998.

[Batory et al. 2003] Batory D., Sarvela J., Rauschmayer A.: Scaling Stepwise Refinement. In: *Proc. of the 25th International Conference on Software Engineering (ICSE 2003)*, Portland, Oregon, USA, May 2003. IEEE Computer Society, 2003. 187-197.

[Beck and Cunningham 1987] Beck K., Cunningham W.: Using Pattern Languages for Object-Oriented Programs. *Technical report, Tektronix Inc.*, 1987. Presented at the OOPSLA-87 Workshop on Specification and Design for ObjectOriented Programming. 1987.

[Beck and Johnson 1994] Beck K., Johnson R.: Patterns Generate Architectures. In: *Proc. of the 8th European Conference on Object-Oriented Programming (ECOOP 1994)*, Bologna, Italy, July, 1994. LNCS 821, Springer, 1994. 139-149.

[Bonnet 1999] Bonnet S.: *Java MVC++ Framework for NMS GUI Applications*. Master of Science Thesis, Department of Information Technology, Tampere University of Technology, 1999.

[Booch 1994] Booch G.: *Object-Oriented Analysis and Design with Applications* (second edition). Benjamin/Cummings, 1994.

[Booch et al. 1999] Booch G., Rumbaugh J., Jacobson I.: *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[Bosch 1998] Bosch J.: Design Patterns as Language Construct. *Journal of Object-Oriented Programming*, Vol. 11, No. 2, May 1998. 18-32.

[Bosch 2000] Bosch J.: *Design and Use of Software Architectures – Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[Bosch 2003] Bosch J.: *Software Architecture*. Book manuscript, October 2003.

[Brown K. 1996] Brown K.: *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk*. Master of Science Thesis, North Carolina State University, 1996

[Brown T. et al. 2004] Brown T., Spence I., Kilpatrick P.: A Relational Architecture Description Language for Software Families. In: *Proc. of the 5th International Workshop* on *Software Product-Family Engineering (PFE 2003)*, Siena, Italy, November, 2003. LNCS 3014, Springer, 2004. 282-295.

[Brown W. et al. 1998] Brown W., Malveau R., McCormic H., Mowbray T.: *AntiPatterns – Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.

[Budinsky et al. 1996] Budinsky F., Finnie M., Vlissides J., Yu P.: Automatic Code Generation from Design Patterns. *IBM Systems Journal*, Vol. 35, No. 2, 1996, 151-171.

[Buschmann et al. 1996] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: *A System of Patterns - Pattern-Oriented Software Architecture*. Wiley, 1996.

[Bünnig et al. 1999] Bünnig S., Forbrig P., Lämmel R., Seemann N.: A Programming Language for Design Patterns. In: *Proc. of the Informatik '99 - Informatik überwindet Grenzen, 29. Jahrestagung der Gesellschaft für Informatik*, Paderborn, Germany, October 1999. Informatik aktuell, Springer, 1999. 400-409.

[Campbell et al. 1992] Campbell R., Islam N., Madany P.: Choices, Frameworks and Refinement. *Computing Systems*, vol. 5, no. 3, 1992. 217-257.

[Carroll 1990] Carroll J.: *The Nurnberg Funnel - Designing Minimalist Instruction for Practical Computer Skill*. Massachusetts Institute of Technology, 1990.

[Chikofsky and Cross 1990] Chikofsky E., Cross II J.: Reverse Engineering and Design Recovery: a Taxonomy. *IEEE Software*, Vol. 7, No. 1, January 1990, 13-17.

[Cleaveland 2001] Cleaveland J.: *Program Generators with XML and Java*. Prentice-Hall, 2001.

[Clements and Northrop 2001] Clements P., Northrop L.: *Software Product Lines – Practices and Patterns*. Addison-Wesley, 2001.

[Conklin 1987] Conklin J.: Hypertext: An Introduction and Survey. *IEEE Computer*, Vol. 20, No. 9, September 1987, 17-41.

[Constantinides et al. 2000] Constantinides C., Bader A., Elrad T., Fayad M., Netinant P.: Designing an Aspect-Oriented Framework in an Object-Oriented Environment. *ACM Computing Surveys*, Vol. 32, Issue 1es, Article No. 41, March 2000.

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

[Coplien 1995] Coplien J.: A Generative Development-Process Pattern Language. In: Coplien J., Schmidt D. (eds.): *Pattern Languages of Program Design*. Addison-Wesley, 1995, 183-237.

[Coplien 1996] Coplien J.: *Software Patterns*. SIGS, New York, 1996.

[Coplien 2004] Coplien J.: *A Pattern Definition*. Available at http://hillside.net/patterns/definition.html. August 2004.

[Coplien and Schmidt 1995] Coplien J., Schmidt D. (eds.): *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[CORBA 2004] CORBA WWW site. Available at http://www.corba.org/. August 2004.

[Cornils 2001] Cornils A.: *Patterns in Software Development*. Ph.D. Thesis, Department of Computer Science, University of Aarhus, Denmark, November 2001.

[Czarnecki and Eisenecker 2000] Czarnecki K., Eisenecker U.: *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, 2000.

[Demeyer et al. 1997] Demeyer S., Meijler T., Nierstrasz O., Steyaert P.: Design Guidelines for Tailorable Frameworks. *Communications of the ACM*, Vol. 40, No. 10, October 1997, 60-64.

[Deursen et al. 2002] van Deursen A., de Jonge M., Kuipers T.: Feature-Based Product Line Instantiation Using Source-Level Packages. In: *Proc. of the 2nd International Conference on Software Product Lines (SPLC 2000)*, San Diego, CA, USA, August 2002. LNCS 2379, Springer, 2002, 217-234.

[Eclipse 2004] Eclipse WWW site. Available at http:/www.eclipse.org. August 2004.

[Eden 2001] Eden A.: Formal Specification of Object-Oriented Design. In: *Proc. of the International Conference on Multidisciplinary Design in Engineering (CSME-MDE 2001)*, Montreal, Canada, November 2001.

[Eden 2002a] Eden A.: A Visual Formalism for Object-Oriented Architecture. In: *Proc. of the 6th World Conference on Integrated Design and Process Technology (IDPT 2002)*, Pasadena, California, USA, June 2002.

[Eden 2002b] Eden A.: A Theory of Object-Oriented Design. *Information Systems Frontiers*, Vol. 4, No. 4, November—December 2002, 379-391.

[Eden et al. 2003] Eden A., Kazman R., Fox C.: *Two-Tier Programming*. Technical report CSM-387, Department of Computer Science, University of Essex, July 2003.

[Egyed and Balzer 2004] Egyed A., Balzer B.: Integrating COTS Software into Systems through Instrumentation and Reasoning. *Journal on Automated Software Engineering (JASE)*, accepted for publication. Available at http://sunset.usc.edu/~aegyed/publications.html. August 2004.

[EJB 2004] EJB WWW site. Available at http://java.sun.com/products/ejb/. August 2004.

[Fayad et al. 1999] Fayad M., Schmidt D., Johnson R., (eds.): *Building Application Frameworks – Object-Oriented Foundations of Framework Design*. Wiley, 1999.

[Florijn et al. 1997] Florijn G., Meijers M., van Winsen P.: Tool Support for Object-Oriented Patterns. In: *Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, Jyväskylä, Finland, June 1997. LNCS 1241, Springer, 1997, 472-496.

[Fontoura et al. 2000] Fontoura M., Pree W., Rumpe B.: UML-F – A Modeling Language for Object-Oriented Frameworks. In: *Proc. of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, Sophia Antipolis and Cannes, France, June 2000. LNCS 1850, Springer, 2000, 63-83.

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

[Froehlich et al. 1997] Froehlich G., Hoover H., Liu L., Sorenson P.: Hooking into Object-Oriented Application Frameworks. In: *Proc. of the of 19th International Conference on Software Engineering (ICSE 1997)*, Boston, Massachusetts, USA, May 1997. IEEE Press, 1997, 491-501.

[Froelich et al. 1998] Froelich G., Hoover H., Liu L., Sorenson P.: *Requirements for a Hooks Tool*. Available at http://www.cs.ualberta.ca/~softeng/papers/ssr04.pdf. August 2004.

[Fuggetta 2000] Fuggetta A.: Software Process: A Roadmap. In: *Proc. of the 22nd International Conference on Software Engineering (ICSE 2000), Future of Software Engineering Track*, Limerick, Ireland, June 2000. ACM Press, 2000, 25-34.

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

[Garlan 2000] Garlan D.: Software Architecture: a Roadmap. In: *Proc. of the 22nd International Conference on Software Engineering (ICSE 2000), Future of Software Engineering Track*, Limerick, Ireland, June 2000. ACM Press, 2000, 91-101.

[Garlan et al. 1995] Garlan D., Allen R., Ockerbloom J.: Architectural Mismatch or Why it's Hard to Build Systems out of Existing Parts. In: *Proc. of the 17th International Conference on Software Engineering (ICSE 1995)*, Seattle WA, April 1995. IEEE Computer Society Press, 1995, 179-185.

[Giarratano and Riley 1989] Giarratano J., Riley G.: *Expert Systems - Principles and Programming*. PSW-KENT Publishing Company, 1989.

[Goebl 1999] Goebl W.: A Survey and a Categorization Scheme of Automatic Programming Systems. In: *Proc. of the 1st International Conference on Generative and Component-Based Software Engineering (GCSE 1999)*. Erfurt, Germany, September 1999. LNCS 1799, Springer, 1999, 1-15.

[Gurp and Bosch 2002] van Gurp J., Bosch J.: Design Erosion: Problems & Causes. *Journal of Systems and Software*, Vol. 61, Issue 2, 2002, 105-119.

[Gurp et al. 2001] van Gurp J., Bosch J., Svahnberg M.: On the Notion of Variability in Software Product Lines. In: *Proc. of the Working IEEE / IFIP Conference on Software Architecture (WICSA 2001)*, Amsterdam, The Netherlands, August 2001. IEEE Computer Society, 2001, 45-55.

[Hakala 2002] Hakala M.: Feature Models, Pattern Languages and Software Patterns: Towards a Unified Approach. In: *Proc. of the 10th Nordic Workshop on Software Development Tools and Techniques (NWPER 2002)*, Copenhagen, August 2002. IT University of Copenhagen, 2002, 189-201.

[Hakala et al. 2001a] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: *Task-Driven Specialization Support for Object-Oriented Frameworks*. Tampere University of Technology, Software Systems Laboratory, Report 22, February 2001. ISBN 952-15-0546X.

[Hakala et al. 2001b] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Architecture-Oriented Programming Using FRED. In: *Proc. of the of 23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Canada, May 2001. IEEE Computer Society, 2001, 823-824 (Formal research demo).

[Hakala et al. 2001c] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Annotating Reusable Software Architectures with Programming Patterns. In: *Proc. of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, The Netherlands, August 2001. IEEE Computer Society, 2001, 171-180.

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

[Hakala et al. 2003] Hakala M., Hautamäki J., Koskimies K., Savolainen P.: Generating Pattern-Based Web Tutorials for Java Frameworks. In: *Proc. of the Scientific Engineering for Distributed Java Applications (FIDJI 2002), International Workshop*. Luxembourg, November 2002. LNCS 2604, Springer, 2003, 99-110.

[Hammouda 2005] Hammouda I.: A Tool Infrastructure for Model-Driven Development Using Aspectual Patterns. In: Beydeda S., Book M., Gruhn V. (eds.): *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*. Springer, 2005.

[Hammouda and Koskimies 2002] Hammouda I., Koskimies K.: Generating a Pattern-Based Application Development Environment for Enterprise JavaBeans. In: *Proc. of the 26th International Computer Software and Applications Conference (COMPSAC 2002)*, Oxford, England, August 2002. IEEE Computer Society, 2002, 856-866.

[Hammouda et al. 2004a] Hammouda I., Katara M., Koskimies K.: A Tool Environment for Aspectual Patterns in UML. In: *Proc. of the Workshop on Directions in Software Engineering Environments (WoDiSEE 2004)*, Edinburgh, Scotland, UK, May 2004. IEE, 2004, 58-65.

[Hammouda et al. 2004b] Hammouda I., Guldogan O., Koskimies K., Systä T.: Tool-Supported Customization of UML Class Diagrams for Learning Complex Systems. In: *Proc. of the 12th International Workshop on Program Comprehension (IWPC 2004)*, Bari, Italy, June 2004. IEEE Computer Society, 2004, 24-33.

[Hammouda et al. 2004c] Hammouda I., Koskinen J., Pussinen M., Katara M., Mikkonen T.: Adaptable concern-based framework specialization in UML. In: *Proc. of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, Linz, Austria, September 2004. IEEE Computer Society, 2004, 78-87.

[Hammouda et al. 2005] Hammouda I., Hautamäki J., Pussinen M., Koskimies K.: Managing Variability Using Heterogeneous Feature Variation Patterns. Accepted in: *Fundamental Approaches to Software Engineering (FASE 2005)*, Edinburgh, Scotland, April 2005.

[Hamu and Fayad 1998] Hamu D., Fayad M.: Achieve Bottom-Line Improvements with Enterprise Frameworks. *Communications of the ACM*, Vol. 41, No. 8, August 1998.

[Harrison N. and Coplien 2004] Harrison N., Coplien J.: *Organizational Patterns of Agile Software Development*. Manuscript. Available at http:/www.easycomp.org/cgi-bin/OrgPatterns?BookOutline. August 2004.

[Harrison N. et al. 1999] Harrison N., Foote B., Rohnert H. (eds.): *Pattern Languages of Program Design 4*. Addison-Wesley, 1999.

[Harrison W. and Ossher 1993] Harrison W., Ossher H.: Subject-Oriented Programming (A Critique of Pure Objects). In: *Proc. of The 8th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1993)*, Washington, DC, USA, September 1993. SIGPLAN Notices, Vol. 28, No. 10, October 1993, 411-428.

[Harrison W. et al. 2000] Harrison W., Ossher H., Tarr P.: Software Engineering Tools and Environments: A Roadmap. In: *Proc. of the 22nd International Conference on Software Engineering (ICSE 2000), Future of Software Engineering Track*, Limerick, Ireland, June 2000. ACM Press, 2000, 261-277.

[Hautamäki 2002] Hautamäki J.: *Task-Driven Framework Specialization – Goal-Oriented Approach*. Licentiate thesis, University of Tampere, Department of Computer and Information Sciences, 2002. Report A-2002-9.

[Helm 1995] Helm R.: Patterns in practice. In: *Proc. of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1995)*. SIGPLAN Notices, Vol. 30, No. 10, October 1995, 337-341.

[Hillside 2004] Hillside Patterns Library WWW site. Available at http://hillside.net/. August 2004.

[Hoek 2000] van der Hoek A.: Capturing Product Line Architectures. In: *Proc. of the 4th International Software Architecture Workshop (ISAW-4)*, Limerick, Ireland, June 2000. ACM Press, 2000, 95-99.

[Hoek et al. 1999] van der Hoek A., Heimbigner D., Wolf A.: *Capturing Architectural Configurability: Variants, Options, and Evolution*. University of California, Irvine, Department of Information and Computer Science, Technical Report CU-CS-895-99, December 1999.

[Hüni et al. 1995] Hüni H., Johnson R., Engel R.: Framework for Network Protocol Software. In: *Proc. of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1995)*. Austin, TX, October 1995. SIGPLAN Notices, Vol. 30, No. 10, October 1995, 358-369.

[Hürsch and Lopes 1995] Hürsch W., Lopes C.: *Separation of Concerns*. Technical Report NU-CSS-95-03, College of Computer Science, Northeastern University, Boston, Massachusetts, February 1995.

[IEEE 2000] Institution of Electrical and Electronics Engineers: *IEEE Recommended Practice for Architectural Description of Software Intensive Systems.* New York, NY, USA, 2000. IEEE Standard 1471-2000.

[J2EE 2004] J2EE WWW site. Available at http://java.sun.com/j2ee/. August 2004.

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

[Jacobson et al. 1999] Jacobson I., Rumbaugh J., Booch G.: *The Unified Software Development Process*. Addison-Wesley, 1999.

[JavaFrames 2004] JavaFrames / Fred WWW site. Available at http://practise.cs.tut.fi/fred/. August 2004.

[Jazayeri et al. 2000] Jazayeri M., Ran A., van der Linden F.: *Software Architecture for Product Families – Principles and Practice*. Addison-Wesley, 2000.

[JHotDraw 2004] JHotDraw WWW site. Available at Internet: http:/members.pingnet.ch/gamma/JHD-5.1.zip. August 2004.

[Johnson 1992] Johnson R.: Documenting Frameworks Using Patterns. In: *Proc. of the 7th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1992)*, Vancouver, Canada, October 1992. SIGPLAN Notices, Vol. 27, No. 10, October 1992, 63-76.

[Johnson 1993] Johnson R.: How to Design Frameworks. In: *Tutorial Notes of the 8th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1993)*, Washington, DC, USA, September 1993. Available at ftp://st.cs.uiuc.edu/pub/papers/frameworks/OOPSLA93-frmwk-tut.ps. August 2004.

[Johnson 1997] Johnson R.: Frameworks = (Components + Patterns). *Communications of the ACM*, Vol. 40, No. 10, 39-42.

[Johnson and Foote 1988] Johnson R., Foote B.: Designing Reusable Classes. *Journal of Object-Oriented Programming*, Vol. 1, No. 5, June/July 1988, 22-35.

[Johnson and Russo 1991] Johnson R., Russo V.: *Reusing Object-Oriented Design*. Technical Report UIUCDCS 91-1696, University of Illinois, 1991.

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

[Keller et al. 1999] Keller R., Schauer R, Robitaille S., Pag P.: Pattern-based Reverse-engineering of Design Components. In: *Proc. of the 21st International Conference on Software Engineering (ICSE 1999)*, Los Angeles, USA, 1999. IEEE Computer Society Press, 1999, 226-235.

[Kellomäki and Mikkonen 2000] Kellomäki P., Mikkonen T.: Design Templates for Collective Behavior. In: *Proc of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, Sophia Antipolis and Cannes, France, June 2000. LNCS 1850, Springer, 2000, 277-295.

[Kiczales et al. 1997] Kiczales G., Lamping J., Menhdhekar A., Maeda C., Lopes C., Loingtier J., Irwin J.: Aspect-oriented programming. In: *Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, Jyväskylä, Finland, June 1997. LNCS 1241, Springer, 1997, 220-242.

[Kim and Benner 1996] Kim J., Benner K.: An Experience Using Design Patterns - Lessons Learned and Tool Support. *Theory and Practice of Object Systems (TAPOS)*, Vol. 2, No. 1, 1996, 61-74.

[Koskimies and Mössenböck 1995] Koskimies K., Mössenböck H.: Designing a Framework by Stepwise Generalization. In: *Proc. of the 5th European Software Engineering Conference (ESEC 1995)*, Sitges, Spain, September 1995. LNCS 989, Spinger, 1995. 479-497.

[Krasner and Pope 1988] Krasner G., Pope S.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. In: *Journal of Object-Oriented Programming*, August/September, 1988, 26-49.

[Lago et al. 2004] Lago P., Niemelä E., van Vliet H.: Tool Support for Traceable Product Evolution. In: *Proc. of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, Tampere, Finland, March 2004. IEEE Computer Society, 2004, 261-269.

[Lamsweerde 2000] van Lamsweerde A.: Formal Specification: A Roadmap. In: *Proc. of the 22nd International Conference on Software Engineering (ICSE 2000), Future of Software Engineering Track*, Limerick, Ireland, June 2000. ACM Press, 2000, 147-160.

[Lange and Nakamura 1995] Lange D., Nakamura Y.: Interactive Visualization of Design Patterns Can Help in Framework Understanding. In: *Proc. of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1995)*, Austin, Texas, USA, October 1995. SIGPLAN Notices, Vol. 30, No. 10, October 1995, 342-357.

[Lea 1994] Lea D.: Christopher Alexander – An Introduction for Object-Oriented Designers. *ACM SIGSOFT Software Engineering Notes*, Vol. 19, No. 1, 1994, 39-46.

[Martin 1995] Martin R.: Discovering Patterns in Existing Applications. In: Coplien J., Schmidt D. (eds.): *Pattern Languages of Program Design*. Addison-Wesley, 1995, 365-393.

[Martin et al. 1998] Martin R., Riehle D., Buschmann F. (eds.): *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.

[Meijler et al. 1997] Meijler T., Demeyer S., Engel R.: Making Design Patterns Explicit in FACE – A Framework Adaptive Composition Environment. In: *Proc. of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE 1997)*, Zurich, Switzerland, September 1997. LNCS 1301, Springer, 1997, 94-111.

[Meszaros and Doble 1998] Meszaros G., Doble J.: A Pattern Language for Pattern Writing. In: Martin R., Riehle D., Buschmann F. (eds.): *Pattern Languages of Program Design 3*. Addison-Wesley, 1998, 529-574.

[Meuter et al. 2001] Meuter W., D'Hondt M., Goderis S., D'Hondt T.: Reasoning with Design Knowledge for Interactively Supporting Framework Reuse. In: *Proc. of the 2nd Workshop on Soft Computing Applied to Software Engineering (SCASE 2001)*, Enschede, The Netherlands, February 2001.

[Mikkonen 1998] Mikkonen T.: Formalizing Design Patterns. In: *Proc. of the 20th International Conference on Software Engineering (ICSE 1998)*, Kyoto, Japan, April 1998. IEEE Computer Society, 1998, 115-124.

[Mikkonen and Pruuden 2001] Mikkonen T., Pruuden P.: Practical Perspectives on Software Architectures, High-level Design, and Evolution. In: *Proc. of the 4th International Workshop on Principles of Software Evolution (IWPSE 2001)*, Vienna University of Technology, Austria, September 2001. ACM Press, 2001, 122-125.

[Miller 1956] Miller G.: The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychology Review*, Vol. 63, 1956, 81-97.

[Niere et al. 2002] Niere J., Schfer W., Wadsack J., Wendehals L., Welsh J.: Towards Pattern-Based Design Recovery. In: *Proc. of the 24th International Conference on Software Engineering (ICSE 2002)*. Orlando, Florida, USA, May 2002. ACM Press, 2002, 338-348.

[Odenthal and Quibeldey-Cirkel 1997] Odenthal G., Quibeldey-Cirkel K.: Using patterns for design and documentation. In: *Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, Jyväskylä, Finland, June 1997. LNCS 1241, Springer, 1997, 511-529.

[Ommering 1998] van Ommering R.: Koala, a Component Model for Consumer Electronics Product Software. In: *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, Las Palmas de Gran Canaria, Spain, February 1998. LNCS 1429, Springer, 1998, 76-86.

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

[Ortigosa et al. 2000] Ortigosa A., Campo M., Salomon R.: Towards Agent-Oriented Assistance for Framework Instantiation. In: *Proc. of the 15th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*, Minneapolis, Minnesota, October 2000. SIGPLAN Notices, Vol. 35, No. 10, October 2000, 253-263.

[Pace and Campo 2001] Pace A., Campo M.: An Empirical Study about Separation of Concerns Approaches. In: *Proc of the 2nd Argentine Symposium on Software Engineering (ASSE 2001), as part of the 30th Argentine Conference on Computer Science and Operational Research (JAIIO 2001)*, Buenos Aires, Argentina, September 2001.

[Pace et al. 2003] Pace J., Trilnik F., Campo M.: Assisting the Development of Aspect-Based Multi-Agent Systems Using the Smartweaver Approach. In: *Proc of the 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2002)*, Orlando, Florida, USA, May 2002. LNCS 2603, Springer, 2003, 165-181.

[Parnas 1972] Parnas D.: On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, Vol. 15, No. 12, December 1972, 1053-1058.

[Pasetti 2002] Pasetti A.: *Software Frameworks and Embedded Control Systems*. LNCS 2231, Springer, 2002.

[Peltonen and Selonen 2004] Peltonen J., Selonen P.: An Approach and a Platform for Building UML Processing Tools. In: *Proc. of the Workshop on Directions in Software Engineering Environments (WoDiSEE 2004)*, Edinburgh, Scotland, UK, May 2004. IEE, 2004.

[Portland 2004] Portland Pattern Repository WWW site. Available at http://c2.com/ppr/. August 2004.

[Perry and Wolf 1992] Perry D., Wolf A.: Foundations for the Study of Software Architecture. *Software Engineering Notes*, Vol. 17, No. 4, October 1992, 40-52.

[Pree 1994] Pree W.: Meta Patterns – A Means of Capturing the Essential of Reusable Object Oriented Design. In: *Proc. of the 8th European Conference on Object-Oriented Programming (ECOOP 1994)*, Bologna, Italy, July 1994. LNCS 821, Springer, 1994, 150-162.

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

[Pree and Koskimies 1999] Pree W., Koskimies K.: Framelets - Small is Beautiful. In: Fayad M., Schmidt D., Johnson R. (eds.): *Building Application Frameworks - Object-Oriented Foundations of Framework Design*. Wiley, 1999, 411-414.

[Pree et al. 1995] Pree W., Pomberger G., Schappert A., Sommerlad P.: Active Guidance of Framework Development. *Software-Concepts and Tools*, Vol. 16, No. 3, 136-145, 1995.

[Pree et al. 2002] Pree W., Fontoura M., Rumpe B.: Product Line Annotations with UML-F. In: *Proc. of the 2nd International Software Product Lines Conference (SPLC 2002)*, San Diego, California, USA, August 2002. LNCS 2379, Springer, 2002, 188-197.

[Rational 2004] Rational Rose WWW site. Available at http://www-136.ibm.com/developerworks/rational/products/rose. August 2004.

[Riehle 1997] Riehle D.: Composite Design Patterns. In: *Proc. of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1997)*, Atlanta, Georgia, October 1997. SIGPLAN Notices, Vol. 32, No. 10, October 1997, 218-228.

[Riehle 2000] Riehle D.: *Framework Design – A Role Modeling Approach*. Ph.D. Thesis, ETH Zürich, Institute of Computer Systems, February 2000.

[Rising 2000] Rising L.: *The Pattern Almanac 2000*. Addison-Wesley, 2000.

[Riva et al. 2004] Riva C., Selonen P., Systä T., Tuovinen A., Xu J., Yang Y.: Establishing a Software Architecting Environment. In: *Proc. of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, Oslo, Norway, June 2004. IEEE Computer Society, 2004, 188-200.

[Robbins 1999] Robbins J.: *Cognitive Support Features for Software Development Tools*. Ph.D. Thesis, Information and Computer Science, University of California, Irvine, September 1999. Technical Report UCI-ICS-99-39.

[Robbins et al. 1997] Robbins J., Hilbert D., Redmiles D.: Extending Design Environments to Software Architecture Design. In: *Proc. of the 11th Knowledge-Based Software Engineering Conference (KBSE 1996)*, Syracuse, New York, USA, September 1996. IEEE Computer Society, 1996, 63-72.

[Roberts and Johnson 1996] Roberts D., Johnson R.: Evolving Frameworks - A Pattern Language for Developing Object-Oriented Frameworks. In: *Proc. of, the 3rd Conference on Pattern Languages and Programming (PLoP 1996)*, Allerton Park, IL, September 1996.

[Ruhe 2000] Ruhe G.: Methodological Contributions to Professional Education and Training. In: *Proc. of the 24th International Computer Software and Applications Conference (COMPSAC 2000)*, Taipei, Taiwan, October 2000. IEEE Computer Society, 2000, 11-16.

[Rumbaugh et al. 1999] Rumbaugh J., Jacobson I., Booch G.: *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[Savolainen 2003] Savolainen P.: *Ohjelmistokehysten erikoistamistutoriaalit Fred-ympäristössä*. Diplomityö, Tietotekniikan osasto, Tampereen teknillinen yliopisto, helmikuu 2003.

[Schmidt 1997] Schmidt D.: Applying Design Patterns and Frameworks to Develop Object-Oriented Communications Software. In: Peter Salus (eds.): *Handbook of Programming Languages*, Vol. I, Macmillan Computer Publishing, 1997.

[Schmidt and Buschmann 2003] Schmidt D., Buschmann F.: Patterns, Frameworks, and Middleware: Their Synergistic Relationships. In: *Proc. of the 25th International Conference on Software Engineering (ICSE 2003)*, Portland, Oregon, USA, May 2003. IEEE Press, 2003, 694-704.

[Sebesta 1999] Sebesta R.: *Concepts of Programming Languages*. Addison-Wesley, 1999.

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

[Shaw and Garlan 1996] Shaw M., Garlan D.: *Software Architecture - Perspectives on an Emerging Discipline*. Upper Saddle River, NJ, Prentice Hall, 1996.

[Smith and Williams 2002] Smith C., Williams L.: *Performance Solutions – A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[Soukup 1995] Soukup J.: Implementing Patterns. In: Coplien J., Schmidt D. (eds.): *Pattern Languages of Program Design*. Addison-Wesley, 1995, 395-412.

[Sutton and Rouvellou 2002] Sutton S., Rouvellou I.: Modeling of Software Concerns in Cosmos. In: *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*. Enschede, The Netherlands, April 2002. ACM Press, 2002, 127-133.

[Taligent 1994] A Taligent white paper: *Building Object-Oriented Frameworks*. Available at http://lhcb-comp.web.cern.ch/lhcb-comp/Components/postscript/buildingoo.pdf. August 2004.

[Tarr et al. 1999] Tarr P., Ossher H., Harrison W., Sutton J.: N Degrees of Separation: Multidimensional Separation of Concerns. In: *Proc. of the 21st International Conference on Software Engineering (ICSE 1999)*, Los Angeles, CA, USA, May 1999. ACM Press, 1999, 107-119.

[Together 2004] Together ControlCenter WWW site. Available at http://www.borland.com/together/. August 2004.

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

[UML 2004] Unified Modeling Language WWW site. Available at http://www.uml.org/. August 2004.

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

[Viljamaa A. 2004] Viljamaa A.: *Specification of Framework Reuse Interfaces for Task-Oriented Framework Specialization*. Ph.D. Thesis, Unpublished manuscript, Department of Computer Science, University of Helsinki, September 2004.

[Viljamaa J. 1997] Viljamaa J.: *Tools Supporting the Use of Design Patterns in Frameworks*. Report C-1997-25, University of Helsinki, Department of Computer Science, 1997.

[Viljamaa J. 2002] Viljamaa J.: *Automatic Extraction of Framework Specialization Patterns*. Licentiate thesis, University of Helsinki, Department of Computer Science, 2002. Report C-2002-47.

[Viljamaa J. 2003] Viljamaa J.: Reverse Engineering Framework Reuse Interfaces. In: *Proc. of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, Helsinki, Finland, September 2003. ACM Press, 2003, 217-226.

[Viljamaa J. 2004] Viljamaa J.: *Applying Formal Concept Analysis to Extract Framework Reuse Interface Specifications from Source Code*. Ph.D. Thesis, Department of Computer Science, University of Helsinki, 2004.

[Vlissides 1997] Vlissides J.: Patterns: The Top Ten Misconceptions. *Object Magazine*, Vol. 7, No. 1, March 1997, 30-33.

[Vlissides et al. 1996] Vlissides J., Coplien J., Kerth N. (eds.): *Pattern Languages of Program Design 2*. Reading, MA, Addison-Wesley, 1996.

[Wild 1996] Wild F.: Instantiating Code Patterns — Patterns Applied to Software Development. *Dr. Dobb's Journal*, Vol 21., No 6., June 1996, 72-76.

[XML 2004] XML WWW site. Available at http://www.w3.org/XML/. August 2004.

[Yacoub et al. 2000] Yacoub S., Xue H., Ammar H.: POD: A Composition Environment for Pattern-Oriented Design. In: *Proc. of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 2000)*, Santa Barbara, California, July-August 2000. IEEE Computer Society Press, 2000. 263-272.

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

Appendix A:
Abstract Factory Pattern Specification

The Abstract Factory pattern was discussed in Subsection 2.1.4 and in Subsection 4.3.4. The pattern was outlined in Figure 5. A more detailed specification is given here. The title of each role and constraint is underlined. The role's multiplicity is shown after the role title: "?" for [0..1], "*" for [0..*], and "+" for [1..*]. By default the multiplicity is 1. Child roles are intended under their parent role. Role's properties and dependencies (except the implicit parent-child dependency) are listed under the role name. A complete reference on the used Java pattern semantics, its role types, and properties can be found from the JavaFrames documentation [JavaFrames 2004].

AbstractFactory: Java class role
Dependencies:
None
Properties:
defaultTemplate: public abstract class <#:roleName> { }
description: This class declares an interface for operations that create abstract product objects.
taskTitle: Provide the abstract factory class to create products
taskDescription: Provide a Java type for the role <#:roleName> <p><#:description></p>

> **CreateProduct: Java method role**
> **Dependencies:**
> /AbstractProduct
> **Properties:**
> *defaultTemplate*: public abstract <#:/AbstractProduct.i.name> create<#:/AbstractProduct.i.name>();
> *description:* This method is used to create <#:/AbstractProduct.i.name> objects. Implement this method in subclasses.
> *taskTitle*: Provide a method to create new <#:/AbstractProduct.i.name> objects
> *taskDescription:* Provide a Java method for the role <#:roleName> <p><#:description></p>
>
> > **ReturnAP: Return type constraint**
> > **Dependencies:**
> > None
> > **Properties:**
> > *value:* /AbstractProduct.i

AbstractProduct: Java class role *
Dependencies:
None
Properties:
defaultTemplate: public abstract class <#:roleName> { }
description: This class declares an interface for a type of product object.
taskTitle: Provide a new product type
taskDescription: Provide a Java type for the role <#:roleName> <p><#:description></p>

ConcreteProduct: Java class role
Dependencies:
/AbstractProduct
/ProductFamily

Properties:
defaultTemplate: public class <#:roleName> extends <#:/AbstractProduct.i.longName> { }
description: This class inherits <#:/AbstractProduct.i.name> and defines a <#:/ProductFamily.name> product object to be created by the corresponding concrete factory.
taskTitle: Provide a new <#:/AbstractProduct.i.name> product for the <#:/ProductFamily.name> product family
taskDescription: Provide a Java type for the role <#:roleName> <p><#:description></p>

Constructor: Java constructor role ?

Dependencies:
None
Properties:
defaultTemplate: public <#:parent.i.shortName>() { }
description: This is a constructor of the <#:parent.i.name> class.
taskTitle: Provide optional constructor
taskDescription: Provide a Java constructor for the role <#:roleName> <p><#:description></p>

InheritAP: Inheritance constraint

Dependencies:
None
Properties:
value: /AbstractProduct.i

ConcreteFactory: Java class role *

Dependencies:
/AbstractFactory
/ProductFamily
Properties:
defaultTemplate: public class <#:/ProductFamily.name>Factory extends <#:/AbstractFactory.i.longName> { }
description: This class inherits <#:/AbstractFactory.i.name> and implements the operations to create concrete product objects.
taskTitle: Provide the factory class to create <#:/ProductFamily.name> objects
taskDescription: Provide a Java type for the role <#:roleName> <p><#:description></p>

CreateProduct: Java method role

Dependencies:
/AbstractFactory/CreateProduct
Properties:
defaultTemplate: public <#:/AbstractProduct.i.name> <#:/AbstractFactory/CreateProduct.i.signature> { }
description: This method is used to create <#:/AbstractProduct.i.name> objects. It implements the <#:/AbstractFactory/CreateProduct.i.signature> method.
taskTitle: Implement the <#:/AbstractFactory/CreateProduct.i.signature> method
taskDescription: Provide a Java method for the role <#:roleName> <p><#:description></p>

OverrideCP: Overriding constraint

Dependencies:
None
Properties:
value: /AbstractFactory/CreateProduct.i

UseConstructor: Code fragment role

Dependencies:
/ConcreteProduct/Constructor

Properties:
defaultTemplate: return new <#:/ConcreteProduct/Constructor.i.signature>;
description: The <#:/ConcreteProduct.i.name> class has defined a constructor. You should use this constructor to create <#:/ConcreteProduct.i.name> objects.
taskTitle: Use the constructor of <#:/ConcreteProduct.i.name>
taskDescription: Provide a Java code fragment for the role '<#:roleName>'.
<p><#:description></p>

InheritAF: Inheritance constraint

Dependencies:
None
Properties:
value: /AbstractFactory.i

ProductFamily: Issue role *

Dependencies:
None
Properties:
description: This is a new product family. After creating a new product family, the system will guide you to give a concrete implementation for each abstract product interface. In addition you are guided to provide a concrete factory class that can create these concrete products.
taskTitle: Create a new product family
taskDescription: <#:description>

Appendix B: Figure Pattern Specification

The Figure pattern was discussed in Subsection 6.2.3. The pattern was outlined in Figure 29. A more detailed specification is given here. The notation was explained in Appendix A.

FigureManager: Java class role
This role is already bound to the FigureManager class.

initFigures: Java method role
This role is already bound to the initFigures method.

Figure: Java class role
This role is already bound to the Figure class.

draw: Java method role
This role is already bound to the initFigures method.

getName: Java method role
This role is already bound to the initFigures method.

MyManager: Java class role
Dependencies:
/FigureManager
Properties:
defaultTemplate: public class <#:roleName> extends <#:/FigureManager.i.longName> { }
description: This class manages your figure types.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java type for the role '<#:roleName>'. <p><#:description></p>

inheritance: Inheritance constraint
Dependencies:
None
Properties:
value: /FigureManager.i

initFigures: Java method role
Dependencies:
/FigureManager/initFigures
Properties:
defaultTemplate: public void initFigures() { }
description: This method registers the available figure types to the framework.
taskTitle: Implement the <#:/FigureManager/initFigures.i.signature> method
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint
Dependencies:
None
Properties:
value: /FigureManager/initFigures.i

addFigure: Code fragment role
Dependencies:
/MyFigure
Properties:
defaultTemplate: addFigure(new <#:/MyFigure.i.name>());
description: Registers figure <#:/MyFigure.i.name> to the framework.
taskTitle: Add figure <#:/MyFigure.i.name> to the framework
taskDescription: Provide a Java code fragment for the role '<#:roleName>'.
<p><#:description></p>

MyFigure: Java class role +
Dependencies:
/Figure
Properties:
defaultTemplate: public class <#:roleName> extends <#:/Figure.i.longName> { }
description: This is one of your figure classes.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java type for the role '<#:roleName>'. <p><#:description></p>

inheritance: Inheritance constraint
Dependencies:
None
Properties:
value: /Figure.i

draw: Java method role
Dependencies:
/Figure/draw
Properties:
defaultTemplate: public void draw(java.awt.Graphics g, int x, int y) { //TODO draw the figure
}
description: This method is used to draw the figure.
taskTitle: Implement the <#:/Figure/draw.i.signature> method
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint
Dependencies:
None
Properties:
value: /Figure/draw.i

getName: Java method role
Dependencies:
/Figure/getName
Properties:
defaultTemplate: public String getName() { return "<#:parent.i.name>"; }
description: This method returns the name of the figure.
taskTitle: Implement the <#:/Figure/getName.i.signature> method
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint
Dependencies:
None
Properties:
value: /Figure/getName.i

Appendix C: Instance Semantics Pattern Specification

The Instance Semantics pattern was discussed in Subsection 7.2.2. The pattern was outlined in Figure 37. A more detailed specification is given here. The notation was explained in Appendix A.

InstanceSemantics: Java class role
This role is already bound to the InstanceSemantics class.

changingReferencedObject: Java method role
This role is already bound to the changingReferencedObject method.

parseReference: Java method role
This role is already bound to the parseReference method.

referencedObjectChanged: Java method role
This role is already bound to the referencedObjectChanged method.

getDefaultRoleName: Java method role
This role is already bound to the getDefaultRoleName method.

getRoleTargetKind: Java method role
This role is already bound to the getRoleTargetKind method.

isValidChildSemanticsClass: Java method role
This role is already bound to the isValidChildSemanticsClass method.

MyInstanceSemantics: Java class role
Dependencies:
/InstanceSemantics
Properties:
defaultTemplate: public class <#:roleName> extends <#:/InstanceSemantics.i.longName> { }
description: This semantics class represents a role that can be bound to a target domain entity.
taskTitle: Provide an instance semantics class
taskDescription: Provide a Java type for the role '<#:roleName>'. <p><#:description></p>

inheritance: Inheritance constraint
Dependencies:
None
Properties:
value: /InstanceSemantics.i

Constructor: Java constructor role
Dependencies:
/MyInstanceSemantics/Type
Properties:
defaultTemplate: protected <#:parent.i.shortName>(<#:/MyInstanceSemantics/Type.i.name> t) { super(t); }
taskTitle: Provide constructor
taskDescription: Provide a Java constructor for the role '<#:roleName>'.
<p><#:description></p>

getDefaultRoleName: Java method role

Dependencies:
/InstanceSemantics/getDefaultRoleName
Properties:
defaultTemplate: public String getDefaultRoleName(applause.Pattern parentRole) {
    //TODO determine a default name for a new role object
    return getRoleTargetKind(); }
description: Returns a default name for this semantics object. For example: "MyClass", "method", "File", etc.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:
value: /InstanceSemantics/getDefaultRoleName.i

getRoleTargetKind: Java method role

Dependencies:
/InstanceSemantics/getRoleTargetKind
/MyInstanceSemantics/TARGET_KIND_STRING
Properties:
defaultTemplate: public String getRoleTargetKind() {
    return <#:/MyInstanceSemantics/TARGET_KIND_STRING.i.name>; }
description: Returns the string that represents the "class" (or the "type" or the "kind") of the entities this semantics represents. For example: "Java class", "source file", etc.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:
value: /InstanceSemantics/getRoleTargetKind.i

isValidChildSemanticsClass: Java method role

Dependencies:
/InstanceSemantics/isValidChildSemanticsClass
Properties:
defaultTemplate: public boolean isValidChildSemanticsClass(Class c) {
    //TODO determine if the given role semantics class can be a child role of this role
    if (super.isValidChildSemanticsClass(c)) { return true; } return false; }
description: Determines whether a semantics object of the given class can be declared inside this semantics object.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:
value: /InstanceSemantics/isValidChildSemanticsClass.i

TARGET_KIND_STRING: Java field role
Dependencies:
None
Properties:
defaultTemplate: public static final String TARGET_KIND_STRING; //TODO write the kind name here, e.g., "Java method"
description: The string that represents the "class" (or the "type" or the "kind") of the entities this semantics represents. For example: "Java class", "source file", etc.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java field for the role '<#:roleName>'. <p><#:description></p>

parseReference: Java method role
Dependencies:
/InstanceSemantics/parseReference
Properties:
defaultTemplate: protected Object parseReference() {
```
    //TODO parse the reference string here and return the object bind to this role
    String refString = getReference();
    if (refString == null || getPattern() == null) { return null; } }
```
description: Resolves the reference string to an object. This is invoked automatically when the reference string has been changed, the semantics object is initialized or resolveReferencedObject() is invoked explicitly.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint
Dependencies:
None
Properties:
value: /InstanceSemantics/parseReference.i

changingReferencedObject: Java method role
Dependencies:
/InstanceSemantics/changingReferencedObject
Properties:
defaultTemplate: protected void changingReferencedObject() {
```
    //TODO if the old referenced object has listeners remove them here
    Object o = getReferencedObject(); }
```
description: Invoked when the referenced object is about to change. Remove all listener relationships to the referenced object here. Note that the invocation to this method is not guaranteed to be immediately followed by an invocation to referencedObjectChanged().
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint
Dependencies:
None
Properties:
value: /InstanceSemantics/changingReferencedObject.i

referencedObjectChanged: Java method role
Dependencies:
/InstanceSemantics/referencedObjectChanged
Properties:
defaultTemplate: protected void referencedObjectChanged() {
```
    //TODO if the new referenced object should have listeners add them here
    Object o = getReferencedObject(); }
```

description: Invoked when the referenced object has changed. The referenced object may have been changed to some non-null object or null. Add appropriate listeners to the refereced object (and/or the suitable context) here. Note that the invocation to this method is not guaranteed to be immediately preceded by an invocation to changingReferencedObject().
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:
value: /InstanceSemantics/referencedObjectChanged.i

Type: Java class role

Dependencies:
None
Properties:
defaultTemplate: public static class <#:roleName> extends InstanceSemantics.Type { }
description: This inner semantics type class is used to create instances of the enclosing class.
taskTitle: Provide an instance type class
taskDescription: Provide a Java inner class for the role '<#:roleName>'.
<p><#:description></p>

getName: Java method role

Dependencies:
/MyInstanceSemantics/Type/ID_STRING
Properties:
defaultTemplate: public String getName() {
    return <#:/MyInstanceSemantics/Type/ID_STRING.i.name>; }
description: Returns the unique name of the type. This is used for distinguishing types and storing and restoring type information of semantics objects during serialization.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'.
<p><#:description></p>

createSemantics: Java method role

Dependencies:
/MyInstanceSemantics/Constructor
Properties:
defaultTemplate: public Semantics createSemantics() {
    return new <#:/MyInstanceSemantics/Constructor.i.name>(this); }
description: Creates a new semantic object of this type.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'.
<p><#:description></p>

Constructor: Java constructor role

Dependencies:
None
Properties:
defaultTemplate: public <#:parent.i.shortName>(Class semanticsClass) {
    super(semanticsClass); }
taskTitle: Provide constructor
taskDescription: Provide a Java constructor for the role '<#:roleName>'.
<p><#:description></p>

ID_STRING: Java field role

Dependencies:
None
Properties:
defaultTemplate: public static final String ID_STRING; //TODO write the role type name here, e.g., "Java method role"
description: The unique name of the type. This is used for distinguishing types and storing and restoring type information of semantics objects during serialization.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java field for the role '<#:roleName>'.
<p><#:description></p>

Appendix D: Constraint Semantics Pattern Specification

The Constraint Semantics pattern was discussed in Subsection 7.2.3. The pattern was outlined in Figure 38. A more detailed specification is given here. The notation was explained in Appendix A.

ConstraintSemantics: Java class role

This role is already bound to the ConstraintSemantics class.

createHandler: Java method role

This role is already bound to the createHandler method.

isValidValue: Java method role

This role is already bound to the isValidValue method.

getDefaultRoleName: Java method role

This role is already bound to the getDefaultRoleName method.

getRoleTargetKind: Java method role

This role is already bound to the getRoleTargetKind method.

MyConstraintSemantics: Java class role

Dependencies:
/ConstraintSemantics
Properties:
defaultTemplate: public class <#:roleName> extends <#:/ConstraintSemantics.i.longName> { }
description: This semantics class represents a constraint role that establishes a constraint for its parent role. E.g., a Java class role may have inheritance constraint.
taskTitle: Provide a class for constraint semantics
taskDescription: Provide a Java class for the role <#:roleName> <p><#:description></p>

inheritance: Inheritance constraint

Dependencies:
None
Properties:
value: /ConstraintSemantics.i

Constructor: Java constructor role

Dependencies:
/MyConstraintSemantics/Type
Properties:
defaultTemplate:

```
protected <#:parent.i.shortName>(<#:/MyConstraintSemantics/Type.i.name> t) {
super(t); }
```

taskTitle: Provide constructor
taskDescription: Provide a Java constructor for the role '<#:roleName>'.
<p><#:description></p>

getDefaultRoleName: Java method role

Dependencies:
/ConstraintSemantics/getDefaultRoleName
Properties:
defaultTemplate: public String getDefaultRoleName(applause.Pattern parentRole) {
    //TODO determine a default name for a new constraint role object
    return getRoleTargetKind(); }
description: Returns a default name for this semantics object. For example: "MyClass", "method", "File", etc.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:
value: /ConstraintSemantics/getDefaultRoleName.i

getRoleTargetKind: Java method role

Dependencies:
/MyConstraintSemantics/TARGET_KIND_STRING
/ConstraintSemantics/getRoleTargetKind
Properties:
defaultTemplate: public String getRoleTargetKind() {
    return <#:/MyConstraintSemantics/TARGET_KIND_STRING.i.name>; }
description: Returns the string that represents the "class" (or the "type" or the "kind") of the entities this semantics represents. For example: "Java class", "source file", etc.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:
value: /ConstraintSemantics/getRoleTargetKind.i

TARGET_KIND_STRING: Java field role

Dependencies:
None
Properties:
defaultTemplate: public static final String TARGET_KIND_STRING; //TODO write the kind name here, e.g., "Java inheritance"
description: The string that represents the "class" (or the "type" or the "kind") of the entities this semantics represents. For example: "Java class", "source file", etc.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java field for the role '<#:roleName>'. <p><#:description></p>

createHandler: Java method role

Dependencies:
/MyConstraintSemantics/Handler
/ConstraintSemantics/createHandler
Properties:
defaultTemplate: public applause.semantics.BindingHandler createHandler() {
    return new <#:/MyConstraintSemantics/Handler.i.name>(); }
description: This method creates and returns an instance of <#:/MyConstraintSemantics/Handler.i.name>.
taskTitle: Provide '<#:roleName>'

taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint
Dependencies:
None
Properties:
value: /ConstraintSemantics/createHandler.i

isValidValue: Java method role
Dependencies:
/ConstraintSemantics/isValidValue
Properties:
defaultTemplate: public boolean isValidValue(String value) {
    //TODO check the given value
    return true; }
description: Determines whether the given string is valid for the "value" script.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint
Dependencies:
None
Properties:
value: /ConstraintSemantics/isValidValue.i

Type: Java class role
Dependencies:
None
Properties:
defaultTemplate: public static class <#:roleName> extends ConstraintSemantics.Type { }
description: This inner semantics type class is used to create instances of the enclosing class.
taskTitle: Provide a constraint type class
taskDescription: Provide a Java inner class for the role '<#:roleName>'.
<p><#:description></p>

getName: Java method role
Dependencies:
/MyConstraintSemantics/Type/ID_STRING
Properties:
defaultTemplate: public String getName() {
    return <#:/MyConstraintSemantics/Type/ID_STRING.i.name>; }
description: Returns the unique name of the type. This is used for distinguishing types and storing and restoring type information of semantics objects during serialization.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'.
<p><#:description></p>

createSemantics: Java method role
Dependencies:
/MyConstraintSemantics/Constructor
Properties:
defaultTemplate: public Semantics createSemantics() {
    return new <#:/MyConstraintSemantics/Constructor.i.name>(this); }
description: Creates a new semantic object of this type.
taskTitle: Provide '<#:roleName>'

taskDescription: Provide a Java method for the role '<#:roleName>'.
<p><#:description></p>

Constructor: Java constructor role

Dependencies:
None
Properties:
defaultTemplate: public <#:parent.i.shortName>(Class semanticsClass) {
    super(semanticsClass); }
taskTitle: Provide constructor
taskDescription: Provide a Java constructor for the role '<#:roleName>'.
<p><#:description></p>

ID_STRING: Java field role

Dependencies:
None
Properties:
defaultTemplate: public static final String ID_STRING; //TODO write the role type name here, e.g., "Java inheritance constraint"
description: The unique name of the type. This is used for distinguishing types and storing and restoring type information of semantics objects during serialization.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java field for the role '<#:roleName>'.
<p><#:description></p>

Handler: Java class role

Dependencies:
None
Properties:
defaultTemplate: public static class <#:roleName> extends ConstraintSemantics.Handler { }
description: This handler class is used to check the constraint.
taskTitle: Provide a constraint handler class
taskDescription: Provide a Java inner class for the role '<#:roleName>'.
<p><#:description></p>

checkConstraint: Java method role

Dependencies:
None
Properties:
defaultTemplate:

```
public int checkConstraint(Object targetEntity, Object evaluatedValueExpression,
boolean fixConstraint, boolean showMsg) {
        //TODO create a message that will be shown if the constraint is violated
        String violationMsg = "";
        constraintViolationMessage(violationMsg);
        //TODO check if the constraint is violated, return "INVALID | FIXABLE"
        //or "INVALID | UNFIXABLE" if it is
        //TODO if fixConstraint is true you can try to fix the violation here
        if (fixConstraint) { revalidateConstraint(); }
        return VALID | FIXABLE; }
```

description: This method is called in order to (1) determine whether the constraint is violated, (2) determine whether the constraint can be automatically fixed, and (3) fix the constraint automatically.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'.
<p><#:description></p>

getOwnerRoleTargetClass: Java method role

Dependencies:
None
Properties:
defaultTemplate: public Class getOwnerRoleTargetClass() { //TODO return the element type that this constraint is associated with }
description: Returns the class of the entity that is bound to the role (i.e. Semantics) that owns this constraint. The returned class is used when checking the constraint to make sure that the class of the target entity is acceptable. Note that also subclasses of the returned class are considered acceptable.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'.
<p><#:description></p>

getAcceptableValueClass: Java method role

Dependencies:
None
Properties:
defaultTemplate: public Class getAcceptableValueClass() { //TODO return the expected type of the constraint value }
description: Returns the acceptable class of the result object of evaluation of the "value" script. Note that also the subclasses of the returned class are considered acceptable.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'.
<p><#:description></p>

cleanUp: Java method role

Dependencies:
None
Properties:
defaultTemplate: protected void cleanUp() { //TODO if you have some constraint-specific listeners etc., remove them here
```
	super.cleanUp(); }
```
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'.
<p><#:description></p>

getMyConstraintSemantics: Java method role

Dependencies:
None
Properties:
defaultTemplate: public <#:/MyConstraintSemantics.i.name>
```
get<#:/MyConstraintSemantics.i.name>() {
	return (<#:/MyConstraintSemantics.i.name>)getSemantics(); }
```
description: Convenience method to return the semantics object.
taskTitle: Provide 'get<#:/MyConstraintSemantics.i.name>'
taskDescription: Provide a Java method for the role '<#:roleName>'.
<p><#:description></p>

initialize: Java method role

Dependencies:
None
Properties:
defaultTemplate: protected void initialize() { //TODO if you have some constraint-specific listeners etc., add them here
```
	super.initialize(); }
```

taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'.
<p><#:description></p>

Appendix E:
Semantics Wizard Pattern Specification

The Semantics Wizard pattern was discussed in Subsection 7.2.4. The pattern was outlined in Figure 39. A more detailed specification is given here. The notation was explained in Appendix A.

SemanticsWizard: Java class role
This role is already bound to the SemanticsWizard class.

getFirstWizardSheet: Java method role
This role is already bound to the getFirstWizardSheet method.

getSemanticsTypeName: Java method role
This role is already bound to the getSemanticsTypeName method.

isAutomaticlyExecutable: Java method role
This role is already bound to the isAutomaticlyExecutable method.

isAvailableFor: Java method role
This role is already bound to the isAvailableFor method.

isPostponeWizard: Java method role
This role is already bound to the isPostponeWizard method.

ApplauseWizardPage: Java class role
This role is already bound to the ApplauseWizardPage class.

performFinish: Java method role
This role is already bound to the performFinish method.

validatePage: Java method role
This role is already bound to the validatePage method.

canFinish: Java method role
This role is already bound to the canFinish method.

canFlipToNextPage: Java method role
This role is already bound to the canFlipToNextPage method.

createControl: Java method role
This role is already bound to the createControl method.

MySemantics: Java class role
Dependencies:
None
Properties:
bindingMethod: Locate Only
defaultTemplate: public class <#:roleName> { }
description: This is the role semantics for which you want to create UI
taskTitle: Locate your semantics class

taskDescription: Provide a Java type for the role '<#:roleName>'. <p><#:description></p>

MyWizard: Java class role +

Dependencies:
/SemanticsWizard
/MySemantics
Properties:
defaultTemplate: public class <#:/MySemantics.i.name>Wizard implements <#:/SemanticsWizard.i.longName> { }
description: This wizard allows the user to use your <#:/MySemantics.i.name> role type. Typically, in the case of instance semantics, there is three different wizards: one to create a new unbound role, one to create an element that is then bound to the role, and one to locate an existing element that is then bound to the role. For constraint semantics you usually need a wizard to set the constraint value and a wizard to repair the constraint.
taskTitle: Provide a new wizard for <#:/MySemantics.i.name>
taskDescription: Provide a Java type for the role '<#:roleName>'. <p><#:description></p>

inheritance: Inheritance constraint

Dependencies:
None
Properties:
value: /SemanticsWizard.i

isPostponeWizard: Java method role

Dependencies:
/SemanticsWizard/isPostponeWizard
Properties:
defaultTemplate: public boolean isPostponeWizard() { //TODO return true if this wizard is used to create an unbound role
    return false; }
description: Returns true if this wizard is used to create a pattern role that should be left unbound at the moment. I.e., the role in the pattern represents an issue that will be fixed later.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:
value: /SemanticsWizard/isPostponeWizard.i

getSemanticsTypeName: Java method role

Dependencies:
/SemanticsWizard/getSemanticsTypeName
Properties:
defaultTemplate: public String getSemanticsTypeName () {
    return <#:/MySemantics.i.name>.Type.ID_STRING; }
description: Returns the identifying semantics type name of <#:/MySemantics.i.name>. The wizard is associated with this kind of role semantics.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:

value: /SemanticsWizard/getSemanticsTypeName.i

isAvailableFor: Java method role

Dependencies:
/SemanticsWizard/isAvailableFor
Properties:
defaultTemplate: public boolean isAvailableFor (applause.Pattern parent, applause.ChildBinding task) {
//TODO check if this wizard is available for the given parent and task
return true; }
description: Determines if this wizard is available.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:
value: /SemanticsWizard/isAvailableFor.i

isAutomaticlyExecutable: Java method role

Dependencies:
/SemanticsWizard/isAutomaticlyExecutable
Properties:
defaultTemplate: public boolean isAutomaticlyExecutable() {
//TODO check if this wizard can be performed without user interactions
return false; }
description: Returns true if this wizard can be executed automatically; there is no need for interaction with the user after he/she has decided to perform this wizard.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:
value: /SemanticsWizard/isAutomaticlyExecutable.i

getFirstWizardSheet: Java method role

Dependencies:
/SemanticsWizard/getFirstWizardSheet
/MyWizardPage/Constructor
Properties:
defaultTemplate: public fi.tut.cs.practise.lib.wizard.WizardSheet getFirstWizardSheet (applause.Pattern parent, applause.ChildBinding task, applause.semantics.Semantics s) {
return new <#:/MyWizardPage/Constructor.i.name>(parent, task, (<#:/MySemantics.i.name>)s); }
description: Creates and returns an instance of <#:/MyWizardPage.i.name>.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:
value: /SemanticsWizard/getFirstWizardSheet.i

MyWizardPage: Java class role
Dependencies:
/ApplauseWizardPage
/MyWizard
Properties:
defaultTemplate: public class <#:/MyWizard.i.name>Page extends <#:/ApplauseWizardPage.i.longName> { }
description: The UI sheet of <#:/MyWizard.i.name>. Here you shoud provide the user interface for the wizard.
taskTitle: Provide UI sheet for <#:/MyWizard.i.name>
taskDescription: Provide a Java type for the role '<#:roleName>'. <p><#:description></p>

inheritance: Inheritance constraint
Dependencies:
None
Properties:
value: /ApplauseWizardPage.i

_semantics: Java field role
Dependencies:
None
Properties:
defaultTemplate: private <#:/MySemantics.i.longName> <#:roleName>;
description:
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java field for the role '<#:roleName>'.

_parent: Java field role
Dependencies:
None
Properties:
defaultTemplate: private applause.Pattern <#:roleName>;
description:
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java field for the role '<#:roleName>'.

_task: Java field role
Dependencies:
None
Properties:
defaultTemplate: private applause.ChildBinding <#:roleName>;
description:
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java field for the role '<#:roleName>'.

_valueField: Java field role
Dependencies:
None
Properties:
defaultTemplate: private org.eclipse.swt.widgets.Text <#:roleName>;
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java field for the role '<#:roleName>'.

_nameField: Java field role

Dependencies:
None
Properties:
defaultTemplate: private org.eclipse.swt.widgets.Text <#:roleName>;
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java field for the role '<#:roleName>'.

Constructor: Java constructor role

Dependencies:
/MyWizardPage/_task
/MyWizardPage/_semantics
/MyWizardPage/_parent
Properties:
defaultTemplate: public <#:parent.i.shortName>(applause.Pattern p, applause.ChildBinding t, <#:/MySemantics.i.longName> s) {

```
        super(<#:/MySemantics.i.name>.Type.ID_STRING);
        //TODO set title
        setTitle("page title");
        //TODO set more specific title, e.g., "Locate element"
        setDescription("page description");
        //TODO short description for this wizard page
        <#:/MyWizardPage/_parent.i.name> = p;
        <#:/MyWizardPage/_task.i.name> = t;
        <#:/MyWizardPage/_semantics.i.name> = s; }
```

taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java constructor for the role '<#:roleName>'.
<p><#:description></p>

canFinish: Java method role

Dependencies:
/ApplauseWizardPage/canFinish
Properties:
defaultTemplate: public boolean canFinish() { return isPageComplete(); }
description: Checks if the finishing of the wizard can be done when this page is shown in the wizard. The Finish button will be disabled if this method returns false.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:
value: /ApplauseWizardPage/canFinish.i

validatePage: Java method role

Dependencies:
/ApplauseWizardPage/validatePage
Properties:
defaultTemplate: public void validatePage() { // TODO setPageComplete(true); }
description: Checks if values in this page are sufficient to move to the next page or perform the finishing. If the page is valid you must invoke setPageComplete(true) otherwise setPage-Complete(false).
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint
Dependencies:
None
Properties:
value: /ApplauseWizardPage/validatePage.i

createControl: Java method role
Dependencies:
/ApplauseWizardPage/createControl
/MyWizardPage/_nameField
/MyWizardPage/_valueField
Properties:
defaultTemplate: public void createControl(org.eclipse.swt.widgets.Composite parent) {
    //TODO create more advanced user interface for your wizard page
    //This is a default implementation with simple text fields to set name and value
    GridLayout layout = new GridLayout();
    layout.numColumns = 1;
    Composite c = new Composite(parent, SWT.NONE);
    c.setLayout(layout);
    c.setLayoutData(new GridData(GridData.FILL_BOTH));
    Label l1 = new Label(c, SWT.LEFT);
    l1.setText("Name");
    <#:/MyWizardPage/_nameField.i.name> = new Text(c,
SWT.SINGLE|SWT.BORDER);
    <#:/MyWizardPage/_nameField.i.name>.setLayoutData(new GridData(Grid-
Data.FILL_HORIZONTAL));
    <#:/MyWizardPage/_nameField.i.name>.setText("");
    //TODO give default name
    Label l2 = new Label(c, SWT.LEFT);
    l2.setText("Value");
    <#:/MyWizardPage/_valueField.i.name> = new Text(c,
SWT.SINGLE|SWT.BORDER);
    <#:/MyWizardPage/_valueField.i.name>.setLayoutData(new Grid-
Data(GridData.FILL_HORIZONTAL));
    <#:/MyWizardPage/_valueField.i.name>.setText("");
    //TODO give default value
    setControl(c); //Remember to call setControl!
    validatePage(); }
description: Creates the top level control for this wizard page under the given parent compos-
ite.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint
Dependencies:
None
Properties:
value: /ApplauseWizardPage/createControl.i

canFlipToNextPage: Java method role
Dependencies:
/ApplauseWizardPage/canFlipToNextPage
Properties:
defaultTemplate: public boolean canFlipToNextPage() {
    return false; //Usually this page is enough }
description: Returns whether the next page could be displayed.
taskTitle: Provide '<#:roleName>'

taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:
value: /ApplauseWizardPage/canFlipToNextPage.i

performFinish: Java method role

Dependencies:
/ApplauseWizardPage/performFinish
Properties:
defaultTemplate: public boolean performFinish() { //TODO }
description: Performs finishing of the wizard. The wizard invokes this method if the Finish button is pressed when this page is shown in the wizard. This method return true if the finishing was succesful.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint

Dependencies:
None
Properties:
value: /ApplauseWizardPage/performFinish.i

locateElement: Code fragment role ?

Dependencies:
/MyWizardPage/_nameField
/MyWizardPage/_valueField
/MyWizardPage/_task
/MyWizardPage/_semantics
/MyWizardPage/_parent
Properties:
defaultTemplate: //Default implementation to bind an element to the role

```
	String name = <#:/MyWizardPage/_nameField.i.name>.getText();
	String value = <#:/MyWizardPage/_valueField.i.name>.getText();
	if (name.equals("")) name = value;
	//TODO check name and value and return false if there is no element to bind
to this role
	ChildPattern pat =
<#:/MyWizardPage/_parent.i.name>.getAssembly().createChildPattern(<#:/MyWizar
dPage/_parent.i.name>, name, <#:/MyWizardPage/_semantics.i.name>);
	<#:/MyWizardPage/_semantics.i.name>.setReference(value);
	// If a task is defined, perform that task with the new pattern
	if (<#:/MyWizardPage/_task.i.name> != null) {
<#:/MyWizardPage/_parent.i.name>.getAssembly().performTask(<#:/MyWizardPag
e/_task.i.name>, pat); }
	return true;
```

description: This code fragment provides a default implementation to locate an existing element and to bind it to the role.
taskTitle: Locate an existing element and bind it to the role
taskDescription: <p><#:description></p>

unboundRole: Code fragment role ?

Dependencies:
/MyWizardPage/_nameField
/MyWizardPage/_task

/MyWizardPage/_semantics
/MyWizardPage/_parent
Properties:
defaultTemplate: //Default implementation to create a new unbound role

```
        String name = <#:/MyWizardPage/_nameField.i.name>.getText();
        //TODO check that the role name is ok
        ChildPattern pat =
<#:/MyWizardPage/_parent.i.name>.getAssembly().createChildPattern(<#:/MyWizar
dPage/_parent.i.name>, name, <#:/MyWizardPage/_semantics.i.name>);
        //If task exists, perform that task by providing the newly created pattern to it
        if (<#:/MyWizardPage/_task.i.name> != null) {
<#:/MyWizardPage/_parent.i.name>.getAssembly().performTask(<#:/MyWizardPag
e/_task.i.name>, pat); }
        return true;
```

description: This code fragment provides a default implementation to create a new unbound role.
taskTitle: Create a new unbound role
taskDescription: <p><#:description></p>

newConstraint: Code fragment role ?

Dependencies:
/MyWizardPage/_nameField
/MyWizardPage/_valueField
/MyWizardPage/_semantics
/MyWizardPage/_parent
Properties:
defaultTemplate: //Default implementation to create a new constraint role

```
        String name = <#:/MyWizardPage/_nameField.i.name>.getText();
        String value = <#:/MyWizardPage/_valueField.i.name>.getText();
        if (name.equals("")) name = value;
        //TODO check name and value and return false if the constraint cannot be
created
        ChildPattern pat =
<#:/MyWizardPage/_parent.i.name>.getAssembly().createChildPattern(<#:/MyWizar
dPage/_parent.i.name>, name, <#:/MyWizardPage/_semantics.i.name>);
        s = new Script(); s.setInterpreter(ScriptingUtils.getInterpreter("SXI"));
        //TODO select a suitable interpreter for the value string
        s.setSource(value);
        <#:/MyWizardPage/_semantics.i.name>.putProperty("value", s);
        return true;
```

description: This code fragment provides a default implementation to create a new constraint.
taskTitle: Create a new constraint role
taskDescription: <p><#:description></p>

createElement: Code fragment role ?

Dependencies:
/MyWizardPage/_nameField
/MyWizardPage/_valueField
/MyWizardPage/_task
/MyWizardPage/_semantics
/MyWizardPage/_parent
Properties:
defaultTemplate: //Default implementation to create a new element and to bind it to the role

```
        String name = <#:/MyWizardPage/_nameField.i.name>.getText();
        String value = <#:/MyWizardPage/_valueField.i.name>.getText();
```

```
		//TODO use the name and value strings to create a new element
<#:/MyWizardPage/_semantics.i.name>.setReference(value);
		ChildPattern pat =
<#:/MyWizardPage/_parent.i.name>.getAssembly().createChildPattern(<#:/MyWizar
dPage/_parent.i.name>, name, <#:/MyWizardPage/_semantics.i.name>);
		//If a task exists, perform that with the new pattern
		if (<#:/MyWizardPage/_task.i.name> != null) {
<#:/MyWizardPage/_parent.i.name>.getAssembly().performTask(<#:/MyWizardPag
e/_task.i.name>, pat); }
		return true;
```
description: This code fragment provides a default implementation to create a new element and to bind it to the role.
taskTitle: Create an element and bind it to the role
taskDescription: <p><#:description></p>

Appendix F: Eclipse Plugin Pattern Specification

The Eclipse Plugin pattern was discussed in Subsection 7.2.5. The pattern was outlined in Figure 40. A more detailed specification is given here. The notation was explained in Appendix A.

AbstractUIPlugin: Java class role

This role is already bound to the AbstractUIPlugin class.

### JavaFramesPluginInitializer: Java class role

This role is already bound to the JavaFramesPluginInitializer class.

addExtensions: Java method role

This role is already bound to the addExtensions method.

addImages: Java method role

This role is already bound to the addImages method.

MyPlugin: Java class role

Dependencies:
/AbstractUIPlugin
Properties:
defaultTemplate: public class <#:roleName> extends <#:/AbstractUIPlugin.i.longName> {
    private static <#:roleName> _instance = null;
    public static <#:roleName> getInstance() { return _instance; }
    public <#:roleName>(org.eclipse.core.runtime.IPluginDescriptor descriptor) { super(descriptor); _instance = this; } }
description: This is the Eclipse plugin class.
taskTitle: Provide a plugin class
taskDescription: Provide a Java type for the role '<#:roleName>'. <p><#:description></p>

inheritance: Inheritance constraint

Dependencies:
None
Properties:
value: /AbstractUIPlugin.i

MyPluginInitializer: Java class role

Dependencies:
/MyPlugin
/JavaFramesPluginInitializer
Properties:
defaultTemplate: public class <#:/MyPlugin.i.name>Initializer extends <#:/JavaFramesPluginInitializer.i.longName> { }
description: This class initializes the plugin.
taskTitle: Provide an initializer class for your plugin
taskDescription: Provide a Java type for the role '<#:roleName>'. <p><#:description></p>

inheritance: Inheritance constraint

Dependencies:
None
Properties:

value: /JavaFramesPluginInitializer.i

addExtensions: Java method role
Dependencies:
/JavaFramesPluginInitializer/addExtensions
Properties:
defaultTemplate: public void addExtensions(fi.tut.cs.practise.root.Root r) { //TODO }
description: Add your extensions in this method.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint
Dependencies:
None
Properties:
value: /JavaFramesPluginInitializer/addExtensions.i

addWizardExtension: Code fragment role
Dependencies:
/MyWizardExtension
Properties:
defaultTemplate: r.addExtension(new <#:/MyWizardExtension.i.name>());
description: Each new wizard must be added to the plugin.
taskTitle: Add the <#:/MyWizardExtension.i.name> extension
taskDescription: Provide a Java code fragment for the role '<#:roleName>'.
<p><#:description></p>

addRoleTypeExtension: Code fragment role
Dependencies:
/MyRoleTypeExtension
Properties:
defaultTemplate: r.addExtension(new
<#:/MyRoleTypeExtension.i.name>.Type(<#:/MyRoleTypeExtension.i.name>.class)
);
description: Each new role type must be added to the plugin.
taskTitle: Add the <#:/MyRoleTypeExtension.i.name> extension
taskDescription: Provide a Java code fragment for the role '<#:roleName>'.

addImages: Java method role
Dependencies:
/JavaFramesPluginInitializer/addImages
Properties:
defaultTemplate: public void addImages(fi.tut.cs.practise.root.eclipse.RootPlugin r) {
```
	//TODO add plugin-specific image resources here
	<#:/MyPlugin.i.name> plugin = <#:/MyPlugin.i.name>.getInstance();
	//r.addImageDescriptor(YOUR_IMAGE_ID, getImageDescriptor(plugin,
"your_image.gif")); }
```
description: Add your image resources in this method.
taskTitle: Provide '<#:roleName>'
taskDescription: Provide a Java method for the role '<#:roleName>'. <p><#:description></p>

overriding: Overriding constraint
Dependencies:
None
Properties:
value: /JavaFramesPluginInitializer/addImages.i

MyWizardExtension: Java class role *
Dependencies:
None
Properties:
bindingMethod: Locate Only
defaultTemplate: public class <#:roleName> { }
description: This is your new wizard class.
taskTitle: Locate your new wizard
taskDescription: Provide a Java type for the role '<#:roleName>'. <p><#:description></p>

MyRoleTypeExtension: Java class role *
Dependencies:
None
Properties:
bindingMethod: Locate Only
defaultTemplate: public class <#:roleName> { }
description: This is your new role type (the semantics class that implements the role behavior).
taskTitle: Locate your new role type
taskDescription: Provide a Java type for the role '<#:roleName>'. <p><#:description></p>

PluginXML: Issue role
Dependencies:
/MyPlugin
Properties:
description: Each eclipse plugin must have a specific "plugin.xml" file. See Eclipse documentation to create "plugin.xml" file for your plugin.
taskTitle: Create plugin.xml file
taskDescription: <p><#:description></p>

PluginDeployment: Issue role
Dependencies:
/PluginXML
Properties:
description: Finally you must deploy your plugin classes and other files under the Eclipse's plugins folder. You should pack compiled .class files and copy the plugin.xml file and any image resources under your plugin folder. Here is an example "psf-example.bat" file that deploys the PSF Example:

```
	<pre>
	@ECHO OFF
	IF "%ECLIPSE_DIR%" == "" GOTO error
	set PSFEXAMPLE_PLUGIN_DIR=fi.tut.cs.practise.jh.psfexample_1.0.0
	mkdir "%ECLIPSE_DIR%\plugins\%PPSFEXAMPLE_PLUGIN_DIR%\"
	jar cf "%ECLIPSE_DIR%\plugins\%PSFEXAMPLE_PLUGIN_DIR%\psfexampleplugin.jar"
-C classes\ fi\tut\cs\practise\jh\psfexample
	xcopy plugin.xml "%ECLIPSE_DIR%\plugins\%PSFEXAMPLE_PLUGIN_DIR%\" /Q /Y
	xcopy icons\*.* "%ECLIPSE_DIR%\plugins\%PSFEXAMPLE_PLUGIN_DIR%\icons\" /Q
/Y
	ECHO PSF example plugin built successfully.
	GOTO end
	:error ECHO Please specify Eclipse directory as an environment variable (ECLIPSE_DIR).
	ECHO (PSF example plugin not built.)
	:end
	</pre>
```

taskTitle: Deploy your plugin under the Eclipse's plugins directory
taskDescription: <p><#:description></p>

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O. Box 527
FIN-33101 Tampere, Finland