Task-Driven Framework Specialization Goal-Oriented Approach

Juha Hautamäki

University of Tampere

Department of Computer and Information Sciences

Licentiate thesis

January 2002

University of Tampere
Department of Computer and Information Sciences
HAUTAMÄKI, JUHA: Task-Driven Framework Specialization – Goal-Oriented Approach
Licentiate thesis, 98 pp.
January 2002

Abstract

The importance of reusing approved design solutions is widely recognized in software engineering. Object-oriented frameworks, design patterns, etc., are ways to reuse existing knowledge. However, some problems remain, particularly how to guide the application developer to reuse so that the design is eventually implemented in a software project.

FRED (FRamework EDitor) is a prototype of a task-driven architecture-oriented programming environment that can be used to implement architectural solutions. Architecture-specific instructions are given to the tool as specialization patterns; these formal specifications make it possible to automatically compute how to implement design solutions during the software development process. FRED manages the implementation process as a gradually progressing work, where each step is recorded and may have effects to the steps to come. This enables, for instance, documentation and source code generation that uses application-specific names familiar to the application developer. Further, the application developer can be instantly notified if he violates the architectural rules embodied by the given specialization patterns.

This thesis describes the FRED environment. A goal-oriented approach is introduced to model design solutions as a set of specialization patterns. We also explain the mechanism to produce a sequence of programming tasks to implement the solution. To experiment with the environment, an industrial framework was annotated with thirteen specialization patterns.

Keywords: development environment, framework, framework adaptation, framework specialization, pattern, software architecture, software engineering, software reuse.

Acknowledgements

The FRED tool and methodology is a result of prolonged co-operation between academe and industry. First versions of the FRED tool were released during the FRED project (FRamework Editor for Java – Support for Framework Development and Use) 1997-1999. The project was carried out at the University of Tampere and the 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 environment and the concept of representing and implementing architectural solutions have been further evolved. The current version, FRED 1.2, has been developed in the JavaFrames project between the Tampere University of Technology, and the University of Helsinki. The financiers of the JavaFrames project are TEKES (National Technology Agency of Finland), Nokia Networks, Necsom, Sensor SC, and SysOpen.

This thesis is closely related to other thesis and papers written during the current project. Antti Viljamaa [Viljamaa 2001] gives a survey of patterns and frameworks and explores the use of FRED when systemically defining the layered specialization interface of the JHotDraw framework [Gamma 2001]. Markku Hakala [Hakala 2000, 2001], in turn, illustrates the core of the FRED system, namely the details of the specialization patterns and the pattern engine. Developing the FRED methodology and tool has been intensive and interesting teamwork [Hakala et al. 1997, 1998, 1999a, 1999b, 2001a, 2001b, 2001c, 2001d].

The author would like to thank professor Kai Koskimies and the FRED team, especially Markku Hakala, for their comments and advices.

Contents

1 Introduction

In computer science, *architecture*, in general, is the conceptual structure and overall logical organization of a computer or computer-based system from the point of view of its use or design [Oxford 1989]. Focused on software engineering, *software architecture* is the design of the subsystems and components of the software system and relationships between them [Buschmann et al. 1996]. Further, *product line architecture* is a collection of patterns, rules, and conventions for creating members of a given family of software products [Jacobson et al. 1997; Bosch 2000; Jazayeri et al. 2000]. Despite the usefulness of these concepts, there are some problems when implementing the architectural solution; adapting such an abstract design to the concrete software system may be error-prone and requires substantial amount of programming skills and experience.

Object-oriented frameworks [Fayad et al. 1999] can be used to help reusing software architectures as they implement the crucial parts of product line architectures. A such framework captures the programming expertise necessary to solve problems in a particular problem domain; it implements the parts of the design that are common to all applications in that domain, and makes explicit the pieces that need to be customized by the application developer. Though the frameworks have many benefits when reusing architectural design, some problems remain. The framework developer must define and implement the architecture for the intended domain and struggle to make the arising framework flexible and easy to use. The application developer, in turn, must know which framework(s) to use and how to eventually implement the application's logic in terms of the framework.

In the object-oriented community it has been widely noticed that *patterns* [Alexander 1979; Lea 1994; Gamma et al. 1995; Coplien 1996; Vlissides et al. 1996; Jacobson and Nowack 1999] or similar abstract structures can be used to represent design solutions and how they should be implemented. In addition, many authors have stressed the formal specification of this information (see, e.g., [Holland 1993; Alencar et al. 1996; Meijler et al. 1997; Florijn et al. 1997; Froehlich et al. 1997; Mikkonen 1998; Eden et al. 1999]). Provided with formal specifications, a tool can be used to support and partially automate the adaptation process of the architecture and the specialization process of the framework.

Specialization patterns [Hakala et al. 2001a, 2001b] are one approach to formally describe architectural design and how this design should be used during the software development process. This includes, for example, the intended usage of frameworks, design patterns, idioms, and other architectural conventions, like how to comply with some company-specific rules. FRED (FRamework EDitor) is a prototype of a tool that helps the application developer to reuse the design; it provides architecture-oriented task-driven programming environment that uses specialization patterns to generate a dynamic sequence of programming tasks to implement the solution. With a set of specialization patterns, FRED becomes a *specialization wizard* for the described architecture.

Particularly, in case of object-oriented frameworks, a set of specialization patterns can be used to map the necessary specialization actions needed to customize the framework. Augmented with suitable constraints, documentation, and default implementations for the required code elements, the constructed specialization patterns are given to the FRED environment. Based on the given information, FRED generates a dynamically updated task list to actually implement the design. During the normal software development, the environment keeps track of the progress of the pattern-related tasks, verifying that the specialization patterns are bound to the context in the required manner. By doing these automatically generated tasks, step-by-step, the programmer eventually specializes the framework. This kind of *task-driven framework specialization* also agrees with the idea of *piecemeal growth* [Alexander et al. 1975], in which the whole emerges from local acts.

The aim of this thesis is to present an idea of general architecture-oriented task-driven programming environment that provides support for the application developer when reusing and implementing design solutions. It is also explained how this support can be created in terms of formal specifications, namely specialization patterns. To illustrate the concept, it is described how to create a set of specialization patterns for a framework and how the implemented tool (FRED) is used to specialize the framework. However, the purpose is not to present metrics or principles about what is a good specialization pattern for a specific framework or how to design frameworks.

We assume a basic knowledge of object-oriented concepts and frameworks. Experience in an object-oriented programming language or method is helpful but not required.

To give some background and motivation, the problems of the traditional framework development and usage are shortly discussed in Chapter 2. The concept of specialization wizard is presented in Chapter 3. Specialization patterns are explained in Chapter 4. The FRED environment is discussed in Chapter 5. A case framework and its specialization goals and obtained specialization patterns are covered in Chapter 6. Due to the confedential nature of the case framework, detailed descriptions are given in a separate appendix [Hautamäki 2001]. Related work and conclusion are discussed in Chapter 7.

2 Problems in the Usage of Frameworks

Object-Oriented frameworks reuse domain-specific architectural design. However, from the standpoint of the application developer, it may be difficult to understand and use the encapsulated design when specializing the framework. Similarly, from the standpoint of the framework developer, it is difficult to provide efficient reusable solutions that are easy to use and cover essentials of the application domain. For instance, Fayad et al. [1999] enumerates the following challenges dealing with frameworks: *development effort*, *learning curve*, *integratability* with other frameworks, *maintainability*, *validation and defect removal* of the framework and derived applications, *efficiency*, and *lack of standards*. In addition, the framework-centered software development can be divided into the following phases: *framework development phase*, *framework usage phase*, and *evolution and maintenance phase*, where each phase is affected by other ones making the process iterative. People have different viewpoints to the problems depending on the phase they are involved in and if they are actually developing the framework or using it to derive applications.

To motivate the need of tool-support, an overview of the problems is given. The framework development phase is discussed in Section 2.1; the problems of this phase mainly concern the framework developer dealing with domain analysis and implementation alternatives. During the development phase a tool could be used to implement general design solutions and coding conventions, or even to utilize other frameworks. Also, formal specifications could be created to map the intended usage of the framework itself. The problems of the framework usage phase are discussed in Section 2.2; these are encountered by the application developer trying to specialize the framework. To reduce the learning efforts and speed up the application development, a tool with framework-specific specifications could be used to guide the application developer through the specialization process. Finally, the problems of the evolution and maintenance phase are discussed in Section 2.3. Again, a tool with framework-specific architectural descriptions could be used as additional technical documentation.

2.1 Development Phase: Creating Framework

Over and over again it has been noticed that developing extensible and reusable object-oriented frameworks for complex application domains is an iterative process, which requires both domain and design expertise. For instance, Johnson, Foote, and Russo have discussed how to create frameworks [Johnson and Foote 1988; Johnson and Russo 1991; Johnson 1997]. Usually the process starts with domain analysis trying to find the reusable design and the extension points, sometimes called "hot spots" [Pree 1995], 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; inexperienced developers should examine applications written by others and consider writing application in the domain. To help the framework development process, a tool may contain a catalogue of high-level solutions and perhaps more company-specific design decisions that can be used over and over again in different software projects. Otherwise the developers can recall their past experience and former designs.

The first version of the framework is usually a *whitebox* framework meaning that it is mainly specialized by deriving new subclasses and overriding member functions. The term "whitebox" refers to visibility; due to inheritance the internals of parent classes are often visible to subclasses. 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 *blackbox* 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.

In case of frameworks, the iteration is inevitable as changes are motivated by trying to reuse the framework. According to Johnson and Russo [1991] all software requires iteration before it becomes reusable. This follows from the general observation that software never has a desirable property unless the software has been carefully examined and tested in terms of the property. The ultimate test for whether the software is reusable is to reuse it. It is not possible to reuse software until it has been written and it is working, so iteration is inevitable. The only exception is that software that is a reimplementation of existing reusable software might not need iteration; this kind of software is actually just a version of the old one, and the iteration took place already when the old version of the software was designed. Similarly, reusing architectural design with a supporting tool can be seen as a proven way to implement design solutions in a domain-specific context.

Because frameworks require iteration and deep understanding of the application domain, it is hard to keep the framework development phase on schedule. Thus, framework design should never be on the critical path of an important project. On the other hand, a use of a framework validates it when the use does not require any changes to the framework, and helps improve the framework when it points out weaknesses in it. Hence, designers of a framework should collaborate closely with application developers. Also, the efforts to create documentation should not be underestimated; frameworks are powerful and complex, making them hard to learn and use without good documentation. For instance, Booch [1994] says that the 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. From this point of view, it may be very beneficial to use specialization patterns or similar architectural specifications to map the extension points (or hot spots) of the framework.

At some point, the framework seems to be good enough to be released. However, the decision to release the framework may be a risk because no general metrics exist to explicitly determine if the framework has enough reusability, stability, and documentation [Poulin 1994]. Releasing an immature framework may have severe consequences; if the framework is not stable, it is difficult to specify the intended usage of its specialization interface. As a result, this leads to a situation where the derived applications become obsolete or, at least, need additional maintenance when the framework is modified. Unfortunately, it is not possible to beforehand completely test the framework against all possible applications that may be derived from it. This is known as a problem of verifying abstract behavior [Fayad et al. 1999].

2.2 Usage Phase: Deriving Application

Methods to analyze the application domain and to implement object-oriented applications have been widely discussed by number of authors (see, e.g., [Rumbaugh et al. 1991; Coad and Yourdon 1991; Jacobson et al. 1999]). Briefly, analysis builds a model of the real-world situation with domain-specific concepts. Based on this information, a high-level decision is made about the application's overall architecture and subsystems. Object-oriented frameworks implementing product line architectures can help this process as they store experience; problems are solved once and the business rules and design are used consistently. By providing an infrastructure, the framework decreases the amount of architectural decisions that the application developer has to make. Also, when implementing the application there is less code to program, test, and debug, as the application developer writes only the code that extends or specifies the framework behavior to suit the requirements of the application. Application development with object-oriented frameworks has been discussed, for instance, by Mattsson [1996] and Fayad et al. [1999].

While frameworks have many advantages when reusing architectural design, some new problems arise. Application development is seldom based on a single framework; a typical framework usually provides the design for only a part of a software system, such as its user interface, though application specific frameworks sometimes describe an entire application. Thus, 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]. However, the integration process may not be straightforward because the architectural styles [Buschmann 1996; Shaw and Garlan 1996] of two or more frameworks are 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].

Using a framework requires substantial amount of knowledge. The application developer must know which frameworks to use, is there any integration problems between them, and how they should be specialized. This means that the application developer must realize that a particular framework is already implementing a part of the required application-specific solution and identify the similarities between the architecture of the framework and the architecture of the application. Information is required about the intended domain of the framework and its appropriateness for the application under consideration as well as how to actually customize the framework to create the required functionality.

One of the reasons why learning and using a framework is hard, is that comprehending an object-oriented design as such is difficult. Software architecture level of design is concerned with the description of elements from which systems are built, their interactions, patterns that guide their composition, and constraints on these patterns [Shaw and Garlan 1996]. But, 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, 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 the specialization. For instance, whitebox frameworks based on inheritance, require application developers to create many new subclasses with a substantial amount of code [Johnson and Foote 1988]. While most of these new subclasses may be simple, their number and interactions can make the task difficult for an inexperienced programmer. Blackbox frameworks based on object composition require less coding, but still the rules behind the component composition may be hard to understand. Thus, in the software development process, a practical system that smoothly integrates the architectural design with the concrete software products, and verifies that the programmer obeys implicit architectural rules, could be very beneficial.

In any case, both novice and advanced programmers will make errors when using the framework. They may read instructions from the documentation, write the required source code, and then go on to the next problem without checking or knowing whether the code is correct. Finding these errors later may be tricky as the framework(s) hides the flow of control, making it hard to trace. With suitable architectural specifications, a tool can be used to verify the static structure of the source code and to remind the programmer if some architectural violations are detected. Also, bindings between the framework and the application become more explicit as they can be tracked by the tool.

2.3 Evolution and Maintenance Phase

Because of its importance to other applications and software projects, a framework requires routine maintenance to fix errors, assist clients, and respond to their problems and requests. Even if the framework itself is robust and accurate, the surrounding world may change affecting to the architecture and rules of the framework. Altogether, this evolves changes to the framework, its documentation, and tool-support. These modifications, in turn, may cause serious problems for the clients using the framework. Thus, after releasing the framework it should be as stable as possible. On the other hand, it is hard to know if the framework is stable enough without use cases that verifies its specialization interface and functionality.

During the maintenance and evolution phase of the framework, the framework developer must understand the application domain, the overall architecture and its rationale, the reasoning behind the solutions, which part of the overall architecture provides flexibility at each extension point, and why each design alternative was selected. This makes a good technical documentation essential as the person maintaining the framework may not be the original developer. Information is needed at both a high level of abstraction and at a concrete level of detail. In addition, besides frameworks, also applications evolve [Lehman and Belady 1985] making it necessary to modify the framework-related parts of the application. Again, without good documentation it is hard to get the overall picture and locate and understand the involved software components or pieces of code. And, of course, if it is noticed that the framework is functioning incorrectly or lacking some essential features, it causes pressure to change the framework entirely or implement the required functionality from scratch.

A tool with framework-specific formal specifications, like specialization patterns, may help the maintenance process as it explicitly describes the intended usage of the framework and its relations to the derived applications. Particularly, maintaining the application becomes easier because the tool can be used to make the bindings between the framework and the application more explicit. Also, augmented with more informal documentation, the tool may be useful for the person trying to understand the framework more deeply. In addition, new ideas and proposals to improve the framework may arise as the system with formal architectural specifications concretizes the encapsulated design and gives common vocabulary to the framework developers and clients. On the other hand, efforts are needed to maintain these framework-specific specifications, too.

3 Specialization Wizard

Influenced by the the experiences gathered from the existing FRED prototype [Hakala et al. 2001a, 2001b, 2001c, 2001d; Viljamaa 2001] and the ideas of formally describe and gradually implement architectural design, we propose architecture-oriented task-driven system that supports incremental, iterative, and interactive use of architectural design. With a set of formal specifications to specify the architecture and its intended implementation, the system constitutes a *specialization wizard* for the architecture. Using the given specifications as input, the environment is able to generate programming tasks that guide the application developer step-by-step to adapt the encapsulated design. Besides frameworks, a specialization wizard can be constructed for architectural- and coding convention, like enforcing the use of certain design patterns [Gamma et al. 1995] or idioms [Coplien 1992; Vlissides et al. 1996].

The general concept of *pattern* as the basis of formal architectural specifications and the advantages of applying patterns gradually rather than with one isolated action are discussed in Section 3.1. As a result, the architecture-oriented task-driven environment to manage and partially automate the implementation process of design solutions is presented in Section 3.2. Finally, to demonstrate the outlined system, an overview of the framework specialization with the FRED environment is given in Section 3.3.

3.1 About Patterns and Piecemeal Growth

Deploying architectural design to concrete software products means that the application developer must provide the individual building blocks that bring the architectural structure into existence. Rather than implementing the plan at once as an isolated action, this can be seen as task-driven, incremental, and iterative process where the application developer pursues a set of goals by creating and modifying program elements like classes, methods, and so on. If the required steps and participants could be formally specified a generic tool can be used as a specialization wizard that automatically supports this gradual implementation process. Specifying patterns, in general, is discussed in Subsection 3.1.1. The idea of gradually adapt patterns by executing small programming tasks is illustrated in Subsection 3.1.2.

3.1.1 Formal Pattern Specification Languages

Architectural design can be expressed as *design patterns* or similar structures. Originally, architect Christopher Alexander developed the idea of design patterns to enable people to design their own homes and communities [Alexander et al. 1977; Alexander 1979]. His pattern language was a set of patterns, each describing how to solve a particular kind of 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, etc. 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). Thus, the patterns in the pattern language were generative; besides describing the architecture they also described how to implement the architecture in practice.

Obviously, as widely embraced in the object-oriented community [Lea 1994; Gamma et al. 1995; Coplien 1996; Vlissides 1996; Jacobson and Nowack 1999], the concept of patterns is suitable to describe software architectures, too. For instance, Gamma et al. [1995] defines a design pattern as a description of communicating objects and classes that are customized to solve a general design problem in a particular context. Each design patterns should have a name and it should also describe the problem and its context, the solution, and the consequences; this suits best for communicating generic design alternatives. However, when emphasizing the generativity of patterns one can describe patterns as a rule or step-by-step instructions like Buschmann et al. [1996]. Riehle and Züllighoven [1996], in turn, propose a more general pattern definition based on separation of the pattern's finite form and its potentially infinite context. To get a good overview of patterns, see, for instance, Appleton [1997].

Patterns have many benefits. A pattern is an essay that describes a problem to be solved, a solution, and the context in which that solution works. It names a technique and describes its costs and benefits giving software developers a common vocabulary for describing their designs and also a way of making design trade-offs explicit. Shortly, patterns describe recurring solutions that have stood the test of time. In practice, instead of re-inventing architectural design and possible implementation alternatives, patterns allow the solution of problems by providing time-tested combinations that work. In addition, a framework can be designed, documented and used in terms of patterns, where each pattern describes how to solve a small part of the larger design problem. For instance, patterns have been used for documentation of frameworks [Johnson 1992] and for describing the rationale behind design decisions for a framework [Beck and Johnson 1994]. As long as the patterns are powerful enough to describe most initial uses of the framework, it will meet some of the needs of the application developers.

Though patterns have many advantages, their descriptions and specifications seem to be rather informal. This makes it hard to provide automatic tool-support and error recovery. Typically, the documentation of a design pattern uses abstract terms and fixed set of examples while the application developer tries to understand how the documentation is related to the current application-specific problem. In addition, there is no support for error recovery and iteration though the application developer most probably will make errors and wants to change some decisions.

To automatically support the use of patterns, they must be given with a specific tool-oriented formalism. We call such formalism a *formal pattern specification language*. Based on the current stage of the application project and the patterns expressed by the formalism, computations can be made to guide the application developer through the implementation process. However, like with all non-trivial formalisms, essential aspects of the original pattern description may become cluttered with many details making it hard to understand without the tool. Thus, the purpose of the formal pattern specification language should be to enable pattern-specific computations, like a generic specialization wizard tool that fills the gap between the pattern and the concrete code-level implementation, not to displace the more general documentation.

In general, specifying patterns unambiguously with some formalism is hard without losing their universal applicability; the informal description of the pattern may be advisedly ambiguous to propose multiple implementation alternatives. Usually, the constructed specification is not as general as the original pattern description and it may be necessary to create a set of specifications to cover the most important alternatives. In addition, if the formalism is not expressive enough, the essential aspects and vital information of the pattern may be lost. Altogether, this means that a single specification given by the formal pattern specification language may represent only a subset of possible implementation alternatives of the original pattern.

The process to create pattern specifications to express architectural design resembles the process to create reusable software discussed by Johnson and Russo [1991]. Like with reusable code, the iteration may be necessary as changes are motivated by trying to implement the design in practice. It seems that the specification for a particular pattern is never accurate enough unless the design has been carefully examined. In addition, the design may be ambiguous making it even more difficult to create a complete set of specifications with the used formalism. The ultimate test for whether the pattern specification is sufficient is to carry out the design with it.

3.1.2 Tasks and Piecemeal Growth

While a formal pattern specification language can be used to specify patterns, a mechanism is needed to adapt these specifications to the concrete code-level software implementations. Working with small and meaningful tasks to achieve complex results seems to be the strategy that people adapt spontaneously. For instance, Carroll [1990] noticed 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. In addition, when learning and using a new system, users often skip over crucial material if it does not address their current task-oriented concern or they try to read several manuals, composing their own instructional procedure on the fly. It has also been noticed that it is difficult to understand a complex design until it has been used [Johnson 1992]. Therefore, from the application developer's standpoint, trying to implement the design in small pieces is more beneficial than studying the theory at once.

Affected by this kind of intuition, we believe that to be useful and easy to use, the mechanism to adapt pattern specifications should be based on the idea of *piecemeal growth* [Alexander et al. 1975]. Alexander says that a process, which allows the whole to emerge gradually from local acts, will guide planning and construction. This piecemeal growth is evolutionary, dynamic, and continuous. Instead of each act of design or construction being an isolated event creating a "perfect" element, every system is changing and growing all the time, in order to keep its use in balance. The acts to implement the design can be seen as a dynamic sequence of tasks. This task list cannot be simply expressed by static and linear step-by-step list because a choice made during the adaptation process may change the rest of the required implementation completely, making it difficult to outline the process.

Thus, rather than concretizing patterns at once, with a single action, they should be adapted with dynamically updated task list containing meaningful programming tasks with task-specific documentation and source code suggestions, guiding the application developer through the implementation process. In addition, during implementation, circumstances may change causing new tasks or making it necessary to revisit the old ones. This can be seen as a goal-directed and iterative activity, where the main priority of the application developer is to know how to do something, rather than to clearly understand why it is done that way. A tool generating a such task list and tracking the bindings between the pattern specifications and the actual code elements could be used to speed up the development process and to improve the quality of the software system. Such a tool would also correspond peoples tendency to work with trial and error when learning new systems.

3.2 Architecture-Oriented Task-Driven System

A formal pattern specification language can be used to represent architectural design and the way of this design should be implemented. Particularly, with a suitable formalism, an *architecture-oriented task-driven system* may be used to support and partially automate the adaptation process. The system is architecture-oriented because it manages and implements software products in terms of architectural specifications. The system is task-driven because integrating and adapting architectural design is based on dynamically generated programming tasks; to enable this kind of intelligent assistance, the system keeps track on the bindings between the given pattern specifications and the source code. Together the system and the given pattern specifications form a *specialization wizard* for a particular architecture. The concept is illustrated in in Figure 1.

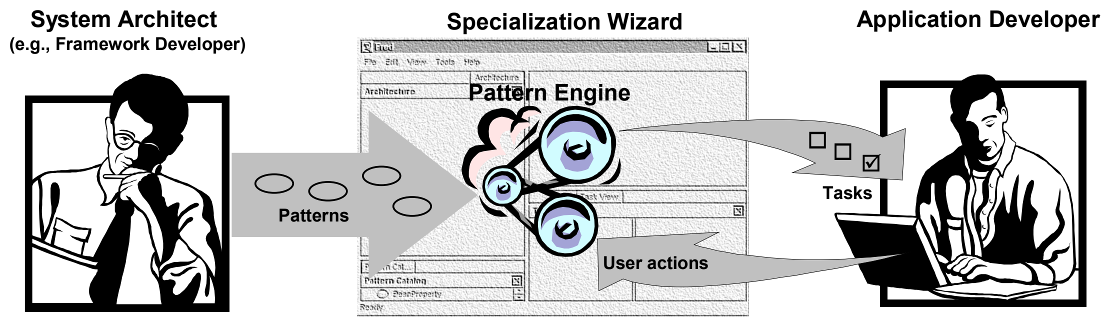


Figure 1. Architecture-oriented task-driven system.

The architecture-oriented task-driven system has two kinds of users. System architects are interested in modeling architectural design in terms of the formal pattern specification language to enforce coding conventions and design solutions. For instance, the framework developer or expert may create a set of pattern specifications to construct a specialization wizard that enforces the intended usage of the framework across software projects. The application developer, in turn, may be searching a suitable solution for his current problem and adapts the most relevant pattern specifications by executing the generated programming tasks. From the standpoint of the application developer, the specialization wizard to adapt a particular design has the following benefits [Hakala et al. 2001a]:

- *Support for incremental, iterative and interactive specialization process*. The specialization wizard provides the application developer a dynamically adjusted list of fine-grained programming tasks. The application developer should be able to execute these specialization tasks in small portions, see their effect in the produced source code, and go back to change something, if needed. In this way, the application developer has better control and understanding of the process and of the produced system.
- *Specialized instructions*. The problem with traditional documentation is that it has to be given beforehand using abstract terms and fixed set of examples. However, in an incremental adaptation process the specialization wizard can gather application-specific information and adjust the documentation as well. This makes the instructions much easier to follow as the documentation adapts to the terms and structures of the application.
- *Architecture-sensitive source-code editing*. The architectural rules that must be followed in the adaptation process can be seen much 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. Thus, immediate response can be provided as tasks to fix any violations of these architectural rules.

- *Open-ended adaptation process*. The adaptation of architectural design should be open-ended in the sense that it can be resumed even for an already completed application. This is important for the future maintenance and extension of the application.

The core of the architecture-oriented task-driven system is the *pattern engine*; actually it is an interpreter [Aho et al. 1986] for the formal pattern specification language that takes pattern specifications as input and generates tasks as output. Thus, it is the pattern engine that maintains the dynamic “things-to-do”-list. The application developer proceeds through the generated tasks while the pattern engine gets notified of the user actions and generates new tasks to indicate errors and missing pieces of code. If the application developer completes a task it can be removed from the task list. Correspondingly, if a required program element is missing or the application developer violates some of the architectural rules implied by the pattern specifications, a new task is generated to fix the situation. In that way, the design is instantiated gradually and each task may affect the tasks to come.

The formalism behind pattern specifications must be able to describe the participants and interactions of the design. Based on this information, the pattern engine verifies that the application developer obeys the pattern-specific constraints and implements the required participants. Further, because the system continuosly tracks the bindings between the given pattern specifications and the actual program elements, the application developer can be instantly informed if a specific program element conforms to the pattern. Thus, the environment is able to enforce certain pre-defined coding conventions and architectural solutions, which can be seen as an advantage when describing, for instance, the intended usage of a framework.

The pattern engine may also include rules and heuristics to determine if a task can be completed automatically. For example, tasks to provide method parameters can be resolved implicitly by analyzing the signature of the method. Usually however, the system cannot rely on such heuristics alone. By requiring explicit commitment from the application developer, the specialization wizard may behave in a more predictable way.

Besides the pattern engine to manage the task list, the environment should provide source code and project handling facilities to enable normal software development. To enable instant feedback and dynamically updated task list, the pattern engine must be integrated to the other development facilities. For instance, it should receive notifications from the source editor to calculate violations and errors against pattern specifications. Thus, as mentioned before, the specialization wizard can be seen as a higher level typing system.

Another important benefit that comes with the integrated pattern engine is that documentation can be modified and generated runtime. Consider, for example, instructions for framework specialization. The problem with traditional documentation is that it has to be written before the specialization takes place. Therefore, the documentation has to be given in terms of abstract concepts of the framework, not with the concrete concepts of the application. By providing tasks incrementally, the pattern engine gathers information about the application and customize the documentation with application-specific terms, reflecting the choices the application developer has already made. At the same time, when adapting the design to the concrete implementation, the application developer sees the process step-by-step making it easier to understand the architectural implications of the design.

Similarly, the source code can be generated that is particularly customized to the current application-specific context. The interactive nature of the specialization wizard makes it natural to show the coding suggestions immediately to the application developer so that he can tailor the code according to the instructions given by the pattern engine. Thus, while the application developer can use the task list and adaptive documentation to learn the design, he can also concentrate on implementing more complicated features letting the environment to generate much of the non-interesting source code. Finally, if the user follows the ever-changing task list, a point is reached where there is no mandatory tasks left and the design has been bound to the context.

Finally, the development process should be open-ended in the sense that it can be resumed even for an already completed application. Thus, the specialization wizard environment must be able to save the current stage of the development process and the bindings between the pattern specifications and concrete software elements. This supports the iterative nature of software development.

3.3 Usage with Frameworks

A framework without supporting tools and documentation is often hard to use. Therefore, to reduce learning efforts and to effectively support the specialization process, a framework usually comes with manuals and toolkits supporting application development. Johnson and Foote [1988] define a toolkit as a collection of high-level tools that allow a user to interact with a framework to configure and construct new applications. Ideally, one should be able to 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. The problem is that there is no general platform to create framework-specific tool-support. Instead, toolkits are created from scratch, requiring considerable development and implementation efforts and consuming limited resources of the framework developers. However, we argue that a suitable formal pattern specification language and the architecture-oriented task-driven system discussed in the previous section can be used to constitute a specialization wizard for any object-oriented framework.

Typically, a framework has a set of extension points (or "hot spots"). The process to find these extension points and figure out the intended usage of the framework is illustrated in Subsection 3.3.1. From the application developer's standpoint, the framework specialization can be seen as a goal-oriented activity; this is discussed in Subsection 3.3.2.

3.3.1 Identifying Extension Points

The architecture of a framework contains a set of classes and the way instances of those classes collaborate that embodies an abstract design for solutions to a family of related problems [Roberts and Johnson 1996; Johnson and Foote 1988]. Specializing a framework means that the application developer must finalize the adaptation of the architectural design partially implemented by these framework elements. Our empirical experiences with large frameworks have shown that there are two different approaches to create pattern specifications to describe how the framework should be specialized; we call these *goal-oriented* and *method-oriented* approaches.

The *method-oriented* approach [Viljamaa 2001, Hakala et al 2001b] 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 a such framework, the method-oriented approach enables systematic mapping of the specialization interfaces and it also allows automated support to create pattern specifications, as some heuristics (see, e.g., [Krämer and Prechelt 1996]) can be used to identify and specify the usage of these interfaces. For instance, *template* and *hook methods* can be identified when trying to inspect the extension points of an object-oriented framework [Pree 1995; Schauer et al 1999]. As a result, the method-oriented approach produces a set of pattern specifications describing the interfaces of the framework.

The *goal-oriented* approach, in turn, is based on an analysis of the expected behavior of the application developer. It assumes that the application developers try to use the framework by setting meaningful goals that they pursue by doing sequence of programming tasks. From the system architect that creates the pattern specifications this requires identification of these application-specific goals and how they should be achieved by the application developer. Thus, rather than mapping the specialization interface as such, the goal-oriented approach provides solutions to practical specialization problems addressing the current task-oriented concern of the application developer. In this thesis, we concentrate on the goal-oriented approach.

3.3.2 Goal-Oriented Approach

When starting to use a framework, one usually has a particular objective in mind or at least a hint of the desired outcome. We call such objectives pursued by the application developer as *specialization goals*. The solution to achieve a specialization goal may be well known and documented, or it can be found by examining existing applications based on the framework and by interviewing the framework developers. This resembles an implementation case [Pasetti and Pree 2000; Pasetti 2001] that describes how functionality for an application in the framework domain can be implemented using the constructs offered by the framework.

Achieving a specialization goal means that some of the framework's extension points must be satisfied. In case of specialization goals, these hot spots are not necessarily isolated from each other; instead, when pursuing the goal, the application developer may struggle with a number of derived subclasses and methods and their complex interactions. The informal framework documentation does not necessarily describe these steps precisely, but has a more general view about the framework and its use.

As an example, Figure 2 presents specialization goals of 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. The example framework provides a skeleton to create such a system and the framework expert has identified the specialization goals that most probably will interest the application developer. Note that the example is slightly simplified; new goals may be identified and the goals shown in the figure may be further divided into more specific sub goals.

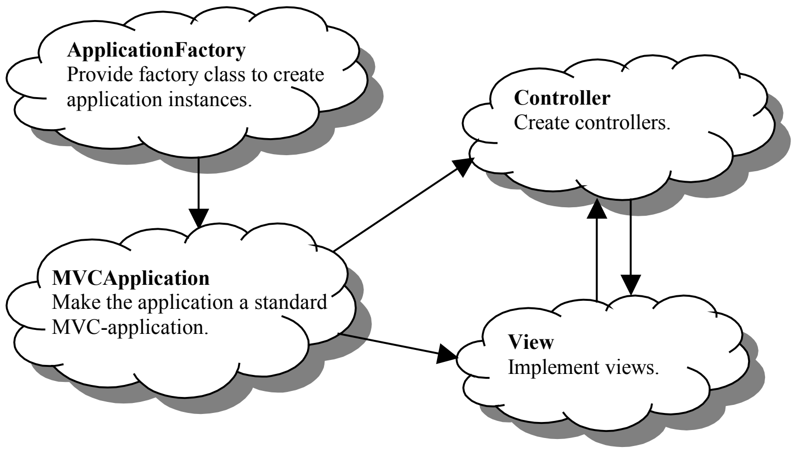

ApplicationFactory
Provide factory class to create application instances.
Controller
Create controllers.
MVCApplication
Make the application a standard MVC-application.
View
Implement views.

Figure 2. Set of specialization goals.

To create goal-oriented pattern specifications the framework expert must recognize typical specialization goals, analyze the architectural aspects involved in these goals, and find the required tasks expected to be carried out by the application developer. Typically, from the standpoint of the application developer, specialization goals constitute a linked structure where achieving one goal leads to another. The order, in which the specialization goals are pursued, is a recommendation only; the application developer may revisit the solutions during the framework specialization, providing more functionality, or undoing previous specialization choices.

Figure 3 illustrates the goal-oriented approach for task-driven framework specialization. In the figure, the framework expert has used the formal pattern specification language to create a set of pattern specifications describing solutions for the identified specialization goals. The architecture-oriented task-driven system takes these specifications as input, thus constructing a specialization wizard to specialize the framework. The pattern engine computes the given specifications against the current situation, generating tasks to construct or modify application elements. The task list is updated automatically according to the stage of the application's source code, in which violated constraints and missing participants are indicated as new tasks. When doing tasks, the system allows the application developer to select actual code elements or it may generate default code as instructed in the pattern specifications.

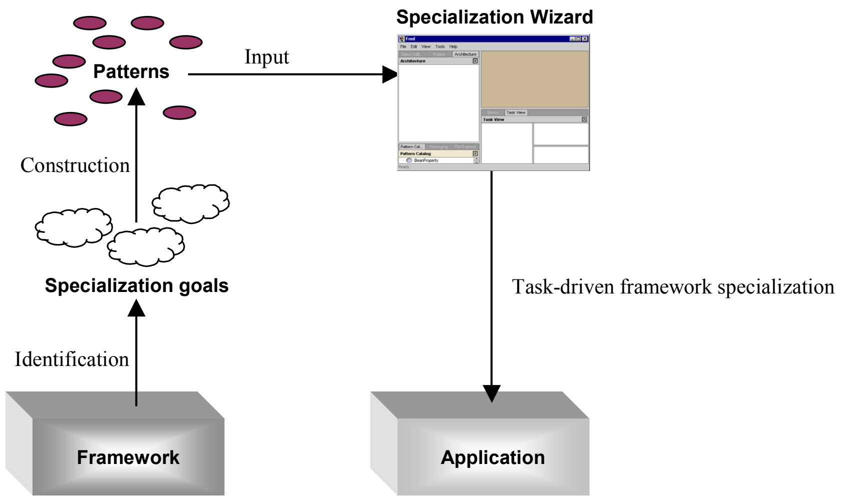


Figure 3. Goal-oriented approach for task-driven framework specialization.

One method to construct pattern specifications for a particular specialization goal is to first derive an example specialization that achieves the goal. This example specialization helps the pattern modeler, usually the framework expert, to identify the required program elements and their interactions. This process is similar to object-oriented analysis on the architecture level: central concepts of the example solution are identified and associated with pattern elements. Participants to represent object-oriented concepts like classes and methods are easy to find. However, the solution includes interactions and other elements, like required method calls, that may not be seen directly from the example specialization. The construction of pattern specifications is discussed more precisely in Chapter 4.

4 Specialization Patterns

The concept of *specialization patterns* [Hakala et al. 2001a, 2001b] can be seen as one approach to formally describe parts of software architecture and how these parts should be implemented and integrated in real software products. Constructed specialization patterns are like formal generative patterns [Alexander 1979] that can be used systematically under the guidance of the architecture-oriented task-driven system (FRED) to produce a number of implementations based on the encapsulated design. Hence, a set of specialization patterns composes a tool-supported pattern language for a particular system. Currently the concept of specialization patterns has been used for Java programs, but in principle, they could be used to express design solutions for other languages like C++ [Stroustrup 1991] and XML [W3C 2001].

The basic concepts of the specialization patterns are introduced in Section 4.1. The mechanism to automatically generate programming tasks to implement the encapsulated design is discussed in Section 4.2. To facilitate the presentation of the specialization patterns, a textual representation format is given in Section 4.3. The process to create specialization patterns for a specific design is discussed in Section 4.4. The actual implementation of specialization patterns is explained in Section 5.4 when we are talking about the FRED environment.

4.1 Basic Concepts

To understand specialization patterns and their use, the essential concepts are explained here. We call the basic building blocks of specialization patterns as *roles*; they are used as blueprints for the actual *program elements*. Associations between roles and program elements are called *contracts*. Relations between roles are affected by *dependencies* and *multiplicity* settings. Each role may have a set of *properties*; constraint properties can be used as the basis of structural validation while template properties enable adaptive documentation and code generation.

Roles, contracts, and program elements are discussed in Subsection 4.1.1, dependencies and multiplicity in Subsection 4.1.2, and properties in Subsection 4.1.3.

4.1.1 Roles, Contracts, and Program Elements

The application consists of *program elements*. In case of Java, a program element may be a class, method, field, constructor, or just an arbitrary piece of source code. A specialization pattern consists of *roles* where each role represents a program element or a set of elements in the intended solution; a role is an abstraction of the required program element(s). By using the given role specifications, the pattern engine (recall Section 3.2) generates tasks for the application developer. When executing tasks, the application developer implicitly creates connections between the roles and the program elements.

The commitment of a program element to play a particular role is called a *contract*; hence, the parties of the contract are the role and the program element associated to that role. If a program element violates its contract(s), the pattern engine may ask the application developer to fix the element. Whether the program element violates the contract is determined automatically by checking that the element obeys the role specification.

When a program element has a contract to play a particular role, that program element is *cast* in the role (or the role is cast to the program element). Alternatively, we can say that the program element is *bind* to the role. Similarly, creating connections between the roles of the specialization pattern and the program elements of the application is called *casting* or *instantiating* the specialization pattern. The collection of contracts is called a *cast*. Figure 4 illustrates these definitions.

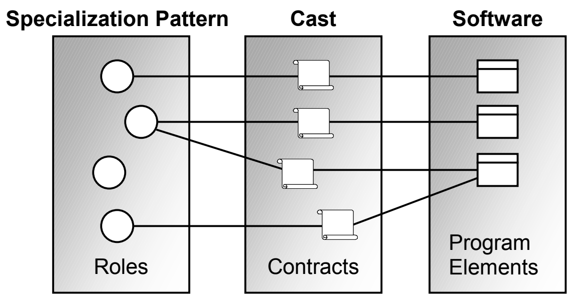


Figure 4. Roles, contracts, and program elements.

Note that a role may have several contracts, as there can be multiple program elements bind to the same role. For instance, a role may represent a base class requiring that the application developer provides that particular base class and casts it in the role. Another role, in turn, may represent all subclasses of that base class.

4.1.2 Dependencies and Multiplicity

Program elements are not self-contained entities. For instance, when implementing a Java method it usually requires the return type and the enclosing class. Equally, roles representing these program elements are not isolated from each other; pre-condition to cast a program element in a specific role may require that another program element has been cast in some other role. Intuitively, there is a *dependency* between two roles if the one needs the other in some way. To go back to the previous example, a role representing a Java method may need the required return type expressed by another role; the method is not syntactically correct until its return type has been specified, so the other role must be cast first. This leads to the mechanism in which the casting process is not carried out in one go but dependencies affect the order in which the casting process proceeds and contracts are made.

Another important concept that affects how the pattern engine generates tasks is *multiplicity*. As shown in Figure 4, roles may have multiple contracts, as there can be multiple program elements cast in the same role. To indicate the number of required contracts each role has the multiplicity property that defines the minimum and maximum number of contracts for that role in relation to its dependencies. At this point, to avoid confusion, we may just think that multiplicity denotes if the role stands for a single program element or a set of elements. This is more clarified in Section 4.2 when we discuss the casting process.

According to the dependencies and multiplicity, if there is not enough program elements cast in the role, the pattern engine advises to provide the missing elements. Practical limits for multiplicity are: one to one (1), zero to one (?), one to infinity (+), and zero to infinity (*). A specialization pattern with a collection of roles with dependencies and multiplicity settings makes the environment capable of representing and implementing a complicated architectural design having infinite number of related program elements.

4.1.3 Properties: Constraints and Templates

A role is usually representing a particular kind of program element. Accordingly, we discuss of class roles, method roles, field roles, and so on. For each kind of role, there is a set of *properties* that can be associated with the role; the meaning of the role is defined with these properties. Without this additional semantics, the specialization pattern could only describe the design and the solution in the level of abstract participants and their dependencies without knowing the actual meaning of those participants and how they should be implemented.

A property may specify a requirement for the concrete program element cast in the role; the property is violated if the provided element doesn't fulfill this requirement. We call such properties as *constraints*. For instance, a specific "inheritance" property may be used to denote that a Java class should extend a particular base class. Other constraint properties could be, for example, "overriding" and "return type" for method roles. To provide structural validation, the pattern engine must keep track of broken constraints and notify the application developer to correct the situation.

Some properties, in turn, can be used for code generation or documentation rather than specifying requirements for program elements. We call these properties as *templates*. For instance, a role may have a "default name" property that specifies the name of the represented program element; this name can then be used when the specialization wizard generates a default implementation for that program element. Other properties could be, for instance, "task title" to generate a text that notifies the application developer when a new program element should be cast in the role, "description" to generate role-specific documentation, and "default implementation" to generate implementation for a new program element.

Roles with template properties enable context-sensitive documentation and source code generation that adapt to the terms and structure of the application. This is possible because a template may refer to another role (making the owner role depend on that other role) and the pattern engine can use the implementation context to generate text. That is, the previously cast program elements are known by the environment, making it possible to use this knowledge to generate dynamically updated instructions, where changes in the casting process affect the documentation.

4.2 Casting Process

So far, it has been discussed that specialization patterns consist of roles having dependencies, multiplicity settings, and properties. When a specialization pattern is used, its roles are cast to the program elements. However, during the casting process, the complexity and terminology of specialization patterns is not shown to the application developer; instead, he proceeds step-by-step by doing small application-specific programming tasks.

The pattern engine considers the given specialization pattern as a *pattern graph*; this is discussed in Subsection 4.2.1. Based on the given graph, the pattern engine generates tasks to guide the application developer during the casting process; this is explained in Subsection 4.2.2. To illustrate the mechanism a short example is given in Subsection 4.2.3.

4.2.1 Pattern Graph

Roles of the specialization pattern can be organized into a *pattern graph* [Hakala et al. 2001a, 2001d]. In this graph roles are denoted by nodes while directed edges are used to denote the dependencies between them. The multiplicity setting (1, ?, +, *) is shown after the role name indicating if the role stands for a single program element or a set of elements. For instance, in Figure 5, there is a pattern graph of a simple specialization pattern called Inheritance. This specialization pattern specifies a common situation where the application developer derives subclasses from a particular base class and wants to override the methods of the base class. The required program elements, namely the base class and its subclasses, overriding methods, and methods to be overridden are represented as roles. To demonstrate different kind of roles, the field and implementation roles have been added to the specialization pattern, too.

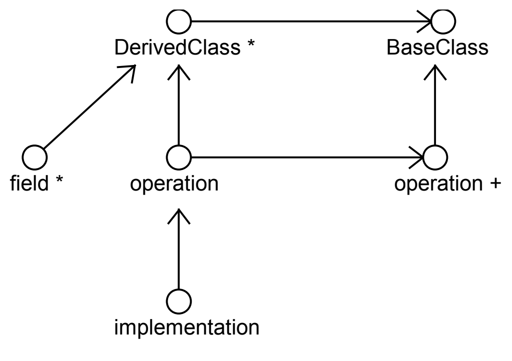


Figure 5. Pattern graph of the Inheritance pattern.

The pattern graph in Figure 5 has enough information to demonstrate the casting process and the overall structure of the given specialization pattern. However, as discussed in Subsection 4.1.3, roles themselves have no fine-grained semantics. Instead, roles must be augmented with constraint- and template properties. The role-specific properties of the Inheritance pattern are shown in Figure 6. References to other roles are indicated with <# r> tags, where r is the name of the referred role. Note that the reference also implies a dependency between the roles; a role with a property that refers to another role cannot be cast until another role is cast and provides the required program element for the property. In addition, if the role is referred within a template property, the tag is replaced by the name of the program element(s) cast in the referred role; this enables the adaptive documentation. More convenient notation to present specialization patterns is given in Section 4.3.

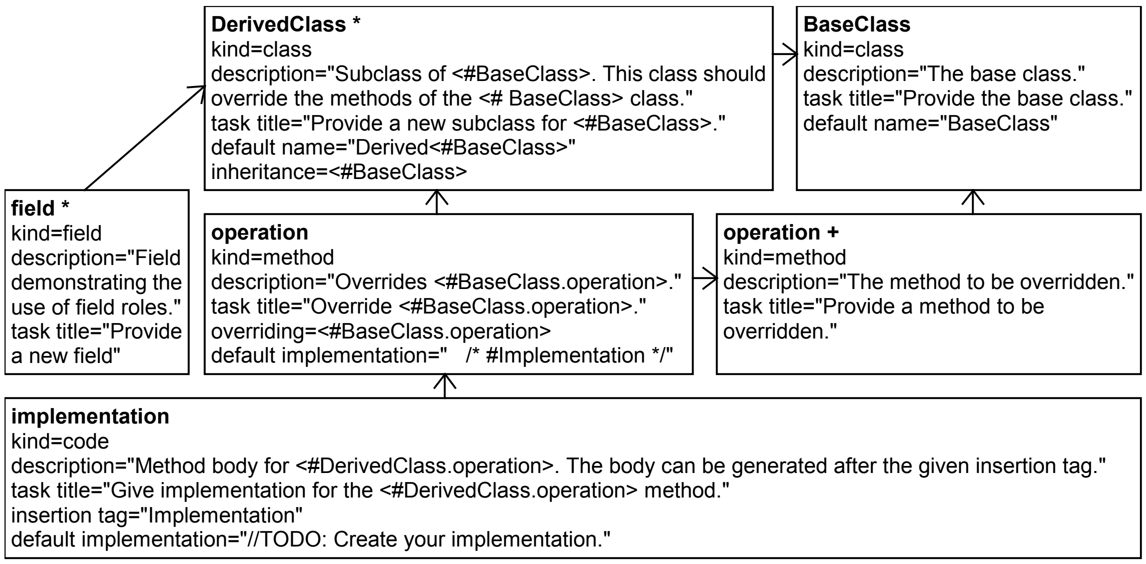

DerivedClass *
kind=class
description="Subclass of <#BaseClass>. This class should override the methods of the <# BaseClass> class."
task title="Provide a new subclass for <#BaseClass>."
default name="Derived<#BaseClass>"
inheritance=<#BaseClass>
BaseClass
kind=class
description="The base class."
task title="Provide the base class."
default name="BaseClass"
field *
kind=field
description="Field demonstrating the use of field roles."
task title="Provide a new field"
operation
kind=method
description="Overrides <#BaseClass.operation>."
task title="Override <#BaseClass.operation>."
overriding=<#BaseClass.operation>
default implementation="   /* #Implementation */"
operation +
kind=method
description="The method to be overridden."
task title="Provide a method to be overridden."
implementation
kind=code
description="Method body for <#DerivedClass.operation>. The body can be generated after the given insertion tag."
task title="Give implementation for the <#DerivedClass.operation> method."
insertion tag="Implementation"
default implementation="//TODO: Create your implementation."

Figure 6. Pattern graph with role-specific properties.

4.2.2 Generating Tasks

The pattern engine uses the given specialization patterns to generate tasks. In case of Java programs, typical tasks include creation of new program elements, like classes and methods, as well as modifying their source code to obey the constraints. Figure 7 illustrates the mechanism; computation to automatically update the tasks list is possible because the pattern engine knows the existing contracts, roles, and program elements. It also gets notified if the program elements are modified by the application developer. Tasks are generated role-basis; if a constraint is violated a task is generated to fix the situation. Similarly, based on the multiplicity settings, a task is generated to point out or create a suitable program element if there is not enough program elements cast in the role.

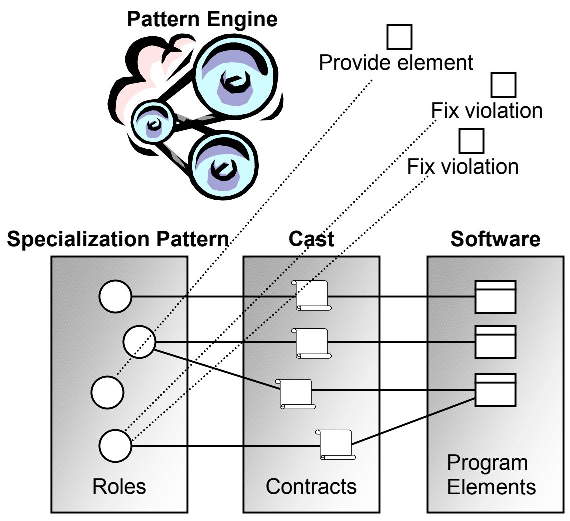


Figure 7. Tasks are generated by the pattern engine.

Tasks are either *mandatory*, *optional*, *unworkable*, or *complete*. A mandatory task must be completed, optional task may be completed, and unworkable task cannot be completed until its state changes to mandatory or optional. The order in which the tasks can be done is defined by dependencies (recall Subsection 4.1.2). For instance, if the role s depends on the role r, the task to provide a program element for s is unworkable until the task to provide a program element for r is done; if the role r defines a specific return type for a Java method, the method cannot be cast in the role s until the return type is fixed.

The mechanism ensures that when starting the casting process for a specialization pattern, the first mandatory or optional tasks are based on the roles that are not depending on any other role. After completing some of these first tasks, other tasks to cast the depending roles become available; thus, properties in those depending roles are requiring some information and when that information becomes available new tasks can be shown in the task list. By using the dependency information and multiplicity settings, the pattern engine can determine the order in which the tasks can be done and if the task is optional or mandatory.

The *casting graph*, illustrated in Figure 8, can be used to represent the state of the casting process. In this graph, nodes denote tasks. Directed edges between the task nodes denote *dependency instances* corresponding the dependencies defined in the pattern specification. Labels of the task nodes are ordered pairs, where the first part denotes the corresponding role the task is based on and the second part identifies the task among other tasks based on the same role. The state of the task (mandatory, optional, or complete) is placed after the label; for convenience, unworkable tasks are not shown.

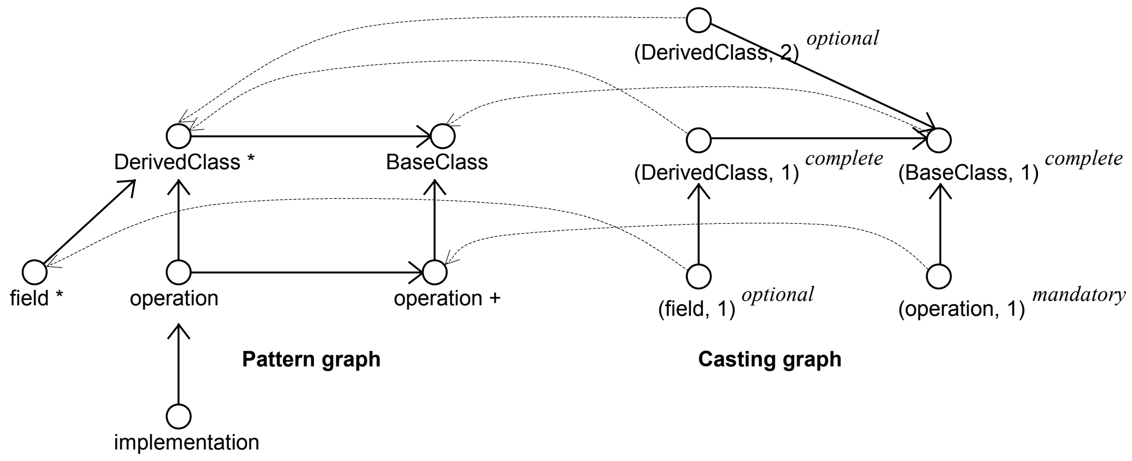


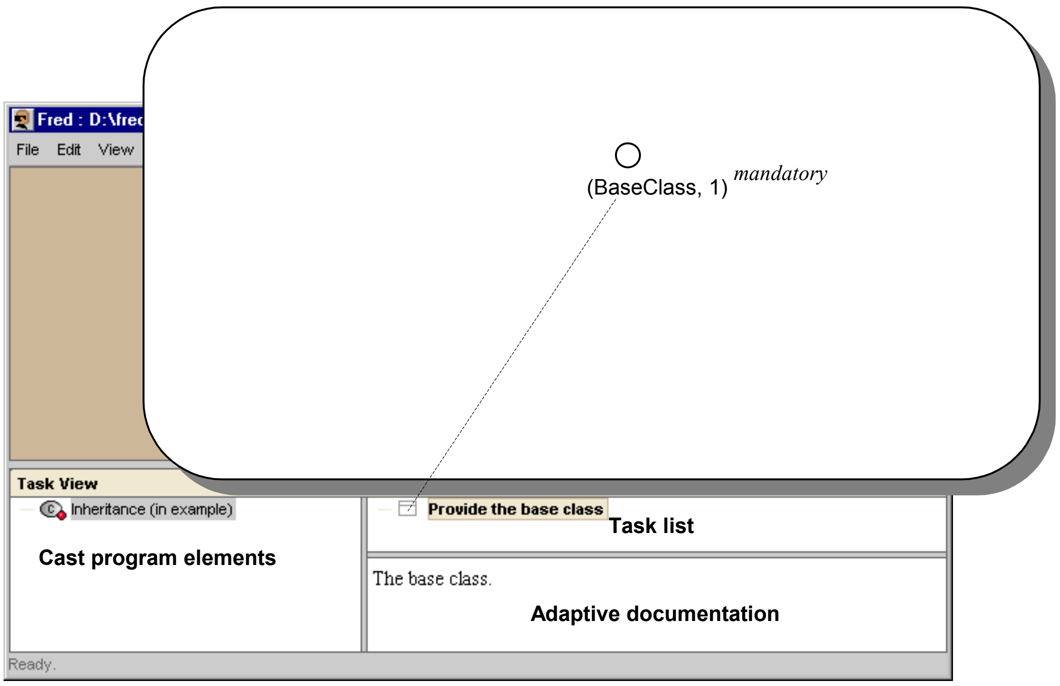
Figure 8. A pattern graph and a casting graph of the Inheritance pattern.

In the figure above, there is a casting graph of the Inheritance pattern. In the example situation, the application developer has already completed the tasks to provide the base class and a new subclass. In the casting graph, one can see that there is a mandatory task to provide at least one operation for the created base class. In addition, there are optional tasks to provide another derived class and a field for the created subclass. Note that the task based on the DerivedClass.operation role is unworkable, as this role depends on BaseClass.operation; the application developer must first provide the method to be overridden before he can complete this task.

To use pattern specifications effectively, the pattern engine needs a development environment that works in the interaction with the user. For instance, the FRED environment is interactive, notifying the pattern engine whenever the application developer manipulates the source code. The pattern engine compares the current stage of the source code to the given pattern definition and updates the casting graph by changing states of the tasks and adding new ones. FRED shows this casting structure as a task list related to the current problem; by doing the tasks in the task list, the application developer generates or modifies the source code, thus enforcing the pattern engine to evaluate again. Descriptive names, as well as task-specific adaptive documentation and source code suggestions are obtained from the properties belonging to the role that the task is based on.

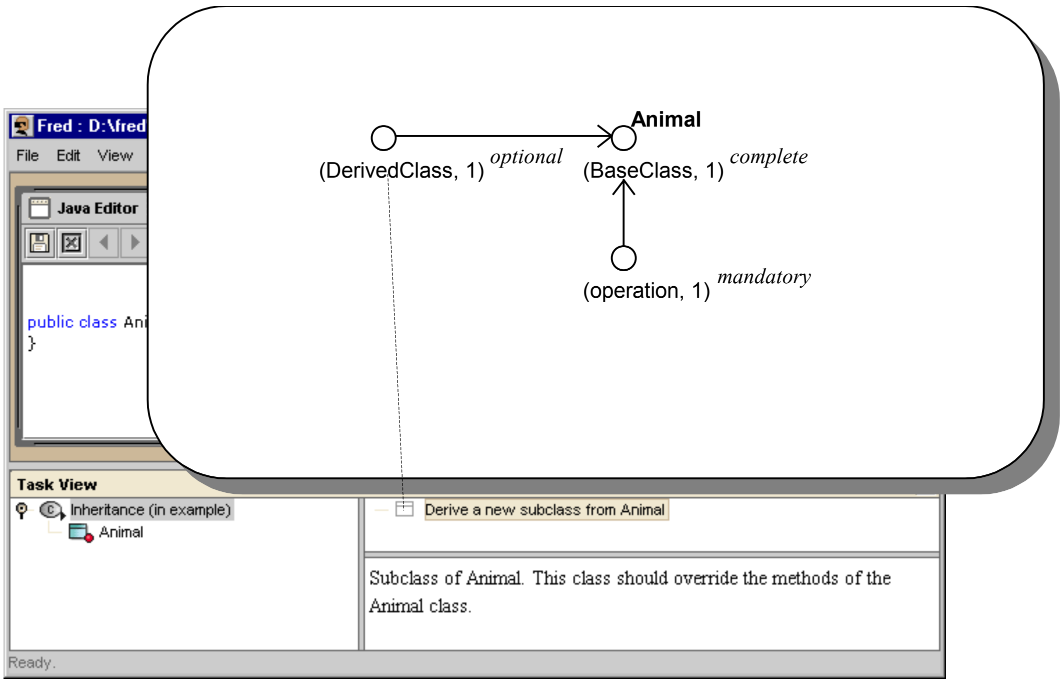
4.2.3 Casting Example

To demonstrate how the pattern engine casts roles and generates tasks, we use the FRED environment to adapt the Inheritance pattern step-by-step in figures 9 – 12.



The base class.

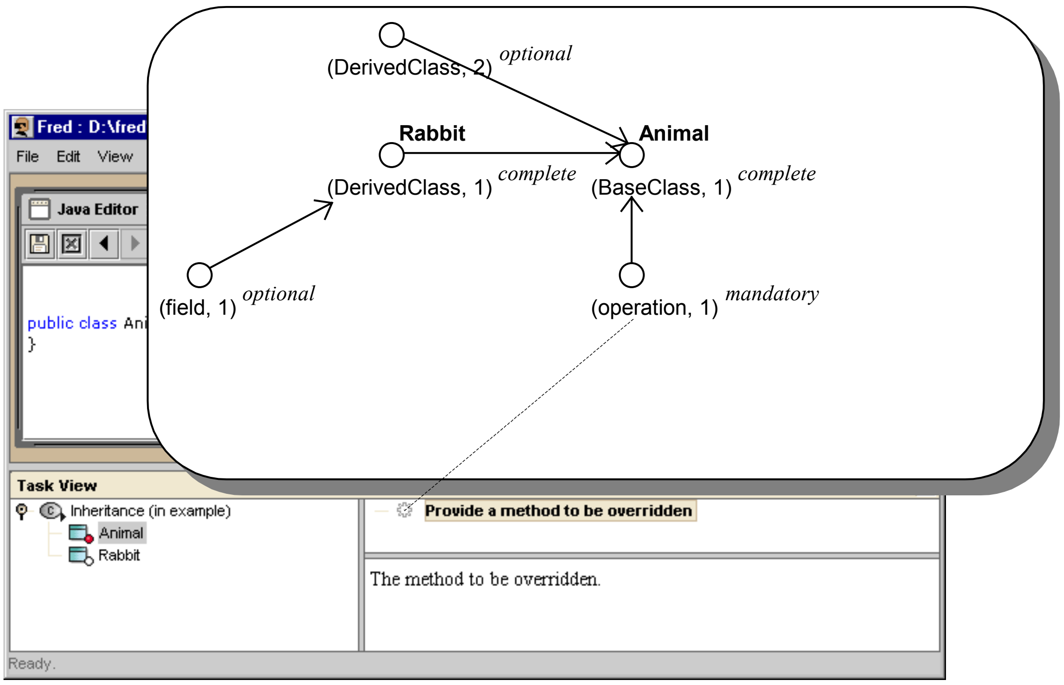
Figure 9. Casting the roles of the Inheritance pattern: Provide the base class.



```
public class Ani
}
```

Subclass of Animal. This class should override the methods of the Animal class.

Figure 10. Casting the roles of the Inheritance pattern: Derive a new subclass from Animal.



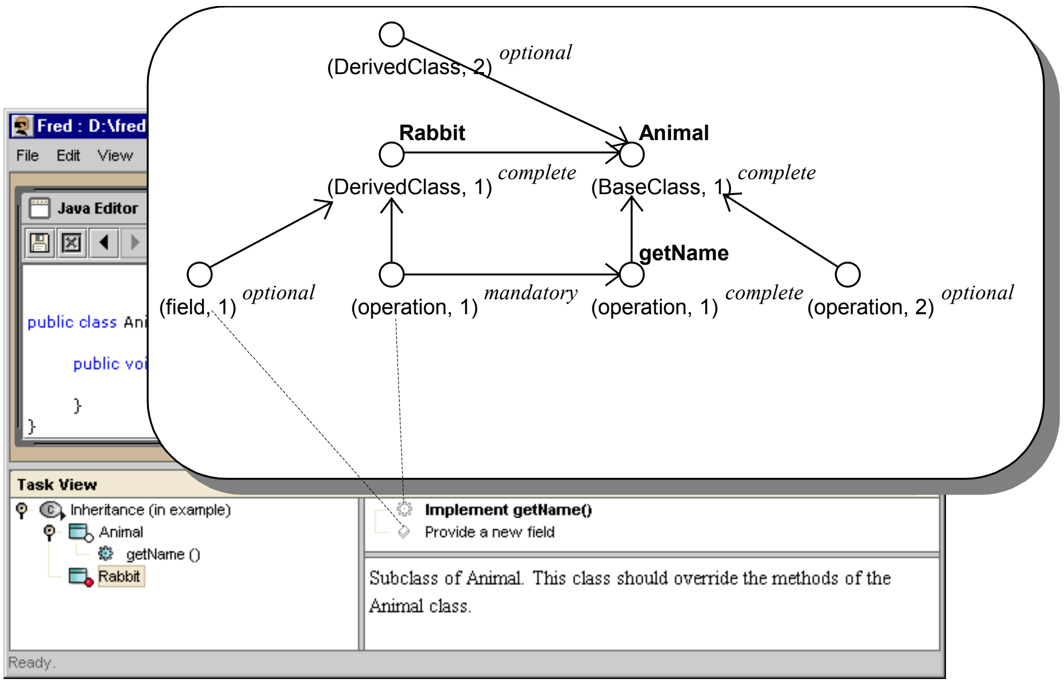
```
public class Ani
}
```

Inheritance (in example)
Animal
Rabbit

Provide a method to be overridden

The method to be overridden.

Figure 11. Casting the roles of the Inheritance pattern: Provide a method to be overridden.



```
public class Ani
    public voi
    }
}
```

Inheritance (in example)
Animal
getName ()
Rabbit

Implement getName()
Provide a new field

Subclass of Animal. This class should override the methods of the Animal class.

Figure 12. Casting the roles of the Inheritance pattern: Implement getName().

In Figure 9, the application developer has started to use the Inheritance pattern; the pattern engine takes the pattern definition and notices that all other roles depend on the BaseClass role directly or indirectly (recall the pattern graph in Figure 5). Thus, the algorithm creates a new task (BaseClass,1) to enforce the application developer to provide the base class. The task is mandatory because of the multiplicity settings of the role. FRED shows this task in the task list; the descriptive title is obtained from the "task title" property of the role. In addition, the kind of the required element (class) and its proposed default name are specified by the properties, too. Note that it is up to the user interface how the specialization pattern and tasks are presented to the application developer; FRED 1.2 uses a simple tree view that shows the cast program elements on the left and the task list and adaptive documentation on the right.

In Figure 10 the application developer has completed the task (BaseClass,1) by providing a new base class. The state of this task has been changed from mandatory to complete. Next, he can complete the tasks (DerivedClass,1) and (operation,1). Again, FRED shows these tasks in the task list; by selecting a cast program element on the left, the tasks to continue casting are shown on the right. As shown in the figure, the adaptive documentation uses the name of the actual base class provided by the application developer.

As the casting process continues, the application developer completes the optional task (DerivedClass,1) and derives a new subclass from the base class, as shown in Figure 11. Again, the pattern engine evaluates and adds a new optional task (DerivedClass,2) to the casting graph. This is because the role DerivedClass has multiplicity from zero to infinity; so, there is a chance that the application developer may create another subclass. Note that the pattern engine is continuously notified for the changes in the source code; if the application developer modifies the source code of the subclass and violates the "inheritance" property, a new task is added to the casting graph to enforce the application developer to fix the situation.

In Figure 11, the application developer prepares to create a new method for the base class by executing the mandatory task (operation,1). The situation continues in Figure 12, where the application developer has created the getName method for the base class. The pattern engine generates a new mandatory task to enforce the application developer to override the created method. In addition, because of the multiplicity settings, an optional task appears to create another method for the base class. At some point, when doing the tasks in the evolving task list, the application developer has completed all mandatory tasks and the design has been integrated to the concrete code-level implementation.

4.3 Notation

To facilitate the presentation of specialization patterns in terms of roles and properties a textual notation is needed. To keep the presentation simple and readable, we use *role tables* explained in Subsection 4.3.1. In addition, to illustrate the overall structure of specialization patterns, a simple graphical notation called *pattern diagram* is given in Subsection 4.3.2. To demonstrate the notation and framework-specific specialization patterns, an example is given in Subsection 4.3.3. However, this is not a thorough specification of a complete modeling language like UML [Rumbaugh et al. 1999], but a more simple notation that is adequate to present the structure of specialization patterns.

4.3.1 Role Table

Usually specialization patterns are managed by the pattern engine, but to discuss and explain patterns we need an understandable textual notation. In this thesis, to give specialization patterns a textual format, we use *role tables*. The notation is illustrated in Role Table 1; the given role table specifies the Inheritance pattern discussed in the previous sections.

Inheritance
This specialization pattern describes how to create a base class and derive new subclasses from it.

Role	**Kind**	**Properties**
BaseClass	class	**description:** The base class. **task title:** Provide the base class. **default name:** BaseClass
operation +	method	**description:** The method to be overridden. **task title:** Provide a method to be overridden.
DerivedClass *	class	**description:** Subclass of <#BaseClass>. This class should override the methods of the <#BaseClass> class. **task title:** Provide a new subclass for <# BaseClass>. **default name:** Derived<#BaseClass> **inheritance:** <#BaseClass>
operation	method	**description:** Overrides <#BaseClass.operation>. **task title:** Override <# BaseClass.operation>. **overriding:** <#BaseClass.operation > **default implementation:** /* #Implementation */
implementation	code	**description:** Method body for the <# DerivedClass.operation> method. The body can be generated after the given insertion tag. **task title:** Give implementation for the <# DerivedClass.operation> method. **insertion tag:** Implementation **default implementation:** //TODO: Create your implementation.
field *	field	**description:** Field demonstrating the use of field roles. **task title:** Provide a new field

Role Table 1. Role table for the Inheritance pattern.

With role tables, each role must be described in terms of its kind and properties. Multiplicity is indicated after the role name (1, ?, +, *); for convenient, if not explicitly marked, the multiplicity is one to one indicating that there must be exactly one program element cast in the role. Nesting of roles can be used to specify containment relations, which can be seen as a constraint establishing an implicit dependency between the roles. For instance, if the role r contains the role s, the program element cast in the role s must be enclosed by the program element cast in the role r. Other properties are explicitly enumerated.

The values of constraint- and template properties may refer to other roles; in a role table, such references are of the form <# r>, where r is the name of the referred role. Note that this reference establishes a dependency (recall Subchapter 4.1.2) between the roles. By convention, if <# r> appears within a template property, it is considered as a macro and it is replaced by the name of the program element(s) cast in the role r. In case of constraint properties, references to other roles imply relationships that must be satisfied by the program element(s) cast in the role having the constraint. For instance, <#BaseClass> refers to the Java class cast in the BaseClass role; inside a template this tag is replaced with the name of the base class, while when used with the "inheritance" constraint the pattern engine checks the source code against this reference.

4.3.2 Pattern Diagram

From a pure textual representation, it may be difficult to figure out the overall structure of the specialization pattern. Pattern graphs (recall Subchapter 4.2.1) may help to outline the structure, but a lot of the information, like which roles are enclosing other ones, is not explicitly shown. To make the structure of a specialization pattern more illustrative, the overall composition of roles can be given as a *pattern diagram* shown in Figure 13. Note that the example diagram corresponds the pattern definition graph shown in Figure 5, but some of the constraints are denoted by visual indicators or with name conventions.

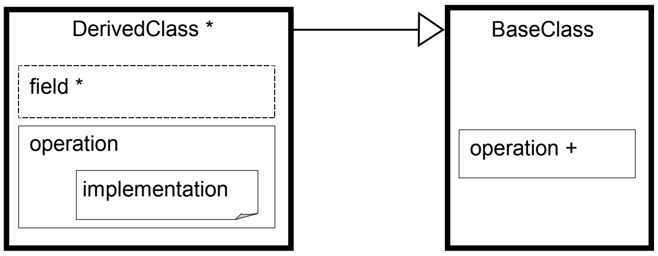


Figure 13. The pattern diagram for the Inheritance pattern.

The symbols used in the pattern diagram are summarized in Table 1. In a pattern diagram, roles to represent common language elements, like classes, constructors, methods, and fields are quite apparent, but roles can also be used to represent an arbitrary piece of code and issues that have no direct analogy with the current programming language; for instance, to remind that the application developer should update a particular configuration file. In the pattern diagram, class roles have thick, method roles thin, and field roles dashed border. A code role is shown as a box with a bent corner. Dependencies are denoted by different arcs, name conventions, or visual composition, where enclosed roles depend on enclosing ones.

■	Class role. This role represents a Java class.
□	Method role. This role represents a Java method (or constructor).
⬚	Field role. This role represents a Java field.
▭	Code role. This role represents a piece of code (e.g., a method body).
▢	Issue role. This role can be used to remind important steps or to group other roles.
──▷	Inheritance dependency between class roles. Corresponds the "inheritance" constraint.
──→	Dependency between roles.
1, ?, +, *	Multiplicity symbols. Multiplicity defines the minimum and maximum number of program elements that can be cast in a single role in respect to its dependencies. Practical combinations are: one to one (1), zero to one (?), one to infinity (+), and zero to infinity (*). For convenient, if no multiplicity symbol is attached to the role, the multiplicity is one to one.

Table 1. Symbols for pattern diagrams.

4.3.3 Example

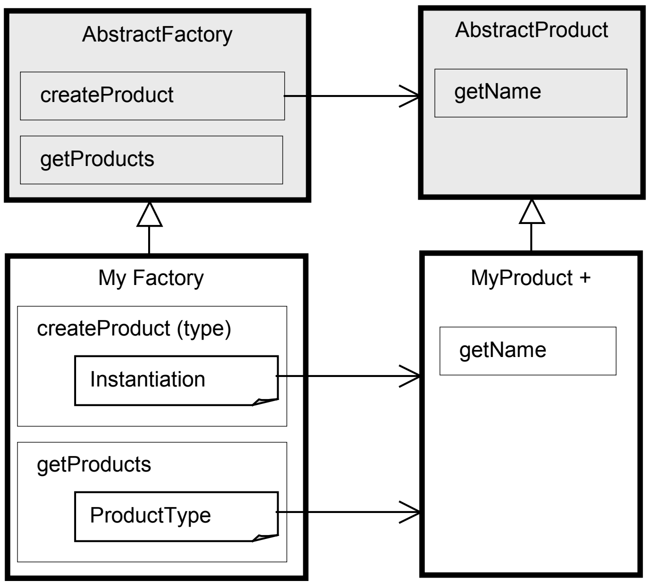
To demonstrate the use of role tables and pattern diagrams, we have constructed the ApplicationFactory pattern shown in Role Table 2. It describes a solution for the corresponding specialization goal that was presented in Figure 2. This particular specialization pattern specifies how the application developer should create a factory class that is needed to launch applications using the example MVC framework.

Naturally, before the specialization pattern in Role Table 2 can be used, the essential framework elements must be pointed out by the pattern modeler; otherwise it may be difficult for the application developer to find out the required base classes or methods to be overridden. Thus, the framework developer creates the specialization pattern and casts the essential program elements in the framework-specific roles. The application developer, in turn, must provide the application-specific elements. To make this division among the roles more explicit, they have been accordingly grouped in the role table. In addition, we have followed the naming convention that we have found useful in framework-specific specialization patterns; if a role is assumed to be cast to a unique program element of the framework, it has the same name as the element.

As explained in the previous sections, the casting process begins as the pattern engine generates tasks based on the given pattern specification. In case of the ApplicationFactory pattern, the first tasks are pointed to the framework expert to provide the base classes and the methods to be overridden. When using the framework, the application developer continues the casting process and executes the tasks to create suitable subclasses and to override the required methods. All the time, the environment keeps track of the bindings between the roles and the program elements, adjusting the documentation and augmenting the task list. The mechanism to deliver specialization patterns from the framework project to the application is explained in Section 5.3, when we are discussing the FRED environment.

AbstractFactory

This specialization pattern describes how to create product classes and a factory class to instantiate them.



Roles played by program elements in the framework

Role	Kind	Properties
AbstractFactory	class	**description:** Base class for factory classes.
createProduct	method	**description:** Creates and returns the given product. Override in subclasses.
getProductTypes	method	**description:** Creates a new array containing a class object for each product class. Override in subclasses.
AbstractProduct	class	**description:** Base class for products.
getName	method	**description:** Returns the product name. Override in subclasses.

Application-specific roles

Role	Kind	Properties
MyFactory	class	**description:** The factory class. This class is used to create products and it must extend <# AbstractFactory>. **task title:** Provide the factory class. **inheritance:** <# AbstractFactory>
createProduct	method	**description:** Implements <# AbstractFactory.createProduct>. **task title:** Implement <# AbstractFactory.createProduct> to create products. **overriding:** <# AbstractFactory.createProduct> **default implementation:** /* #Instantiation */ throw new ClassNotFoundException("Unsupported product type " + <# type>);
type	parame-ter	**description:** The parameter expressing the required type. **type:** Class

Instantiation	code	**description:** Within the <# MyFactory.createProduct> method create and return the <# MyProduct> product.
		task title: Instantiate the <# MyProduct> product.
		default name: Instantiate <# MyProduct>
		insertion tag: Instantiation
		default implementation:
		if (<# type> == <# MyProduct>.class) {
		  return new <# MyProduct>();
		}

getProductTypes	method	**description:** Implements <# AbstractFactory.getProductTypes>.
		task title: Implement <# AbstractFactory.getProductTypes> to return available product types.
		overriding: <# AbstractFactory.getProductTypes>
		default implementation:
		return new Class[] {
		/* #ProductType */
		};

ProductType	code	**description:** Within the <# MyFactory.getProducts> method add the product class <# MyProduct> to the class array. This array contains all available application types. **task title:** Add the product class <# MyProduct> to the class array. **default name:** Type <# MyProduct> **insertion tag:** ProductType **default implementation:** <# MyProduct>.class,
MyProduct +	class	**description:** The product class. This class must extend <# AbstractProduct>. **task title:** Provide a new product class. **inheritance:** <# AbstractProduct>
getName	method	**description:** Implements <# AbstractProduct.getName>. **task title:** Implement <# AbstractProduct.getName> to return the product name. **overriding:** <# AbstractProduct.getName> **default implementation:** //TODO: Return the product name.

Role Table 2. ApplicationFactory pattern.

4.4 About Writing Patterns

Though the purpose of this thesis is not to be a comprehensive reference manual or style guide for pattern writing, it is relevant to give a short introduction to the pattern modeling. Subsection 4.4.1 illustrates how the common process to write patterns is related to the process to create specialization patterns. Subsection 4.4.2, in turn, presents a simple method to outline specialization patterns. To read more about methodologies and conventions to write (design) patterns see, for example, Meszaros and Doble [1998]. The actual formalism to construct specialization patterns is given in Section 5.4.

4.4.1 Pattern Modeling and Specialization Patterns

Like developing object-oriented frameworks, writing good patterns is difficult; 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]. Creating patterns is an iterative process that requires deep understanding of the problem. The validity of patterns 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.

According to Gamma et al. [1995], a 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. In a way, also specialization patterns have these elements; particularly each specialization pattern has a solution in terms of roles and their interactions. In addition, specialization patterns may also have problem and consequence descriptions as template properties or some other informal documentation attached to them. However, it should not be forgotten that specialization patterns are formal representations making them difficult to use without a tool-support, though a tool may be used to generate, for instance, informal HTML documentation from the role specifications. After all, converting an informal pattern specification, like a design pattern, into some fomalism usually requires trade-offs of the universality of the original pattern.

Specialization patterns are represented with a formal pattern specification language and evaluated with the pattern engine; this makes the process to write and test specialization patterns to resemble more like a programming project. However, the general issues taking into consideration when writing patterns hold also with specialization patterns. For instance, Buschman et al. [1996] summarizes the following criteria that patterns 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*. The interested persons trying to use the patterns should contact the pattern author(s) 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.

In case of specialization patterns, the architecture-oriented task-driven system provides tools to create and evaluate them in practice. During this process, non-anonymous review and writer's workshop can be seen as a group of testers or clients trying to use the constructed specialization patterns with the environment. Specialization patterns – particularly the framework-specific ones – are not necessarily very general but try to describe a practical solution for a specific programming problem. Particularly, if a set of specialization patterns is going to represent the specialization interface of a framework, they should be thoroughly tested and evaluated before publication. Feedback from other framework experts and the application developers is essential to provide a comprehensive specialization support.

4.4.2 Outlining Specialization Patterns

When creating specialization patterns, the pattern modeler should think the design in terms of roles representing the required program elements. To demonstare the modeling process, Figure 14 presents a simple framework that can be used to draw figures. The framework provides an user interface where the user can select among figure types and draw them by pressing the "Draw" button. Creating new figure types can be seen as a specialization goal; the application developer creates new figure classes by subclassing the Figure base class. These new figure types are provided to the framework system by deriving a manager class from the FigureManager base class and by overriding the initFigures method.

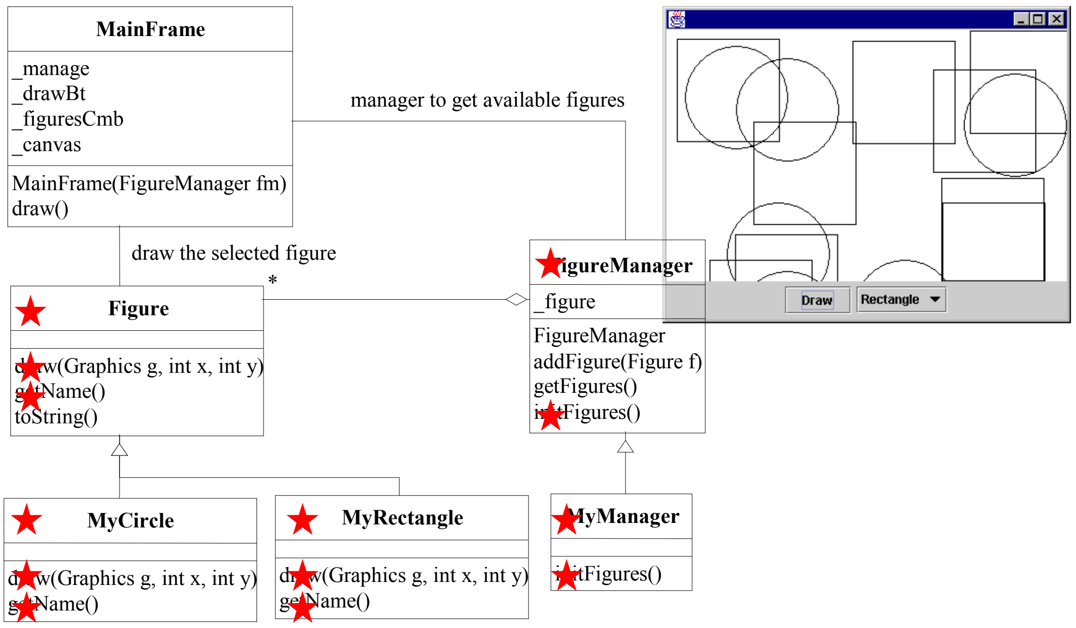


Figure 14. Example specialization and its essential participants.

In the figure above, the pattern modeler has identified the need of new figure types as a specialization goal; to analyze the participants he has made an example specialization with couple of figure types and the manager class. By examining this example specialization, the pattern modeler has marked the program elements as participants that should be represented when the solution is described. Note that this includes both the existing program elements provided by the framework and the program elements that must be provided by the application developer. Usually the application-specific part must be abstracted to support a variety of possible implementations. For instance, the MyCircle and MyRectangle classes obviously require a common role in the final specialization pattern.

After figuring out the participants, the skeleton of the specialization pattern can be outlined, for instance, by drawing a pattern graph or a pattern diagram as described in Subsection 4.2.1 and Section 4.3. In Figure 15, the pattern modeler has drafted the overall structure of the Figure pattern. Roles of the pattern graph are based on the observations made during the example specialization.

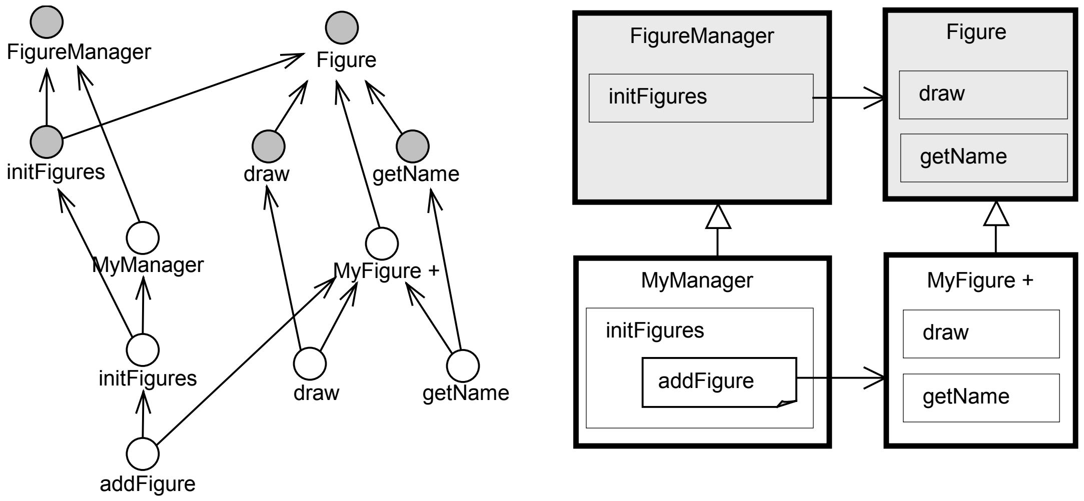


Figure 15. Pattern graph and pattern diagram for the Figure pattern

To make the pattern graph in Figure 15 more illustrative, gray nodes are used to represent the roles cast to the framework elements. If a participant in the example specialization needs another participant, a dependency is drawn between the corresponding roles. Similarly, if the role is representing a single program element or a set of program elements, its multiplicity must be set accordingly. In this way, the structure of the specialization pattern emerges from the given example specialization. By using code roles, like addFigure, it is also possible to generate source code dynamically. For instance, the required piece of code to register new figure types is generated inside the initFigures method whenever a new figure type is added to the framework system. Details of the casting process are managed by the pattern engine as discussed in the previous chapters.

Finally, after modeling the overall structure of the Figure pattern, properties are set for each role (recall Subsection 4.1.3). For instance, constraints are added to enforce the inheritance and override relations. In addition, template properties are used to provide adaptive documentation and code templates; for example, the addFigure role representing a piece of Java code may have the "default implementation" property that produces the registration code for new figure types. Thus, the example specialization can be used to figure out the static structure of the specialization pattern(s) in terms of roles. The pattern modeler then completes this structure with more detailed documentation and constraints by augmenting the obtained roles with a set of properties. The outcome can be tested by giving the specialization pattern to the pattern engine and trying to reproduce the example specialization.

Finding roles and their interaction is not always straightforward. Roles must be obtained from examples, informal documents, diagrams, discussions with the domain-experts, or just from the source code of existing applications. This leads to a situation where we have ambiguous information about how to model the specialization pattern. It may be necessary to provide multiple pattern variations to model a single design. Altogether, this makes the pattern modeling iterative; it is common that the pattern modeler (or the application developer using the pattern) notices that some parts of the specialization pattern should be modified. This may be dangerous as the existing role casts may become obsolete. The problem of changing a specialization pattern resembles the problem that occurs if the specialization interface of a framework is changed so that existing specializations become outdated. Therefore specialization patterns should be adequately tested before they are utilized in large scale.

Usually a single specialization pattern is not enough to specify a complicated solution. Instead, multiple specialization patterns may be needed to cover various implementation alternatives and subsystems. The outcome is a kind of pattern language with tightly interrelated specialization patterns. Composing the pattern language to represent, for instance, the specialization interface of a framework, requires the grouping of specialization patterns and determining the order in which they should be used.

The pattern modeler must identify the underlying principles and systems to organize the specialization patterns together into useful configurations.

5 FRED Environment

FRED (FRamework EDitor) is a prototype of architecture-oriented task-driven development environment that utilizes specialization patterns to model and implement design solutions. FRED has been implemented in Java for Java, and it has been used as a specialization wizard for large frameworks. First versions of FRED were released in 1998 and 1999 [Hakala et al. 1997, 1998, 1999a, 1999b]. Since then, the environment and the concept of formalizing and implementing architectural design have been further evolved [Hakala 2000; Hakala et al. 2001a, 2001b, 2001c; Viljamaa 2001]. The current version, FRED 1.2, has been developed in a joint research project between the Tampere University of Technology, and the University of Helsinki. The project has been funded by TEKES (National Technology Agency of Finland) and several software companies.

The aim of this chapter is to demonstrate FRED as a concrete prototypical environment that experiments with the ideas of specialization patterns, specialization wizards, a pattern engine, architecture-oriented programming, and task-driven framework specialization, discussed in the previous chapters. An overview of FRED is given in Section 5.1. FRED has tools to create and use specialization patterns; these are briefly introduced in Section 5.2. Annotating frameworks with specialization patterns and specializing them with FRED is explained in Section 5.3. The actual formalism to declare specialization patterns is presented in Section 5.4. Encountered problems and benefits are discussed in Section 5.5.

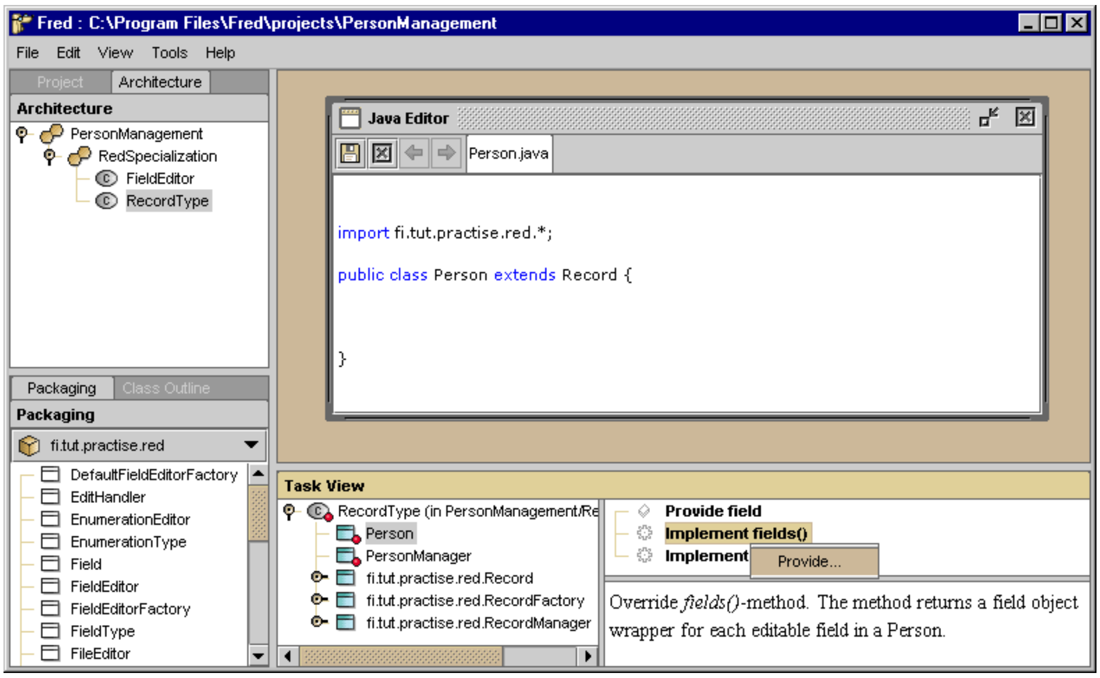
5.1 Overview

One of the constituting ideas of FRED is to provide an architecture-oriented task-driven programming environment that allows the precise specification of architectural design and its use. In practice this means that the system architect or domain expert creates a set of specialization patterns to describe how to implement design solutions; this includes, for instance, the use of design patterns, coding conventions, and framework extension points. Based on the given specialization patterns, the application developer is then guided with small and context-sensitive programming tasks to achieve a specific goal. Thus, FRED supports both the implementation of high-level design solutions – like design pattern [Gamma et al. 1995], architectural conventions [Venners 1998], and the MVC paradigm [Krasner and Pope 1988] - and low-level domain-specific design solutions like how to use a particular framework.

As mentioned in Chapter 4, specialization patterns consist of roles having dependencies, properties, and multiplicity settings. When using a specialization pattern, its roles are cast to program elements. The use of specialization patterns is integrated into the FRED environment, so that pattern-based verifications are made simultaneously with the software development. As a specialization wizard for a particular architecture, FRED maintains a task list in which missing program elements and violated constraints are indicated as tasks. All along, the environment ensures the consistency of the (Java) source code according to the given pattern specifications. The complexity and terminology of specialization patterns is not shown to the application developer; instead, the system architect (e.g., the framework expert) creates the specialization pattern and the application developer follows this encapsulated solution step-by-step by doing ordinary programming tasks appearing in the task list.

As an integrated development environment, FRED contains a number of views or tools. Some of the tools are used to manage the creation, selection, and casting of specialization patterns (Architecture View, Task View, Pattern Catalog, Pattern Editor). Other tools are needed to provide traditional Java development facilities, like tools to manage Java projects, source code editing, and compilation (Project View, Packaging View, Class Outline, Java Editor, and so on). Using specialization patterns with these tools is closely integrated to the software development process, making FRED an architecture-sensitive typing system in which violations against the architecture are instantly notified to the application developer.

A typical user interface of FRED is shown in Figure 16. As an example, the application developer has opened a new Java project called PersonManagement to create a Java application that stores personalia. At some point, the application developer has noticed that a particular framework, in this case the Red framework [Hakala et al. 2001d; Pree and Koskimies 1999], could be used to provide the user interface facilities to manage the information (this framework is used as an example in Subsection 5.3.2). The selected framework and its specialization patterns – provided by the framework developer - are shown in the Architecture View (see Subsection 5.2.3). One of the specialization patterns associated with the selected framework is the RecordType pattern that helps the application developer to integrate application-specific data structures to the framework system; in the figure, the application developer has opened this specialization pattern in the Task View (see Subsection 5.2.4). By doing the tasks in the task list (recall the casting example in Subsection 4.2.3), either by pointing out the required program elements or by letting the FRED environment to generate default implementations, the application developer eventually specializes the framework and creates a person data structure compatible with the framework system.



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

public class Person extends Record {


}
```

Override *fields()*-method. The method returns a field object wrapper for each editable field in a Person.

Figure 16. The user interface of FRED 1.2.

Note that we call the user interface in Figure 16 "typical", because the user can customize FRED by opening and closing tools and dragging them around. In addition, the FRED environment can be extended with plug-in extensions, making it easy to test new ideas and conventions. In fact, the pattern engine and the current tools are implemented as plug-ins. Similarly, it sounds very interesting to explore the possibility to integrate the pattern engine plug-in and the pattern tools with some commercial Java development environment (e.g., Visual Cafe, JBuilder, or VisualJ++). This is shortly illustrated in Subsection 5.5.3. However, the FRED API goes beyond the subject of this thesis.

5.2 *Pattern Tools*

FRED 1.2 has four tools to manage specialization patterns. The Pattern Editor is a semi-graphical programming tool that is used to give the exact pattern specifications. The Pattern Catalog is used to browse the created specialization patterns in the pattern repository. The Architecture View manages the specialization patterns used in the current project; it can also be used to import partially cast specialization patterns from other (framework) projects. The casting process is completed with the Task View. Advanced editors and tools may be implemented as the development of FRED continues. The Pattern Editor is presented in Subsection 5.2.1. The Pattern Catalog is discussed in Subsection 5.2.2. The Architecture View is introduced in Subsection 5.2.3. The Task View is presented in Subsection 5.2.4.

5.2.1 Pattern Editor

To give exact specialization pattern declarations, FRED includes a special programming language that is discussed more precisely in Section 5.3. To make it convenient to create specialization patterns with the language, some kind of editor or programming tool is needed; currently, specialization patterns are created with the semi-graphical Pattern Editor shown in Figure 17.

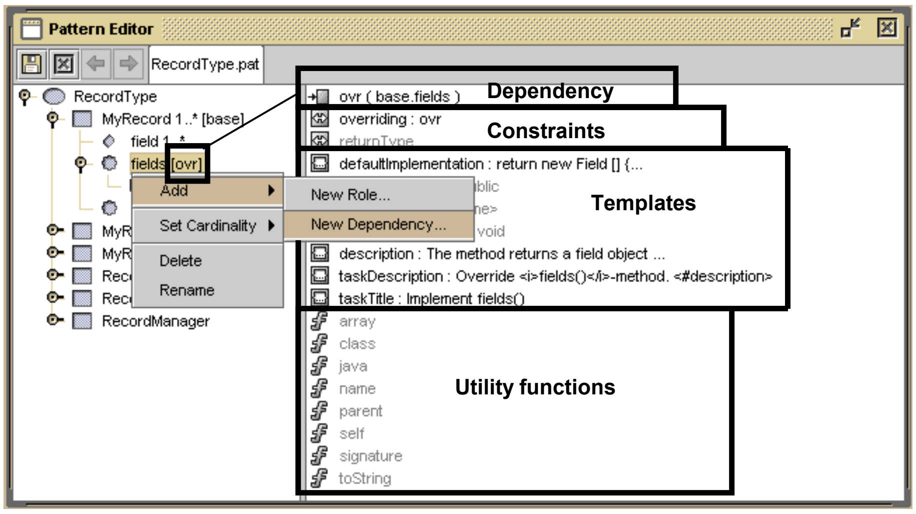


Figure 17. Pattern Editor.

In Figure 17, the framework developer is creating a specialization pattern called RecordType. This pattern is used to cover a particular specialization goal; namely, how the application developer could made his data structures compatible with the user interface facilities provided by the framework. Roles of the specialization pattern are shown as tree-like structure on the left pane. On the right, there are the dependencies and properties associated with the selected role. These are used to define the role-specific constraints and documentation. For instance, the "description" property is a template that defines the documentation for the role.

Using the Pattern Editor is simple; roles are created with a popup-menu while properties are set by clicking a corresponding property and writing or selecting a suitable expression (expressions are explained in Subsection 5.4.3). Creating a role under a particular parent role implicitly establishes a containment constraint between the represented program elements. In addition, multiplicity and dependencies between roles are given with the popup-menu. As a technical detail, note that in the current FRED version a dependency must have a name. This differs from the pattern graph presentation discussed in Subsection 4.2.1 as, in FRED 1.2, the edges in the pattern graph must be named and references to other roles are made by using these dependency names; however this will be changed in the future, as it is possible to determine unambiguously the referred program element by using the role names directly. Created specialization patterns are stored into the pattern repository; they can be browsed with the Pattern Catalog tool discussed in the next subsection. Stored specialization patterns are employed to a particular project with the Architecture View discussed in Subsection 5.2.3.

Currently, as the development of the pattern engine and the used pattern specification language has been evolved, we have noticed that XML [W3C 2001] could be used as a file format for the specialization patterns. A converter has been implemented that transforms specialization patterns into XML-files and back. This approach enables XML-based tools to create and manage pattern specifications. It is also possible to create specialization patterns by writing XML-files without using any dedicated programming tool, though this may be tedious.

5.2.2 Pattern Catalog

Created specialization patterns are stored into the pattern repository; Figure 18 shows the Pattern Catalog tool, which is used to browse this repository. Currently there is no support to categorize or search specialization patterns; instead, they are shown as a simple alphabetically ordered list. This becomes a problem when the number of specialization patterns increases; we need a system that structures the specialization patterns in the repository and guides the application developer to select the most relevant specialization patterns.

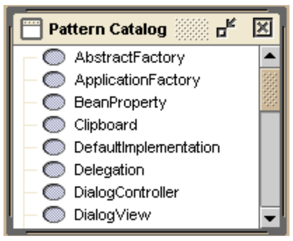


Figure 18. Pattern Catalog.

One possible solution to the categorization problem is to group specialization patterns in the same way than the Java classes are grouped under packages. Pattern providers could publish their specialization patterns under certain "pattern packages". Another approach is to attach some kind of selection criterias to the specialization patterns enabling the use of intelligent pattern selection wizards. Also, as the file format to store specialization patterns is going to be changed into XML, browsers supporting XML become available to inspect the repository.

5.2.3 Architecture View

When employing a specialization pattern from the pattern repository, it is natural to think that the pattern is used as a part of the current software architecture. In FRED, to represent the software architecture and its subsystems, each Java project is considered to have a *specialization architecture* (see Subsection 5.3.1). The specialization architecture may contain other specialization architectures and employed specialization patterns. To manage this logical container we use the Architecture View shown in Figure 19. It gives overall representation of the specialization architectures ( ) and the employed patterns ( ).

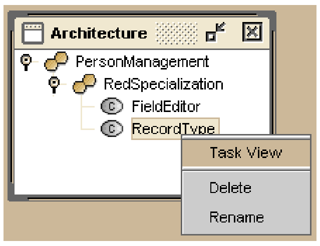


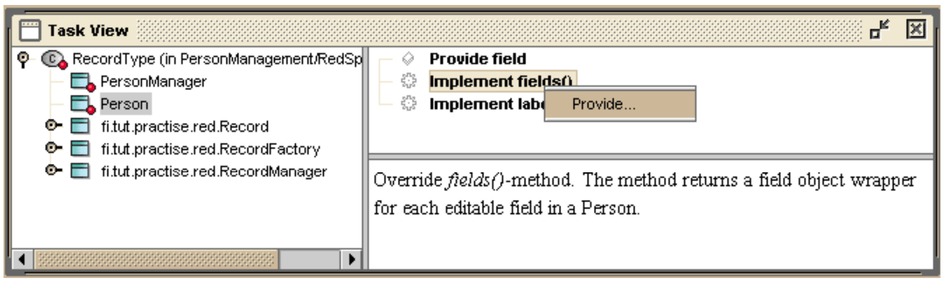
Figure 19. Architecture View

In the figure above, the application developer has opened the Architecture View for the PersonManagement project. RedSpecialization is a specialization architecture that has been imported to the application project from an existing framework project; it contains the specialization patterns provided by the framework developer that can be used to specialize the framework. In addition, one could import more specialization architectures to utilize other frameworks and, of course, employ specialization patterns directly from the pattern repository. To start the casting process, the application developer opens the Task View that is discussed in the next subsection. Working with frameworks is explained more precisely in Section 5.3.

Like other pattern tools in FRED, also the Architecture View could be further developed. The structure of specialization architectures and employed specialization patterns evolves during the development process as the application developer adds or removes specialization patterns and subsystems (like frameworks). To support this process, the Architecture View could provide more information about the purpose of various specialization architectures, specialization patterns, and their intended interactions. In addition, it would be useful to get an overall picture of specialization architectures in terms of participating program elements and their relationships to the employed specialization patterns.

5.2.4 Task View

To cast the roles of the employed specialization patterns to actual program elements, the pattern engine generates tasks for the application developer. The Task View, shown in Figure 20, is a tool to support this process. The tool represents the opened specialization pattern in terms of dynamically updated task list, adaptive role-specific documentation, and program elements that has been cast in the roles so far. The pattern engine of FRED updates the task list repeatedly as explained in Section 4.2. Typically, a tool to complete a task is the source editor but it could also be more dedicated task-specific tool. The source editor is used to write the required code, or the code can be automatically generated, if possible. The FRED development environment is tightly integrated with the casting process and the consistency of the (Java) source code according to the role-specific constraints is verified simultaneously with the normal software development. Violated constraints and missing program elements cause new tasks until the whole pattern is instantiated to the application-specific context.


Task View
RecordType (in PersonManagement/RedSp
PersonManager
Person
fi.tut.practise.red.Record
fi.tut.practise.red.RecordFactory
fi.tut.practise.red.RecordManager
Provide field
Implement fields()
Implement labe
Provide...

Override *fields()*-method. The method returns a field object wrapper for each editable field in a Person.

Figure 20. Task View.

As an example, in Figure 20, the application developer has opened the employed RecordType pattern that comes with the imported framework-specific specialization architecture. By doing meaningful and relatively small programming tasks, the application developer has already created the Person and PersonManager classes. Different visual indicators can be used to illustrate optional and mandatory tasks, as well as to even more clarify the casting process. For instance, red circles in the left pane indicate that there are still some tasks to do with the created classes.

To further develop the Task View, it could be augmented with features that help the application developer to estimate the required amount of work. For instance, the tool could estimate the number of required tasks. Also, advanced users may need more information about the underlaying specialization pattern and its roles. For the pattern modeler it would be nice if he could modify the underlaying specialization pattern during the casting process. This approach probably leads, in some level, to the integration of the Pattern Editor and Task View.

5.3 Using Frameworks: Patterns across Architectures

By definition, software architecture is the design of the subsystems and components of the software system and relationships between them [Buschmann et al. 1996]. Patterns can be seen as parts or building blocks of this entity, describing and encapsulating architectural rules and implementation instructions. In FRED, the concept of software architecture and its subsystems is represented in terms of *specialization architectures*. Each project is considered to have a specialization architecture that, in turn, may contain other specialization architectures and employed specialization patterns. The concept of specialization architecture is essential in order to understand how a set of partially instantiated specialization patterns can be passed through the framework project to the application.

Specialization architectures are discussed in Subsection 5.3.1. Organizing specialization patterns in framework projects is explained in Subsection 5.3.2. Using the organized specialization patterns in the application project is discussed in Subsection 5.3.3.

5.3.1 Specialization Architecture

Specialization patterns are used to implement design solutions in software projects. To group the employed specialization patterns inside the project, FRED uses *specialization architectures*. Roughly, the specialization architecture can be seen as an expression of the actual software architecture or its subsystem. However, in FRED, their function is to be logical containers for other specialization architectures and employed specialization patterns; thus, they are not necessarily a comprehensive or accurate image of the actual software architecture. When a specialization pattern is employed from the pattern repository, it is attached to the selected specialization architecture in the Architecture View (recall Subsection 5.2.3).

Each project creates its own system of specialization architectures with employed specialization patterns. If the project is a framework, it may have specialization patterns that will guide the application developer during the specialization process. Hence, to enable the use of specialization patterns originally employed in the framework project, specialization architectures can be imported from one project to another. Now, if the application project imports specialization architectures from the framework one, the contained specialization patterns are imported, too. Thus, for each specialization architecture, the employed specialization patterns act as an interface to adapt the architecture. The principle is illustrated in Figure 21.

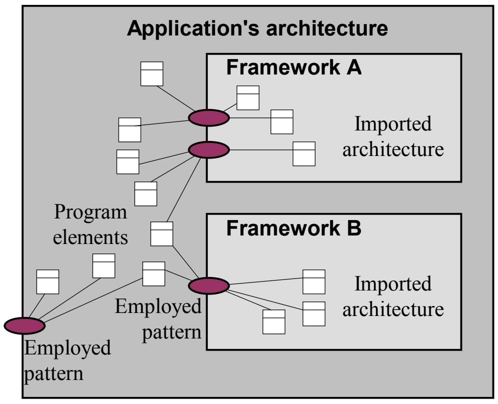


Figure 21. Specialization patterns across specialization architectures.

The use of specialization architectures is a kind of architectural composition. When utilizing a framework, its specialization architectures become a part of the application's architecture. With the imported specialization architecture and its specialization patterns, the application developer continues the casting process and completes the specialization. In addition, the application developer can import other architectures (or the same architecture more than once) or employ more general specialization patterns directly from the pattern repository. Also, the categorization of projects to a framework- or an application project is not strict; intuitively, a project becomes a framework if it has some uncompleted specialization patterns that may be used by other projects.

5.3.2 Framework Project: Pattern Organization Phase

Specialization patterns are employed from the pattern repository and organized into specialization architectures. These specialization architectures can then be imported by other projects to continue the casting process. Therefore, to provide a comprehensive pattern language to specialize the framework with FRED, the framework expert must create the required specialization patterns, employ them to the framework's architecture, and point out the essential framework elements by casting the framework-specific roles. We call this the *pattern organization phase*.

To briefly demonstrate the pattern organization phase, a short example is given here. *Framelets* are small frameworks consisting of handful of classes and used as reusable, tailorable building blocks for creating components [Pree and Koskimies 1999]. The Red framework is a simple framelet used to demonstrate FRED [Hakala et al. 2001d]. Red provides user interface facilities to maintain a list of Record-objects and to edit their fields. Typically, the Red framelet is used by deriving a new Record subclass with some application-specific fields. Once the application developer has created this new record type, the framelet provides facilities to automatically generate dialogs to set the values of the instantiated Record-objects. In Figure 22, an application is started that uses the Red framelet and defines a new record type for personalia. In the figure, the framelet has provided a dialog to update the fields of the selected person.

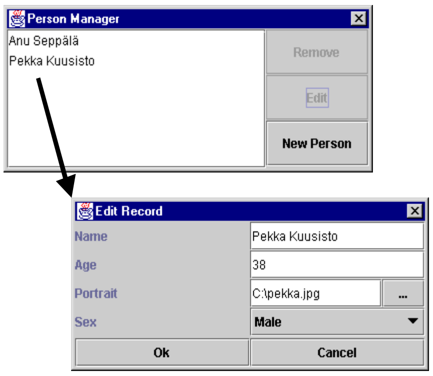


Figure 22. Typical views provided by the Red framelet.

Clearly, from the application developer's standpoint, one of the specialization problems is how to create a new record type that complies with the framework system. The framework expert, in turn, identifies this request as a specialization goal pursued by the application developer. To construct a specialization wizard for Red, the framework expert models (with the Pattern Editor tool) the expected specialization as a specialization pattern called RecordType. Solutions to other specialization goals, like how to support new types of fields in addition to standard types, may be modeled, too. Together these specialization patterns compose a pattern language to specialize Red.

The created specialization patterns are stored into the pattern repository. To organize specialization patterns inside the framework's specialization architecture and to cast the framework-specific roles, the framework expert must employ the created patterns in the Architecture View and cast the framework-specific parts in the Task View. This is illustrated in Figure 23. Without this organization process, the application developer himself would have to employ these specialization patterns from the repository and cast the framework-specific roles.

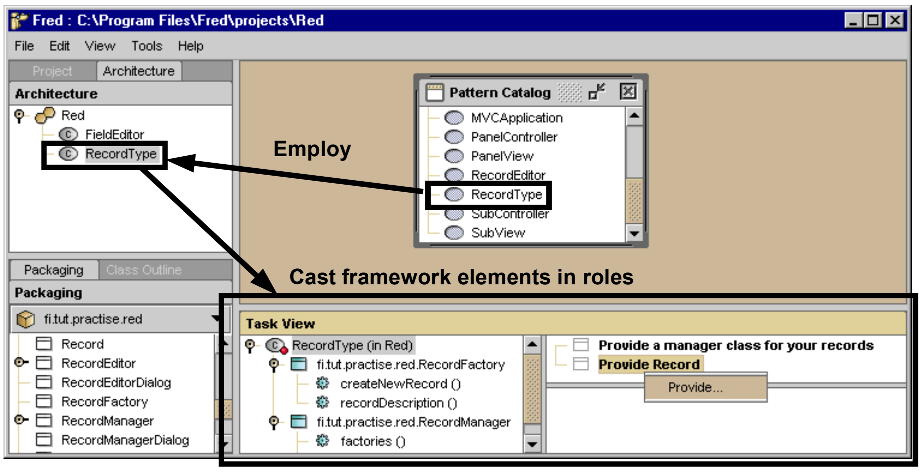
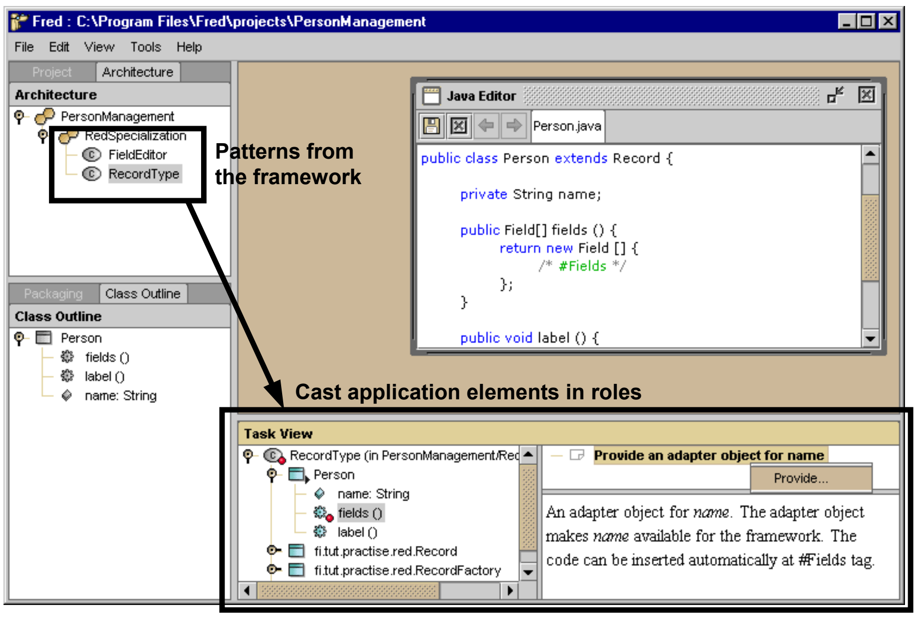


Figure 23. Organizing specialization patterns in the framework project.

As a conclusion, making framework-specific specialization patterns with FRED 1.2 has the following phases; creation of the patterns, employing them to the framework's specialization architecture, and casting the framework-specific roles. It may be considered how this process could be simplified in the future.

5.3.3 Application Project: Continue Casting

Let us now assume that the application developer wants to create a person manager application using the Red framelet discussed in the previous subsection. After creating a new project for the application, the application developer selects the framelet from a library of existing projects. The specialization architecture of Red is imported to the application project and added as a subsystem to the application's specialization architecture. The situation is illustrated in Figure 24. The Architecture View shows that there are two specialization patterns that came with the imported Red framework. The application developer decides to work with the RecordType pattern and opens it in the Task View. In the figure, the application developer has completed tasks to create a new record type (Person) and some of its methods and fields. Small red circles indicate the mandatory tasks for the application developer. By doing tasks step-by-step, the application developer eventually specializes the Red framelet and enables the Red-based user interface to edit the person repository.



```
public class Person extends Record {

    private String name;

    public Field[] fields () {
        return new Field [] {
            /* #Fields */
        };
    }

    public void label () {
```

Provide an adapter object for name

An adapter object for *name*. The adapter object makes *name* available for the framework. The code can be inserted automatically at #Fields tag.

Figure 24. Using framework-specific specialization patterns in the application project.

As we can see in Figure 23 and Figure 24, both the framework expert and the application developer works with the same user interface and tools when using FRED. There is no restrictions on the application project to become a framework if the application developer adds some partially instantiated specialization patterns. Eventually the system of imported specialization architectures may lead to a chain of projects, where the most abstract project could, for instance, describe a set of abstract interfaces. The second project may, in turn, implement essential parts of these interfaces while a third project completes the specialization. Thus, in a way, FRED supports the use of layered architectural structures.

5.4 Pattern Definition Language (PDL)

In FRED, the formalism to construct specialization patterns is a special class-based object-oriented programming language. Here this language is simply called the Pattern Definition Language (PDL). By definition, a programming language is a notation for writing programs [Sethi 1989]; in this case, the pattern engine is an interpreter [Aho et al. 1986] that takes a program made with PDL as input and makes the computation to implement and integrate the specified design solution into the current software project.

This section is a tutorial giving the main constructs of the language and how they are meant to be used. Typically, the best way to learn programming is by doing; by experimenting and using the language to build up specialization patterns. Subsection 5.4.1 presents the role types and the corresponding PDL base classes. Subsection 5.4.2 explains how role-specific properties are given as functions. Subsection 5.4.3 discusses function expressions. Subsection 5.4.4 explains how to create adaptive documentation with tags.

5.4.1 Role Types

As mentioned in Subsection 4.1.1, a specialization pattern consists of roles. When using PDL to represent specialization patterns, roles are declared by deriving subclasses from the PDL base classes. For instance, to declare a new method role we must derive a new subclass from the MethodClause base class. To give the role-specific properties, some of the functions must be overridden in the derived PDL class. In addition, dependencies and multiplicity can be given. This may sound difficult, but with a programming tool, like the Pattern Editor, it is rather easy to create roles, set dependencies, and write function expressions.

Roles can be categorized to different role types depending on the kind of program element they are used to represent. Currently, PDL is capable to express the role types enumerated in Table 2. Roles to represent Java classes, constructors, methods, and fields are quite evident. However, because specialization patterns must be able to express also complex interactions and more fine-grained details precisely, we need some additional roles. The issue role is rather versatile; it can be used to group other roles, or to denote a subject that needs additional attention from the application developer. The code role is used to express a piece of code, for instance, inside method or constructor bodies. The parameter, exception, and inheritance roles are needed to express context-sensitive method parameters, exception types, and inheritance relations. Note that because of the prototypical nature of FRED 1.2, there is some lability if a constraint property should actually be a role. For instance, for class roles there is the "inheritance" constraint property which basically has the same meaning – though less flexibility - as the inheritance role type. In the future, the set of role types and properties will be more consistent.

Pattern	The root of the pattern structure. Defines the pattern name.
Class	Represents Java classes; name, kind (class or interface), base class, and modifiers.
Constructor	Represents constructors; modifiers and default implementation.
Method	Represents methods; name, return type, modifiers, default implementation, and overriding.
Field	Represents fields; name, type, modifiers, and default initialization.
Issue	Used as a reminder or to group other roles.
Code	Represents a piece of code, for instance, inside a Java method or constructor.
Parameter	Represents method- or constructor parameters; name, type and position.
Exception	Represents the type of an exception thrown by a Java method or constructor.
Inheritance	Represents a context-sensitive inheritance relation between two Java classes.

Table 2. FRED role types.

In FRED, role types are declared as PDL classes. Each role type has a corresponding base class that defines a suitable set of properties and the semantics associated with that role type. The base classes are shown in Figure 25; when we are creating a new role with the Pattern Editor we are actually deriving a new PDL class from one of these base classes. Thus, when declaring a specialization pattern with PDL, the modeler creates the corresponding roles by subclassing particular base classes. Constraint and template properties are given simply by overriding some of the inherited functions (see the next subsection). As a PDL interpreter, the pattern engine of FRED checks the application against the constraint properties and generates tasks as explained in Chapter 4. Note that for historical reasons, PDL classes for various role types have the "Clause" suffix and the root class for the specialization pattern structure is named "Contract"; this will change in the future, when a new version of PDL and the pattern engine is released.

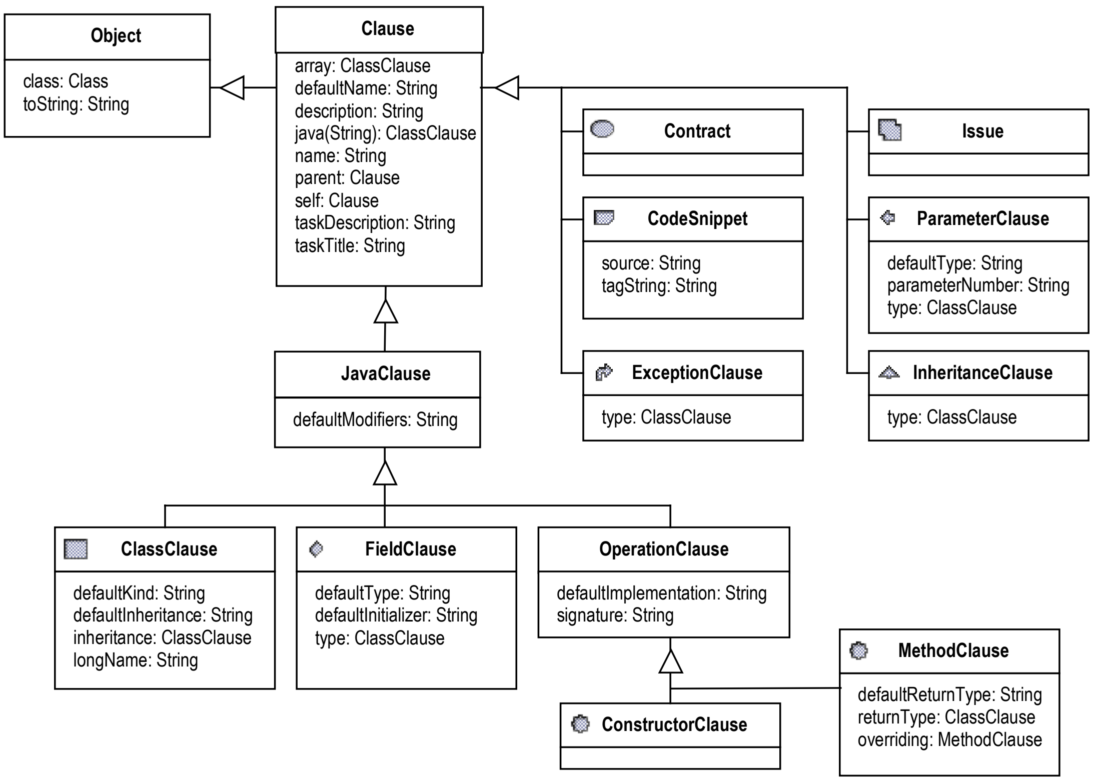


Figure 25. Base classes of PDL to represent different role types.

Technically speaking, the current Pattern Catalog tool actually shows the subclasses of the Contract class stored in the pattern repository when it enumerates the available specialization patterns. The Architecture View and the Task View tools, in turn, are dealing with the instantiated PDL objects. For instance, when a specialization pattern is employed from the pattern repository to the current project (in the Architecture View), the corresponding PDL classes are instantiated. Also, when it is said that a program element is cast in the role, or the application developer makes a contract between the element and the role, the program element is actually bound to a particular PDL object instantiated from the derived PDL class.

5.4.2 Functions

Each PDL class comes with a set of functions to express role-specific properties. The value of the property is defined as the return value of the corresponding function. For template properties the return value is a string expression while for constraint properties it is a reference to a particular role. Depending on the semantics of the current property, the return value is then used to check constraints or to generate adaptive documentation or source code. The base classes and functions of PDL used in FRED

1.2 are enumerated in Table 3. Here the functions are divided into three categories: utility functions ( ), constraint functions ( ), and template functions ( ). Utility functions, like the parent function, cannot be overridden in the derived subclasses. Function expressions are discussed more precisely in the next subsection.

Object

Base class of all PDL classes.

class: Class

Returns the PDL class of this object.

toString: String

Returns this object as string expression.

Clause

Base class of PDL classes declaring roles.

defaultName: String

Returns the default name of this PDL object. Default expression: "<#name>".

description: String

Returns description. This is the role-specific documentation.

taskDescription: String

Returns task description. This is the role-specific documentation for the generated task to cast the role. Default expression: "<#description>".

taskTitle: String

Returns task name for the generated task to cast the role. Default expression: "Provide <#defaultName>".

array(String): ClassClause

Returns a new ClassClause object that represents an array of Java classes represented by this PDL object. This function takes the Java class name as a parameter.

java(String): ClassClause

Returns a ClassClause object representing a particular Java class. This function takes the Java class name as a parameter.

name: String

If this PDL object is bound to a program element the function returns the element's name, otherwise it returns the PDL object's name (i.e., the role name).

parent: Clause

Returns the parent PDL object containing this object.

self: Clause

Returns this PDL object itself.

Contract

Declares the root of the specialization pattern structure. Declaring a new specialization pattern is started by deriving a new subclass from this base class.

JavaClause

Common base class for PDL classes declaring Java-specific pattern roles.

defaultModifiers: String

Returns default modifiers for the Java code element. For fields the default expression is "private". For classes, methods, and constructors the default expression is "public".

ClassClause

Declares a class role. New class roles are created by deriving new subclasses from this base class. It can be used to define name, kind (class or interface), modifiers, and base class of the represented Java class.

inheritance: ClassClause

Returns a ClassClause object representing the type that must be inherited by the Java class.

defaultInheritance: String

Returns the name of the default type to be inherited by the Java class.

defaultKind: String

Returns the kind of the Java class (interface or class). Default expression: "class".

longName: String

Returns the long name of the Java class. If no Java class has been bound to this PDL object, evaluates the defaultName function.

FieldClause

Declares a field role. New field roles are created by deriving new subclasses from this base class. It can be used to define type, modifiers, and default initialization of the Java field.

type: ClassClause

Returns a ClassClause object representing the type of the Java field.

defaultType: String

Returns the default type name of the Java field. Default expression: "int".

defaultInitializer: String

Returns the default initialization statement for the Java field.

OperationClause

Base class for PDL classes declaring Java operation roles, i.e., methods and constructors.

defaultImplementation: String

Returns the default code that is used when the code is generated for the Java operation.

signature*: String*

Returns the signature of the Java operation.

ConstructorClause

Declares a constructor role. New constructor roles are created by deriving new subclasses from this base class. It can be used to define modifiers and default implementation of the Java constructor. Note that context-sensitive parameters are defined with the ParameterClause classes, exceptions with the ExceptionClause classes, and constructor bodies with the CodeSnippet classes.

MethodClause

Declares a method role. New method roles are created by deriving new subclasses from this base class. It can be used to define return type, modifiers, default implementation, and override relation of the Java method. Note that context-sensitive parameters are defined with the ParameterClause classes, exceptions with the ExceptionClause classes, and constructor bodies with the CodeSnippet classes.

returnType: ClassClause

Returns a ClassClause object representing the return type of the Java method.

overriding: MethodClause

Returns a MethodClause object representing the Java method to be overridden by the Java method bound to this PDL object.

defaultReturnType: String

Returns the default return type name of the Java method. Default expression: "void".

ParameterClause

Declares a parameter role that is used to describe a single context-sensitive method or constructor parameter. New parameter roles are created by deriving new subclasses from this base class. It can be used to define type and place number of the parameter.

type: ClassClause

Returns a ClassClause object representing the type of the parameter.

defaultType: String

Returns the default type name of the parameter. Default expression: "int".

parameterNumber: String

Returns the place number of the parameter as string expression. Default expression: "1".

ExceptionClause

Declares an exception role that is used to describe a context-sensitive exception thrown by a Java method or constructor. New exception roles are created by deriving new subclasses from this base class. It can be used to define type of the Java exception.

type: ClassClause

Returns a ClassClause object representing the type of the Java exception.

InheritanceClause

Declares an inheritance role that is used to describe a context-sensitive inheritance relation between Java classes. New inheritance roles are created by deriving new subclasses from this base class. It can be used to define the type to be inherited.

type: ClassClause

Returns a ClassClause object representing the type to be inherited.

CodeSnippet

Declares a code role that is used to denote a specific piece of code inside a Java method or constructor. New code roles are created by deriving new subclasses from this base class.

source: String

Returns a string expression representing the piece of code. The string expression is given as normal Java code with some optional tags, which are replaced with application-specific names when the code is inserted into the application's source code.

tagString: String

Returns the identifier, which is used to locate the insertion point. This identifier can be written between the /*# */ tag to the suitable location in the application's source code, e.g., /*#myIdentifier*/. The code is inserted below this tag. If the identifier is not defined or if it is not found, the application developer must explicitly insert the code.

Issue

Declares an issue role that can be used to group other roles or to denote concepts which are hard to formalize or have no direct analogy with the current programming language; e.g., to remind that the application developer should update a particular configuration file. Issue roles are created by deriving new subclasses from this base class.

Table 3. PDL base classes and functions used in FRED 1.2.

Actually, in PDL, also role classes are considered as functions; using a role or dependency name in an expression returns a set of program elements cast to that role. We call such functions *role functions*. If there are no program elements cast to the role, the role function returns an empty set.

5.4.3 Expressions

Constraint- and template properties are set by implementing functions. The base class of the role may have default implementations but if they cannot be used, properties are set by overriding the corresponding functions. A function contains an expression that is evaluated when the function is called. The result of the evaluation is returned and used as the value of the property. For instance, by overriding the overriding function, a PDL class declaring a method role sets a constraint for the represented Java method. During the casting process, after a particular Java method is cast to the role, the overriding function is evaluated by the pattern engine to check that the actual Java method overrides another Java method that is cast in the referred role.

Writing expressions to override PDL functions is rather straightforward. An expression can contain role names (considered as functions returning a set of cast program elements), and other function calls separated by dots. The result of the expression must obey the function's return type. Each role establishes a namespace and names in the same namespace must be unique. Names are resolved by searching the current namespace and the namespaces of the referred roles and so on to the root of the specialization pattern. For example, the expression "BaseClass.operation.signature" evaluates the signature function in the operation role enclosed by the BaseClass role enclosed by the root of the pattern structure. Though there may be multiple Java methods cast in that operation role, the system associates the expression with the relevant one, determined by the casting graph.

However, as mentioned in Subsection 5.2.1 when discussing the Pattern Editor, there is some unnecessary complexity in the current version because the name of the referred role cannot be used directly. Instead, the dependency between two roles must be named (except the implicit containment relation) and the name of that dependency must be used. For instance, if a method role depends on the role BaseClass.operation, we must use the given dependency name when writing the expression; we may name this dependency "OVR" and define it to represent the dependency between the current role and BASE.operation where "BASE", in turn, is another dependency name declared in the enclosing role and referring to the BaseClass role. Now the expression for the overriding function turns from " BaseClass.operation" to "OVR". Figure 26 illustrates the situation.

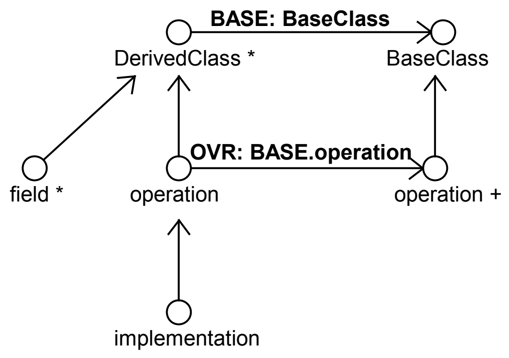


Figure 26. Pattern graph with named dependencies.

The use of additional dependency names will be removed in the future as it is possible to unambiguously determine the program element cast in the referred role without using explicitly given dependency names. Also, to avoid unnecessary complexity, we have not used dependency names in the notation for specialization patterns, though they are actually used when the specialization patterns are created with the Pattern Editor.

As a PDL interpreter, the pattern engine of FRED utilizes the given function expressions and enforces the application developer to obey the role-specific constraints. Checks are made simply by evaluating functions and comparing the result against the program element(s) cast to the role. Similarly, the template functions are used to generate adaptive documentation and source code by evaluating string expressions. Because the pattern engine is integrated to the source editor facilities, violated constraints can be instantly detected when the application developer modifies the source code of the application; violated constraints effect tasks to repair violations.

5.4.4 Tags

Rather than roles and program elements, some functions contain string expressions that may be used to generate documentation, task titles, default implementations, and so on. To make this text dynamic, string expressions may contain macro tags enumerated in Table 4.

<#*expression*>	This macro is expanded with the string that is obtained when the given expression is evaluated.
<%*expression*>	This macro is expanded with the string that is obtained when the given expression is evaluated. The first letter of the string is written upper case.

Table 4. Macro tags for template properties in FRED 1.2.

A macro tag is expanded with the string that is obtained when the expression inside the tag is evaluated. Thus, the result of the string evaluation may vary, depending on the previously made actions. This means that unlike with traditional documentation, the role-specific documentation can adapt the application-specific terms that were unknown when the documentation was given. Further, the generated source code may be customized with application-specific names. Note that if the expression inside a tag evaluates a PDL object, FRED implicitly converts this object to string expression by calling its toString method.

To illustrate the use of macro tags, consider that the pattern modeler wants to give a short description for a role representing a particular Java class. The modeler has derived a new subclass from the ClassClause base class to represent the desired Java class; the description function is overridden with the following string expression:

```
The base class of <#name> is <#BASE> that has operation <#BASE.myMethod.signature>
```

Here it is supposed that "BASE" is the name of the dependency that has been declared between the current class role and the role representing its base class. The role referred by "BASE" has a method role myMethod, which in turn has the signature function inherited from the MethodClause base class; thus, the macro tag "<#BASE.myMethod.signature>" returns the signature of the Java method cast to the myMethod role. In PDL, each role class inherits also the name function that returns the name of the role or the names of the program elements cast to that role; here the macro tag "<#name>" simply returns the name of the Java class cast to the current class role.

Another example; the string expression below could be used to override the defaultName function for a particular method role. Here it is assumed that MyField is a specific field role. For instance, if the name of the actual Java field (that is bound to the MyField role) is "button", the string expression evaluates "getButton". Thus, this template specifies a naming convention for a method associated with a particular field.

```
get<%MyField>
```

Augmented with hyper text links and macro tags, the FRED environment constructs a flexible documentation and source code that adapts to the current task-specific problem of the application developer. This can be seen as an advantage when the application developer is trying to learn or use a complex design solution.

5.5 Experiences

Besides minor experiments, like the JavaBeans programming [Sun 2001] and the Red framelet [Hakala et al. 2001d; Pree and Koskimies 1999], FRED has been applied to two major frameworks: a public domain graphical editor framework by Erich Gamma (JHotDraw) [Gamma 2001] and an industrial framework by Nokia [Bonnet 1999]. JHotDraw is a well-structured, relatively large (about 150 classes), yet sophisticated Java framework for implementing graphical editors. It enables the user to define various kinds of figures as well as handles to grab them, connectors to link them together, and tools to manipulate them. About ten patterns were needed to annotate the main parts of the specialization interface of JHotDraw [Viljamaa 2001]. The industrial framework, in turn, has about 300 classes and it is intended for creating GUI components for a family of network management systems. After analysing the specialization goals of the framework, a collection of thirteen patterns was defined to cover a major part of its specialization interface. These are shortly discussed in Chapter 6.

So far, the experiences gained from various case studies have been encouraging, showing that the FRED approach is sufficiently powerful to define the specialization interface of a real framework, and that FRED 1.2 – though still a prototype - scales up for industry-sized frameworks. Benefits of FRED are discussed in Subsection 5.5.1. The encountered problems are presented in Subsection 5.5.2. The possibility to integrate FRED with a third party development environment is shortly discussed in Subsection 5.5.3.

5.5.1 Benefits

Besides and due to the general benefits of architecture-oriented task-driven system enumerated in Section 3.2 (support for incremental, iterative and interactive specialization process; specialized instructions; architecture-sensitive source-code editing; open-ended adaptation process), FRED enables *extensive applicability*, *reduced learning efforts*, *intelligent code generation*, and *restricted specialization*.

Extensive applicability. Although the original motivation of FRED was to support the framework specialization process, it turned out that the architecture-oriented task-driven approach has much wider scope. Due to the extremely general character of the underlying concept, FRED can be used to support all kinds of architectural conventions and rules. For instance, we have used FRED to support JavaBeans programming, where the framework is thin if non-existent and the architecture relies on just a set of architectural and coding conventions.

Reduced learning efforts. The incremental specialization process with context-sensitive specialization instructions facilitates the understanding of the framework and architectural design by supporting learning-by-doing. During this process, the user may experiment with different aspects of the architecture; the environment illustrates the intended usage of design solutions by generating new tasks to fix possible errors and to provide missing participants. In a way, FRED resembles a human tutor, which rather than giving a lecture with abstract terms and beforehand, guides the process continuously using the terms related to the specific task at hand. Hence, FRED can be used also as a training aid in a company, complementing traditional framework documentation.

Intelligent code generation. In addition to novice users, also the expert users are served. While the specialization process can be actually carried out by persons who are not thoroughly familiar with the framework, an experienced user can utilize FRED to automatically produce a lot of essential and strictly regulated, but uninteresting code. Unlike with other wizards and development environments, the code is not generated as large and static lump. Because also the code generation proceeds piecemeal, the application developer is not overwhelmed by the generated code but can reason the rationale of it.

Restricted specialization. To ensure quality and robustnes of software products, it is often essential that programmers obey some company related rules and conventions. By providing the specialization interface as specialization patterns, FRED can be used to restrict and remind programmers during the development process. Though this may sound limiting, it assures that the essential aspects will be perceived by the application developers.

5.5.2 Problems

Though FRED has many advantages compared to traditional programming environments, it is still a prototype. The problems enumerated here are *categorization*, *reusability*, *dependencies across patterns*, *kinds of dependency*, *verification of methods*, *debugging*, *environment*, and *incomplete implementation*. Minor bugs that are not relevant to the scope of this thesis are not discussed here. Various problems to implement design patterns are discussed, for instance, by Bosch [1998] and Soukup [1995]. In any case, additional functions and role types would increase the expression power of PDL, making it possible to create even more sophisticated tool-support.

Categorization. In the current version, the created specialization patterns cannot be categorized in the pattern repository; they are shown as a simple list in the Pattern Catalog tool. This becomes a problem when the number of patterns increases. We need a system that guides the user to select the most suitable patterns. One possible solution is to categorize patterns in the same way than the Java classes are categorized under packages. Pattern providers could publish their patterns under certain "pattern packages". Another approach is to attach some kind of selection criterias to the specialization patterns enabling the use of intelligent pattern selection wizards.

Reusability. One of the main problems is that specialization patterns cannot be easily used to derive new patterns. The structure of a specialization pattern is statically declared as PDL class declarations, and to define a new slightly different pattern we must create a copy of the original pattern and modify the PDL code (with the Pattern Editor); thus, there is no "inheritance" mechanism between pattern definitions. This makes it difficult to reuse specialization patterns as building blocks of other specialization patterns. For instance, a framework could use a specialization pattern that models a general design solution but the documentation of that pattern cannot be changed to be more convenient for the application developer. In the current version, the framework expert must create a new slightly different framework-specific copy of the original specialization pattern, save it to the pattern repository, and finally instantiate it to the framework project. However, it would be more convenient to create a new specialization pattern and reuse the original as much as possible.

Dependencies across patterns. The current implementation doesn't support dependencies between the roles of the separate specialization pattern declarations. However, specialization patterns may be closely related requiring common program elements during the casting process. Now this can be only mentioned in the role-specific documentation and the pattern modeler must rely on the alertness of the application developer; after using one of the related specialization patterns, when instantiating the second one, the application developer must explicitly point out the already associated program element. Alternatively, separate specialization patterns could be implemented as one big pattern structure. But this approach doesn't sound very elegant, like trying to create one monolithic entity, and it only gets around the actual problem. Therefore, we are extending the model for increasing modularity within individual specialization patterns.

Kinds of dependency. If the role s depends on the role r, it means that the role s cannot be cast before the role r is cast. However, it would be useful if we could add more complicated semantics to the dependency relations. For instance, it could be useful to say that the role s cannot be cast if the role r is cast. Adding a comprehensive set of dependencies and making it possible to create dependencies over specialization patterns would substantially increase the expression power of PDL.

Verification of methods. FRED does not provide techniques to verify the semantics of a method, that is, to check the behaviour of a software system. FRED provides an extended architecture-specific typing system, augmented with capability to generate default implementations, but there is no way to programmatically check that the application developer really makes sensible actions inside the code. Defining the abstract semantics of a method (e.g., by pre- and post-conditions) and checking the implementation against such specifications is beyond the current research scope. However, FRED could be augmented with a richer set of statically verifiable constraints.

Debugging. Debugging specialization patterns is not supported. At least the checking of the syntax of PDL expressions (e.g., mistyped dependency- and role names) would be helpful. Now the pattern modeler must instantiate the pattern and use it step-by-step with the `Task View` to find out possible errors. As the experience about pattern modelling increases, more sophisticated tools may be implemented.

Environment. The current PDL implementation doesn't effectively support other programming languages than Java. However, specialization patterns to generate and handle various textual configuration files, XML, C++, etc. would be useful. Also, the tool set of FRED could be integrated into some 3rd Party IDE, making it more professional; we do not see any reason why this integration could not be done, provided that the IDE offers reasonable integration capabilities, especially to access its source editor. We argue that the concept of specialization patterns doesn't inherently exclude a system to support other languages and task-driven environments. The integration is shortly discussed in the next subsection.

Incomplete implementation. Because of the prototypical nature of FRED 1.2, there are some low-level problems deriving from the incomplete implementation of the current FRED version and PDL. For instance, the current version doesn't support Java inner classes; they cannot be bound to the pattern roles making it difficult to work with the code where inner classes are used. However, the concept of specialization patterns doesn't make such limitations.

5.5.3 Integrating FRED

Figure 27 illustrates how the FRED system could be integrated with a third party development environment. The main components of FRED are: the *pattern repository* to store specialization patterns, the *FRED project management* to handle FRED-specific information for projects, the *pattern engine* to update the task list and manage the casting process, and the *pattern tools* (Pattern Catalog, Pattern Editor, Architecture View, Task View) to provide the user interface and enable the use of specialization patterns.

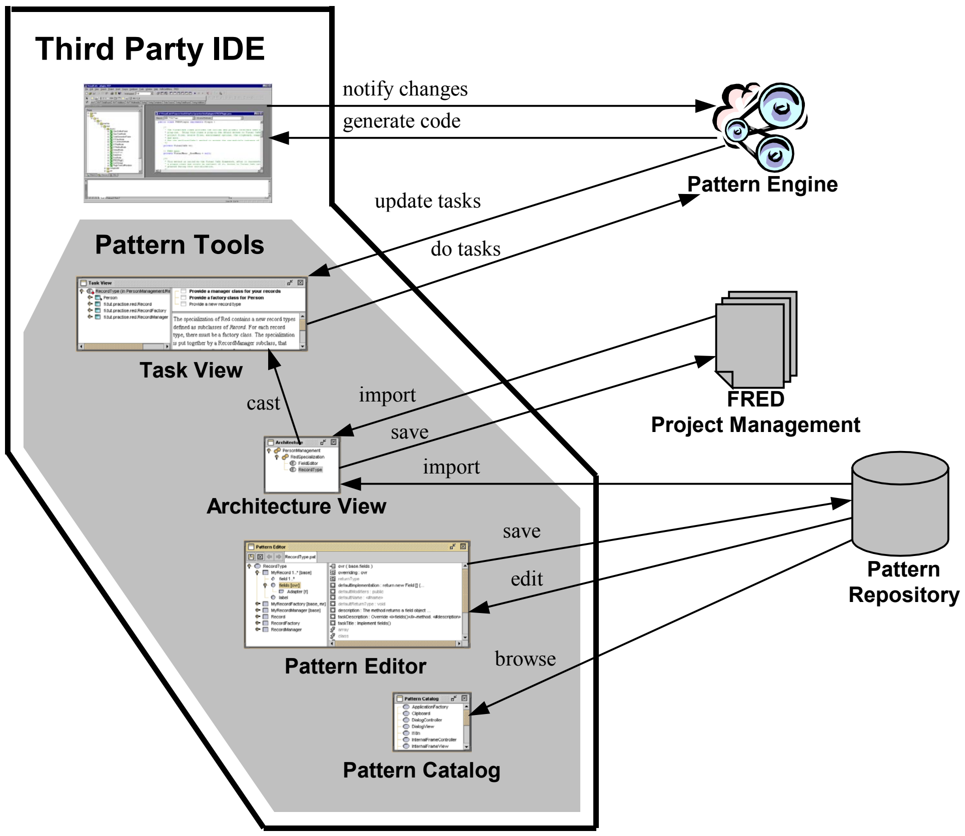


Figure 27. Integrating FRED with third party IDE.

To use specialization patterns effectively, the development environment must work in the interaction with the application developer. For instance, the FRED environment is incremental, notifying the pattern engine whenever the user manipulates the source code with the source editor. The pattern engine compares the current stage of the source code to the given pattern definition and updates the internal casting structure by changing states of the tasks and adding new ones. The Task View shows this casting structure as a task list related to the current problem; by doing the tasks in the task list, the application developer generates or modifies the source code, thus enforcing the pattern engine to evaluate again.

To cast roles to program elements and to continuously check the role-specific constraints the pattern engine must have an access to the parse information maintained by the development environment. This includes the details of classes, methods, fields, and constructors. In addition, it may be necessary to obtain the source code of a particular program element, like the body of a method or a constructor and do some FRED-specific parsing. Currently we have made some very tentative experiments with third party development environments and the results have been encouraging. In principle, we could perhaps turn a suitable Java development environment into architecture-oriented task-driven system. Clearly, this topic belongs to the future works of FRED and must be carefully examined.

6 Case Study

Nokia produces a family of NMS (Network Management System) and EM (Element Manager) applications that are used to manage the network or network element. They have a Java GUI 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 specialization patterns so that FRED can be used as a specialization wizard when creating user interfaces with the platform. The main work of the case study consists of becoming familiar with the framework, identifying its specialization goals and extension points, annotating the framework with a set of specialization patterns, writing documentation, like specialization instructions to be attached to the patterns, testing the installation, and reporting the work.

The process to study the framework and to find out specialization goals is discussed in Section 6.1. The construction phase and the obtained specialization patterns are presented in Section 6.2. The pattern organization phase as well as the use of the constructed specialization wizard is summarized in Section 6.3. The case framework is confidential, thus, technical details are omitted. Details of the constructed framework-specific specialization patterns are presented in a separate appendix [Hautamäki 2001].

6.1 Studying the Framework

The most difficult and time consuming part of the case study was to learn the framework and to analyze the use of the specialization interface. Though the case framework had good documentation, the source code was not available, nor any realistic use cases. However, the author learned the essentials of the framework in couple of weeks and was able to create a draft of the required specialization patterns. First the specialization goals were found by reading the documentation. Then, as discussed in Subsection 4.4.2, the specialization patterns were constructed by deriving example specializations to achieve the observed specialization goals and by analyzing the resulted participants.

The specialization goals of the case framework are shown in Figure 28. The constructed specialization patterns are enumerated under the corresponding specialization goals. In the figure, the specialization goals are grouped into three sections. Firstly, it is supposed that the application developer 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, to create other views, like dialogs and frames, the application developer must implement additional view- and controller classes as described by the used MVC paradigm. Thirdly, the application developer may utilize features like the internationalization service or the clipboard. The order, in which the application developer pursues these specialization goals is a recommendation only; solutions can be revisited during the life cycle of the specialization process, providing more specific functionality, or undoing previous specialization choices.

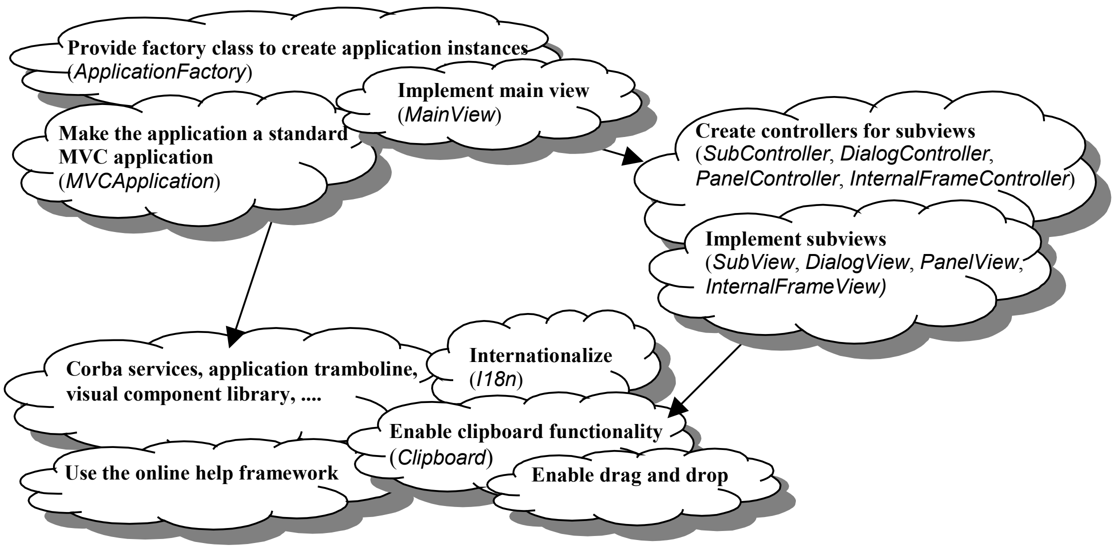

Provide factory class to create application instances
(*ApplicationFactory*)
Implement main view
(*MainView*)
Make the application a standard MVC application
(*MVCApplication*)
Create controllers for subviews
(*SubController*, *DialogController*, *PanelController*, *InternalFrameController*)
Implement subviews
(*SubView*, *DialogView*, *PanelView*, *InternalFrameView*)
Corba services, application tramboline, visual component library, ...
Internationalize
(*I18n*)
Enable clipboard functionality
(*Clipboard*)
Use the online help framework
Enable drag and drop

Figure 28. Specialization goals of the case framework.

Note that the map of the specialization goals is not necessary complete; instead, new problems may arise during the use of the case framework, forcing the application developer to invent new ways to specialize the framework. These empirical experiences can be used as a feed-back to refine the existing specialization patterns and to create new ones. Also, because of the limitations of the current FRED implementation, not all of the specialization goals can be efficiently supported by specialization patterns.

6.2 Constructing Specialization Patterns

As discussed in Subsection 4.4.2, constructing a specialization pattern requires that the participants of the desired outcome are analyzed and represented as roles and their interactions. In the case study, we have used the available documentation and some example specializations as a source of the analysis. As shown in Figure 28, the specialization patterns to pursue the specialization goals can be organized as patterns to make the application compatible with the framework system, patterns to implement controllers and views, and patterns to utilize services and other features of the case framework. During the pattern modelling process, iteration was occurred as each specialization pattern was tested by trying to re-produce the corresponding example specialization. Similarly, the attached role-specific documentation was continuosly revisited.

Based on the categorization of the specialization patterns, the constructed application patterns are presented in Subsection 6.2.1, the controller- and view patterns in Subsection 6.2.2, and the service patterns in Subsection 6.2.3. Due to the confidential nature of the case framework, detailed descriptions with role tables and pattern diagrams are given in a separate appendix [Hautamäki 2001].

6.2.1 Application Patterns

The specialization patterns in Table 5 are categorized as application patterns because they are used to associate a particular application with the framework system and to create the main controller and main window for it. The ApplicationFactory pattern describes how an application is instantiated with a specific factory class. The MVCApplication pattern specifies how to make the application a standard MVC application in terms of the case framework. The MainView pattern defines the application's main user interface and the interactions between the user interface and the main controller.

After using the patterns in Table 5, the application developer should have a working skeleton application compatible with the framework's system; it can be launched with the factory class and it has the main controller and the corresponding main view. To create dialogs, panels and frames the application developer uses the controller- and view specialization patterns discussed in the next subsection.

ApplicationFactory

This specialization pattern defines the constructional relationship between an application and the factory class used to create this application. One should use the MVCApplication pattern to make the application a standard MVC application, and the MainView pattern to build up the main window.

MVCApplication

Each application instance created with the ApplicationFactory pattern should be a standard MVC application. The MVCApplication pattern defines the required functionality for such an application. This includes the creation of the application's main controller. After transforming the application into MVC application, one should use the MainView pattern to provide the main view. Controller- and view patterns can be used to create other user interface elements like dialogs and frames.

MainView

This specialization pattern defines the application's main user interface and the interactions between the user interface and the main controller. After creating the main user interface, one should use controller- and view patterns to create other frames, dialogs, etc.

Table 5. Application patterns.

6.2.2 Controller- and View Patterns

The case framework is based on the MVC paradigm [Krasner and Pope 1988]. The aim of this architecture is to provide a clear separation between the graphical user interface and the rest of the application. As explained in the framework's documentation, the fundamental principles in the case framework are that every view object is managed by exactly one controller object and that every controller is managed by a parent controller. In addition, there are different kinds of controllers that handle different kinds of views. The view-controller interactions may concern the call-backs from the view to the controller, triggered by user actions, or the orders from the controller to the view.

To support the use of the MVC system, we have created specialization patterns to create suitable controller and view pairs. These specialization patterns are enumerated in Table 6. A new frame window can be created with the SubController and SubView patterns. Panels can be created and used with the PanelController and PanelView patterns. Dialogs, in turn, are produced with the DialogController and DialogView patterns. Finally, internal frames (windows opened in the desktop area of the parent window) can be created with the InternalFrameController and InternalFrameView patterns.

SubController	The SubController pattern is used to create controllers 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 a new frame window. After using this pattern, one should use the SubView pattern to create the frame's user interface.
SubView	The SubView pattern is used to create user interface for new frame windows.
PanelController	The PanelController pattern is used to create controllers for panel components. It describes a parent-child relation between a parent controller and a panel controller handling the panel. After using this pattern, one should use the PanelView pattern to create the panel's user interface.
PanelView	The PanelView pattern is used to create user interface for new panel components.
DialogController	The DialogController pattern is used to create controllers for dialogs. It describes a parent-child relation between a parent controller and a dialog controller handling the dialog. After using this pattern, one should use the DialogView pattern to create the dialog's user interface.
DialogView	The DialogView pattern is used to create user interface for new dialogs.
InternalFrameController	The InternalFrameController pattern is used to create controllers for internal frames, i.e., 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. After using this pattern, one should use the InternalFrameView pattern to create the internal frame's user interface.
InternalFrameView	The InternalFrameView pattern is used to create user interface for new internal frames.

Table 6. Controller- and view patterns.

6.2.3 Service Patterns

In addition to the MVC system, the case framework provides other features which are useful for building applications, like internationalization, clipboard, and various GUI components. Unlike the MVC system, some of these features are more like components or behavioral aspects of the application. For instance, the use of the internationalization service is typically splattered inside methods of the view- and controller classes.

As an example, we have implemented specialization patterns for some of the sub architectures of the case framework. The I18n pattern is used to internationalize user interface components. The Clipboard pattern, in turn, can be used to enable clipboard facilities for arbitrary user interface components. Other features of the case framework that could be supported by FRED are drag and drop functionality and the system to create online help, like tooltips, etc.

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 Swing JTextField and JTextArea classes and so also the components inherited from those base classes support text copy/paste internally.

DragAndDrop [1]

Drag and drop is a Java feature first introduced in Java 2 environment. 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, to enable the drag and drop feature, the application developer must implement number of classes dealing with the drag source and the drop target. Clearly, a specialization pattern could be used here to support the use of this complex mechanism. Though the specialization pattern for drag and drop is not currently implemented, it would be similar (and more complex) to the Clipboard specialization pattern.

HelpFramework [2]

The online help framework provides the GUI designer a way to add help to the GUI application and standardises help between GUIs of applications. It contains methods for GUI applications and visual help request mechanisms for the users of the applications. There should be no major obstacles to construct a specialization pattern to support the usage of the online help framework. The specialization pattern would be similar (and slightly more complex) to the I18n specialization pattern.

Table 7. Service patterns.

6.2.4 Other Features

The constructed specialization wizard for the case framework is not a slot machine that independently generates applications; some decisions cannot be automatized and some aspects cannot be effectively supported by FRED. As discussed in Subsection 5.5.2, one of the main problems of FRED 1.2 is the lack of behavioral verifications; this makes it difficult to support solutions that involve changes in method bodies. Thus, FRED is more suitable to express structural solutions, like how to create MVC application, than behavioral aspects.

[1] The DragAndDrop pattern is not currently implemented.

[2] The HelpFramework pattern is not currently implemented.

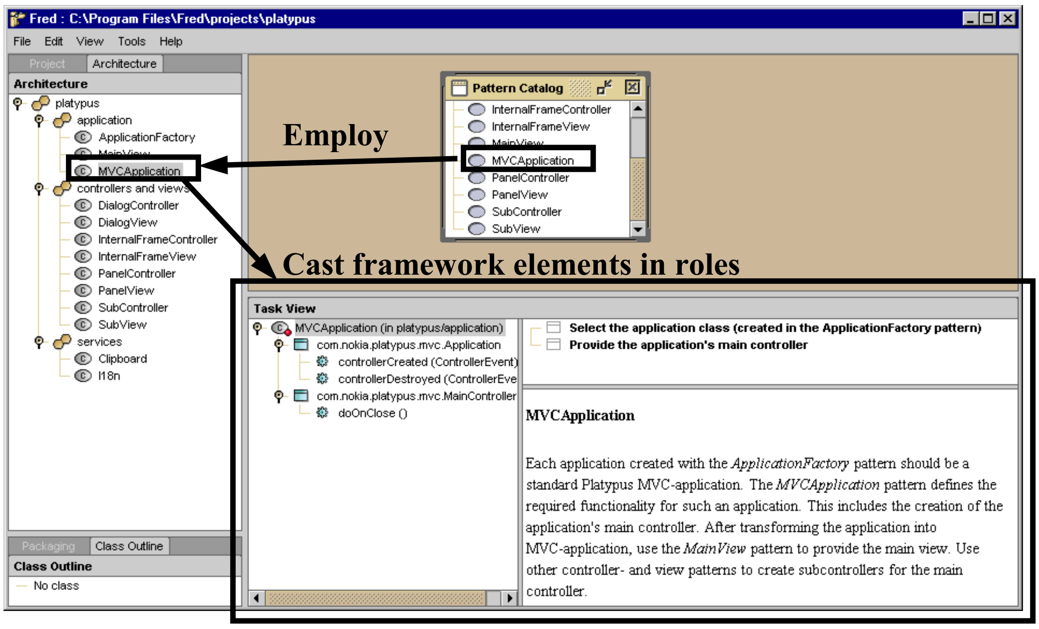
For instance, to ease writing multi-threaded code the case framework includes a collection of classes. Typically, using these classes means that the application developer creates a new thread with few lines of code and implements the application-specific logic. From the standpoint of FRED, the place where this thread is created nor the application-specific logic cannot be specified beforehand, unless the application domain is very limited. During the implementation, benefits of the specialization pattern would be minimal.

Similarly, using a specialization pattern to support the usage of individual components, like buttons and scroll bars, could be tedious. In addition, there are existing RAD (Rapid Application Development) tools that can be used to compose the user interface with these GUI components. However, if the usage of the component is part of a more complex architectural solution, FRED could be used at some point to teach or guide the application developer.

6.3 Constructing and Using Specialization Wizard

As described in Subsection 5.3.2, the pattern modeler must create a framework project and organize the framework-specific specialization patterns. The organization is done by employing the constructed specialization patterns from the pattern repository and placing them into the specialization architectures of the framework project. Because the specialization goals and the corresponding specialization patterns are already categorized into three sections (recall Figure 28), it is natural to create a specialization architecture for each of these groups. Also, to ensure that the application developer will use right base classes, override particular methods, and so on, some of the roles must be cast to these framework elements. The pattern organization phase is illustrated in Figure 29. After organizing the specialization patterns, the framework project is closed and ready to be utilized by the application developer.

The application developer uses the case framework by importing its specialization architectures and the contained specialization patterns. Figure 30 illustrates the situation. In the figure, the application developer has created a new application project. Because the user interface of the application must be implemented with the case framework, the application developer imports the specialization architecture of the framework project. To make the application compatible with the framework system, the application developer must use at least the ApplicationFactory, MVCApplication, and MainView patterns.

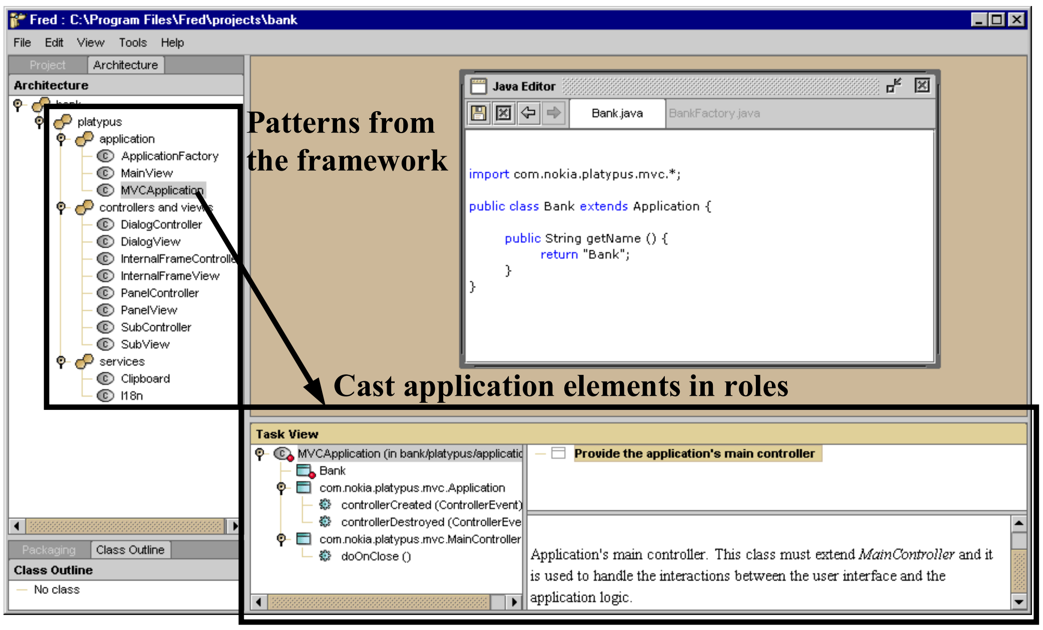


Select the application class (created in the ApplicationFactory pattern)
Provide the application's main controller

MVCApplication

Each application created with the *ApplicationFactory* pattern should be a standard Platypus MVC-application. The *MVCApplication* pattern defines the required functionality for such an application. This includes the creation of the application's main controller. After transforming the application into MVC-application, use the *MainView* pattern to provide the main view. Use other controller- and view patterns to create subcontrollers for the main controller.

Figure 29. Organizing the specialization patterns for the case framework.



```
import com.nokia.platypus.mvc.*;

public class Bank extends Application {

    public String getName () {
        return "Bank";
    }
}
```

Provide the application's main controller

Application's main controller. This class must extend *MainController* and it is used to handle the interactions between the user interface and the application logic.

Figure 30. The application developer is specializing the case framework.

One of the issues that this case study can be criticited is that the obtained specialization wizard has not been used by ordinary application developers creating real industrial applications; such use cases would most propably point out some weaknesses and missing specialization patterns. However, in the case of small example applications, the system worked satisfactorily. For instance, when making the application compatible with the MVC system, most of the required source code can be generated automatically. Also, the specialization wizard helped the application developer to remember some non-trivial method invocations and implementations. For a novice user, the specialization wizard was illustrative, making it easier to start making compatible applications. One of the most important uses of the constructed specialization wizard could be its suitability to teach programmers as they are simultaneously creating meaningful applications with the case framework.

7 Related Work and Conclusion

FRED (FRamework EDitor) is a prototype of a task-driven architecture-oriented programming environment that can be used as a specialization wizard to adapt architectural design. Specialization instructions are given to the tool as specialization patterns [Hakala 2001, Hakala et al. 2001a, 2001b]; these formal specifications make it possible to automatically compute how to implement architectural solutions during the software development process. This includes, for example, the intended usage of object-oriented frameworks [Fayad et al. 1999] and other architectural conventions, like design patterns [Gamma et al. 1995], and idioms [Coplien 1992; Venners 1998].

The main advantage of FRED is that it manages the adaptation and specialization process as a gradually progressing work, where each step is recorded and may have effects to the steps to come. This enables, for instance, documentation and source code generation that uses application-specific names familiar to the application developer. Further, the application developer can be instantly notified if he violates the architectural rules embodied by the given specialization patterns. The system can be used both to learn the described architectural solution and to rapidly generate much of the non-interesting source code, letting the application developer to concentrate more complex issues.

One of the constituting ideas behind FRED is the concept of patterns [Alexander et al. 1975, 1977; Alexander 1979] and, as noticed by number of authors in the object-oriented community, their applicability to describe software architectures (see, e.g., [Lea 1994]). For instance, *design patterns* [Gamma et al. 1995], *metapatterns* [Pree 1994, 1995], *architectural patterns* [Buschmann et al. 1996], *modeling patterns* [Coad 1992; Coad et al. 1995], *idioms* [Coplien 1992; Vlissides et al. 1996], *analysis patterns* [Fowler 1997], and *cookbooks* [Krasner and Pope 1988, Pree 1995] are informal description about a particular object-oriented implementation problem and its solution. In this connection, we should also mention *antipatterns* [Akroyd 1996; Brown et al. 1998] describing common bad design and how to avoid it.

The close relationship between patterns and frameworks was noted, for instance, by Johnson [1992] when he proposed using design patterns as an aid to document frameworks. Since then, patterns have been used to provide higher level descriptions of frameworks [Hueni et al. 1995; Schmidt 1997]. Many authors (e.g., [Riehle 2000] and [Florijn et al. 1997]) have also recognized the close relationship between the framework's flexibility points or "hot spots" [Pree 1995] and design patterns.

Typically, patterns and cookbooks are rather informal when explaining the problem and its solution. For instance, a cookbook is a documentation containing problem-specific instructions that describe in an informal way how to use a framework. However, instructions for specializing a framework cannot be expressed as a static and linear step-by-step task list, because a choice made during the specialization process may change the rest of the list completely. Also, it is common that the application developer makes mistakes and revisits the implementation continuosly; this kind of incremental and iterative software development is not directly supported by informal pattern descriptions. Thus, formal specifications are needed to enable incremental, iterative, and interactive tool-support with error recovery capabilities and to make the modeling of the solution more systematic.

One of the earliest works to specify architectural solutions were *contracts* [Helm et al. 1990; Holland 1993]. The proposed mechanism used pre- and post conditions to monitor the program execution. Thus, unlike with specialization patterns, contracts are used to specify reusable program fragments and restict their behavior at run-time, not to verify architectural rules at development time while typing in source code.

Since contracts, number of formalisms have been proposed to specify patterns and frameworks. For instance, *hooks* [Froelich et al. 1997] are semi-formal templates for describing the extension points of a framework. A hook represents the required sequence of actions as an algorithm that is intended to be carried out by the application developer. *UML-F* [Fontoura et al. 2000], in turn, is a UML [Rumbaugh et al. 1999] extension to specify the usage of a framework. These UML-F descriptions can then be executed with a special framework instantiation tool. However, the approach utilizes standard UML case tools, not a FRED-like incremental, iterative, and interactive programming environment.

One approach is *LePUS* [Eden et al. 1999; Eden and Grogono 2001] that is a symbolic logic language for the specification of recurring structural solution aspects of patterns. Further, Eden et al. [1999] have implemented a tool that locates, generates, and verifies pattern instances based on LePUS formulas. Yet another logic based formalism is presented by Alencar et al. [1996] that resembles the FRED system in that they recognize the possibility to implement a pattern by doing a sequence of tasks.

Besides formalisms, attempts have been made to create tool-support for adapting architectural design. For instance, *active cookbooks* [Pree et al. 1995] outlines a system to provide guidance for framework specialization. However, unlike with FRED, it is not explained how the system keeps consistent during the software development. The *SmartBooks* system [Ortigosa and Campo 1999; Ortigosa et al. 2000], in turn, seem to be more precise, letting the user to write instantiation rules describing the necessary tasks to specialize a framework. This reminds the use of specialization patterns in the FRED environment, but while SmartBooks assumes that the application developer precisely knows what kind of application he wants, our approach is more forgiving, guiding the application developer to gradually construct the application. Also, it seems that SmartBooks doesn't keep up the bindings between the rules and the produced source code.

Various other tools are, for example, *COGENT* [Budinsky et al. 1996], *fragment* tools [Florijn et al. 1997], *PSiGene* [Schütze et al. 1999], *Pattern-Lint* [Sefika et al. 1996], *SNIP* [Wild 1996], *POE* [Kim and Benner 1996], and *FACE* [Meijler et al. 1997]. Typically, these tools provide a way to specify and instantiate patterns. What makes them different from the FRED environment is that our approach is a pragmatic one, assuming that the application developer is not going to instantiate patterns as perfect monumental manifestations, but rather with small backwardable steps. Thus, our specialization patterns can be seen as small pattern languages [Alexander 1979] for writing software, while FRED provides a meaningful programming environment for that purpose.

References

[Aho et al. 1986] Aho A., Sethi R., Ullman J.: *Compilers – Principles, Techniques, and Tools*. Bell Telephone Laboratories, 1986. Reproduced by Addison-Wesley.

[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.

[Alencar et al. 1996] Alencar P., Cowan C., Lucena C.: A Formal Approach to Architectural Design Patterns. In: *Proceedings of the 3rd International Symposium of Formal Methods Europe*, 1996, 576-594.

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

[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.

[Appleton 1997] Appleton B.: Patterns and Software – Essential Concepts and Terminology. *Object Magazine Online*, Vol. 3, No. 5, 1997.
Internet: www.enteract.com/~bradapp/docs/patterns-intro.html, August 2001.

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

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

[Booch 1994] Booch G.: Designing an Application Framework. *Dr. Dobb's Journal*, Vol. 19, No. 2, February 1994.

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

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

[Brown 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.

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

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

[Coad et al. 1995] Coad P., North D., Mayfield M.: *Object Models – Strategies, Patterns, and Applications*. Prentice Hall, 1995.

[Coad and Yourdon 1991] Coad P., Yourdon E.: *Object-Oriented Analysis - Second Edition.* Prentice-Hall, 1991.

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

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

[Demeyer 1998] Demeyer S.: Analysis of Overridden Methods to Infer Hot Spots. In: *Proceedings of the ECOOP'98 Workshops, Demos, and Posters (Workshop Reader)*, Brussels, Belgium, July 1998, LNCS 1543, 1998, 66-67.

[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.

[Eden et al. 1999] Eden A., Hirshfeld Y., Lundqvist K.: LePUS – Symbolic Logic Modeling of Object Oriented Architectures - A Case Study. In: *Proceedings of the Second Nordic Workshop on Software Architecture (NOSA'99)*, University of Karlskrona/Ronneby, Ronneby, Sweden, 1999.

[Eden and Grogono 2001] Eden A., Grogono P.: A Theory of Object-Oriented Software Architecture. Submitted to: *International Journal of Software Engineering and Knowledge Engineering, Special Issue on Software Architecture*. Internet: www.cs.concordia.ca/~faculty/eden/articles/atoosa/atoosa.pdf, September 2001.

[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: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, 1997, 472-496.

[Fontoura et al. 2000] Fontoura M., Pree W., Rumpe B.: UML-F – A Modeling Language for Object-Oriented Frameworks. In: *Proceedings of the 14th European Con-*

ference on Object-Oriented Programming (ECOOP'00), Sophia Antipolis and Cannes, France, June 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: *Proceedings of the of 19th International Conference on Software Engineering (ICSE'97)*, Boston, Massachusetts. IEEE Press, 1997, 491-501.

[Gamma 2001] Gamma E.: *JHotDraw Framework, Download*. Internet: members.pingnet.ch/gamma/JHD-5.1.zip, August 2001.

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

[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: *Proceedings of the 17th International Conference on Software Engineering*, Seattle WA, April 1995

[Hakala 2001] Hakala M.: *The Pattern Engine*. Manuscript. September 2001.

[Hakala 2000] Hakala M.: Task-Based Tool Support for Framework Specialization. In: *Proceedings of OOPSLA'00 Workshop on Methods and Tools for Framework Development and Specialization*. Tampere University of Technology, Software Systems Laboratory, Report 21, October 2000.

[Hakala et al. 2001a] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Generating Application Development Environments for Java Frameworks. To appear in: *The 3rd International Conference on Generative and Component-Based Software Engineering (GCSE'01)*, Erfurt, September 2001.

[Hakala et al. 2001b] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Annotating Reusable Software Architectures with Programming Patterns. To appear in: *The Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, Amsterdam, August 2001.

[Hakala et al. 2001c] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Architecture-Oriented Programming Using FRED. In: *Proceedings of the of 23rd International Conference on Software Engineering (ICSE'01)*, Toronto, Canada. IEEE Computer Society, 2001, 823-824.

[Hakala et al. 2001d] 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. 1999a] Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J.: Managing Object-Oriented Frameworks with Specialization Templates. In: *Work-*

shop on Object Technology for Product-Line Architectures. European Software Institute, Spain, 1999, 87-98.

[Hakala et al. 1999b] Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J., Koskimies K., Paakki J.: Task-Driven Framework Specialization. In: Penjam J. (eds.): *Software Technology, Fenno-Ugric Symposium*, Institute of Cybernetics, Tallinn Technical University, August 1999, 65-74.

[Hakala et al. 1998] Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J.: Pattern-Oriented Framework Engineering Using FRED. In: *Object-Oriented Technology*. LNCS 1543, 1998, 105-109.

[Hakala et al. 1997] Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J.: *Design of a Java Framework Engineering Tool*. University of Tampere, Department of Computer Science, Report A-1997-12, November 1997. ISBN 951-44-4256-3.

[Hautamäki 2001] Hautamäki J.: *Appendix: Specialization Patterns for the Nokia Case Framework*. Internal document, September 2001.

[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.

[Helm et al. 1990] Helm R., Holland I., Gangopadhyay D.: Contracts – Specifying Behavioral Composition in Object-Oriented Systems. In: *Proceedings of The 5th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'90)*. SIGPLAN Notices, Vol. 25, No. 10, 1990, 169-180.

[Holland 1993] Holland I.: *The Design and Representation of Object-Oriented Components*. Ph.D. thesis, Northeastern University, 1993.

[Hueni et al. 1995] Hueni H., Johnson R., Engel R.: Framework for Network Protocol Software. In: *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*. Austin, TX, October 1995.

[Jacobson and Nowack 1999] Jacobson E., Nowack P.: Frameworks and Patterns – Architectural Abstractions. In: Fayad M., Schmidt D., Johnson R. (eds.): *Building Application Frameworks – Object-Oriented Foundations of Framework Design*. Wiley, 1999.

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

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

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

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

[Johnson 1992] Johnson R.: Documenting Frameworks Using Patterns. In: *Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92)*, Vancouver, Canada, October 1992, 63-76.

[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.

[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.

[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.

[Krämer and Prechelt, 1996] Krämer C., Prechelt L.: Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: *Proceedings of the Working Conference on Reverse Engineering*, Monterey, 1996. IEEE CS Press, 1996, 208-215.

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

[Lehman and Belady 1985] Lehman M., Belady L. (eds): *Program Evolution – Processes of Software Change*. Academic Press, 1985.

[Mattsson 1996] Mattsson M., *Object-Oriented Frameworks – A Survey of Methodological Issues*. Licentiate thesis, LU-CS-TR, 96-167, Department of Computer Science, Lund University, 1996.

[Meijler et al. 1997] Meijler T., Demeyer S., Engel R.: Making Design Patterns Explicit in FACE – A Framework Adaptive Composition Environment. In: *Proceedings of the 6th European Software Engineering Conference (ESEC'97)*. LNCS 1301, 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.

[Mikkonen 1998] Mikkonen T.: Formalizing Design Patterns. In: *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*. IEEE Press, 1998, 115-124.

[Ortigosa et al. 2000] Ortigosa A., Campo M., Salomon R.: Towards Agent-Oriented Assistance for Framework Instantiation. In: *Proceedings of the 15th Conference on*

Object-Oriented Programming, Systems, Languages, and Applications (OOP-SLA'00), Minneapolis, Minnesota, October 2000, 253-263.

[Ortigosa and Campo 1999] Ortigosa A., Campo M.: SmartBooks – A Step Beyond Active-Cookbooks to Aid in Framework Instantiation. In: *Technology of Object-Oriented Languages and Systems* 25. IEEE Press, June 1999, 131-140.

[Oxford 1989] Oxford: *The Oxford English Dictionary – Second Edition*. Oxford University Press, 1989.

[Pasetti 2001] Pasetti A.: *A Software Framework for Satellite Control Systems – Methodology and Development*. Ph.D. thesis, University of Constance, 2001.

[Pasetti and Pree 2000] Pasetti A., Pree W.: Two Novel Concepts for Systematic Product Line Development. In: Donohoe P. (eds.): *Software Product Lines: Experience and Research Directions (First Software Product Lines Conference, Denver, Colorado)*, Kluwer Academic Publishers, 2000.

[Poulin 1994] Poulin J.: Measuring Software Reusability. In: *Proceedings of the International Conference on Software Reuse*, Rio de Janeiro, November 1994.

[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 1994] Pree W.: Meta Patterns – A Means of Capturing the Essential of Reusable Object Oriented Design. In: *Proceedings of the 8th European Conference on Object-Oriented Programming*, Bologna, Italy, July 1994.

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

[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.

[Riehle 2000] Riehle R.: *Framework Design – A Role Modeling Approach*. Ph.D. thesis, ETH Zürich, Institute of Computer Systems, February 2000.

[Riehle and Zűllighoven 1996] Riehle D., Zűllighoven H.: Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems*, Vol. 2, No. 1, 1996, 3-13.

[Roberts and Johnson 1996] Roberts D., Johnson R.: Evolving Frameworks - A Pattern Language for Developing Object-Oriented Frameworks. In: *Proceedings of, the 3rd Conference on Pattern Languages and Programming (PLoP'96)*, Allerton Park, IL, September 1996.

[Rumbaugh et al. 1991] Rumbaugh J., Blaha M., Premerlani W., Frederick E., Lorensen W.: *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

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

[Schauer et al 1999] Schauer R., Robitaille S., Martel F., Keller R.: Hot Spot Recovery in Object-Oriented Software with Inheritance and Composition Template Methods. In: *Proceedings of the IEEE International Conference on Software Maintenance 1999 (ICSM '99)*, Keble College, Oxford, England. IEEE Computer Society Press, 1999, 220-229.

[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.

[Schütze et al. 1999] Schütze M., Riegel J., Zimmermann G.: PSiGene - A Pattern-Based Component Generator for Building Simulation. *Theory and Practice of Object Systems (TAPOS)*, Vol. 5, No. 2, 1999, 83-95.

[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 18th IEEE International Conference on Software Engineering*, Berlin 1996. IEEE Computer Society Press 1996, 387-396.

[Sethi 1989] Sethi R.: *Programming Languages – Concepts and Constructs*. Addison-Wesley, 1989.

[Shaw and Garlan 1996] Shaw M., Garlan D.: *Software Architecture - Perspectives on an Emerging Discipline*. Upper Saddle River, NJ, Prentice Hall, 1996.

[Soukup 1995] Soukup J.: Implementing Patterns. In: Coplien J., Schmidt D. (eds.): *Pattern Languages of Program Design*. Addison-Wesley, 1995, 395-412.

[Stroustrup 1991] Stroustrup B.: *The C++ Programming Language – Second Edition*. Addison-Wesley, 1991.

[Sun 2001] *JavaBeans*. Internet: http://java.sun.com/products/javabeans, August, 2001.

[Venners 1998] Venners B.: *The Event Generator Idiom*.
Internet: http://www.javaworld.com/ javaworld/jw-09-1998/jw-09-techniques.html.

[Viljamaa 2001] Viljamaa A.: *Pattern-Based Framework Annotation and Adaptation - A Systematic Approach*. Licentiate thesis, University of Helsinki, Department of Computer Science, 2001.

[Vlissides et al. 1996] Vlissides J., Coplien J., Norman L. (eds.): *Pattern Languages of Program Design 2*. Reading, MA, Addison-Wesley, 1996.

[W3C 2001] World Wide Web Consortium: *Extensible Markup Language (XML)*. Internet: http://www.w3.org/XML, August 2001.

[Wild 1996] Wild F.: Instantiating Code Patterns — Patterns Applied to Software Development. *Dr. Dobb's Journal*, Vol 21., No 6., June 1996, 72-76.