TASK-DRIVEN SPECIALIZATION SUPPORT FOR OBJECT-ORIENTED FRAMEWORKS

Markku Hakala[1], Juha Hautamäki[1], Kai Koskimies[1],
Jukka Paakki[2], Antti Viljamaa[2], Jukka Viljamaa[2]

[1]Software Systems Laboratory, Tampere University of Technology
P.O.Box 553, FIN – 33101 Tampere, Finland
E-mail: {markku.hakala, csjuha, kk}@cs.tut.fi

[2]Department of Computer Science, University of Helsinki
P.O.Box 26, FIN – 00014 University of Helsinki, Finland
E-mail: {antti.viljamaa, jukka.viljamaa, jukka.paakki }@cs.helsinki.fi

A framework is a collection of classes implementing the shared architecture of a family of applications. It is shown how the extension points ("hot spots") of a framework can be specified formally in such a way that the specification can be used to automatically generate a task-based wizard for guiding the framework specialization process. The extension points are specified as parameterized patterns, which define various constraints over the parameters. The tool (FRED) allows the application developer to bind actual system elements to the pattern parameters or generate default code as instructed in the pattern specification. The tool keeps track of the broken constraints and generates necessary programming tasks to remedy them. We argue that this kind of tool support could be the core of a programming environment for architecture-oriented programming, guaranteeing that the static requirements of the architecture are satisfied. In this sense the parameterized pattern concept represents an architecture-level (static) typing system, and the tool is a structure-oriented editor that both guides the user and checks that the application conforms to the given architecture. The tool has been implemented in Java for Java, and it has been evaluated against a real industrial framework. We will explain the underlying concepts of FRED and the main characteristics of the tool, demonstrate the approach with a simple example, and summarize our experiences with the approach so far.

1. INTRODUCTION

Background

Product line architecture is a system of rules and conventions for creating software products for a given domain ([JGJ97], [Bos00], [JRL00]). *Object-oriented frameworks* are an established way to implement product line architectures [FSJ99]. A framework implements the invariant part of an architecture and defines its specialization interface. A new product can be derived from the framework by providing the application-specific part written against the specialization interface.

The specialization interface of a framework, like any software interface, can be regarded as a contract between two systems. A system (framework) defines a contract that specifies the requirements the system expects from another system (application-specific part). Any system that fulfills the contract can play the role of the latter system.

Typically, the documentation provided together with a framework describes in varying formats the specialization interface of the framework. In addition, the features of the implementation language can be used to express some aspects of the specialization interface (e.g. interface classes in Java). Unfortunately, the former method relies on informal descriptions that cannot be exploited for automated specialization support and verification, and the latter method can capture only a fraction of the rules associated with specialization interfaces.

Design patterns [Gam95] have gained popularity as a means to describe architectural units of frameworks [Joh92]. Many authors have also recognized the close relationship between the flexibility points of frameworks ("hot spots" [Pre95]) and design patterns (e.g. [Rie00], [Hak99], and [FMW97]). It seems that the notion of a design pattern is a promising starting point for systematic approaches to the specification of framework architectures and specialization interfaces.

Our Contribution

This paper proposes a task-driven approach to framework specialization. Our model embodies an algorithm that maintains a dynamic "things-to-do"-list in co-operation with the framework developer. The idea is to provide interactive specialization instructions that adapt to the current specialization problem. In addition, code generation, and to some extent, verification, can be implemented on top of the model.

Motivated by the notions of contract and (design) pattern we define a new kind of contractual interface concept that we call a *programming pattern* (simply referred as a pattern). A pattern is a collection of *roles* for various language structures (classes, methods, attributes etc.), and a set of *constraints* on the roles. Each role can be bound to an actual instance of a language structure in a system (e.g. a class role is bound to a particular class).

Structural aspects of a design pattern in the sense of [Gam95] can be represented as a programming pattern, but our pattern does not take a stand on the purpose or scope of the pattern. A programming pattern is a mechanism for providing programming assistance that can be used to specify also idioms, coding conventions, and framework-specific architectural properties.

We have implemented a development environment called *FRED (FRamework EDitor for Java)* to demonstrate the potential of pattern-based framework engineering. FRED introduces semi-graphical editors for defining patterns and for carrying out the bindings. It supports the framework specialization process by guiding the application developer through a task list based on the pattern definitions. At the same time, it verifies that the patterns are bound to the context in the required manner. FRED is freely available at http://practise.cs.tut.fi/fred.

The FRED methodology and programming environment have been validated in a real-world case study carried out for one of our industrial partners.

The rest of this paper is organized as follows. In Chapter 2 we discuss ways to describe software architectures and possibilities to utilize architectural descriptions to generate programming tasks. The model adopted for the architectural descriptions is presented in detail in Chapter 3. In Chapter 4 we use a small case study to demonstrate the practical implications of the FRED environment. Finally, conclusions are drawn in Chapter 5 together with some comparisons with the related work on the field.

2. The Vision: Task-Based Framework Specialization

Traditionally, the planning of software architecture has been understood as a part of the software design phase, and architectures have been mainly described with standard modeling languages (such as UML) or with dedicated architecture description languages (ADL). To some degree, these abstract descriptions make it possible to assess the quality of the system at the architectural level, but they fall short in supporting the construction of the actual executable system based on the architecture.

Especially the construction of product families calls for systematic architecture-centric methodologies and tools that support the implementation of both the reusable core of the family and the products derived from it. In particular, we need an environment that guarantees that the application-specific code conforms to the underlying architecture.

Explicit architectural descriptions are needed to enable tool support for framework specialization. We propose an architecture-oriented programming environment that takes a definition of an architecture as a set of patterns and provides interactive, architecture-specific guidance for the development process.

Within the environment, the application of an architecture results in a corresponding set of pattern instances binding the produced source code elements to the roles defined in the patterns. Both patterns and pattern instances are considered as explicit programming entities that can be formally expressed and processed by a tool. The tool takes pattern definitions as input and gradually constructs instances for them by recording the bindings interactively with the developer.

The interaction is carried out by the means of programming tasks, which reflect the unbound roles of the pattern, as well as the broken constraints related to existing bindings. Typical programming tasks include creation of new classes and methods as well as refactoring of program elements to adhere to some semantic constraints. For instance, the creation of a subclass can be seen as a task. Defining an overriding method within that subclass might be another task, which may occur only after the creation of the enclosing class. Hence tasks are executed in ordered sequences, and a completed task may generate a sequence of new tasks.

When doing a task, the developer proceeds according to a well-defined plan, but also makes choices. Creating a new program element for a given task results in a new binding. There might be several alternative ways to do the task, each generating succeeding tasks. The tool creates these new tasks based on the information encoded in the pattern. At the same time the developer sees the effects of the binding step by step, making it easier for her to understand the architectural implications of the pattern.

We argue that this kind of interactive guidance is very beneficial in framework adaptation. 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 specialization. By providing tasks incrementally, the tool can gather information about the specialization and "specialize" the documentation as well using application-specific terms, reflecting the choices the developer has already made.

Similarly, code generation is dynamic in the sense that it adapts to the implementation context captured by previous bindings. The interactive nature of the environment makes it natural to show the automatically produced code immediately to the developer so that she can tailor it according to the instructions given by the tool.

Semantic constraints defined in programming patterns form the basis of code verification. The environment can re-evaluate constraint checks whenever the user manipulates the source code, thus making it possible for the task list to evolve concurrently with the development process to express constraint violations and ways to fix them. We use incremental compilation techniques to enable such interactive response.

3. THE THEORY: PROGRAMMING PATTERNS

As the basis of task-based framework specialization, and task-based programming in general, we propose the notion of a *programming pattern*, as a generative description that can be applied systematically under the guidance of a development tool to produce a number of similar structures.

A programming pattern is a generalization of recurring implementation. It defines a generic structure by the means of *roles*, where each role is an abstraction of a recurring fragment in a set of concrete structures. Each role is characterized by its type, which defines the kinds of program elements (e.g. classes, methods, variables, formal parameters to a method, statements) the role stands for. Furthermore, a role may stand for a single element, or a set of elements.

Applying the pattern is called *instantiation*, and the resulting structure is called a *pattern instance*. Given a pattern with roles, a pattern instance consists of *role instances*, each of which maps to a role in a pattern. Each role instance represents a *binding* between a concrete program element within a software system, and a role in the pattern.

As a programming pattern is composed of roles, placeholders for syntactical elements in a programming language, a programming pattern is essentially a template for producing and validating code. Thus, it should not be confused with design patterns [Gam95]. However, considering only the solution-part of design patterns, there is a subtle relationship. A design pattern describes a solution as a collaboration of roles. A design pattern manifests itself within a system if a set of objects exist that play these roles according to their responsibilities. Thus, in the context of design patterns, the notion of a role is essentially dynamic, in contrast to our definition where a role is a placeholder of static program elements. However, the documentation of a design pattern typically implicitly assumes a class-based system. Rather than describing the solution by the means of objects, a class-based implementation schema is outlined that implies the actual object collaboration. Thus, also design patterns have a static interpretation. These two views can be defined as static and dynamic perspectives on patterns. Programming patterns therefore fit in this picture as static, implementation-oriented variations of design patterns. For the rest of the discussion, we restrict the usage of word "pattern" to denote our notion of programming patterns.

The instantiation of a programming pattern equals to the process of binding suitable program elements to the roles of the pattern. The instantiation process is incremental and carried out in co-operation of a developer and a development tool. The tool provides the developer with the sequence of programming tasks that instruct the developer in instantiating the generic solution proposed by the pattern. *Production tasks* instruct the developer to instantiate a role. Such an instance will become part of the pattern instance. *Refactoring tasks* assist the user in modifying a program element bound to a role to adhere the constraints imposed by the role.

There might be several ways to do the task, each of which will generate succeeding tasks, based on the information encoded in the pattern. In addition to semantic constraints that can be used as the basis of code verification, this information could include documentation and code templates that are parameterized by the implementation context captured by previous bindings.

If the user follows the ever-changing task list, a point is reached where no more mandatory tasks exist. As a result, the abstract structure defined by the roles has been specialized in a user-defined context. This is recorded as a set of bindings between roles and program elements – the pattern instance.

We will apply patterns for describing specialization interfaces of frameworks. Specializations of a framework share a similar, although not the same structure. This is unavoidable, and even favorable as similarity promotes maintenance and comprehensibility. A specialization is also guaranteed to work best and be less sensitive to internal changes of a framework when implemented the way the framework developer intended. As programming patterns cope with recurring structures, they constitute a convenient way to define the specialization interface of a framework. To provide systematic task-based tool support for framework specialization, the framework developer should provide framework-specific patterns for the development tool. Although framework-independent, when instructed by these patterns the tool produces a programming environment for a particular framework. We call this kind of a tool a *metawizard*.

Syntax

It is possible to separate the syntactic and semantic aspects of programming patterns. In this chapter, we will focus on the syntactic side. We will first define programming patterns and pattern instances as graphs of certain properties, then pattern instantiation as a process of forming the pattern instance graph based on the pattern definition graph. We call the structure of the instance graph the *syntax* of a pattern instance, defined by the grammar of the definition graph. At the end we will provide an algorithm for pattern instantiation.

Pattern Definition Graph

Syntactically, a programming pattern can be presented as a directed acyclic graph, formalized by the following 3-tuple:

$$P = (R, D, c)$$

$$R \subseteq L$$

$$D \subseteq R \times R$$

$$c : R \rightarrow \{0, 1, 2, \ldots\} \times \{1, 2, 3, \ldots, \infty\},\ c(r) = (l, u),\ \ u \geq l$$

This is called the *definition graph*. The vertices R of the graph are called *roles*, and the directed edges D are called *dependencies*. In the definition of a role, we assume language L, which defines the set of possible names. Each role is identified by a unique name within the pattern. Furthermore, a dependency from r to s is an ordered pair (r, s), where role r is called the *depender*, and role s is called the *dependee*.

The third part of the definition, function c, is called the *cardinality function*. For each role, it returns an ordered pair (l, u), called the *cardinality constraint*, where l is called the lower bound and u is called the upper bound. This bounds the number of instances of the role, in relation to its dependees. Precise interpretation is provided later when discussing pattern instantiation.

Figure 1 shows a graphical representation of an example pattern definition graph. The nodes represent roles and the directed arcs represent dependencies. The cardinality constraint for a role is placed in parentheses after the label of the associated node.

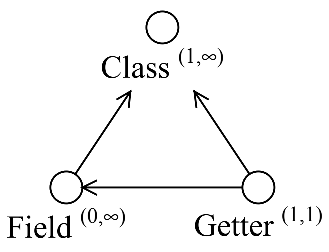


Figure 1. An example pattern definition graph.

Note that no semantic information is encoded in the notation, even though the names of the roles may intuitively suggest associated semantics (role named *Class* should probably be bound to a class, etc.). We can however assume semantic constraints stating e.g. that *Class* represents an implementation class, *Field* represents a private attribute within that class, and *Getter* represents a method for accessing the *Field* within a *Class*. Given this semantics, the pattern would represent a reusable structure for a class having a number of attributes and an accessor method for each attribute. Nevertheless, we will explain the patterns first in a purely syntactic way, and continue to the semantics thereafter.

Pattern Instance Graph

A pattern instance can be presented as a directed acyclic graph as well. Instance P' of pattern $P = (R, D, c)$ can be formalized by the following 3-tuple:

$$P' = (R', D', s)$$

$$R' \subseteq R \times \boldsymbol{Z}^{+}$$

$$D' \subseteq R' \times R',\ \forall\ ((r, x), (s, y)) \in D' : (r, s) \in D$$

$$s : R' \rightarrow \{\ \mathit{mandatory}, \mathit{optional}, \mathit{done}\ \}$$

In this *instance graph*, the vertices R' are called *role instances*. Each role instance is a manifestation of some role in the associated definition graph. Role instances are of form (r, x) where r is that role and x, as a positive integer, identifies the instance amongst all instances of role r. The directed edges D' between role instances are called *dependency instances*. Each dependency instance (a, b) manifests a dependency defined between the two roles that a and b are instances of.

Role instances correspond to production tasks within a tool environment. Function s, called the *state function*, maps each role instance to a *state*. As a task, role instance can be *done* or *undone*. Furthermore, an undone task may be considered as *mandatory* or *optional*. To understand the state function, and instance graph in general we must look at the instantiation process.

Pattern Instantiation

The process of creating a pattern instance based on a pattern is called *instantiation*. Instantiation of a pattern takes place in a development tool environment. It is an incremental process, which is carried out co-operatively by the tool and the developer. In the perspective of the tool, the instantiation process can be described by an algorithm that assumes a definition graph $P = (R, D, c)$ and an instance graph $P' = (R', D', s)$ and augments P' with new role instances, if possible. The newly created role instances are mandatory or optional production tasks, to be carried out by the developer. Whenever the user completes a task, the state of the role instance is changed to *done*, and the tool should re-evaluate the algorithm to determine whether it is possible to create new role instances based on existing role instances.

Figure 2 shows a graphical representation of a pattern instance, based on the pattern in Figure 1. The nodes represent role instances and the directed arcs represent dependency instances, respectively. The state of a role instance is written in superscript after the label of the associated node.

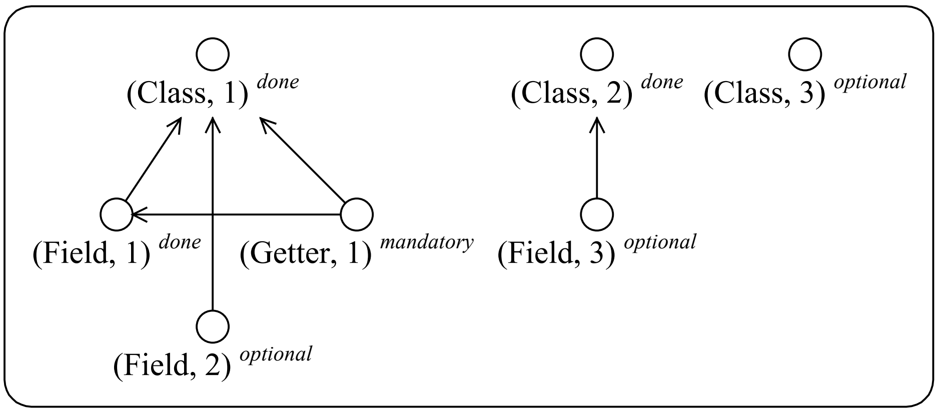


Figure 2. An example pattern instance graph.

Figure 2 portrays a partial pattern instance, i.e. a pattern instance that is in the middle of instantiation. Some tasks have been done, resulting in a graph of role instances. The interpretation of the graph can be based on the semantic outline sketched in Figure 1. The developer has created two classes and an attribute. (*Getter*, 1) represents a task to provide a getter-method for that attribute. As this is a mandatory task, the instance is not yet considered complete instantiation of a pattern. The developer has also a choice of continuing with optional tasks, some of which may lead to new tasks, even mandatory.

Before examining the instantiation algorithm, let us provide some auxiliary definitions.

If within a graph vertex *b* is *reachable* from vertex *a*, i.e. either $a=b$ or there is a path from *a* to *b*, we call *b* an *ancestor* of *a*, and *a* a *descendant* of *b*. Within the instance graph we define binary relation $\pi : R' \times R'$ such that $\pi(a, b)$ if and only if *b* is reachable from *a*. In addition, we define three functions. All of these are defined below.

$$\pi(a, b) \equiv a = b \vee (\exists\, c \in R' : (a, c) \in D' \wedge \pi(c, b))$$

$$I_d(r) = \{ (r, x) \in R' \mid s((r, x)) = done \},\ r \in R'$$

$$I_u(r) = \{ (r, x) \in R' \mid s((r, x)) \neq done \},\ r \in R'$$

$$dependees(r) = \{ s \mid (r, s) \in D \}$$

Given these definitions we can introduce a statement called *syntactic goal*:

$$\begin{aligned}
&\forall\, r \in R :\\
&\quad \{ s_1, \ldots, s_n \} = dependees(r),\\
&\quad \forall\, (d_1, \ldots, d_n) \in \{ (d_1, \ldots, d_n) \in I_d(s_1) \times \ldots \times I_d(s_n) \mid\\
&\qquad\qquad \forall\, t \in R : \forall\, i, j : \pi(d_i, (t, x)) \wedge \pi(d_j, (t, y)) \rightarrow x = y \} :\\
&\qquad Q = \{ q \in I_d(r) \mid \forall\, i : \exists\, (q, d_i) \in D' \},\\
&\qquad l \leq |Q| \leq u,\ (l, u) = c(r)
\end{aligned}$$

If this statement holds the instance graph is considered to be syntactically valid. The responsibility of a tool can therefore be restated as an effort to achieve the syntactic goal. Informally, for each role *r* with a cardinality constraint (l, u), the syntactic goal states the following:

> There should be $n \in (l, u)$ number of instances of *r* for each ordered list ($d_1,\ldots,d_n$) where d_i and d_j are instances of different dependees of *r* such that if both d_i and d_j have an ancestor that is an instance of role *t*, then this ancestor is unambiguous. There should be no other instances of *r*.

When the instance graph conforms to this statement, it is called *complete*. If it is possible to achieve the syntactic goal by adding more role instances to the instance graph, the instance as well as the instantiation is called *partial*. The tool should ensure that any time, the pattern instance is either complete or partial, but not otherwise malformed or in a form that would lead to a malformed pattern by the completion of an existing task. An algorithm is given in following (Algorithm 1) that assumes a pattern $P = (R, D, c)$ and an instance $P' = (R', D', s)$; the latter presents the current instantiation situation. The algorithm augments P' with missing role instances, i.e. production tasks. The process considers each role and decides whether it is necessary to create new instances of that role. This is determined by first constructing all possible combinations of the instances of the dependee roles. Then the algorithm checks if a correct amount of instances exists for each of these combinations. If not, a new role instance is created, with state set to optional or mandatory, depending on whether the lower bound denoted by the cardinality constraint has been exceeded.

When the user carries out a task, the state of the task is changed to *done*, and the algorithm is re-evaluated.

```
FOR EACH $r \in R$ DO
    $\{ s_1, \ldots, s_n \} \leftarrow dependees(r)$
    $(l, u) \leftarrow c(r)$
    FOR EACH $(d_1, \ldots, d_n) \in I_d(s_1) \times \ldots \times I_d(s_n)$
        WHERE $\forall t \in R : \forall i, j : \pi(d_i, (t, x)) \wedge \pi(d_j, (t, y)) \rightarrow x = y$ DO
        $Q \leftarrow \{ q \in I_d(r) \mid \forall i : \exists (q, d_i) \in D' \}$
        IF $|Q| < u \wedge \neg\exists p \in I_u(r) : \forall i : \exists (p, d_i) \in D'$ THEN
            $q \leftarrow (r, x)$ WHERE $(r, x) \notin R' \wedge ((r, x - 1) \in R' \vee x = 1)$
            $R' \leftarrow R' \cup \{ q \}$
            $D' \leftarrow D' \cup \{ (q, d_1), \ldots, (q, d_n) \}$
            $s(q) \leftarrow$ ( IF $|Q| \leq l$ THEN $mandatory$ ELSE $optional$ )
        END
    END
END
```

Algorithm 1. Augmenting the pattern instance with new tasks.

The tool should also allow the developer to backtrack her choices. A role instance can be removed from the instance graph if it has no descendants but itself. After the removal of a role instance the states of other undone tasks should be updated (as a result of the removal, an optional task may become mandatory), and the algorithm given should be re-evaluated.

As an example of the instantiation process, consider Figure 3, which presents three initial steps of an instantiation of a pattern presented in Figure 1. The first step represents the initial situation; the tool has created an instance of role *Class*, as it has no dependencies. In the second step, the developer has completed that task (indicated by a gray arrow). Given the semantics we sketched when describing the example pattern, the completion of this task could have resulted by the implementation of a new class, pointed out by the developer as the program element for the task (*Class*, 1). This has resulted in the re-evaluation of Algorithm 1. Creation of a task has resulted in new tasks, based on the dependencies defined in the pattern (a dependency (*Field*, *Class*) is defined is the pattern definition graph). Therefore, for each (done) instance of *Class*, the tool creates an instance of *Field*. Actually, as the upper bound of the cardinality constraint on *Field* is infinity, a new (optional) task is created from *Field* each time the developer succeeds to complete the earlier one. This eventually leads to a number of *Field* instances for each *Class* instance.

In addition to (*Field*, 1), the first instance of *Field*, a new instance of role *Class* has been created, as the upper bound denoted by the cardinality constraint is not yet reached. Both of the new tasks are optional, as the lower bounds defined in the pattern have been exceeded.

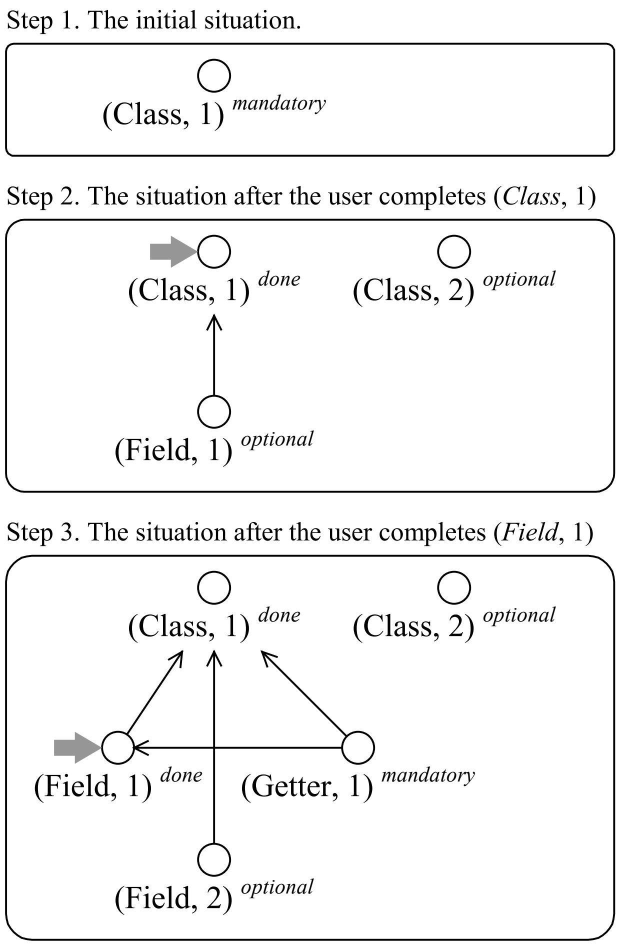


Figure 3. Three initial steps of an example pattern instantation.

Step 3 shows yet another situation, resulted from the previous situation on the completion of task (*Field*, 1). This has resulted in a new instance of *Field*, as well as an instance of *Getter*. The latter is possible as there now exists an instance of both *Class* and *Field*, and given the dependencies (*Getter*, *Class*) and (*Getter*, *Field*) in the definition graph, a *Getter* should be instantiated for each pair of such instances. Note that an instance of *Getter* is not created for pair (*Class*, 2) for two reasons; firstly, the state of that instance is not done, and secondly, that would result in a role instance with ambiguous ancestor of role *Class*, which is prohibited by the syntactic goal, defined earlier as the basis for the instantiation process.

As the fourth step of the instantiation, you may consider Figure 2, resulted from Step 3 on the completion of (*Class*, 2).

Semantics

The definition graph provides a grammar for describing patterns. In the previous subchapter, we provided an algorithm that gradually generates an instance graph based on that grammar. The resulting pattern instance can be regarded as a sentence of a language defined by the pattern. So far, we have discussed only the syntax of these languages. To provide useful tool support, the grammar described by the definition graph has to be decorated with tool-specific semantics.

A semantic system can be constructed by attaching semantic properties to vertices of the definition and instance graphs. Roles are supplied with expressions, role instances with values. An expression attached to role *r* will be evaluated in the context of a role instance, reading and writing values to and from the instance and its ancestors. The approach relates to attribute grammars, where rules attached to the grammatical productions (pattern) are used in calculating values on the concrete syntax tree (pattern instance). Thus, an expression attached to role *r* can be used to express constraints on the instances of *r*. In a development tool environment, role instances are bound to program elements like classes and methods, and the expressions are used to provide code generation and verification, as well as context-sensitive documentation for program development.

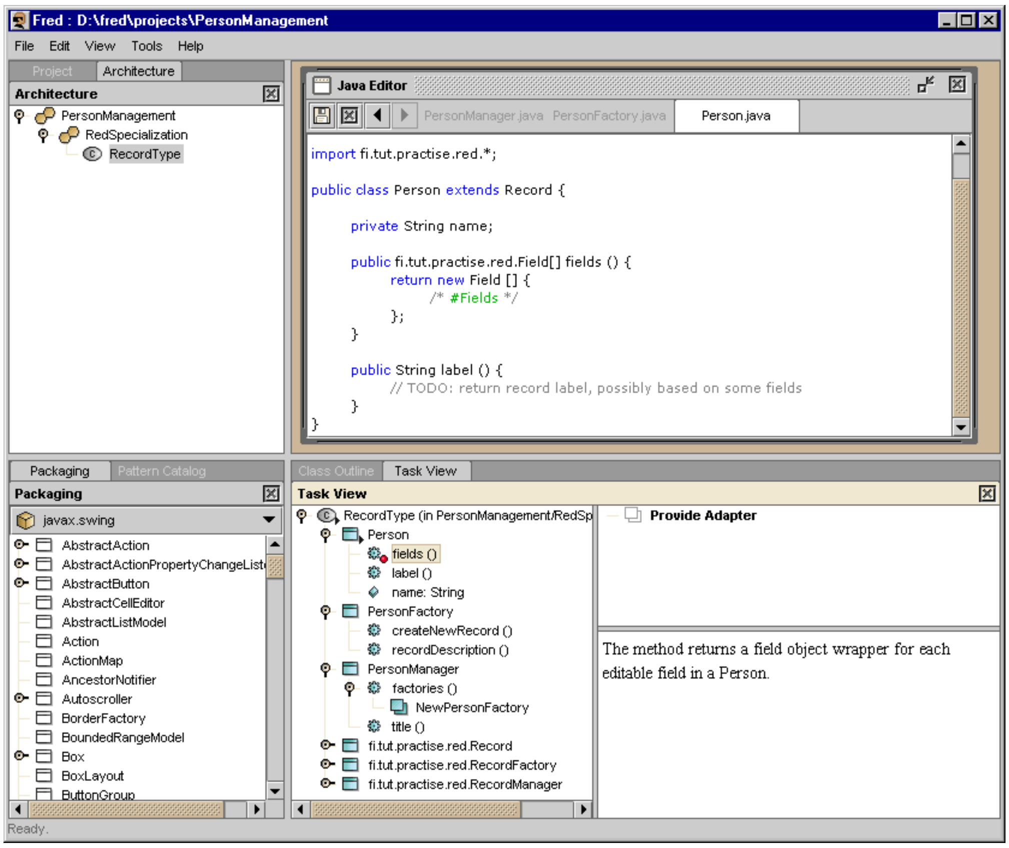
We assume a suitable expression language for defining these expressions. The language should be compatible to the previously mentioned language for naming the roles, so that expressions can refer to other roles by their names. References can be made only to the ancestors of the role. If an expression for role *r* refers to ancestor role *s*, when evaluated in the context of an instance *q*, that reference evaluates to an instance, which is of role *s* and an ancestor of *q*. Although there can be multiple instances of *s*, given the syntactic goal, the tool warrants the unambiguity of the reference.

4. THE TOOL: FRED ENVIRONMENT

FRED (FRamework EDitor for Java) is a prototype tool implementing the model discussed above. It provides a task-driven programming environment for the framework specialization process by guiding the application developer through a task list based on pattern definitions and keeping track of the progress of the tasks, verifying that the patterns are bound to the context in the required manner.

FRED is implemented in Java and intended for providing task-driven assistance for architecture-oriented Java programming. Our original motivation was to support specialization of Java frameworks, but it has later turned out that the approach can be used to guide programming according to various kinds of other architectural or coding conventions as well. As an example, we have modeled parts of the JavaBeans architecture as patterns, obtaining thus an environment for JavaBeans programming.

The user interface of FRED is shown in Figure 4. It contains number of views to manage Java projects and programming patterns. In the figure, the application developer has opened the Architecture view, which shows the project in terms of instantiated patterns and subsystems. The Task View tool shows the task list for a selected pattern instance.



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

public class Person extends Record {

    private String name;

    public fi.tut.practise.red.Field[] fields () {
        return new Field [] {
            /* #Fields */
        };
    }

    public String label () {
        // TODO: return record label, possibly based on some fields
    }
}
```

The method returns a field object wrapper for each editable field in a Person.

Figure 4. User interface of FRED.

The task list is automatically updated according to the stage of the source code. Typically, doing a task opens the source code editor to write the required code, or the code can be automatically generated, if possible. Violated constraints and unbound roles will generate new tasks during the pattern usage, until the whole pattern has been bound to its context.

FRED-specific Semantics for Java Programs

FRED uses patterns to support the development of Java programs, especially in the context of framework specialization. The semantic system is implemented as a class-based object-oriented language, classes presenting roles, role instances being instances thereof, and expressions implemented as class members, roughly divided to *constraints* and *templates*. Currently, both patterns and pattern instances are expressed in a tool-specific storage format, manipulated only by graphical or semi-graphical editors within the tool environment.

Patterns within FRED are used for expressing reusable structures within Java programs. Roles are considered as abstract representations of recurring program elements. Being implemented as classes, each role extends a base class that specifies the kind of program element the role represents: e.g. a class, a method, a field (Java-term for attribute) or a formal parameter to a method. The base class is called the *type* of the role. It defines what kinds of expressions can be attached to the role. A role instance represents a production task for providing a program element denoted by the type. The task is considered done if the user, using the tool, assigns a suitable program element to the instance.

A refactoring task is generated if the program element assigned to a role instance does not conform to the constraints of the role. In such case, the assigned element needs to be modified or the role instance has to be associated with more suitable element. Table 1 lists some typical constraints for role *r* with an ancestor role *s*. Each expression given is evaluated in the context of some instance *q* of role *r*. Therefore, each occurrence of *r* in the descriptions stand for "the program element bound to *q*", and each occurrence of *s* stands for "the program element bound to an instance of *s* that is an ancestor of *q*".

Constraint	Description
contained in *s*	*r* should be contained in *s* (This denotes containment in the syntactic sense. E.g. a class contains methods and attributes whereas method contains formal parameters and statements.)
inherit *s*	*r* should inherit *s* (Classes, interfaces and primitive types are all considered classes in FRED. Inheritance is a generalization over the extends- and implements-relationships of Java.)
override *s*	*r* should override *s*
return *s*	*r* should declare *s* as its return type

Table 1. Some semantic constraints.

Templates are used for generating context-sensitive code and documentation. A template is essentially a string with embedded sub-expressions. Like all expressions, a template is evaluated in the context of a role instance. During the evaluation, all embedded sub-expressions are evaluated and expanded to strings. Table 2 lists some of the most important types of templates in FRED.

Template	Description
taskTitle	Returns an informal string representation of the production task (all role instances are considered production tasks). This is shown to the user when the task is not yet done. It should shortly summarize what the user should do in order to complete the task.
description	Returns a description of the role instance that should document the purpose and characteristics of the role instance.
defaultImplementation	Returns a piece of code that constitutes a default implementation of the role.
defaultName	The default name of a code element. As with implementation-template, this is used for code generation.

Table 2. Some templates.

As an example, consider Figure 5. It defines a minimal pattern and a partial instance thereof. The pattern defines an inheritance relationship between two class roles, and presents a couple of templates to aid in the instantiation of the pattern. The graph has been annotated accordingly with sufficient semantic information. The instance graph in turn shows a snapshot of a moment of instantiation, where the super class has been bound to a Java class named Vehicle (indicated in the figure by a value named *element* attached to a role instance). This has resulted in a production task for providing a subclass. The *taskTitle*-template for that task evaluates literally to "Provide a subclass of Vehicle". Provided the tool requests a proper name for the class (say, Bicycle) from the user, the default implementation is

```
"public class Bicycle extends Vehicle { }",
```

resulted from the *defaultImplementation*-template. Note that before expanding the evaluated sub-expressions within a template, appropriate string conversion must be defined by the tool.

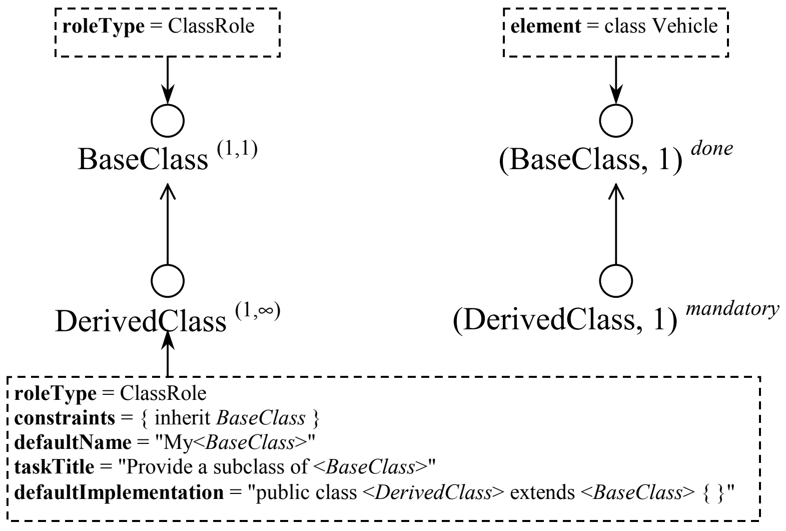

roleType = ClassRole
element = class Vehicle
BaseClass (1,1)
(BaseClass, 1) done
DerivedClass (1,∞)
(DerivedClass, 1) mandatory
roleType = ClassRole
constraints = { inherit *BaseClass* }
defaultName = "My<*BaseClass*>"
taskTitle = "Provide a subclass of <*BaseClass*>"
defaultImplementation = "public class <*DerivedClass*> extends <*BaseClass*> { }"

Figure 5. A minimal example pattern with semantics and a partial instantiation.

A Small Case Study

As an example of task-driven framework specialization in FRED environment, we will go through the specialization of a framelet called Red. Framelets are small frameworks consisting of a handful of classes and used as reusable building blocks for creating components. The Red framelet is an evolved version of a framelet discussed by Pree and Koskimies [Prk98]. Red provides user interface facilities to maintain a list of Record-objects and edit their fields. A specialization of Red typically defines a new Record subclass with some application domain–specific fields. Once the user has defined this new record type and derived some other classes, the framelet provides facilities to automatically generate the user interface to create and modify records. Typical user interface windows provided by Red are shown in Figure 6.

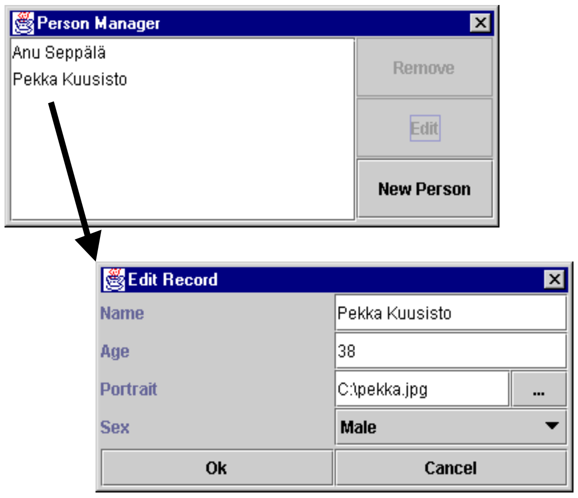


Figure 6. Typical views provided by Red framelet

To illustrate FRED methodology and tool we demonstrate the specialization support for Red. From the application developer's standpoint one of the specialization problems is how to create new record types in a way the framelet is able to provide user interface functionality for them. The specialization problem is expressed as a programming pattern called *RecordType*, encapsulating this particular hot spot of the framework. The definition graph of this pattern is shown in Figure 7.

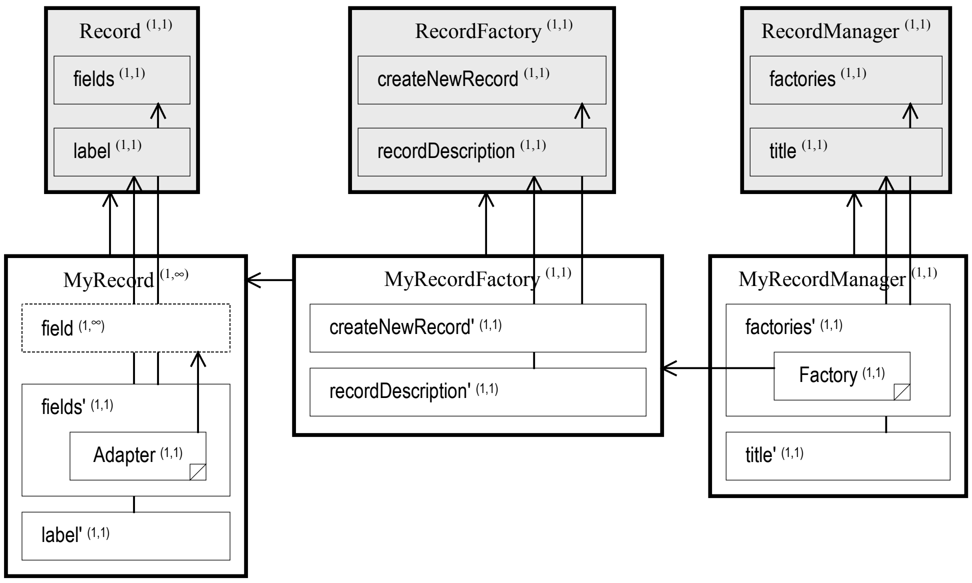
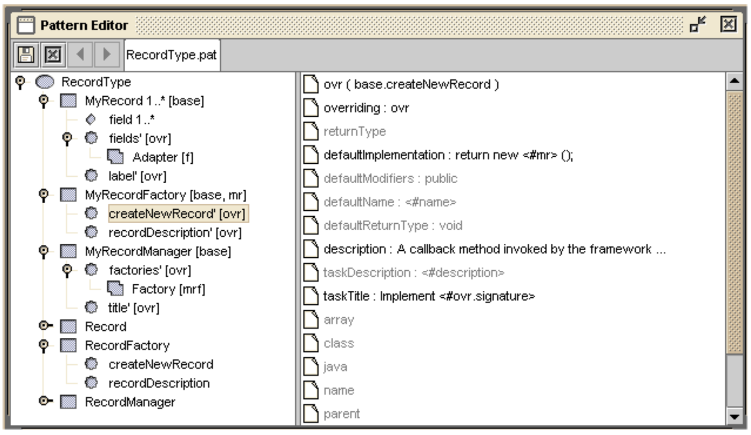


Figure 7. The syntactical definition graph of the RecordType pattern.

To make the graph more compact, we have adopted a slightly developed notation in this figure; the nodes are represented as boxes and the dependencies are represented either by arcs or visual composition. That is, a role enclosed within another role depends on the enclosing role. Moreover, we make a distinction between roles that map to the framework elements and roles that describe the structure of the specialization. The gray boxes denote roles for the framework elements. These roles are associated with semantic constraints that bound the roles to program elements of the framework. This means that there will be exactly one instance of the role, and that role is bound to a certain program element in each instantiation. Instead of a pattern parameter, such constraint will denote a pattern-specific constant. In addition, within this example, we assume that given these *bound roles*, the name of the required program element matches the name of the role.

The semantic constraints associated with the roles of *RecordType* pattern are numerous and thus not shown in the figure. However, the type of the role is indicated by the border decoration; classes have thick, methods thin and attributes dashed border. Roles with a bent corner denote code snippets, i.e. pieces of code that can be written within a method. FRED supports such roles by generating anchor-tags within method implementations when necessary. In FRED, the generated behavior is however always considered as default, and the developer may choose to write her own implementation.

In FRED, the patterns are created with an integrated Pattern Editor tool. It is a semi-graphical programming tool for specifying patterns. Figure 8 shows part of the *RecordType* pattern within the tool window. Within FRED implementation, the roles are organized in a hierarchy and the dependencies are named. The hierarchy combined with cross-references constitutes the definition graph of the pattern. The role hierarchy of a pattern is shown in the left side of the window. After each role, the cardinality constraint, if other than (1, 1), is shown, as well as the names of the associated dependencies. The right side of the window lists the set of constraints, templates and dependency definitions for a role selected in the left side.



```
RecordType
    MyRecord 1..* [base]
        field 1..*
        fields' [ovr]
            Adapter [f]
        label' [ovr]
    MyRecordFactory [base, mr]
        createNewRecord' [ovr]
        recordDescription' [ovr]
    MyRecordManager [base]
        factories' [ovr]
            Factory [mrf]
        title' [ovr]
    Record
    RecordFactory
        createNewRecord
        recordDescription
    RecordManager
```

```
ovr ( base.createNewRecord )
overriding : ovr
returnType
defaultImplementation : return new <#mr> ();
defaultModifiers : public
defaultName : <#name>
defaultReturnType : void
description : A callback method invoked by the framework ...
taskDescription : <#description>
taskTitle : Implement <#ovr.signature>
array
class
java
name
parent
```

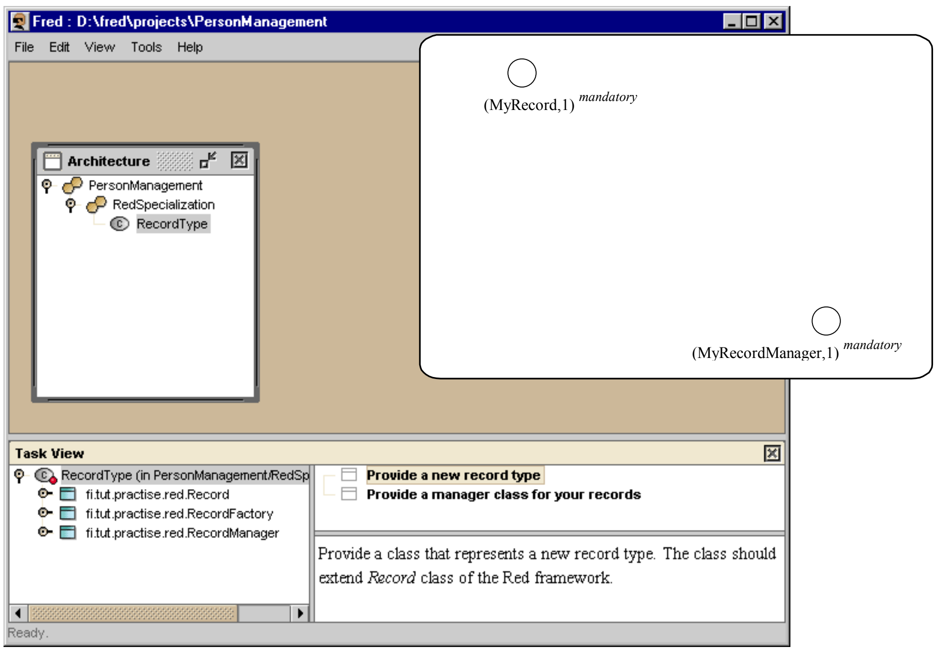
Figure 8. The Pattern Editor tool is used to create patterns in FRED.

Let us now assume that the application developer is creating the person manager application. The aspired user interface of this specialization of Red was shown in Figure 6. The developer chooses to create the application by specializing the Red framelet. Using the *RecordType* pattern specification, FRED acts as the specialization wizard for the Red framework.

FRED begins by instantiating each role that has no dependencies. Figure 9 presents this initial condition where an instance of both *MyRecord* and *MyRecordManager* role has been created. The figure shows both the user interface of FRED in this specialization situation, and the instance graph using the notation introduced in Chapter 3. Note that the visual appearance of the user interface differs slightly from the previous screenshot just because the tool environment can be freely customized.

Considering our syntactic theory on programming patterns, the newly created instances are presented by ordered pairs (*MyRecord*, 1) and (*MyRecordManager*, 1), where the first part denotes the role and the second part identifies the instance amongst all the instances of the role. Therefore, (*MyRecord*, 1) can also be read as "the first instance of *MyRecord*".

The state of both (*MyRecord*, 1) and (*MyRecordManager*, 1) is set to *mandatory*, as the lower bounds of the associated cardinality constraints are 1. Task (*MyRecord*, 1) tells the developer to provide a subclass of *Record* (a framework class that is the only instance of a bound role with the same name). To continue, the application developer may choose either task. However, as tasks are mandatory, the developer need eventually to complete them both.


Fred : D:\fred\projects\PersonManagement
File Edit View Tools Help
Architecture
PersonManagement
RedSpecialization
RecordType
(MyRecord,1) mandatory
(MyRecordManager,1) mandatory
Task View
RecordType (in PersonManagement/RedSp
fi.tut.practise.red.Record
fi.tut.practise.red.RecordFactory
fi.tut.practise.red.RecordManager
Provide a new record type
Provide a manager class for your records
Ready.

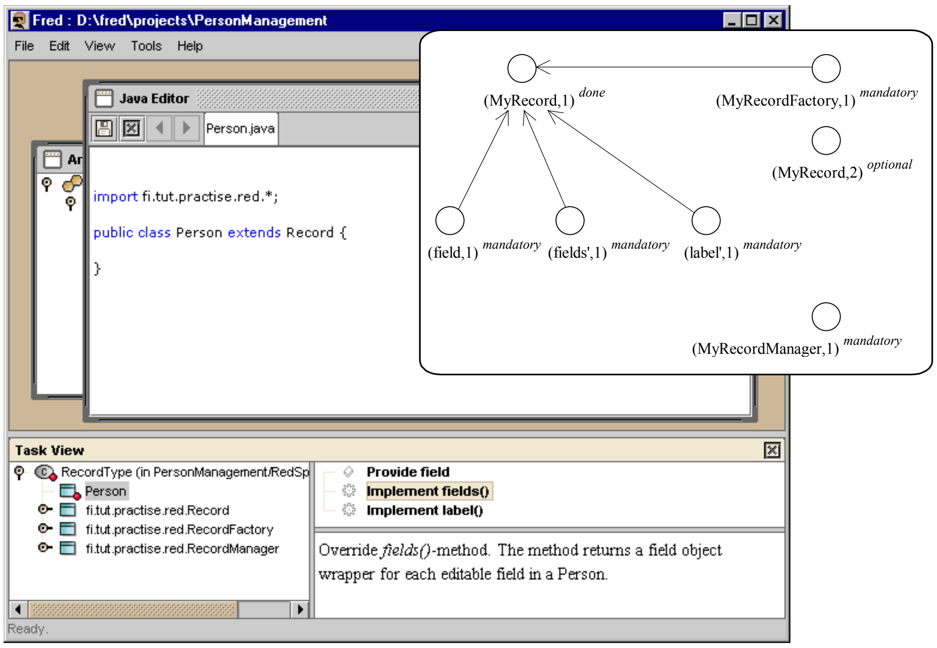
Provide a class that represents a new record type. The class should extend *Record* class of the Red framework.

Figure 9. The developer has just started to specialize the Red framelet.

We assume the developer continues by providing a class for the *MyRecord* role. FRED provides user interface functionality to point out an existing class to be bound to the role, as well as to generate a new class based on the templates and constraints associated with the role.

Because the developer is specializing Red to store information on personnel, he chooses to create a new *Record* subclass named *Person*. The developer types in the desired name of the class for the tool, which generates the default implementation for the class. FRED changes the state of the (*MyRecord*, 1) task from *mandatory* to *done*. The binding between the class and the role is recorded in the role instance so that the violations of the semantic constraints can be reported by the means of refactoring tasks whenever changes are detected in the associated source code. As FRED environment provides an integrated syntax-aware source code editor common to most contemporary development environments, any changes to the source code can be immediately evaluated in a background execution thread within the environment. If suitable heuristics were provided, it would be possible to consider every subclass of *Record* to be bound to the role *MyRecord*. Then, the task would be considered done automatically whenever the developer constructs a suitable class. If such heuristics are unavailable (as with the current FRED release), the user must point out the *Person* class explicitly to adhere the task. It is important to note that such heuristics could never entirely replace the explicit user interaction.

After completion of (*MyRecord*, 1) task, FRED re-evaluates the pattern instance against the definition. A new instance of the *MyRecord* role is created because the upper bound of the associated cardinality constraint is infinite. However, the state of the new (*MyRecord*, 2) instance is *optional*. In addition, a *mandatory* task (*field*, 1) is created to denote a member variable that should play the *Field* role. The (*field*, 1) task is mandatory, as the lower bound of the associated cardinality constraint is 1, requiring there to be at least one instance of *Field* for each instance of *MyRecord*. The (*fields'*,1), (*label'*,1) and (*MyRecordFactory*, 1) instances are created similarly. These tasks request the developer to override the methods declared in the *Record*-class. Figure 10 presents the instance graph at this point. Java editor displays the source code for *Person*, which is yet very minimal, as only one task has been carried out. The implementation of the class is guided by the sequence of programming tasks to follow.


Fred : D:\fred\projects\PersonManagement
File Edit View Tools Help
Java Editor
Person.java
(MyRecord,1) done
(MyRecordFactory,1) mandatory
(MyRecord,2) optional
(field,1) mandatory
(fields',1) mandatory
(label',1) mandatory
(MyRecordManager,1) mandatory
Task View
RecordType (in PersonManagement/RedSp
Person
fi.tut.practise.red.Record
fi.tut.practise.red.RecordFactory
fi.tut.practise.red.RecordManager
Provide field
Implement fields()
Implement label()
Ready.

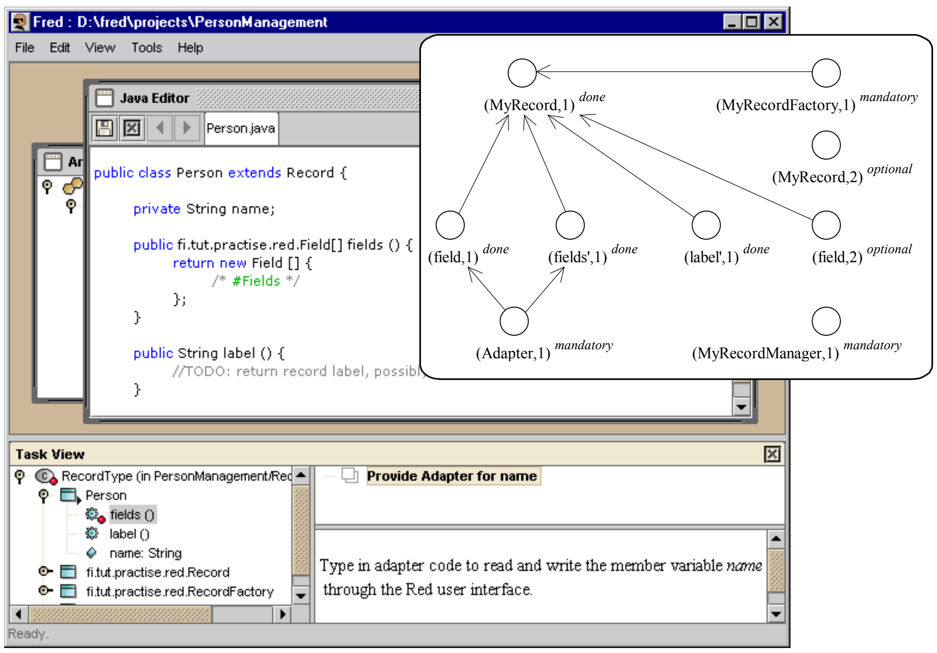
```
import fi.tut.practise.red.*;

public class Person extends Record {

}
```

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

Figure 10. Tasks for the new record type.

To complete the created *Person* class, new tasks (*field*, 1), (*fields'*, 1), and (*label'*, 1) are provided and the application developer may continue in any order she wishes. Task (*field*, 1) represents an instance variable that will become editable in the user interface provided by the framelet. The developer creates a variable called *name* to the *Person* class, and associates this variable as the required program element. For (*fields'*, 1) and (*label'*,1) the developer requests the tool to generate the default implementations.

Figure 11 presents the situation where the developer has done all these tasks and the tool has re-evaluated the pattern instance once again. Two new tasks have been created. The role instance (*field*, 2) offers the possibility to create another member variable. The role instance (*Adapter*, 1) instructs the developer to type in the required adapter code in the overridden *fields* method. In Red specialization, this adapter code provides access to read and write the member variable through the Red user interface. Our pattern definition states that such adapter code must exist for each member variable declared to play the role *field*. Based on the pattern definition it is possible to generate such adapter code, and inject it to the method implementation.



```
public class Person extends Record {

        private String name;

        public fi.tut.practise.red.Field[] fields () {
                return new Field [] {
                        /* #Fields */
                };
        }

        public String label () {
                //TODO: return record label, possibl
        }
```

Provide Adapter for name

Type in adapter code to read and write the member variable *name* through the Red user interface.

Figure 11. The fields method needs an adapter for the created name field.

To carry out the specialization the developer needs to complete all the rest of the mandatory tasks shown, and the required tasks resulting from the completion of these tasks. However, this process need not be a linear process. A mechanism is provided to undo selected tasks, providing the means to backtrack the instantiation process and reconsider the decisions made. Therefore, the pattern instance may evolve during the lifetime of the software development. In FRED, the source code is modified under the supervision of the tool, thus earlier decisions are refined automatically based on the modified source code. Whenever the code no longer complies with constraints of the pattern, the associated refactoring tasks are generated, reminding the user of the architectural rules and conventions.

Related Work and Conclusion

Framework Documentation, Adaptation, and Organization

To tackle the complexities related to framework development and adaptation we need means to document, specify, and organize them. The key question in framework documentation is how to produce adequate information dealing with a specific specialization problem and how to present this information to the application developer. A number of solutions have been suggested, including *framework cookbooks* [KrP88, Pre95], *smartbooks* [OrC99], and *patterns* [Joh92].

As shown in this paper, an application framework's usage cannot be adequately expressed as a static and linear step-by-step task list, because a choice made during the specialization process may change the rest of the list completely. That is why cookbooks [KrP88, Pre95], although a step to the right direction, are not enough. Our model can be seen as an extension of the notion of framework cookbooks.

Another advanced version of cookbooks is the SmartBooks method [OrC99]. It extends traditional framework documentation with instantiation rules describing the necessary tasks to be executed in order to specialize the framework. Using these rules, a tool can be used to generate a sequence of tasks that guide the application developer through the framework specialization process [OCS00]. This reminds our model, but whereas they provide a rule-based, feature-driven, and functionality-oriented system, our approach is pattern-based, architecture-driven and more implementation-oriented.

Fontoura, Pree, and Rumpe present a UML extension *UML-F* to explicitly describe framework variation points [FPR00]. They use a UML *tagged value* (a name-value-pair that can be attached to a modeling element to extend its properties) to identify and document the hot spots. Each of the variation point types has its own tag. In addition, there are tags for differentiating between static and dynamic variation points (i.e. whether or not the variable information is available at compile time) as well as for identifying application-specific classes as opposed to classes belonging to the framework.

Fontoura et al. identify the most common kinds of variation points in frameworks to be *extensible interfaces*, *variable methods*, and *extensible classes*. An extensible interface variation point denotes that a new application specific subclass must be provided for the interface. This can be directly implemented by inheritance in any object-oriented language. The latter two mean changes made to existing methods and classes, respectively. They cannot be directly mapped to constructs of standard object-oriented languages. That is why *implementation transformations* are needed to formalize, for example, design patterns to transform variable methods and extensible classes into extensible interface variation points. Implementation transformations utilize tagged values to denote pattern roles.

Framework adaptation is considered to be a very straightforward process in [FPR00]. UML-F descriptions are viewed as a structured cookbook, which can be executed with a wizard-like framework instantiation tool. This vision resembles closely that of ours. We see the framework specialization problem to be more complex than what is implied in [FPR00], however. The proposed implementation technique is based on adapting standard UML case tools. This does not directly support interactive task-driven framework specialization.

To manage the complexity of large frameworks they should be organized into smaller and more manageable units. *Framelets* provide a way to do just that [PrK99]. A framelet is a small framework with a clearly defined simple interface used for structuring new software architectures and especially for reorganizing legacy code. *Implementation cases* document framelets by giving examples of how applications should extend them [PaP00]. We have gained good experiences with annotating framelets with FRED patterns to make it easy to adapt and combine them in systems (see, e.g. Red example in chapter 4). We think that it is possible to represent implementation cases with programming patterns to enable tool support in the FRED environment.

Pattern Tools and Formalisms

The specification of an architectural unit of a software system as a pattern with roles bound to actual program elements is not a new idea. One of the earliest works in this direction is Holland’s thesis [Hol93] where he proposed the notion of a contract. Like UML's collaborations, and unlike our patterns, Holland’s contracts aimed to describe run-time collaboration. After the introduction of design patterns [Gam95], various formalizations have been given to design patterns resembling our pattern concept (for example, [FMW97], [MDE97], [Mik98], [Rie00]), often in the context of specifying the hot spots of frameworks. Our contribution is a pragmatic, static interpretation of the pattern concept and the infrastructure built to support it in realistic software development. Our view of patterns as a generalized interface mechanism is also somewhat different from the usual picture of a design pattern as a "mini-architecture".

In [EHL99] Eden, Hirshfeld, and Lundqvist present LePUS, a symbolic logic language for the specification of recurring motifs (patterns) in object-oriented architectures. They have implemented a PROLOG based prototype tool and show how the tool can utilize LePUS formulas to locate pattern instances, to verify source code structures' compliance with patterns, and even to apply patterns to generate new code. Furthermore, they claim that LePUS can give basis for defining refinement relations between patterns.

We recognize the need for a rigor formal basis for pattern tools, especially for code validation. Our model, however, is more analogous with programming languages and attribute grammars than with logic formalisms. In addition, we emphasize adaptive documentation and automatic code generation instead of code validation.

In [ACL96] Alencar, Cowan, and Lucena propose another logic-based formalization of patterns to describe *Abstract Data Views* (a generalization of the MVC concept). Their model resembles ours in that they identify the possibility to have (sub)tasks as a way to define functions needed to implement a pattern. They also define parameterized *product texts* corresponding to our code snippets.

Experiences

We have applied FRED to the development of a framework-specific programming environment for an industrial Java framework intended for creating GUI components for a family of network management systems. The framework comprises of about 300 classes. After analyzing the specialization problems of the framework, a collection of 13 patterns was defined to cover the specialization interface of the framework. This work required about 5 man-months for a person not initially familiar with the framework. The experience showed that the FRED patterns are sufficiently powerful to define the specialization interface of a real framework, and that FRED scales up for industry-sized frameworks. Several other benefits were noted in our approach, some of which were not even originally foreseen. In particular, the pattern-directed specialization tool facilitates the understanding of a complex framework architecture, by offering the user a view to single architectural "aspects" (i.e. patterns) at a time, with focused explanations. Hence the tool can be used also as a training aid in a company. The specialization process can be actually carried out by persons who are not thoroughly familiar with the framework. On the other hand, for an experienced user the tool produces automatically a lot of essential and strictly regulated, but uninteresting code. However, certain problems were also recognized in our current approach.

First, it seems to be difficult to capture a pattern specification for a hot spot in one go. Especially for a person who has not been actually using the framework it is hard to know what aspects are actually intended to be specialized. New ways of specializing a framework are found even in the application development process. Hence the tool should make it possible to easily modify the patterns even during the specialization process. Currently this is not possible. A possible solution is to make pattern instances more dynamic, modifiable entities.

Second, sometimes patterns depend on each other. Our current model of patterns does not include dependencies between patterns. In principle, this problem can always be solved by fusing the patterns that depend on each other into a single pattern, but this leads to large, unstructured patterns that are difficult to understand. A possible solution is to make patterns hierarchical and use patterns of patterns.

Third, currently FRED does not provide techniques to define the semantics of a method, that is, to define what the effect of a method should be (except for default implementations of method bodies). Hence there is no way to check that the user has given a method body in a way intended by the framework designer. Defining the abstract semantics of a method (for example by pre- and post-conditions) and checking the implementation against such specifications is, however, beyond our current research scope.

Discussion

We have presented a new approach to architecture-oriented programming and shown how a tool can support it. Our approach is founded on object-oriented frameworks and patterns that specify the design decisions made when developing the framework. The systems are specialized from the framework by following the tasks generated from the patterns. The tool supports architecture-oriented programming by guiding the application developer through the specialization process and by checking that the pattern constraints are not violated.

The idea of applying frameworks for the implementation of system families is not new. For instance, [BCS00] explores the relation of frameworks and system families and presents techniques for defining an interface between a framework and the applications specialized from it. The main advances in our approach when compared to previous ones are that the architectural design decisions are explicitly specified as patterns, and that we have developed a tool that supports framework development and specialization from an architecture- and pattern-oriented perspective. Strong task-driven automation support also makes our work different from previous approaches to framework development based on design patterns.

The notion of a pattern has been developed to support architecture-oriented programming paradigm, and is not meant for replacing the established concept of pattern. However, we pursue to extend our model to enable hierarchies of patterns, where more specific patterns refine more generic ones, thus narrowing the conceptual gap between design patterns and our patterns. We investigate alternative solutions on such hierarchies based on forms of inheritance, prototyping, composition, or patterns of patterns, not to forget explicit support for the separation of dynamic and static perspectives on patterns.

Other work to be done includes more advanced techniques for pattern classification and retrieval, explicit support for the separation of dynamic and static perspectives on patterns, automated facilities to recover patterns from framework code, support for standard architectures like Enterprise Java Beans, and integration of additional software development tools (such as a debugger) into the FRED tool box. Additional industrial case studies with framework development and specialization will be conducted as well.

Acknowledgements

FRED methodology and programming environment have been developed in a joint research project between the University of Tampere, Tampere University of Technology, and the University of Helsinki. The project has been funded by the National Technology Agency of Finland (Tekes) and by several software companies.

References

ACL96 Alencar P., Cowan C., Lucena C., A Formal Approach to Architectural Design Patterns. In Proc. *3rd International Symposium of Formal Methods Europe*, 1996, 576-594.

BCS00 Batory D., Cardone R., Smaragdakis Y., Object-Oriented Frameworks and Product Lines. In: Proc. *First Software Product Lines Conference,* Denver, Colorado. Kluwer, 2000, 227-247.

Bos00 Bosch J., *Design & Use of Software Architectures — Adopting and Evolving a*

Product-Line Approach. Addison-Wesley, 2000.

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

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

FPR00 Fontoura M., Pree W., Rumpe B., UML-F: A Modeling Language for Object-Oriented Frameworks. In: Proc. *ECOOP '00 (LNCS 1850)*, 2000, 63-83.

FSJ99 Fayad M., Schmidt D., Johnson R., (eds.), *Building Application Frameworks — Object-Oriented Foundations of Framework Design*. Wiley 1999.

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

Hak99 Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J., Koskimies K., Paakki J., Task-Driven Framework Specialization. In: Proc. *Fenno-Ugric Symposium on Software Technology (FUSST '99)* (Penjam J., ed.), Sagadi, Estonia, 1999. Technical Report 104/99, Institute of Cybernetics, Tallinn Technical University, 1999, 65-74.

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

JGJ97 Jacobson I., Griss M., Jonsson P., *Software Reuse — Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

Joh92 Johnson R.: Documenting Frameworks Using Patterns. In: Proc. *OOPSLA '92*, Vancouver, Canada, October 1992, 63-76.

JRL00 Jazayeri M., Ran A., van der Linden F., *Software Architecture for Product Families*. Addison-Wesley, 2000.

KrP88 Krasner G., Pope S., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Object-Oriented Programming*, 1988.

MDE97 Meijler T., Demeyer S., Engel R., Making Design Patterns Explicit in FACE — A Framework Adaptive Composition Environment. In: Proc. *ESEC '97 (LNCS 1301)*, 94-111.

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

OCS00 Ortigosa A., Campo M., Salomon R., Towards Agent-Oriented Assistance for Framework Instantiation. In Proc. *OOPSLA '00, Minneapolis, Minnesota USA, ACM SIGPLAN Notices*, 35, 10, 2000, 253-263.

OrC99 Ortigosa A., Campo M., SmartBooks: A Step Beyond Active-Cookbooks to Aid in Framework Instantiation. *Technology of Object-Oriented Languages and Systems* 25, June 1999, IEEE Press. ISBN 0-7695-0275-X

PaP00 Pasetti A., Pree W., Two Novel Concepts for Systematic Product Line Development. In: Donohoe P. (ed.), *Software Product Lines: Experience and Research Directions (First Software Product Lines Conference, Denver, Colorado)*, Kluwer Academic Publishers, 2000.

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

PrK99 Pree W., Koskimies K., Framelets — Small is Beautiful. In: *Building Application Frameworks — Object-Oriented Foundations of Framework Design* (Fayad M., Schmidt D., Johnson R., eds.), Wiley, 1999, 411-414.

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