Finding and documenting the specialization interface of an application framework

Juha Hautamäki and Kai Koskimies* †

Institute of Software Systems, Tampere University of Technology, P.O. Box 553, 33101 Tampere, Finland

SUMMARY

This paper presents an approach to find, specify and use the specialization interface of an object-oriented framework as a set of framework-specific patterns. The approach is based on the assumption that the user tries to reuse a framework by setting meaningful goals in the context of their application and then achieves the goals by performing a sequence of programming tasks. The goals can be refined as informal specialization patterns, which are framework-specific descriptions on how to reach a particular specialization goal. Furthermore, the obtained specialization patterns can be transformed into more precise specifications to enable tool support. As a result, the framework user can use both a cookbook-like informal documentation and supporting tools to specialize the framework. Copyright © 2006 John Wiley & Sons, Ltd.

KEY WORDS: framework; framework specialization; pattern; product-line architecture; variability management; variation point

INTRODUCTION

Object-oriented frameworks [1] are a popular way to implement product line architectures (PLAs) [2]. A framework provides the common core of a family of software products; a new product can be developed by extending the framework with product-specific code. Typical extension mechanisms are inheritance (in the case of white-box frameworks), implementation of interfaces and the creation and configuration of objects (in the case of black-box frameworks).
A central concern of PLAs is variability management [3]: how to capture and describe the required variation in the products based on a PLA, how to support the variation in the PLA design and how to help the product developer in exploiting the supported variation. In this context, a variation point [4] is a requirement imposing variation on the products developed using the PLA.

In the case of a framework, a hot spot [5] is a design solution supporting certain variation in the products. The specialization interface of the framework provides the means to make use of the hot spots and in that way to implement the supported variation. Ideally, the variation points are precisely defined in the specialization interface and linked to the corresponding hot spots of a framework. In practice, however, this is rarely the case. Instead, product-specific requirements and design direct the way in which the product developer uses different frameworks, derives new subclasses, overrides and implements operations and deploys and configures components. If the user does not know whether a framework supports a specific feature of the planned application or how this feature could be implemented, they attempt to study the framework. This slows down product development and is the main reason why framework specialization is so difficult and tedious.

Typically, especially in the case of white-box frameworks, the specialization interface becomes complex even for modest sized frameworks: a product developer must know and understand a substantial part of the framework to be able to extend it. In the absence of generally accepted techniques to describe the specialization interface, framework documentation at best usually tries to capture the specialization interface with informal instructions and examples. Such a description may help a product developer, if the product resembles the examples, but it fails to convey the full capabilities of the framework and it cannot be used as a basis of tool support for guiding and controlling product development.

In software engineering, particularly in the object-oriented community, patterns [6,7] have been used as a technique to document the best practices and experiences [8]. The best known examples of patterns are design patterns [9], which describe solutions to common object-oriented design problems. Design patterns have also been used to explain the architecture of a framework [10].

In our previous work [11,12] we have developed a pattern-based tool support for framework specialization. The tool allows the specification of the specialization interface as a set of formal pattern-like descriptions. During a product development session, the tool interprets such a specification and maintains a task list for carrying out individual programming tasks. In this way the tool guides the product developer through the specialization process and checks that the rules imposed by the framework are followed.

However, even if an advanced pattern specification technique and a supporting tool environment exist, the problem of mining the specialization interface of a given framework remains. In this paper we further develop a systematic goal-oriented approach [13] to find, specify and use the specialization interface of a framework as a set of patterns. While these patterns are used to document the hot spots informally, they provide a specification for more formal pattern descriptions, allowing tool-assisted specialization of a framework [11,12].

Our approach is goal-oriented in the sense that it is based on the analysis of the expected behaviour of the framework user, assuming that the user tries to specialize the framework by setting meaningful goals in the context of their application. An advantage of the goal-oriented approach is that the description of the specialization interface is directly linked to the assumed goals of the product developer. In this paper, the process from goals and informal pattern documentation to tool-supported use of patterns is evaluated with an experiment concerning an industrial framework.
SPECIALIZATION PATTERNS

When using a framework, the structure of the product-specific code and its relationships to the framework code follow the same basic form in different products, although the details (such as method bodies) can vary considerably. There is thus a preferred (or even imposed) solution for every design problem faced by the product developer: indeed, the whole idea of a framework is to relieve the product developer of the burden of finding new design solutions. This is a natural situation to apply the idea of patterns: there is a design problem encountered frequently by software developers and there is a known solution that should be used to solve the problem, exploiting the variation supported by the framework. We will call the representation of such a solution a specialization pattern. We use this term both for an informal and for a formal representation of such a solution, the latter originating from [12].

As opposed to design patterns, specialization patterns are system-specific: they describe a known solution to specialize a particular framework. Some of the classes appearing in a pattern description are actual framework classes, while others represent product-specific classes that vary in different instances of the pattern. However, as design patterns are often applied in framework design, a specialization pattern may represent an application of a known design pattern.

We use a format for a specialization pattern adapted from that used for design patterns (e.g. [9]).

- **Name.** Identifying name, used to communicate a particular part of the specialization interface among product developers and framework architects.
- **Goal.** Description of the specialization goal taken care of by the pattern.
- **Binding time.** The time the variation is fixed (e.g. development time, initialization time, or runtime).
- **Structure.** A model of the structural relationships of the various elements playing some role in the specialization pattern. This can be given, for example, as a Unified Modelling Language (UML) class diagram.
- **Participants.** Explanations of the roles of the various elements in the Structure description.
- **Constraints.** Additional constraints on the participants.
- **Applied design patterns.** The general design pattern(s) applied, if any.
- **Forces.** The facts that shape the solution towards a particular form.
- **Examples.** Example specializations with concrete product-specific classes and method bodies.
- **Related patterns.** Possible other specialization patterns related to the pattern in question.

A pattern language [6,7] is an organized collection of patterns that can be used systematically to provide solutions in a certain problem domain. For a given framework, our aim is to develop a set of specialization patterns that constitute a pattern language for using the framework. A concrete example of specialization patterns is given in the next section.

GOAL-ORIENTED APPROACH FOR MINING SPECIALIZATION PATTERNS

The process used in our goal-oriented approach is outlined as an UML activity diagram in Figure 1. The approach assumes that the framework user is directed by a set of goals. After finding the goals, the pattern writer can write the specialization patterns that will help the framework user to achieve the goals. By formalizing the specialization patterns, a tool-supported pattern interface to specialize the framework is obtained. The process is further explained in the corresponding subsections.
An example framework

Figure 2 presents a toy framework for drawing applications. This example framework is used later in this section to illustrate specialization patterns and the goal-oriented approach. The framework (written in Java) has only three classes (shaded): MainFrame, Figure and FigureManager. The MainFrame class implements a window for drawings; it contains a canvas and user interface components to select available figure types and to draw them to the canvas. By itself, the framework does not provide any figure types that could be drawn. Instead, each application must provide its own figures by subclassing the Figure base class and by registering these new figure types to the framework. In Figure 2, the product developer has derived new figure classes MyCircle and MyRectangle. These new figure types are registered with the application-specific manager class MyManager. The Test class provides the main method to start the application.

Finding specialization goals

The idea of finding goals and using them to predict the use of complex systems is not a new idea. For example, in cognitive psychology, the basic argument is that human cognition is always purposeful, directed to achieving goals and to removing obstacles to those goals [14]. Goal-driven approaches
have been used, for example, in requirements engineering to find and rationalize system functionality [15–18]. However, the goals discussed in this paper are of different character: they are related to the use of a particular framework. These goals are relevant when a product developer tries to satisfy product requirements based on the facilities provided by a framework.

In general, there are three essential features of problem solving [14]:

(i) the behaviour is organized toward a goal;
(ii) the original goal is decomposed into subgoals;
(iii) the solution of the overall problem is a sequence of actions (operators) that will transform the problem state into another problem state.

Typically, a goal is set to solve an encountered problem, so that the hierarchy of problems and subproblems reflect the hierarchy of the goals and subgoals. Thus a goal is a state that the product developer wants to achieve via the subgoals, while the problem and its subproblems can be seen as obstacles that hinder the achievement of the goal.

In requirements engineering, the main sources for identifying goals are considered to be scenarios, use cases, interviews, mission and policy statements, corporate goals, etc. [15]. The observed goals can then be refined into subgoals by asking how these goals should be achieved, while super goals are found by asking why a certain goal is sought [18]. Obviously, similar mechanisms can be used to find framework-specific specialization goals, too. Example specializations can be seen as scenarios and use cases of the framework. Interviewing framework users can clarify why and how the framework is going
to be specialized. Technical documentation and user guides give insight into how the developers of the framework expect their product to be used.

**System of goals**

In this paper, the set of goals and subgoals is called a system of goals. In the case of object-oriented frameworks, these goals must be achieved in order to implement the specialization. Since the system of goals comes from the problem solving process of the product developer, it is subjective and inherent to the developer’s mind. Figure 3 shows the goals to specialize the example framework. In this simple case, the main goal of deriving an application is divided into two subgoals, one for creating new figure types and the other for initializing the application. Here the operators to achieve these goals are typical programming activities, such as creating a class and implementing an operation.

As the system of goals is subjective, the search for the most suitable set of goals can be difficult. This is true also in the case of low-level specialization goals. In addition, the system of goals may evolve during the lifetime of the framework, even though the framework itself remained intact: new problems may arise during product projects, forcing the product developer to invent new ways to use the framework. This reflects the fact that the specialization interface of a framework is seldom uniquely determined by the framework structure but is to some extent a matter of choice. Indeed, there could even be several different systems of goals for the same framework: one for, say, novice developers and another for more experienced developers.

The person searching the framework-specific goals is not necessarily the architect of the framework but typically a person who has good understanding of frameworks in general and has access to (or previous knowledge of) the documentation and existing applications of the framework. The resulting system of goals is a map that outlines the required specialization patterns and their purpose but does not yet describe detailed solutions.

**Writing specialization patterns**

In general, pattern writing is a creative process that requires a lot of skills and experience (e.g. [19,20]). This holds for specialization patterns as well. The pattern writer must decide how the system of goals
is mapped to a pattern language. It is up to the pattern writer to decide which goals are patternized and how detailed the pattern language will be. For example, it may be beneficial to group tightly related goals into a single pattern or to provide alternative patterns for the same goal. In any case, a goal (or goals) becomes the goal part of the corresponding specialization pattern.

One method to come up with the solution part of a goal-oriented specialization pattern is to study an example solution that achieves the goal or goals. The example solution helps the pattern writer to identify the required program elements and their interactions. This process is similar to example-driven object-oriented analysis: essential elements of an example solution are identified and generalized. In the case of a specialization pattern, these elements, typically classes, methods and attributes, become a part of the structural description of the pattern. Direct structural constraints concerning these elements (such as containment relationships, inheritance relationships, etc.) can be usually presented in the structural description (UML class diagram) as well. However, the solution often includes more implicit requirements for the specialization code, such as mandatory method calls, which may not be presented in the structural description. Such requirements are listed in the constraints part of the pattern.

The binding time of the variation is essential information helping the product developer to interpret the structural description. Therefore, it should be clearly marked in the specialization pattern:

(i) if the variants are given as subclasses that are directly linked into the product, the binding time is development time, which is the normal case in white-box frameworks;
(ii) if the used variants are given by classes that are loaded during initialization, the binding time is initialization time;
(iii) if the used variants are selected by the end-user, the binding time is run time.

Table 1 shows a specialization pattern, which guides the product developer to create new figure types following the mechanisms provided by the example framework.

The Figure pattern in Table 1 was found after finding the goals (Figure 3) and creating an example specialization (Figure 2). The pattern presents a solution to achieve the goal ‘create new figure types’. Another specialization pattern was created for the ‘initialize the application’ goal.

Formalizing specialization patterns

The value of an informal specialization pattern stems from its ability to communicate the essential information about a part of the specialization interface in a readable form. While a formal specification would present this information precisely and unambiguously, we do not expect that an average product developer would study such specifications. However, a formal specification of a specialization pattern is required to provide tool support for applying the pattern, in particular to assist the product developer in following the often complex and implicit specialization rules of the framework, and to generate specialization code automatically if possible. The use of formal pattern specifications could be compared to the use of a specialized programming language, in which the ordinary product developer is supported by a set of tools that check the correctness of programs and generates code for legal programs.

In this paper we do not discuss formal approaches to pattern specification in detail. However, to facilitate the discussion we will briefly outline role-based pattern specification along the lines of [11]. We assume this kind of formalization of specialization patterns throughout the paper. Other approaches are introduced, for example, in [21] and [22].
Table I. Specialization pattern to create new figure types.

<table>
<thead>
<tr>
<th>Name</th>
<th>Figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>Add new figure types to the system</td>
</tr>
<tr>
<td>Binding time</td>
<td>Development time</td>
</tr>
</tbody>
</table>

**Structure**

```
@Figure
+clone(): Figure
+draw(g:Graphics, x:int, y:int)
+getName(): String
```

```
@FigureManager
+figures: Figure[]
+addFigure(f: Figure)
+getFigures(): Figure[]
+initFigures()
```

```
@MyFigure
+clone(): Figure
+draw(g:Graphics, x:int, y:int)
+getName(): String
```

```
@MyManager
+initFigures()
```

**Participants**

* Figure: the base class for figure classes
* FigureManager: the base class for the product-specific manager class
* MyFigure: a new figure class
* draw: used to draw the figure
* clone: creates a copy of the figure object
* getName: used to get the name of the figure type
* MyManager: a new manager class
* initFigures: used to register figure prototypes

**Constraints**

* initFigures: must call addFigure to register a prototype of each figure class

**Applied design patterns**

* Prototype

**Forces**

* Each product requires a manager class for the figure types
* A figure type is implemented by creating a corresponding subclass
* Each figure type must be registered within the initFigures method

**Examples**

Java code samples...

**Related patterns**

Use the Initialize Application pattern to test the figure types

---

**Roles, constraints and dependencies**

The basic building blocks of a (specialization) pattern are called *roles*, representing participants in the collaboration of software elements described by the pattern. Each role can be *bound* to one or more concrete software elements, meaning that those elements play the role in the collaboration. The *multiplicity* of a role defines the minimum and maximum number of elements bound to the role. A role has a *type*, determining the kind of software elements that can be bound to the role. For example, a type can be a (e.g. Java) class, a method, an attribute, etc.
Roles are organized hierarchically so that each role may have a number of child roles. In any binding situation, the role hierarchy must correspond to the element hierarchy: if a role is bound to an element, its child role must be bound to a child of the element.

Each role may have a set of constraints. Constraints can be used to enforce naming conventions, required types, exceptions, method parameters, structural relationships and so on. For instance, in object-oriented languages, concepts such as inheritance, overriding and return type can be seen as constraints set to elements bound to certain roles. In any legal instance of a pattern, all the constraints must be satisfied by the elements bound to the roles of the pattern. The type of a role is an implicit constraint limiting the kind of an element bound to the role.

If the constraints of a role cannot be evaluated unless another role is bound as well, the former role depends on the latter. For example, an inheritance constraint of a role may require that the class bound to the role must inherit the class bound to another role; in this case the former role depends on the latter. The dependencies between roles set up the order in which they can be bound.

Role diagrams

We have found it useful to construct a formal tool-independent role-based model of a specialization pattern as a UML class diagram where every role is represented by a class, showing the role hierarchy with nested classes and the dependencies between roles with explicit dependency arrows. We call this kind of representation a role diagram. We use stereotypes to mark the role types as well as the nature of the dependency between roles. The role's multiplicity is shown after the role name; by default the multiplicity is one. The multiplicity symbols used are the following: ‘?’ for [0..1], ‘*’ for [0..*] and ‘+’ for [1..*]. To make role diagrams more illustrative, the roles played by the framework elements are shaded, in contrast to the roles played by the application elements.

The role diagram of the specialization pattern in Table I is presented in Figure 4. Note that the constraint mentioned in the pattern description is represented by a code fragment role (for calling `addFigure`) under the method role `initFigures`; the code fragment role depends on a product-specific figure class that the call instantiates. When the user provides this figure class and binds it to the `MyFigure` role, the `addFigure` code fragment role can utilize this information (thanks to dependencies and bindings it has access to the correct figure class and the enclosing `initFigures` method) and adds a corresponding initialization call. Also note that it is possible to introduce an arbitrary number of figure classes (but at least one); each of these figure classes will have their own initialization calls in the method body.

In the case of the example framework, to construct the role diagram, the pattern writer analyses the specialization pattern that was shown in Table I and identifies the code elements that play a meaningful role in the solution, typically found from the list of participants. This includes both the existing base classes (`Figure` and `FigureManager`) and the specialization code that must be provided by the product developer when specializing the framework. The application-specific part must be abstracted to support a variety of possible solutions. The roles representing Java classes can be further divided into child roles representing methods and attributes. Method roles may in turn have child roles for parameters, return types, code fragments in the body, etc.

The structural relationships exposed by the UML class diagram in Table I must be converted into constraints of the roles in question. This concerns both explicit relationships such as inheritance and implicit relationships such as overriding relationships between methods. Additional requirements given
in the constraints part of the pattern must be formulated as role-specific constraints as well. Also, if several product-specific elements can play a role, the multiplicity of the role must be set accordingly. Similarly, if the solution does not absolutely require a player for a role, the minimum number of bindings is set to zero.

The main advantage of a role diagram is that it gives an ‘architecture’ for a pattern, showing the major components (roles) and their relationships. Without such a model it would be difficult to understand the pattern as a whole, especially in the case of a sizable pattern with tens of roles. By using a role diagram the dependencies can be analysed, possibly unnecessary dependencies can be removed and circular dependencies can be eliminated.
Finally, the role diagram must be transformed into a tool-dependent form. This may comprise the addition of tool-specific information, such as task prompts for binding a role or informal text explaining the meaning of a role, to be displayed by the tool when needed. By creating a set of formal pattern specifications the pattern writer establishes a programming environment for the framework. This environment provides a tool-supported pattern interface to use the specialization interface of the framework.

The principle of using a pattern interface is illustrated in Figure 5. A product developer specializes the framework by instantiating Figure and Initialize Application patterns with a pattern instantiation tool. The instantiation tool helps the user to create, select or modify the required elements to bind the roles. It also checks that the elements do not violate the pattern(s) they are involved in.

**TOOL: JAVA FRAMES**

JavaFrames‡ (formerly known as Fred) [11–13] is an environment providing tools to create and use precise pattern specifications. Currently JavaFrames has been integrated into the Eclipse§ environment and it supports pattern specifications to represent Java [23,24] and UML [25,26] solutions.

In JavaFrames, the pattern writer uses a semi-graphical pattern development tool to compose a new pattern specification. Based on a role diagram, the role structure of the pattern can be constructed.

---

‡http://practise.cs.tut.fi/

§http://www.eclipse.org/
in a straightforward way. The constraints and the role dependencies are then expressed using particular property fields of the roles. Similarly, additional information such as task prompts are added as properties of the roles.

An important feature of JavaFrames is the possibility to associate a role with a default element template. If a default element template exists for a role, the product developer can ask the tool to generate the element to be bound to the role. Since the default element template can refer to the information found in the elements already bound to other roles, the tool can perform powerful code generation based on the known relationships between the roles. In principle, it becomes possible to produce automatically any code that is already implied by earlier choices in the specialization process, taking the specialization rules of the framework into account. This mechanism is discussed in more detail in [11].

The product developer uses a special pattern deployment tool. In principle, the duty of the product developer is to bind the product-specific roles in a way that conforms to the pattern specification. The tool takes the pattern specifications as input and generates programming tasks as output to instantiate and bind the selected patterns. Depending on the current stage of the pattern instantiation, each task may require some actions from the product developer; they may create (using the default template mechanism) or select a new subclass to be bound to a role, fix constraint violations, etc. Besides the default element template, each task also provides adaptive documentation and implementation hints. The list of tasks changes dynamically: when a particular task is done, other tasks become available, as implied by the dependency (and containment) relationships between roles.

In the next section, to evaluate the goal-oriented approach, JavaFrames has been used to annotate a real industrial Java framework with a set of pattern specifications. The obtained goal-oriented pattern interface was then used to specialize the framework with the task-driven guidance of JavaFrames.

EXPERIMENT

The purpose of this experiment was to develop a tool-supported pattern interface for an industrial framework using the goal-oriented approach. Although not a full-scale case study, the experiment evaluates the usability of the obtained pattern interface by developing a non-trivial application based on the framework.

The framework: Nokia Graphical User Interface platform

Nokia produces a family of Network Management System (NMS) and Element Manager (EM) applications that are used to manage the network or network elements. The company has a Java Graphical User Interface (GUI) platform developed to support the implementation of the graphical user interface parts for the variants of this product family [27].

The platform is based on the Model View Controller (MVC) paradigm [28] and it has about 300 classes. MVC is an architectural pattern for interactive systems, dividing components into models, views and controllers. The fundamental principle of the platform is that every view object is managed by exactly one controller object and every controller (except the main controller) is managed by a parent controller. The platform is used as a Java framework to create suitable views and controllers. In addition, the framework provides some useful services for GUI applications, such as the clipboard and internationalization facilities.
Finding goals to use the framework

When starting the experiment, Hautamäki (the first author of this paper) was not familiar with the Nokia platform. The first task was to study the case framework and to find its specialization goals. The framework’s source code or any realistic use cases were not available. Fortunately, the framework was accompanied by good documentation that carefully explained the different methods of use; the goals to specialize the framework were found rather straightforwardly by reading this documentation. During this goal mining process, observations were made on how to use the framework in order to create applications. Finding the goals and learning the basics of the framework took about one week. An overview of the main specialization goals is shown in Figure 6.

In Figure 6, the goals are grouped into three subsets. First, in the Application goals subset, it is supposed that the framework user starts by providing a specific factory class to launch the application, provides a main controller that makes the application compatible with the framework system and implements the main window. Second, in the MVC goals subset, it is supposed that the framework user implements additional view and controller classes as described by the MVC paradigm. Finally, in the Service goals subset the framework user may utilize features like the internationalization service or the clipboard.
Table II. Specialization patterns for the Nokia framework.

<table>
<thead>
<tr>
<th>Application patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>* Application Factory. This pattern defines the constructional relationship between an application and the factory class used to create the application</td>
</tr>
<tr>
<td>* MVC Application. Each application created with the Application Factory pattern should be a standard MVC application in terms of the framework. The MVC Application pattern helps to implement such an application. This includes the creation of the application’s main controller and main view and the interactions between the UI and the main controller</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MVC patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>* MVC Subview. This pattern is used to create a controller and a view for a new subview (i.e. a new frame window). See Table III</td>
</tr>
<tr>
<td>* MVC Panel. This pattern is used to create a controller and a view for a new panel component. The panel is typically added inside a frame window</td>
</tr>
<tr>
<td>* MVC Dialog. This pattern is used to create a controller and a view for a new dialog window</td>
</tr>
<tr>
<td>* MVC Internal Frame. This pattern is used to create a controller and a view for a new internal frame. Such a frame can be opened in the desktop area of the parent window</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Service patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>* The case framework provides services and components that are useful when building GUI applications. Unlike with MVC, their use is typically splattered inside methods of the view and controller classes. Here the corresponding patterns are omitted but it is possible to construct a specialization pattern, e.g. to guide how to internationalize an application [13]</td>
</tr>
</tbody>
</table>

Writing specialization patterns for the framework

After finding the goals, the informal specialization patterns were derived. For each goal, the author studied how the goal was achieved by reading the documentation and by implementing example specializations. Patterns were drafted based on this information and experience. An initial pattern language was finished in about two weeks.

As with the goals in Figure 6, the obtained specialization patterns are divided into three categories, which are summarized in Table II. Application patterns are used to build up a basic MVC application with a main view and its controller. MVC patterns provide support for adding new views, such as dialogs and internal frames. Service patterns support miscellaneous services and features of the framework. A more detailed example of these specialization patterns is given in Table III. However, in this paper, to keep the pattern description short, only the class-level participants are discussed.

Finally, the informal specialization pattern language was transformed into JavaFrames patterns. Role diagrams were used as an intermediate formal representation of the patterns to design the role structure of each pattern. A semi-graphical pattern development tool was used to compose the required roles, constraints and their dependencies. Each role was also attached with a short description and a task title to be shown when the pattern is instantiated. The entire formalization phase took about one week.
Table III. MVC subview specialization pattern.

<table>
<thead>
<tr>
<th>Name</th>
<th>MVC subview</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>Create a new subview, i.e. a frame window that can be opened from the main</td>
</tr>
<tr>
<td></td>
<td>window or from another frame window</td>
</tr>
<tr>
<td>Binding time</td>
<td>Development time</td>
</tr>
<tr>
<td>Structure</td>
<td>SubView: the base class for a new frame window</td>
</tr>
<tr>
<td></td>
<td>SubController: the base class for a new frame window controller</td>
</tr>
<tr>
<td></td>
<td>MyView: a new frame window</td>
</tr>
<tr>
<td></td>
<td>MyController: a new frame window controller</td>
</tr>
<tr>
<td></td>
<td>MyParentController: a parent controller for MyController</td>
</tr>
<tr>
<td></td>
<td>MyControllerListener: a listener interface for MyController events</td>
</tr>
<tr>
<td></td>
<td>MyControllerEvent: an event class for MyController</td>
</tr>
<tr>
<td>Participants</td>
<td>* Each view must be created by a corresponding controller</td>
</tr>
<tr>
<td></td>
<td>* Each controller, except the main controller, is created by a suitable</td>
</tr>
<tr>
<td></td>
<td>parent controller</td>
</tr>
<tr>
<td></td>
<td>* To promote reusability, the subcontroller should not communicate with</td>
</tr>
<tr>
<td></td>
<td>its parent directly; instead, it should offer a listener interface and</td>
</tr>
<tr>
<td></td>
<td>event notifications</td>
</tr>
<tr>
<td>Applied design patterns</td>
<td>* Observer</td>
</tr>
<tr>
<td>Forces</td>
<td>Use the MVC Application specialization pattern to create the main</td>
</tr>
<tr>
<td></td>
<td>window. Use other MVC patterns to create internal frames, dialogs and panels</td>
</tr>
</tbody>
</table>

**Using the pattern interface to specialize the framework**

To demonstrate the use of the resulting tool-supported pattern interface, it was applied to reproduce the Bank example application discussed by Bonnet [27]. Although this application does not belong to the intended scope of the framework, it has been used as a business-neutral example that does not reveal sensitive material but nevertheless makes extensive use of the framework capabilities.
In the example application, a bank has a set of clients who are characterized by their name and password. An account has exactly one owner, which is the bank client. There can be withdrawal and deposit transactions; the account maintains a history of all the transactions made by its owner. The bank has an authentication system where several clients can be logged in concurrently. The bank also keeps a list of the current connections. When a client is successfully logged in, a session is created and their account is opened. During a session, a client can perform transactions and view their transaction history.

The user interface of the Bank application is shown in Figure 7. The main window shows the list of opened sessions. The client can open a session with the authentication dialog, which asks the client’s name and password. If the authentication is successful, a new session window is opened. In this session window, the client can use different views to perform deposit and withdrawal transactions. They can also view the history of their transactions.

Figure 8 shows schematically how the pattern interface was used to create the user interface of the Bank application. First, Application Factory and MVC Application patterns were instantiated to create the required factory class, the main window and controller of the Bank application. Then, the authentication dialog was implemented by instantiating MVC Dialog. The session window and its internal frames for performing transactions and to viewing history were implemented similarly.
Statistics

The final application has 24 classes and 551 code lines (Java source code lines inside the class declaration): they constitute a simple application model and its user interface. Augmenting Figure 8, Table IV shows how the classes of the Bank application are involved in different pattern instances. Each instance is indicated by a number (1 = Bank Application, 2 = Bank Main Window, 3 = Authentication View, 4 = Session Window, 5 = Deposit View, 6 = History View, 7 = Withdraw View). For simplicity, the table presents the pattern instances only in the level of classes, although the instances may also include methods, fields, constructors and code fragments.

As demonstrated in Table IV, a class can be involved in multiple pattern instances. Thus, patterns can be used to group elements in different logical entities and combinations. If an element is modified, the system can check that the element still obeys the pattern-specific rules. If not, a repair task can be generated. Furthermore, also other elements that play roles in the same pattern instance(s) can be checked, as a change in one element may require some modifications in other elements. For example, the SessionController class is a parent controller for the internal frames used in the Bank application. It is mainly created with the MVC Subview pattern but it is also involved as a parent controller when instantiating the MVC Internal Frame pattern to create the transaction and history windows. In this way the pattern interface helps to control implementation details that scatter across classes.

The strength of the obtained pattern interface was also estimated by considering the ratio of automatically generated code for different kinds of application classes. The framework user had to manually create or edit 293 lines while the pattern interface generated 258 lines, 47% of the code lines.
Table IV. The use of patterns across classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Application Factory</th>
<th>MVC Application</th>
<th>MVC Dialog</th>
<th>MVC Subview</th>
<th>MVC Internal Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AccountEvent</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AccountListener</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AuthenticationController</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AuthenticationControllerListener</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AuthenticationEvent</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AuthenticationView</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BankApplication</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>BankApplicationController</td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>BankApplicationFactory</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BankApplicationView</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DepositController</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DepositControllerEvent</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DepositControllerListener</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DepositView</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HistoryController</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HistoryView</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SessionController</td>
<td></td>
<td>4</td>
<td>5, 6, 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SessionView</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transaction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WithdrawController</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WithdrawControllerEvent</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WithdrawControllerListener</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WithdrawView</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Typically, code generation was used to produce a customized skeleton of a class or a method signature. It was also used to produce tailored source code fragments, such as method calls, inside a method body. To analyse the results, the classes were put into four categories according to the generated code ratio.

- **Weak generated code ratio (0–25%).** The lowest generated code ratio, 0% for the Account, AccountEvent, AccountListener and Transaction classes, is due to the pattern interface having no patterns to implement the application model of the banking system; they are not in the scope of the pattern interface. The BankApplicationView class (24%) and the SessionView class (16%), in turn, utilize the pattern interface but they also contain a lot of UI components and event listeners that cannot be predicted in the pattern interface. Using the goal-oriented pattern interface to support the usage of individual components, such as buttons and scroll bars, can be tedious. In addition, there exist Rapid Application Development (RAD) tools that can be used to compose the user interface with these components. However, if the usage of the components is part of a more complex and predictable framework specialization, a specialization pattern could be used at some point to teach and guide the framework user.
• **Fair generated code ratio (26–50%).** Similarly, the AuthenticationEvent (50%), DepositView (40%), HistoryView (47%), SessionController (40%) and WithdrawView (40%) classes must provide some implementations that could not be predicted by the pattern interface. Typically, these are user interface components and their event listeners. What could be done is to create patterns for narrower domains. The tradeoff is the complexity and general applicability of the pattern specifications. However, the pattern interface significantly speeded up the implementation by supporting the specialization rules of the underlying MVC framework.

• **Good generated code ratio (51–75%).** Over a half of the source code of the AuthenticationView (75%), BankApplicationController (72%) and HistoryController (75%) classes was generated automatically. Again, there were some issues that could not be predicted by the pattern interface. For example, the BankApplicationController class has to implement logic to add and remove bank clients. The HistoryController class needs a method to update the history view. The AuthenticationView utilized a special dialog base class provided by the framework to ask the user for their name and password. Clearly, the current version of the pattern interface is more suitable to express structural solutions, such as how to create a skeleton for a new MVC application, than small, unpredictable behavioural aspects. From the standpoint of the pattern interface, the application-specific logic or the place where this kind of behavioural aspect is created cannot be specified beforehand, unless the application domain is very limited.

• **Excellent generated code ratio (76–100%).** The AuthenticationController (94%), AuthenticationControllerListener (100%), BankApplication (100%), BankApplicationFactory (100%), DepositController (81%), DepositControllerEvent (100%), DepositControllerListener (100%), WithdrawController (81%), WithdrawControllerEvent (100%) and WithdrawControllerListener (100%) classes have the highest generated code ratio. The structure of the controller classes and their event classes is very stable from application to application. Although there are some variations between different applications, the implementation of these classes is rather predictable. This makes it easier to the pattern writer to provide pattern specifications that can almost automatically create the required code elements.

**Summary of the experiment**

The constructed specialization patterns provide a new kind of architectural view on the framework, particularly intended for the users of the framework. Being goal-oriented, the specialization patterns link high-level variation requirements ("how to achieve a new variant of . . .") to architecture and design level solutions. This makes it easier for the product developer to understand the framework architecture from the viewpoint of the specialization. Tool-supported specialization patterns thus facilitate learning-by-doing, where the pattern interface can be seen as an interactive goal-oriented tutorial to use the framework. In fact, this benefit of specialization patterns was considered even more important than automated code generation by our industrial partner.

However, the analysis of the reasons for different code generation ratios in different class categories gave some confidence that the extracted specialization patterns succeeded in capturing a significant amount of the dependencies shaping the product-specific code. Particularly, it was noticed that if some
application class can be given in a fully arbitrary way, there is no place for really useful pattern support for that class. Also, the encountered specialization problem may be outside the scope of the pattern interface, making it impossible to support the required variation without extending the pattern interface (and the system of goals). On the other hand, if the variation becomes fully determined by earlier decisions, its implementation can be generated automatically, assuming that a specialization pattern faithfully captures the dependencies between the involved code elements.

Besides one-way code generation, the tool environment supports fragmented design solutions. While classes and other implementation elements can be involved in multiple pattern instances, the programming environment checks that the application code obeys the constraints given in the specialization patterns. This is useful even if automated code generation is not possible or if the application developer relies on manual editing instead of exploiting the default element templates given in the patterns.

Although it is hard to draw definite conclusions, the experiment was encouraging, showing that the goal-oriented approach was sufficiently powerful for extracting the specialization interface of an industrial framework. Of course, this experiment can be criticized on the basis that ordinary software developers creating real industrial applications have not used the created specialization patterns. Such an extensive case study would most probably point out some weaknesses and missing patterns in the used pattern interface and it would allow us to confirm the expected benefit of using the tool-supported pattern interface as an interactive tutorial of the framework.

RELATED WORK

Johnson [10] noticed the close relationship between patterns and frameworks when he proposed using patterns as an aid to document frameworks. Since then, patterns have been used for describing the rationale behind design decisions for a framework [29] and to provide higher-level descriptions of frameworks (e.g. [30]). Many authors (e.g. [22]) have also recognized the close relationship between patterns and the framework’s extension points or hot spots. Specialization patterns can be seen as a pragmatic approach to provide such documentation.

When compared to other goal-driven approaches [15–18], our ‘goals’ are rather low-level and implementation oriented. The process to create and use goal-oriented specialization patterns resembles the idea of cookbooks [28] in which instructions for practical implementation problems are presented as recipe-like descriptions. Another, similar approach is implementation case [31] that describes how functionality for an application in the framework domain can be implemented using the constructs offered by the framework. Besides acting as cookbook recipes, implementation cases can be used as test cases when developing a framework. They do not have any specific formalism; rather, they are a mixture of informal language, pseudo-code and illustrative diagrams.

One approach to represent variability is to express variation in terms of feature models [32,33]. In a feature model, each of the alternative design decisions is presented by a feature, so that selections of these features are reflected in the design and implementation of the software product [34–36]. This resembles the use of goals, where the goal is selected and pursued by the product developer. Similarly, the system of goals could be based on a feature model that describes the variability provided by a specific framework. As illustrated in this paper, specialization patterns could then be created to support the use of the underlying feature model. In fact, as discussed in [37], feature-based variation can be presented as pattern-like entities. Such a feature variation pattern collects together the cross-cutting elements required to realize the feature. Ranging from requirements descriptions to actual
implementation, these artifacts may be created in different phases of the software development process, expressed using different notations and manipulated by different tools.

Besides the goal-oriented approach, experiments with JavaFrames have produced two slightly different approaches to annotate software systems and frameworks with pattern specifications. The structure-based approach [11,23,24] allows the automated support to create pattern specifications, as some heuristics can be used to identify and specify the usage of a framework. In the structure-based approach, the pattern modeller gives relevant parts of the framework and example applications as input and the heuristics are used to generate skeletons of pattern specifications as output. The pattern modeller can then complete these initial versions to get a complete pattern interface for the framework. Thus, the structure-based approach is more suitable for automatic pattern construction and framework documentation, depending on the heuristics used and upon the condition that the target framework has well-defined layered structure. The goal-oriented approach, in turn, is applicable also in situations where the specialization interface is not clear, example specializations do not exist or the pattern writer is not thoroughly familiar with the framework.

The concern-oriented approach [25,26], in turn, is based on the idea that with a proper tool support, formal pattern specifications can be used to make explicit the ways in which different concerns [38] can cooperate or interact with each other at the levels of design and implementation. The goal-oriented approach resembles the concern-oriented approach. However, the concern-oriented approach is intended to describe solutions for rather high-level and general problems. On the other hand, goals in our goal-oriented approach can be seen as a more concrete and atomic issues or subgoals to carry off the concern.

CONCLUSIONS

We have presented a goal-oriented approach to find, specify and use the specialization interface of a framework. The goals in our approach can be seen as implementation-oriented low-level subgoals to fulfill the requirements and design of an application with a specific object-oriented framework. In this way, the goals actually come close to the variation points in the requirements of the framework and the underlying product line architecture: each variation point gives rise to a goal and a corresponding specialization pattern where the use of the variation is documented. Furthermore, these patterns can be transformed into formal specifications and eventually into tool-supported form, which enables the systematic creation of a framework-specific programming environment. An attractive topic for future research is to study to what extent it is possible to systematically derive a tool-supported specialization environment starting from a precisely defined mapping of variability requirements onto design solutions.

REFERENCES


