Annotating Reusable Software Architectures with Specialization Patterns

Markku Hakala, Juha Hautamäki, Kai Koskimies

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

Jukka Paakki, Antti Viljamaa, Jukka Viljamaa

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

Abstract

An application framework is a collection of classes implementing the shared architecture of a family of applications. It is shown how the specialization interface ("hot spots") of a framework can be annotated with specialization patterns to provide task-based guidance for the framework specialization process. The specialization patterns define various structural, semantic, and coding constraints over the applications derived from the framework. We also present a tool that supports both the framework development process and the framework specialization process, based on the notion of specialization patterns. We will outline the basic concepts of the tool and discuss techniques to identify and specify specialization patterns as required by the tool. These techniques have been applied in realistic case studies for creating programming environments for application frameworks.

1. Introduction

An extensible architecture is the cornerstone of large-scale software reuse. Such an architecture can be implemented as an (object-oriented) *application framework* [1]. Application frameworks are steadily growing in industrial importance since they provide a suitable technology for implementing large-scale architecture-centric reusable assets such as *product lines* [2].

An object-oriented framework is a collection of classes that implement the shared architecture and the common functionality of a family of applications. The interface between the common, reusable framework and the application-specific parts built on top of it is realized as a set of extension points, or *hot spots* [3]. The architecture and specialization interface of a framework can be documented with (*design*) *patterns* [4]. Understanding and specializing a framework using patterns and hot spots is a challenging task for which tool support is of vital importance (see, e.g., [5] and [6]). According to our vision, future frameworks are accompanied by framework-specific programming environments that both guide and control application programmers in creating applications according to the conventions of the framework.

FRED (FRamework EDitor) is a prototype tool intended for generating programming environments for Java frameworks [7]. The FRED approach is based on a vision of architecture-oriented programming where the specialization interface of a framework is defined by *specialization patterns*. FRED supports both a framework developer in creating the specialization patterns for the framework, and an application developer in specializing the framework by following a task list generated by the tool, as implied by the patterns. The tool guides the application developer through the task list, dynamically adjusts the list according to the choices made by the adapter, and verifies that the syntactic and semantic constraints of the framework are not violated.

FRED provides thus an interactive environment in which specialization tasks can be executed incrementally in small pieces, allowing the application programmer to observe the effect in the source code, to cancel the tasks if needed etc. Ideologically, FRED is a descendant of the *"cookbook"* concept [8, 3, 9, 10, 11]. Related approaches include also *motifs* [12] and *hooks* [13]. The current implementation of FRED aims at supporting frameworks written in Java, but in principle the approach is not tied to any particular language. FRED is freely available at *http://practise.cs.tut.fi/fred*.

In this paper we will present the conceptual basis of FRED and discuss various techniques to apply this approach in practice. These techniques have been developed and evaluated in the context of two major case studies where we have applied the FRED methodology to realistic

Java frameworks. One of them is *JHotDraw* [14], a framework for implementing graphical editors. JHotDraw was chosen as an example framework because it is commonly known, mature, well structured, relatively large (about 150 classes), implemented in Java, and freely available.

Our second case study is a network management GUI framework with about 300 classes. This framework was developed by Nokia, and the application programming environment produced as a result of the case study is currently being used within Nokia.

We proceed as follows. The specialization pattern concept is introduced in Section 2. In Section 3 we discuss techniques for identifying specialization patterns and using the FRED approach in practice. Related work is discussed in Section 4. Finally, concluding remarks summarizing our experiences and future work topics are presented in Section 5.

2. Specialization patterns

Traditionally, software architecting has been understood as a part of the software design phase, and architectures have been mainly described with standard modeling languages (such as UML). To some degree, these abstract descriptions make it possible to assess the quality of the system at the architectural level and at design time, 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 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.

As the basis of architecture-oriented programming, we propose the notion of a *specialization pattern*. It is a specification of a recurring program structure, which can be instantiated in several contexts to get different kinds of concrete structures. A specialization pattern is given in terms of *roles*, to be played by structural elements of a program. We call the commitment of a program element to play a particular role a *contract*. A role may stand for a single element, or a set of elements. Thus, a role can have multiple contracts, and a program element can play many roles through a number of contracts. *Multiplicity* of a role bounds the number of its contracts.

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

Unlike constraints, some properties affect code generation or user instructions. For instance, most role kinds support a default name property for specifying the name of the program element used when the tool generates a default implementation for the element.

2.1. Pattern definition graph

A specialization pattern can be expressed as a directed acyclic graph called a *pattern definition graph*. Figure 1 shows the definition graph of a pattern representing a reusable structure for a class having a number of fields and an accessor method for each of them. The nodes in the graph represent roles. Class roles are denoted with circles, method roles with white squares, and field roles with black squares in the figure.

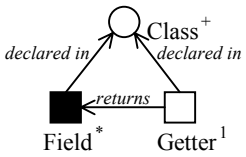

Figure 1. A pattern definition graph

The directed edges between roles are called *dependencies* (e.g. the edge from *Field* to *Class*). If there is a dependency from role *r* to role *s* then *s* is called the *dependee* of *r*. The multiplicity constraint for a role, in relation to its dependees, is placed in parentheses after its name in Figure 1. Multiplicity can be either exactly one (1), from zero to one (?), from one to infinity (+), or from zero to infinity (*).

Some of the properties of the roles are given as labels of the associated dependency arrows. They state that program elements playing the *Field* or *Getter* roles must be declared inside the class playing the corresponding *Class* role, and that each getter must return an object whose type is compatible with the type playing the *Field* role.

2.2. Casting

Applying a pattern is called *casting*, and the resulting structure is called a *cast*. Casting means incremental and interactive binding of suitable program elements to the unbound roles of the pattern. The tool provides the developer with a sequence of tasks that guide the developer in adapting the generic solution proposed by the pattern. *Production tasks* instruct the developer to

instantiate and bind a role. *Refactoring tasks* assist the user in modifying a program element bound to a role to adhere to the constraints imposed by the role.

Since a cast is an instance of a specialization pattern it can be presented as a directed acyclic graph as well. Figure 2 shows a *cast graph* based on the pattern in Figure 1. The nodes in the cast graph represent contracts. Each contract is a manifestation of some role in the associated definition graph. Contracts are of form r_i where r is that role and subscript i, as a positive integer, identifies the contract amongst all contracts of role r. The directed edges between contracts are called *dependency instances*. Each dependency instance between two contracts manifests a dependency defined between the two corresponding roles.

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

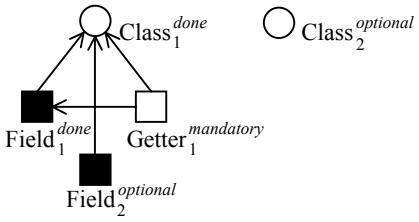

Figure 2. A cast graph

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

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

Figure 2 above portrays a *partial* cast, i.e. a pattern instance that is in the middle of instantiation. Some tasks have been done, resulting in a graph of contracts. The interpretation of the graph can be based on the semantic outline sketched for the graph in Figure 1. The developer has created a class and a field inside it. An unbound contract $Getter_1$ is shown to the user as a task to provide a getter method for that field. As this is a mandatory task, the cast is not yet considered *complete*. The developer has also a choice of continuing with optional tasks, some of which may lead to new tasks, even mandatory.

Figure 3 gives an overview of the whole situation at this point. The dashed arrows show how the contracts of the current cast are related to the roles in the pattern definition graph on the left, and how the bound (done) contracts, on the other hand, are also associated to the pieces of code on the right.

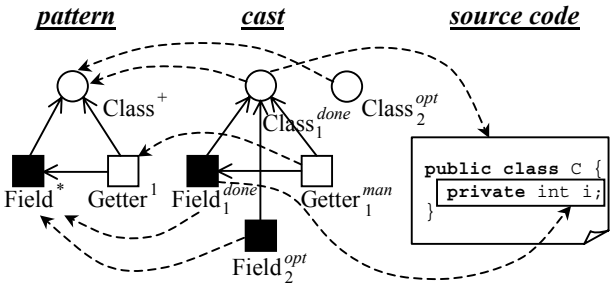

Figure 3. Pieces of code cast to roles of a pattern

3. Working with specialization patterns

FRED is a prototype tool employing the model discussed in Section 2. It is implemented in Java and provides 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 as well to guide programming according to various kinds of other architectural or coding conventions. As an example, we have modeled parts of the *JavaBeans* architecture as patterns, obtaining thus an environment for JavaBeans programming.

In this section we illustrate how FRED can be used to annotate a specialization interface of an object-oriented application framework with specialization patterns. We use JHotDraw [14] to demonstrate our approach. We will not go into the details of FRED or JHotDraw. Instead, we give an overview of the framework annotation process.

The user interface of FRED is shown in Figure 4. It contains a number of views to manage Java projects and specialization patterns. In the figure, the user is specifying specialization patterns for JHotDraw. She writes the patterns with *Pattern Editor* by creating roles and defining their properties and dependencies.

Pattern definitions are typically based on the analysis

of the framework source code and its documentation. The FRED environment provides a dedicated *Java Editor* for browsing and editing source code.

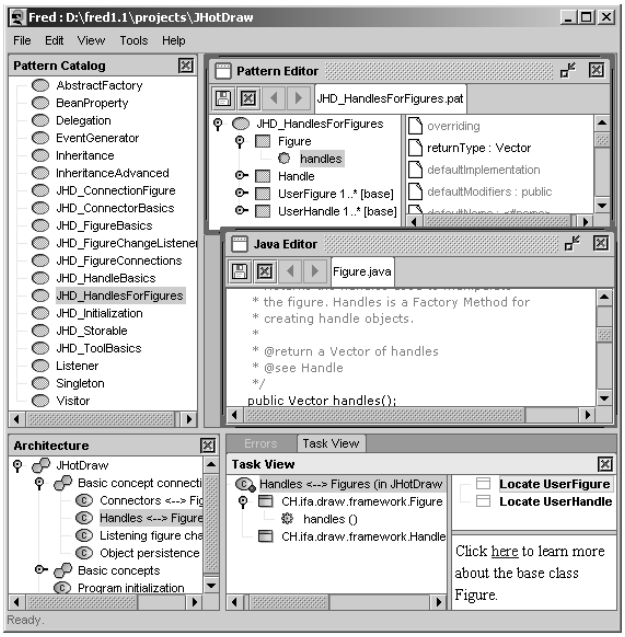

Figure 4. The user interface of FRED

The framework developer can also utilize existing general patterns from FRED's *Pattern Catalog* as templates for her framework-specific patterns. The catalog contains, for example, versions of many generally applicable patterns like the design patterns presented in [4]. When the user wants to see her patterns in action, she can instantiate them and then examine the instantiated patterns in the *Architecture View*. The *Task View*, in turn, shows the task list for the selected pattern instance (cast).

3.1. Identifying framework hot spots

There are a number of useful heuristics that help in identifying and specifying a framework's hot spots. The heuristics we discuss first are most relevant to *white-box frameworks*, which use inheritance as the main specialization technique. An overview of a more general technique is given later on in Section 3.4.

Here we assume that the framework has a layered structure and that its basic concepts are implemented on the highest layer as abstract interfaces. In addition, we assume that we are annotating a fairly mature framework and that we have enough information about the framework's structure and its intended use.

Since any framework can be annotated in numerous different ways, the framework annotator must decide what kind of assistance she wants to give for the framework users. Adding constraints will give the user better guidance. However, at the same time she will loose some of her freedom. The formalization of a framework's specialization interface always reveals only a subset of possible implementation variations. We argue that it is better first to provide patterns for a quite narrow specialization interface, and later modify the patterns and add new ones to enable more advanced ways to use the framework.

Template and *hook methods* are obvious candidates when trying to locate the important hot spots of an object-oriented framework [3, 15]. Most of the hot spots can often be found by analyzing overridden methods, because polymorphism needed in hook methods is usually implemented using method overriding [16].

There are typically hundreds of overriding relationships between methods in any non-trivial application framework. That is why it is best to concentrate first on the main concepts of the framework and their relationships. The main concepts of the framework usually map fairly consistently with the top-level interfaces in the framework implementation.

Figure 5 represents the highest-level interfaces of the JHotDraw framework as a UML class diagram. By implementing these interfaces directly (or indirectly by adapting the default implementations provided with the framework) the user can have different kinds of figures, handles to grab them, connectors to link them together, and tools to manipulate them in her drawing application.

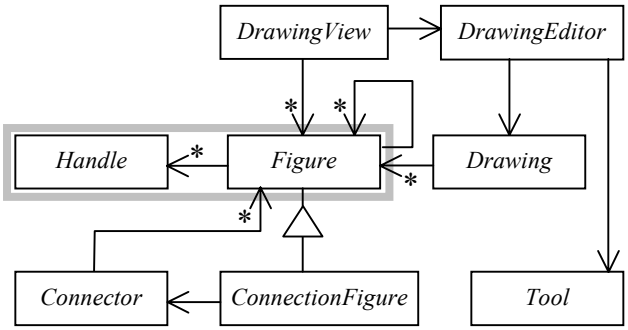

Figure 5. A hot spot in the JHotDraw framework

In our case study we needed 10 specialization patterns to annotate the main parts of the specialization interface of JHotDraw. As an example of a real, although somewhat simplified specialization pattern, we look at *HandlesForFigures*. As its name suggests, its purpose is to assist the application developer to define handles for her figure objects. This hot spot involves two framework interfaces highlighted in Figure 5.

Before explaining the details of this specialization pattern, it is important to distinguish between framework roles and application roles. A *framework role* is a role that

will be bound to a framework source code entity by the framework developer. Framework roles can be deduced from the source code directly. For example, there usually exists a one-to-one mapping between a framework interface representing a certain framework's concept and a framework role in a pattern describing ways to implement that interface.

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

3.2. Specifying class and method roles

Figure 6 represents a more detailed UML class diagram describing the hot spot related to the relationship between *Figure* objects and *Handle* objects in JHotDraw. The diagram shows that there can be a number of handles for each figure, and that each handle object knows its owner figure. The methods, which are relevant to that particular relationship and actually implement the association, are highlighted in the diagram.

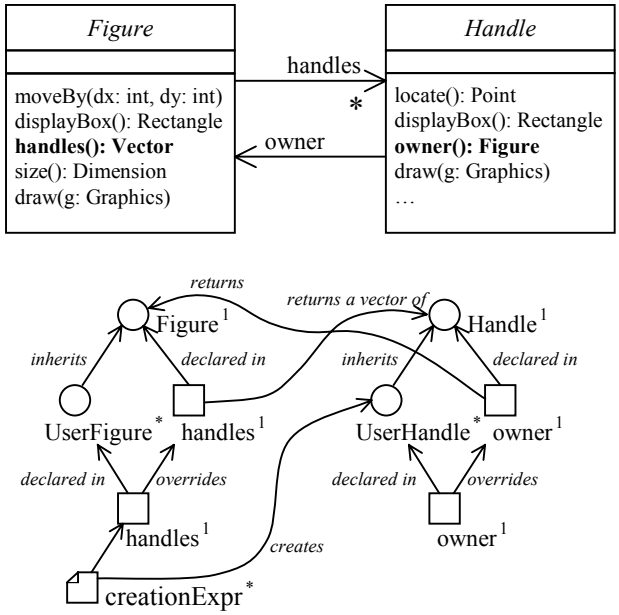

Figure 6. A pattern for defining handles for figures

Below the class diagram in Figure 6 there is a graph representation of the *HandlesForFigures* pattern. A part of it is also visible in *Pattern Editor* in Figure 4. The pattern specifies that if the user wishes to have handles in her figures she must bind her *Figure* subclasses to the *UserFigure* role and then provide or generate a method, which overrides the *handles* method declared in *Figure* interface. Similarly, she must override the *owner* method in her *Handle* subclasses.

There are two class roles for both basic concepts involved in this hot spot: a framework role and a related application role. The framework role is named after the framework class it has been derived from (e.g. *Figure*). The corresponding application role (identified with the *"User"* prefix in Figure 6) represents the set of possible application-specific subclasses to be derived from the framework class. Thus, it has a dependency to the framework role and an associated inheritance constraint.

There is one method role for each class role in Figure 6. Those method roles that denote methods that are to be declared within framework classes have constraints restricting the types of their return values (e.g. *Handle*'s *owner* method must return a *Figure*). The application method roles, on the other hand, have overriding constraints referring to the corresponding framework method roles.

3.3. Roles for method implementation

So far we have discussed only class and method roles that guide the user to a particular hot spot of the framework. They indicate the classes she must inherit and the methods she must override, in order to adapt the framework. Specialization patterns can, however, be used also for representing various ways to code the actual implementation. For instance, we can use patterns to describe algorithms defined in method bodies, to show data fields that are needed to implement the algorithms, and to describe constructors required to initialize the fields.

In FRED, we describe method bodies and field initialization clauses with *code snippet roles*. Figure 6 shows an example of a code snippet (*creationExpr*), which has a dependency (*creates*) to the *UserHandle* class role. The snippet is used here to give the user a possibility to generate code for creating a number of concrete *Handle* objects that she can then return in a vector from her handles method.

In general, code snippets can be created under application roles as required. There can be many code snippets for one role, e.g. to describe algorithmic options. The framework developer should specify snippets as generalizations of the most representative examples among the existing implementations. The guiding roles and constraints for method bodies should be specified as suggestions, not as mandatory constraints. The same applies for field and constructor roles. On the other hand, selecting

the implementation strategy for a hook method to be overridden usually fixes the variation possibilities for the related fields and constructors, too.

3.4. Goal-oriented identification of hot spots

JHotDraw is a typical example of a framework, which uses inheritance and method overriding as a means to provide extensibility. FRED can be used also with frameworks that involve more advanced reuse techniques, such as *reflection* or *dynamic object composition*. For these cases we have come up with a more general way of identifying the hot spots of a framework. It is based on an analysis of the expected behavior of the framework specializer.

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 framework user 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's users. This resembles an *implementation case* [17] 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 isolated from each other; instead, when pursuing the goal, the user may struggle with a number of hot spots 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.

To create goal-oriented specialization patterns 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 framework user, specialization goals constitute a linked structure where achieving one goal leads to another.

As an example, Figure 7 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 [8]. 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 framework user. 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.

One method to construct specialization patterns for a specific goal is to first derive an example specialization that achieves the goal. This example specialization helps the pattern modeler to identify the required program elements and their interactions. This process is similar to object-oriented analysis on the architecture level: central concepts of a specialization pattern are identified and associated with roles. In this way it is usually easy to find class and method roles. However, other aspects of the specialization pattern like constraints may be more implicit in an example specialization.

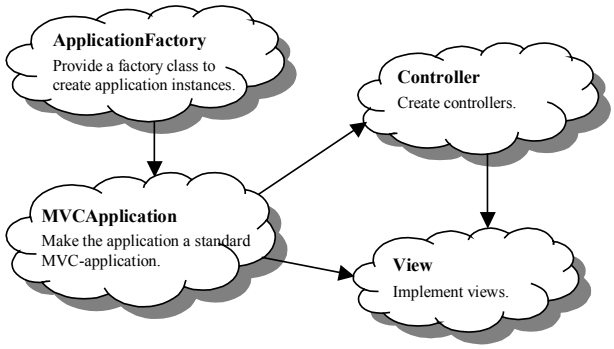

Figure 7. A set of specialization goals

Thus, the pattern modeler creates a set of specialization patterns to describe solutions for the identified specialization goals while the specializer utilizes this knowledge by doing tasks generated by FRED. This approach for finding specialization patterns was used in the network management GUI case study. In this way, we identified 13 specialization goals for the framework, each giving rise to a specialization pattern.

3.5. Pattern initialization

Once the framework developer has specified the framework hot spots as specialization patterns, she must make the framework annotation ready to be used. This *pattern initialization* means instantiating the specialization patterns and associating the framework roles with the corresponding framework source code elements. The rest of the roles will be left for the application developer to bind.

The *Architecture View* of FRED displays the instantiated patterns in the current project. In Figure 4, the user has selected an instance of the *HandlesForFigures* pattern described above. On the left hand side of the *Task View*,

one can see the current cast with bindings to classes *Figure* and *Handle*, as well as to method *handles*. The contracts for the application-specific subclasses are not bound yet, so they are represented on the right as tasks for the user to provide the missing classes. An HTML description of the selected task is shown below the task list.

3.6. Task-driven framework specialization

After the framework developer has initialized her patterns the framework annotation is finally ready to be used. At this point, FRED will create a task list for the framework adapter to systematically derive an application from the framework. Each task in the task list represents a certain contract and its associated role and constraints. The task list is inherently dynamic: solving a task may generate additional tasks. For instance, the creation of a class for an application role typically implies that some (abstract) method inherited from a framework class must be overridden and tuned for the specific application. In such a case, the creation of the application class is a task, which implicitly generates another task for method overriding as described in Section 2.

The generated tasks together with the cast representing the already bound roles are shown in FRED's *Task View*. The tasks can either guide the user in providing program elements to be bound to roles or instruct on how to fix possible constraint violations the already bound elements cause.

Some of the tasks are mandatory, while some of them are optional. Also, some tasks are mutually ordered and must be solved in a certain sequence. For example, from the pattern shown in Figure 6 FRED generates a list of tasks to attach handles for figures in an editor application. The list contains, e.g., an optional task for providing a class for the role *UserFigure* (see Figure 4) and, if the user carries out this task, a mandatory task for providing a method that overrides the *handles* method inherited from the framework class *Figure*.

Executable code for realizing the tasks can be provided in several ways: by coding from scratch, by introducing a binding to a suitable class or method that already exists, or by tailoring a default code snippet generated by FRED. For these, FRED provides a dedicated *Java Editor*, which parses the source code incrementally as the user types it in. Changes in the source code are monitored and their validity is continuously checked against the constraints specified in the patterns. Possible violations of constraints immediately result in new refactoring tasks. Hence, the proper use of the framework is constantly validated and supervised by the system. Besides the standard *Java Editor*, also more high-level (framework-specific) tools can be provided by extending FRED's general tool API.

When walking through the task list, the application can be developed step-by-step under the interactive guidance and documentation provided by FRED. This disciplined process makes sure that all the core hot spots of the framework are traversed and that the framework is extended by concrete code that is necessary to make the application complete. FRED keeps track of the status of the tasks, and the application is considered complete and executable when the user has done all the mandatory tasks.

In addition to following the tasks generated from the patterns defined for the framework, the application developer can herself instantiate general patterns that are suitable for her application and use them for producing application code. Typically these patterns would involve coding convention patterns like JavaBeans component patterns or *Singleton* design pattern [4].

4. Related work

4.1. Tool support for framework specialization

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* [8, 3] and *patterns* [5].

As emphasized in this paper, instructions for adapting a framework cannot be adequately expressed as a static and linear step-by-step task list, because a choice made during the specialization process may change the rest of the list completely. That is why cookbooks, although a step to the right direction, are not enough. Our model can be seen as an extension of the notion of framework cookbooks.

Another advanced version of cookbooks is the *SmartBooks method* [9]. 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 [10]. This reminds our model, but while SmartBooks method provides a rule-based, feature-driven, and functionality-oriented system, our approach is pattern-based, architecture-driven, and more implementation-oriented.

Froehlich, Hoover, Liu, and Sorenson suggest semi-formal templates for describing specialization points of frameworks [13] in the form of *hooks*. A hook presents a recipe as an imperative algorithm. This algorithm is intended to be read, interpreted, and carried out by the

framework adapter. Tool support has been suggested, but as the solution description within a hook is given in procedural form it may be hard to support the non-linearity of software engineering process.

Fontoura, Pree, and Rumpe present a UML extension *UML-F* to explicitly describe various kinds of framework variation points [11]. 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 variation point type has a dedicated 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. view UML-F descriptions 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 [11], however. The proposed implementation technique is based on adapting standard UML case tools. This does not directly support interactivity in 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 exactly that [18]. A framelet is a small framework with a clearly defined simple interface used for structuring new software architectures and especially for reorganizing legacy code. We have gained good experiences with annotating framelets with FRED patterns to make it easy to adapt and combine them into software systems.

4.2. Pattern-based tool support

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 [19] 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 [4], various formalizations have been given to design patterns resembling our pattern concept (for example, [6], [20], [21], and [22]), 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 its piecemeal application in realistic software development.

In [23] Eden, Hirshfeld, and Lundqvist present *LePUS*, a symbolic logic language for the specification of recurring *motifs* (structural solution aspects of patterns) in object-oriented architectures. They have implemented a PROLOG based prototype tool and show how the tool can utilize LePUS formulas to locate pattern instances, to verify source code structures' compliance with patterns, and even to apply patterns to generate new code.

In [24] 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.

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.

5. Conclusions

We have presented a new tool-supported approach to architecture-oriented programming based on Java frameworks. We envisage application development shifting towards using platforms like object-oriented frameworks, which support extensive reuse. So far there is relatively little tool support for this kind of software development, where the central problem is to build software according to the rules and mechanisms of the framework.

FRED represents a possible approach to produce adequate environments for framework-centric programming. It supports architecture-oriented programming by providing tasks, which guide the adaptation of reusable architectures realized as object-oriented application frameworks. The tasks are generated dynamically based on specialization patterns that specify the specialization interface of the framework.

We are aware of some restrictions in our current specialization pattern model. For instance, it does not allow dependencies between patterns, and it does not provide enough modularity within patterns. Also, currently in FRED there is no way to check that the user has defined a method body as intended by the framework designer. To allow more control, FRED could be augmented with a richer set of statically verifiable constraints like, for example, in *CoffeeStrainer* [25].

In order to further validate and enhance our methodology of using specialization patterns, we are going to apply it to a range of frameworks of varying sizes and characteristics. At the same time we will investigate ways to make it easier to import existing code into FRED for systematic management.

In the current version of FRED, annotating a framework with specialization patterns includes many trivial details that could well be automated. For example, many roles, dependencies, constraints, and default values could

be deduced directly from the framework source code and existing example applications using various kinds of heuristics. We could also apply the techniques developed for automatic discovery of design patterns from source code (see, e.g., [26]).

A possibility to automate trivial (one-to-one) role bindings would also greatly ease pattern initialization, i.e. the process of binding the specified patterns to the framework entities so as to provide an initial set of annotations for the user.

Also, since new ways of adapting a framework are found even in the application development process, the tool should make it possible to easily modify the patterns during the specialization process. Currently this is not possible. A potential solution to this problem is to make pattern instances more dynamic, modifiable entities.

Despite the restrictions mentioned above, our experiences with real frameworks confirm our belief that the fairly pragmatic approach of FRED matches well with the practical needs. Our future work includes integration of FRED with contemporary development environments and building FRED-based support for standard architectures like *Enterprise Java Beans*.

Acknowledgements

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

References

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

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

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

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

[5] Johnson R., Documenting Frameworks Using Patterns. In: Proc. *OOPSLA '92*, Vancouver, Canada, 1992, pp. 63-76.

[6] Florijn G., Meijers M., van Winsen P., Tool Support for Object-Oriented Patterns. In: Proc. *ECOOP '97*, Jyväskylä, Finland, 1997, LNCS 1241, pp. 472-496.

[7] FRED Site. http://practise.cs.tut.fi/fred, 2001.

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

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

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

[11] Fontoura M., Pree W., Rumpe B., UML-F: A Modeling Language for Object-Oriented Frameworks. In: Proc. *ECOOP '00*, Sophia Antipolis and Cannes, France, 2000, LNCS 1850, pp. 63-83.

[12] Lajoire R., Keller R., Design and Reuse in Object Oriented Frameworks: Patterns, Contracts and Motifs in Concert. In: *Object Oriented Technology for Database and Software Systems*, Alagar V., Missaoui R. (eds.), World Scientific Publishing, Singapore, 1995, pp. 295-312.

[13] Froehlich G., Hoover H., Liu L., Sorenson P., Hooking into Object-Oriented Application Frameworks. In: Proc. *ICSE '97*, Boston, Massachusetts, 1997, pp. 491-501.

[14] JHotDraw 5.1 source code and documentation. http://members.pingnet.ch/gamma/JHD-5.1.zip, 2001.

[15] Schauer R., Robitaille S., Martel F., Keller R., Hot Spot Recovery in Object-Oriented Software with Inheritance and Composition Template Methods. In: Proc. *IEEE Int. Conf. on Software Maintenance 1999 (ICSM '99)*, Keble College, Oxford, England, 1999, IEEE Press, pp. 220-229.

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

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

[18] 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, pp. 411-414.

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

[20] Meijler T., Demeyer S., Engel R., Making Design Patterns Explicit in FACE — A Framework Adaptive Composition Environment. In: Proc. *ESEC/FSE '97*, Zurich, Switzerland, 1997, LNCS 1301, pp. 94-110.

[21] Mikkonen T., Formalizing Design Patterns. In: Proc. *ICSE '98*, Kyoto, Japan, 1998, IEEE Press, pp. 115-124.

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

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

[24] Alencar P., Cowan C., Lucena C., A Formal Approach to Architectural Design Patterns. In: Proc. *3rd International Symposium of Formal Methods Europe (FME '96)*, St Hugh's College, Oxford University, England, 1996, pp. 576-594.

[25] Bokowski B., CoffeeStrainer — Statically-Checked Constraints on the Definition and Use of Types in Java. In: Proc. *ESEC/FSE '99*, Toulouse, France, 1999, ACM Software Engineering Notes 21, 6, 1999, pp. 355-374.

[26] Krämer C., Prechelt L., Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: Proc. Working Conference on Reverse Engineering (WCRE '96), Monterey, California, 1996. IEEE Press, pp. 208-215.