ANNOTATING FRAMEWORKS WITH PROGRAMMING PATTERNS

Jukka Viljamaa, Antti Viljamaa

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

A 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 specified formally in such a way that the specification can be used to provide task-based guidance for the framework specialization process. The specialization interface is annotated with programming patterns, which define various constraints over the applications derived from the framework. The tool (FRED) allows the application developer to bind actual system elements to the pattern parameters or to 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. The tool has been implemented in Java for Java, and it has been evaluated against a real industrial framework.

1. INTRODUCTION

Object-oriented *application frameworks* have become an established way to implement reusable software architectures [FSJ99]. Common concepts and algorithms for a given domain can be embodied in a framework and only variable details need to be implemented in applications derived from the framework.

In general, the more flexible a framework is, the more difficult it is to clearly specify its *specialization interface*. On the other hand, efficient use of a framework is possible only if comprehensive and illustrative documentation exists for it. To solve this *framework specialization problem* the reuse interface should be organized into smaller units often called framework's *hot spots* [Pre95].

Typically, the documentation provided together with a framework describes in varying formats its extension points and intended use. In addition, the features of the implementation language can be used to express some aspects of the specialization interface. 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 hot spots of frameworks and design patterns (e.g. [Rie00], [Hak99], and [FMW97]). It seems that the design pattern concept is a promising starting point for systematic approaches to the specification of framework architectures and specialization interfaces.

We introduce a notion of *programming pattern* as a model for describing and restricting the properties and relationships of a group of target language structures. Programming patterns form the basis of our *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.

We have implemented a development environment called *FRED (FRamework EDitor for Java)* to demonstrate the potential of task-driven framework engineering. FRED introduces semi-graphical editors for defining programming patterns and for using them to guide framework adaptation. This paper shows how the specialization interface of a framework can be annotated with programming patterns to specify constraints that regard larger wholes than just separate types. FRED processes programming patterns dynamically to generate instructions on how to fix constraint violations and, more generally, how to proceed with the framework adaptation.

The FRED methodology and programming environment have been validated in a real-world case study carried out for one of our industrial partners. FRED is freely available at http://practise.cs.tut.fi/fred.

The rest of this paper is organized as follows. In Chapter 2 we discuss the benefits of a task-driven approach to framework specialization. The programming pattern model adopted for describing framework specialization interfaces is explained in Chapter 3. Chapter 4 presents the FRED environment from the framework developer's point of view. We discuss ways to identify the hot spots of a framework and to annotate them with programming patterns. Finally, conclusions are drawn in Chapter 5.

2. TASK-BASED FRAMEWORK SPECIALIZATION

Traditionally software architectures have been described with standard modeling languages (such as UML) or with dedicated architecture description languages. 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.

In particular, we need explicit architectural descriptions 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, which makes 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.

3. PROGRAMMING PATTERNS

In this chapter we describe the main characteristics of the *programming pattern model* adopted to be the basis of our task-driven programming environment. For formal definition of the model and details of the related algorithms refer to [Hak00].

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

Applying the pattern is called *instantiation*, and the resulting structure is called a *pattern instance*. A pattern instance consists of role instances, each of which maps to a role in a pattern, and represents a binding between a concrete program element and a role in the pattern.

The instantiation of a programming pattern means incremental and interactive binding of suitable program elements to the roles of the pattern. The tool provides the developer with the sequence of *programming 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.

Pattern Syntax

It is possible to separate the syntactic and semantic aspects of programming patterns. Syntactically, a programming pattern can be expressed as a directed acyclic graph called the *definition graph* (see Figure 1 for an example). The nodes in the graph represent *roles* (e.g. *Class* and *Field*). Each role is identified by a unique name within the pattern.

The directed edges between roles represent *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*. For each role there is also a *cardinality constraint*, which bounds the number of instances of the role, in relation to its dependees. The cardinality constraint for a role is placed in parentheses after its name.

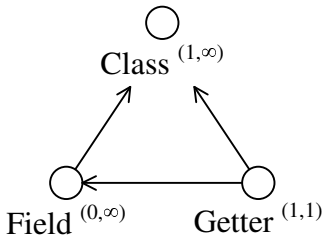

Figure 1. An example pattern definition graph.

Note that no semantic information is encoded in the graphical notation introduced above. 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.

A pattern instance can be presented as a directed acyclic graph as well. Figure 2 shows a pattern instance based on the pattern in Figure 1. In the *instance graph*, the nodes 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 between role instances are called *dependency instances*. Each dependency instance manifests a dependency defined between the two roles that its end points are instances of.

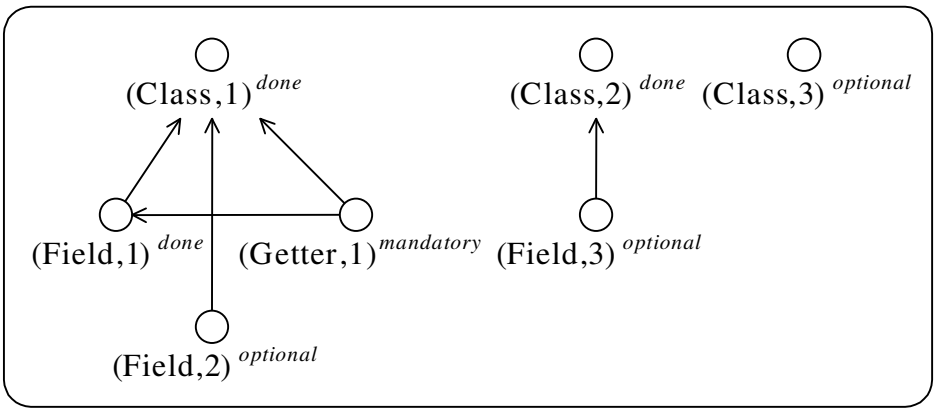

Figure 2. An example pattern instance graph.

Role instances correspond to production tasks within a tool environment. That is why each role instance has a *state*. As a task, role instance can be *done* or *undone*. Furthermore, an undone task may be considered as *mandatory* or *optional*. The state of a role instance is written in superscript after the label of the associated node. To understand how the states of individual role instances change, and the dynamics of instance graph in general, we must look at the pattern instantiation process.

Pattern Instantiation

Instantiation of a pattern takes place in a development tool environment as an incremental interaction between the tool and the developer. From the perspective of the tool, the instantiation process can be described by an algorithm that assumes a definition graph and an instance graph, which it augments with new role instances whenever possible. The newly created role instances imply mandatory or optional production tasks to be carried out by the developer. As the user completes a task, the state of the corresponding role instance is changed to *done*, and the tool re-evaluates the algorithm to determine whether it is possible to create new role instances.

Figure 2 above 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 for the graph 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*. The developer has also a choice of continuing with optional tasks, some of which may lead to new tasks, even mandatory.

The instantiation algorithm 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.

Pattern Semantics

So far, we have discussed only the syntactical properties of programming patterns. 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 nodes 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* (i.e. other instance nodes reachable from the current instance).

This approach relates to *attribute grammars* [Knu68], 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 allow expressions to refer roles by their names. References can be made only to the ancestors of the role, which owns the expression. If an expression

for role *r* refers to an 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*, the tool warrants the unambiguity of the reference.

4. ANNOTATING FRAMEWORKS IN FRED ENVIRONMENT

FRED (FRamework EDitor for Java) is a prototype tool implementing the model discussed above in Chapter 3. It 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.

In this chapter we use *JHotDraw* [JHD00] — a simple, yet sophisticated Java framework for implementing graphical editors — to illustrate how FRED can be used to annotate a specialization interface of a framework with programming patterns. We do not go into details of FRED of JHotDraw. Instead, we give an overview of the framework annotation process.

The user interface of FRED is shown in Figure 3. It contains a number of views to manage Java projects and programming patterns. In the figure, the user is specifying programming patterns for JHotDraw. She has opened the *Architecture view*, which shows the instantiated patterns. The *Project view* below allows the user to browse source files associated with the project. The *Task view*, in turn, shows the task list for the selected pattern instance. Finally, on the main workspace area, the *Pattern Editor* and *Java Editor* enable her to create and manage programming patterns and edit source files, respectively.

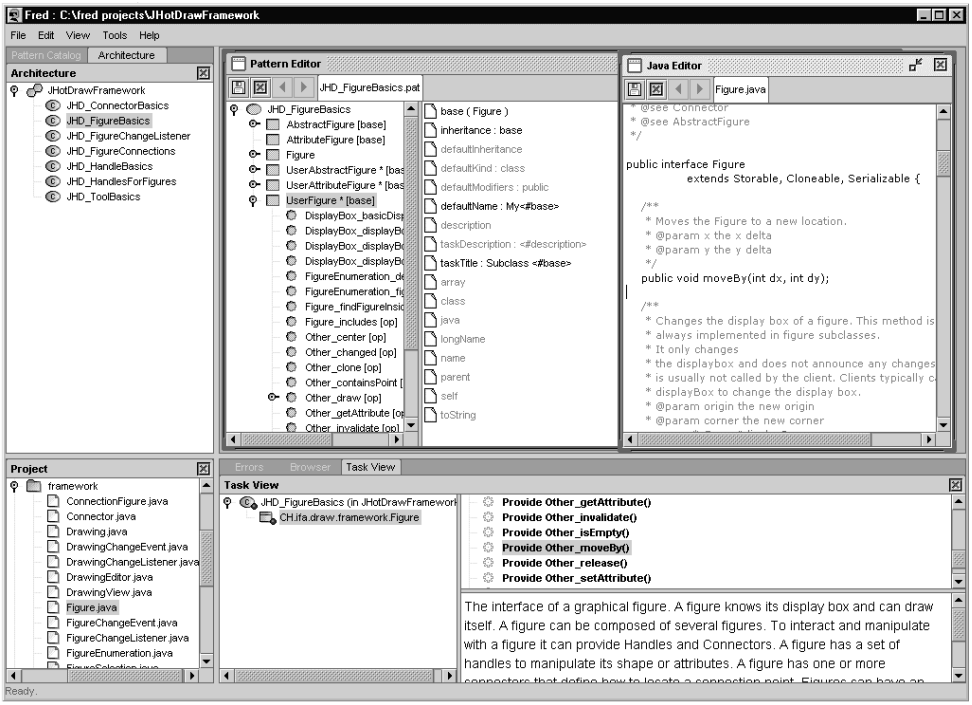

Figure 3. The user interface of FRED.

There are a number of useful heuristics that help in identifying and specifying framework's hot spots. The heuristics we are about to discuss are most relevant to *white-box frameworks*, which use inheritance as the main specialization technique. 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. Formalization of framework's specialization interface reveals always only a subset of possible implementation variations. That is why we argue that it is better to first 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.

Identifying Framework Hot Spots

Template and *hook methods* are obvious candidates when trying to locate the important hot spots of a framework [Pre95, SRM99]. 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 [Dem98, FAM00].

There are usually 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 frameworks and their relationships. We argue that the main concepts of the framework usually map fairly consistently with the top-level interfaces (roots of inheritance chains) in the framework implementation.

Every interface does not necessarily imply a hot spot, however. There are interfaces that are used only internally in the framework and which thus are irrelevant to the framework adapter. They must be identified and left out based on intuition and the documentation available for the framework.

Besides selecting the right interfaces as parts of hot spots, it is important to note that there are different kinds of hot spots that need different kinds of programming patterns. Figure 4 represents the highest-level interfaces of JHotDraw framework as a UML class diagram. The main concepts of the framework are written in bold in the figure. Each of them will have its own *basic pattern*, which allows the application developer to define new *Figure*, *Handle*, *Connector*, and *Tool* subclasses in order to have different kinds of figures, handles to grab them, connectors to link them together, and tools to manipulate them, respectively.

To show how to derive a new application specific subclass implementing a basic framework concept, we must take into account the typical layered structure of a framework. It consists of (1) *the interface layer* that defines the framework's basic objects, their services, and the connections between them; (2) *framework core implementation layer* that supplies the default behavior of the framework by partially implementing the interfaces with abstract classes; and (3) *default component layer* that comprises full default implementations (concrete classes) for commonly recurring circumstances. The programming patterns should allow the end-user deriving application from the framework to extend concepts from any of these layers. The different options can be represented in one pattern, for example, to make it easy for the framework user to choose the abstraction level she wishes to work on.

There are also patterns that describe interactions between two or more concepts (e.g. *HandlesForFigures* and *FigureConnections* in Figure 4) as well as patterns for adding

certain properties or behavioral aspects to subclasses (e.g. *FigureChangeListener*). In these patterns we assume that the user already has derived her application specific subclass using a basic pattern and now wishes to use another pattern to add some functionality to the already existing subclass.

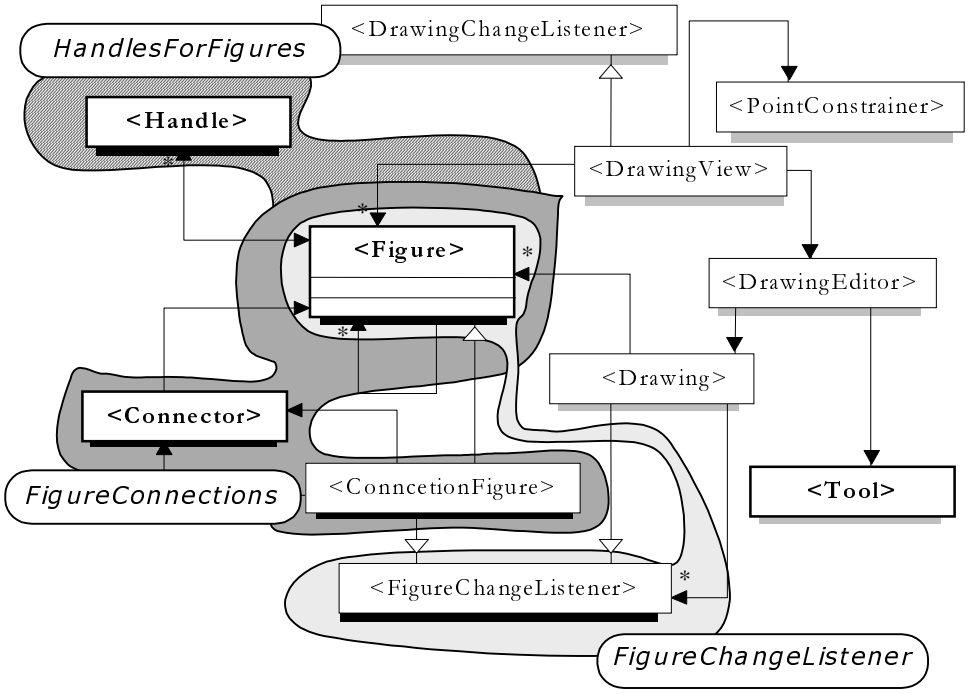

Figure 4. The main concepts and hot spots of JHotDraw framework.

Deducing Class and Method Roles Based on Source Code

Once we have identified the hot spots we want to annotate and decided which kinds of patterns we need to define, we start working out the details of each individual pattern. At this point it is important to distinguish between framework and user 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 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. This implies that a framework role can be usually automatically bound to the source code entity it was deduced from.

A *user role*, on the other hand, is a role that will be bound to an application source code entity later on. User roles typically depend on the framework roles and contain constraints that guide the framework user as she derives her application from the framework. The structure and constraints of user 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.

Figure 5 gives an example of deducing a programming pattern based on source code. It represents a pattern for providing new *Figure* subclasses as well as the related class hierarchy in JHotDraw it was derived from. The idea of the pattern is to describe each layer of the framework (except the top layer) with two *class roles*: a framework role and a user role. This enables the framework specializer to inherit

from each layer without forcing her classes to be similar to the framework classes on the same layer. For example, we may want to make sure that all user classes will be concrete, whereas framework classes on intermediate levels can, of course, be abstract.

Figure 5 shows how the layering of the class roles reflects the three framework layers: (1) interface layer, (2) framework core implementation layer, and (3) default component layer. Note that the highest-level framework class role has the same name as the main interface of the pattern (*Figure*). Similarly, the lower level framework roles are named after the framework classes they have been derived from. The user roles have the same names with *"User"* prefix. Both user and framework roles have a dependency to the immediate super class role and an associated inheritance constraint.

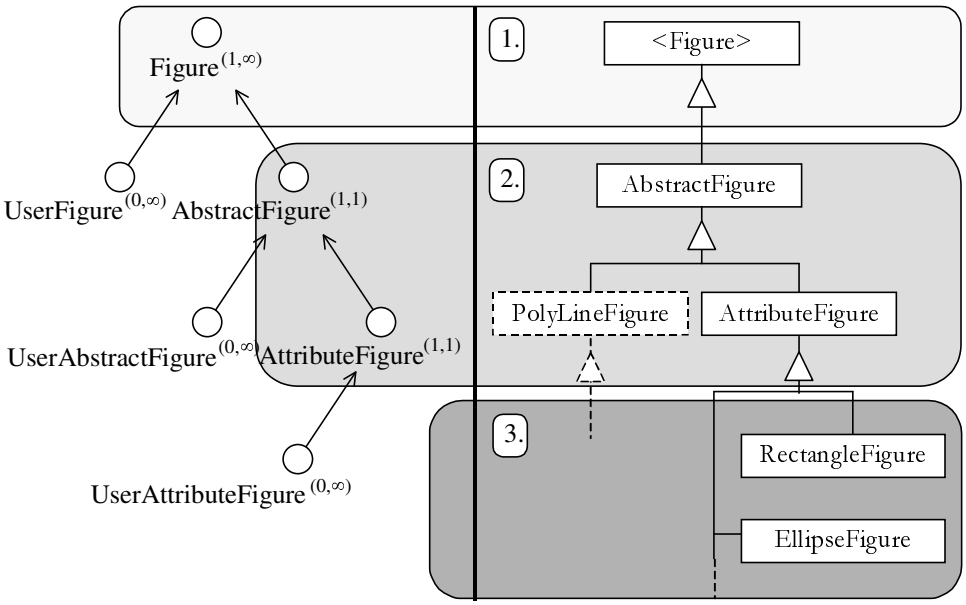

Figure 5. A programming pattern for new Figure subclasses and the related class hierarchy.

To actually implement an interface we need to implement all methods in it. That is why we need also method roles in our patterns. The methods of an interface (or a class) can be divided into *groups* that correspond to the main concepts of the framework. A method can be related to none, one, or many main concepts depending on its signature (name, parameter and return type names) and related comments. Method groups can be formed for desired combinations of concepts as needed.

The method grouping can be used to specify *method roles* only for relevant methods of interfaces to the current pattern. Other methods will be taken care of by other patterns. Actually these other methods manifest the ways the current pattern depends on other patterns. Unfortunately, at the moment there is no way to force the user to take these inter-pattern dependencies into account in FRED. The only way to express them is to use pattern descriptions and comments.

Specifying Roles for Method Bodies, Fields, and Constructors

So far we have concentrated on guiding the user to the hot spots of the framework by providing her with class and method roles showing the classes she must inherit and methods she must override in order to adapt the framework. Programming patterns can, however, be used also for representing various ways to code the actual implementation. 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*. Code snippets can be created under user roles as required. There can be many code snippets for one role, e.g. to describe algorithmic options. Snippets should be specified by the framework developer 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, once the user has selected her implementation strategy for a hook method to be overridden, that usually fixes the variation possibilities for the related fields and constructors, too.

Automating the Annotation Process

At the moment programming patterns for annotating a framework need to be defined from scratch with the *Pattern Editor* by the framework developer. This process includes many trivial details that could well be automated. For example, most roles, dependencies, constraints, and default values could be deduced directly from the source code of the framework using heuristics similar to ones described above.

We are currently improving the *Pattern Editor* to make it possible to turn a piece of source code (a hot spot) into a programming pattern. The tool will show the class hierarchy of the framework for the user as a tree, from which she selects the relevant interfaces and classes and groups them according to hot spots they belong to. After that the tool can produce initial pattern versions based on the user's groupings and the dependencies deduced from source code. This initial version can then be modified by the developer to get the exact amount of flexibility needed.

A possibility to automate trivial role bindings would also greatly ease *pattern initialization*, i.e. the process of binding the specified patterns to the framework entities to make the annotation set ready to be used.

5. Conclusion and Future Work

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 applications are derived from the framework by following the tasks generated from the patterns. The FRED tool supports architecture-oriented programming by guiding the application developer through the specialization process and by checking that the pattern constraints are not violated.

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 approximately 300 classes. After analyzing the framework, a collection of 13 patterns was defined to cover its reuse interface. 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.

The notion of a programming pattern that has been developed to support architecture-oriented programming paradigm 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.

Another aspect of the FRED environment we wish to develop further is the assistance for framework developers to annotate their frameworks. In order to make it easier to import existing code to FRED, we are investigating ways to automate the heuristics described in Chapter 4.

It seems also 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 adapting 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 potential solution to this problem is to make pattern instances more dynamic, modifiable entities.

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. Other members of the FRED team are Markku Hakala, Juha Hautamäki, Kai Koskimies, and Jukka Paakki.

REFERENCES

Dem98 Demeyer S., Analysis of Overridden Methods to Infer Hot Spots. In: Proc. *ECOOP'98 Workshops, Demos, and Posters (Workshop Reader)*, Brussels, Belgium, July 1998, LNCS 1543, 1998, 66-67.

FAM00 FAMOOS project's home page. http://iamwww.unibe.ch/~famoos/, 2000.

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

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.

Hak00 Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J., Task-Driven Specialization Support for Object-Oriented Frameworks. Submitted for publication, http://www.cs.helsinki.fi/research/fred/reports/ecoop01final.pdf, 2000.

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.

JHD00 JHotDraw 5.1 source code and documentation. http://members.pingnet.ch/gamma/JHD-5.1.zip, 2000.

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

Knu68 Knuth D., Semantics of Context-Free Languages. *Mathematical Systems Theory* 2, 1968, 127-145.

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

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

SRM99 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, IEEE Computer Society Press, 1999, 220-229.