Generating Pattern-based Documentation for Application Frameworks

Markku Hakala, Juha Hautamäki, Kai Koskimies and Pekka Savolainen

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

Application frameworks are a popular technique to implement product-line architectures. The problem of communicating the relevant properties of a framework for application developers is studied. It is argued that a conventional API specification is not sufficient for a framework, but a pattern-based specification of the extension interface is required. A technique to generate a pattern-based browser for the extension interface of a framework is described, relying on an existing tool developed for the generation of a programming environment for a framework.

1 Introduction

A central problem in software development is the understanding of existing systems. This is particularly important in the case of software product-lines ([Bos00], [ClN02]). A product-line is comprised of a set of reusable software assets that form the platform for a family of applications sharing similar structure and functionality. Such a platform is built around the common architecture of the application family, called the product-line architecture. The product-line approach is becoming increasingly popular in software industry as a systematic technique to achieve high level of reuse. In the product-line approach, you do not reuse only a collection of common services but the whole design of the applications as well. Today product-lines are the most important technological means for enterprises to gain shortened time-to-market, better quality of software products, and reduced costs.

In the case of product-lines, the underlying architecture and its implementation serve as an implementation platform for applications. In this type of application development, the central problem is how to map the requirements of the application to the concepts provided by the platform, rather than to the mechanisms of a general-purpose programming language. For example, the Enterprise JavaBeans architecture is based on the concepts of entity beans and session beans, and the central problem in EJB based application development is to map the appropriate parts of the domain model to these platform concepts. Hence it is of utmost importance for an application developer to understand thoroughly the platform architecture - in the same sense as it is important to understand the structures of the used programming language in conventional programming. It is indeed justified to use the term *architecture-oriented programming* in the context of product-lines.

How to make the architecture and the conventions of the platform understandable for an application programmer, then? The traditional way is to document the application programming interface (API) of the platform in terms of interfaces and informal explanations, possibly supported by examples of the usage of the platform and hyperlinks helping to access related concepts. For example, JavaDoc can generate such an API documentation automatically directly on the basis of the source code, provided that certain commenting

conventions have been followed. Indeed, the idea of automatically generated documentation is very desirable especially in the case of product-lines, because this guarantees that the documentation can be kept up to date. However, to understand the architecture of the platform and the principles of how the platform is supposed to be extended just by browsing through the API documentation becomes very difficult, if not impossible.

Frameworks have become a popular technique to implement product-line architectures within the object-oriented paradigm [FSJ00]. A framework is a collection of classes constituting the common structure and functionality of a family of OO systems. A framework differs from a conventional class library in that a framework is reused as a whole, rather than just using some of its services. A framework constitutes a skeleton for the applications, with holes for application-specific code that is called under the control of the framework using callback mechanisms. This is sometimes called the Hollywood principle: "don't call us, we'll call you".

Typically, a framework is extended by providing application-specific subclasses (implementations) for some of the framework's classes (interfaces), thus allowing application-specific code to be called by the framework. Hence such base classes in the framework can be regarded as a "specialization interface" of the framework. One could therefore expect that documenting this interface in an API-like fashion (for example, using JavaDoc or equivalent facilities) would be sufficient for the application developer to understand the relevant parts of the platform. Unfortunately, this is far from sufficient. The reason is that single extension points (like subclasses) are usually not independent of each other. Typically, the application developer has logical extension tasks that concern several extension points of the framework, and they have to be given in a consistent manner. As a concrete example, each introduction of an application-specific subclass for a framework class requires code in some other application class for instantiating that subclass. Hence these two extension points depend strongly on each other. Without understanding the relationships of individual extension points an application developer cannot hope to understand the platform as an implementation paradigm.

In this paper we will discuss a more general notion of an interface, based on the concept of a pattern. We will call it simply a *pattern interface*. We will show how the specialization interface of a framework can be documented using pattern interfaces instead of traditional API-like service interfaces. We will also demonstrate how such documentation can be generated automatically on the basis of existing knowledge about the specialization interface and previous applications of the framework.

The idea of documenting framework architecture through its design pattern instances has been presented before [Joh92], even with tool support [MCK97]. In this paper, however, we focus on documenting the specialization interface of a framework, rather than its architecture in general. In many cases these views are close, because the design pattern instances often appear in the borderline between a framework and its application. Another difference is that we investigate how the documentation could be produced automatically, based on some specification of the specialization interface.

A motivation for this work was the need for one of our industrial partners to establish a web site for the open source software available for all the units of the company. This web site should not only offer downloading, but also a systematic documentation of the available software systems so that the users can quickly obtain an understanding of the usage of the system. In the case of a software component this can be done with a conventional API specification, but in the case of an open source framework a more elaborated approach is required. Our vision is that a person interested in the framework could browse the documentation of the specialization interface in an intuitive manner through the web, and that such a facility could be produced automatically for the framework, using information that has been gathered from a sample application of the framework.

This work is part of a project investigating tool support for Java frameworks ([Hak01a], [Hak01b]). In that project, a tool prototype called Fred has been developed for assisting the specialization process of a framework. Fred makes use of so-called specialization patterns as a basis to generate an architecture-driven programming environment for the framework. Here we will interpret Fred's specialization patterns as a realization of pattern interfaces, and demonstrate how the automatic generation of pattern interface documentation can be accomplished using Fred. An advantage of Fred is that a sample specialization can be easily extracted from Fred after an application developer has accomplished her task. Otherwise the basic ideas presented in this paper are not dependent on Fred.

We proceed as follows. In the next section we will discuss the pattern interface concept and show how it can be used to specify the specialization interface of a framework. In Section 3 we will briefly introduce the Fred environment. In Section 4 we present a technique to generate documentation of the specialization interface of a framework based on the facilities provided by Fred. In Section 5 we present a practical example of the generated documentation for a small framework. Finally, we discuss some related work and present some concluding remarks.

The technique described in Section 4 is currently under implementation.

2 Patterns as interfaces

Basically, an interface can be regarded as a contract between two software systems. A system defines (or refers to) a contract that specifies the requirements the system expects from another system. Any system that fulfills the contract can play the role of the latter system. This principle, sometimes called design-by-contract [Mey88], has been well understood, but we argue that the conventional service-oriented interface concept can support this only in a limited way.

UML has introduced the notion of a collaboration as a "society of roles and other elements that work together to provide some co-operative behavior" [BRJ99]. This is intuitively close to a contract in the above sense. According to UML, a role is a slot that may hold an instance of a classifier (UML's term for a type-like entity). We make use of a similar collaborative interface concept that we call a *pattern interface*, or simply a *pattern*. A pattern is a collection of *roles* for various language structures (classes, methods, attributes etc.), and a set of *constraints* on the roles. Each role is a slot that can be *bound* to an actual instance of a program element in a system: a class role is bound to a particular class, a method role is bound to a particular method, an attribute role is bound to a particular attribute etc. The constraints can specify the cardinality of the source structures bound to a role, or the relationships between the source structures bound to different roles. Note that (the structural aspect of) a design pattern in the sense of [Gam95] can be represented as our pattern, but our pattern is a more general concept that does not take a stand on the purpose of the pattern.

As an example of a pattern, consider a framework containing a base class "SomeClass" that is assumed to be subclassed by the application. In the subclass, a particular method of the base class (say, "doSomething") is assumed to be overridden. Further, the subclass is assumed to be instantiated in the "main" method of the root class of the application, taking care of the initialization of the application. These are closely related tasks that have to be reflected in the specification of the specialization interface of the framework. We can specify this part of the interface as a pattern consisting of the following roles: base class, method (in base class), derived class, method (in derived class), root class, method (in root class), and creation code (in method of root class). These roles have several constraints concerning the actual source elements bound to the roles. For example, certain source elemenst are required to be located

within other elements, the derived class is required to inherit the base class, the method in the derived class is required to override the method in the base class etc.

In the context of frameworks, the strength of the pattern concept is that it allows various levels of binding. At the most abstract level, none of the roles in the pattern is bound to a concrete program element. This kind of pattern represents an abstract architectural solution without references to actual systems. For example, an abstract pattern might specify the idea of extending a framework by overwriting a method in an application-specific subclass, without referring to any particular framework. At the next level, a certain subset of the roles can be bound to concrete program elements. Such a pattern (instance) represents an interface a software system offers for other systems to hook up with it. In the case of a framework, a partially bound pattern specifies part of the specialization interface of the framework, fixing the roles that the elements of the framework play in the specialization but leaving the application-specific roles open. Finally, in some pattern (instance) all the roles of the pattern may be bound to actual program elements. In this case the pattern specifies either a fully implemented specialization, or an architectural solution occurring completely within the framework.

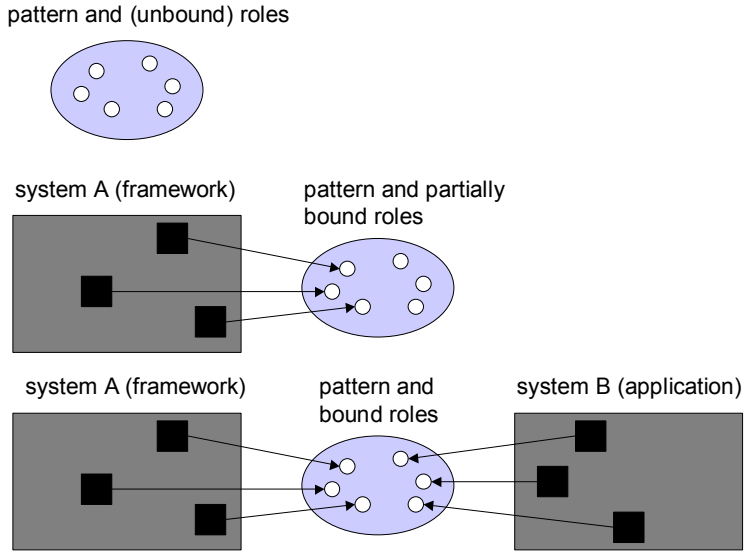

Figure 1. Pattern as an abstract architectural specification (top), as the specification of an open specialization interface (middle), and as the specification of a completed specialization (bottom).

An application developer wants to achieve certain specialization goals when extending a framework. A goal is typically a collection of functional properties which constitutes a *feature*, that is, a logical unit of behavior that can be observed in an application. A framework supports a set of features, but in a generic way: each feature supported by the framework can be realized in different forms, within the variance allowed by framework architecture. Ideally, a pattern specifies all the software elements that should be taken into account when extending a framework to realize a particular feature in an application-specific manner. Certain roles in the pattern are bound to framework elements which support all variants of this feature, and the roles representing implementations of certain variants. Typically all these program elements have strong relationships with each other, specified by the constraints of the pattern.

We have investigated several real frameworks when extracting their specialization interface for Fred [Hak01a, Hau02, Vil01]. Even in middle-sized frameworks consisting of several hundreds of classes the number of features appears to be relatively small: it is the variance within the features that gives the frameworks their expressive power rather than the number

of features. We have observed typically less than twenty features in such frameworks. This results in a relatively modest number of patterns describing the specialization interface for a framework. On the other hand, those patterns are fairly sizable, consisting of tens of roles. Nevertheless, documenting and explaining those patterns for the application developer is much more focused and efficient way of communicating the essential properties of the framework than, say, generating the interface specifications using JavaDoc.

3 Fred Environment

Fred (FRamework EDitor for Java) is a tool for generating an architecture-driven software development environment for a given framework. Fred allows precise specification of design abstractions, like (design) patterns, architectural and coding conventions, and framework extension points. Design abstractions are modelled as patterns, discussed in the previous chapters. Given the description of the specialization interface as patterns, Fred supports the framework specialization process by guiding the application developer through a task list based on these pattern definitions. Fred keeps track of the progress of the tasks, verifying that the requirements of the framework architecture are followed as required in the patterns. Fred is unique in that the accompanying textual documentation is automatically adapted to the application context (e.g. application-specific names and terms), and that the integrated Java editor is "architecture-sensitive": it immediately notifies the user of an architectural conflict. In a sense, Fred can be compared to language-sensitive editors, but in Fred the rules to be checked are those of the architecture, not those of the underlying programming language.

In Fred, the pattern concept is called a *specialization pattern*. This is essentially a concrete realization and extension of the pattern concept discussed in the previous section. As a pattern, a specialization pattern is given in terms of roles, to be played by (or bound to) structural elements of a program, such as classes or methods. The same role can be played by a varying number of program elements. This is indicated by the *multiplicity* of the role; it defines the minimum and maximum number of bindings that may be created for the role. A single program element can participate in multiple patterns.

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 role, there is a set of *properties* that can be associated with the role. For instance, for a class role there is a property "inheritance" specifying the required inheritance relationship of each class associated with that role. Properties like this, specifying requirements for the concrete program elements playing the role are called *constraints*. For example, a simple inheritance pattern might consist of roles Base and Derived, with a constraint stating that the class bound to Derived must inherit the class bound to Base. Another constraint might state that the program element bound to a particular role must contain the element bound to another role; we call this a containment constraint. It is the duty of the tool to keep track of broken constraints and instruct the user to correct the situation. Other properties affect code generation or user instructions; for instance, most role kinds support a property "default name" for specifying the (default) name of the program element used when the tool generates a default implementation for the element.

Roughly speaking, Fred generates a task for any role-element binding that can be created at that point, given the bindings made so far. A task prompting the creation of a binding is mandatory if the lower bound of the multiplicity of the corresponding role is 1, and there are no previous bindings for the role; otherwise the task is optional. Fred generates a task prompt also for an existing binding that has been broken (e.g., by editing actions).

The central part of the user interface of the Fred environment shows the current bindings of the roles for a selected pattern, structured according to the containment relationship of the roles. Since this relationship corresponds to the containment relationship of the program elements playing the roles, the given view looks very much like a conventional structural tree-view of a program. In this view, a red spot marks undone mandatory tasks, optional tasks are marked with a white spot. The actual to-do tasks are shown with respect to this view: for each bound role selected from the view, a separate task pane shows the tasks for binding the child roles, according to the containment relationship of the roles. The user interface of Fred is shown in Figure 2.

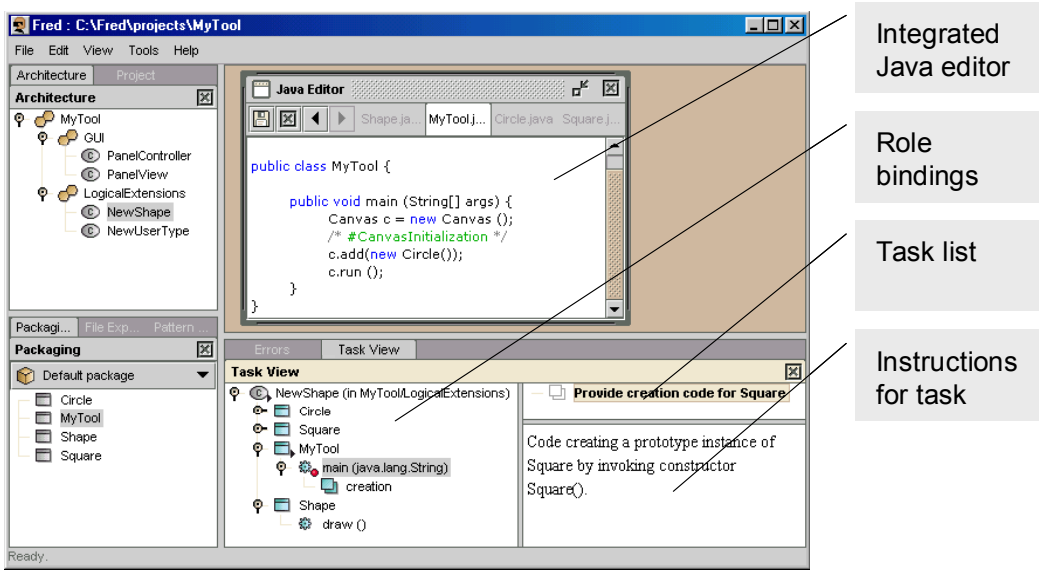

Figure 2. Fred user interface

The application is built following the tasks generated by the tool. The tasks can be carried out by indicating the existing program element that plays the role, by asking the system to generate a default form of for the bound element as specified by the pattern, or simply by typing the element using the Java editor and then binding it to the role. The system checks the bound element against the constraints of the pattern and generates remedial tasks if necessary. The task list evolves dynamically as new tasks become possible after completing others. The application programmer immediately sees the effect of the actions in the source code. Each individual task can be cancelled and redone. Hence the development process becomes highly interactive and incremental, giving the application programmer full control over the process.

An important feature of Fred is its support for adaptive user guidance: the task prompts and instructions are dynamically customized for the particular application at hand. This is achieved by giving generic task title and instruction templates in the pattern specifications, with parameters that are bound at pattern instantiation time. The actual parameters can be taken, for instance, from the names of the concrete program elements bound so far to the roles of the pattern.

Fred is freely available at **http://practise.cs.tut.fi/Fred**.

4 Generating pattern-oriented documentation with Fred

Normally, Fred has been used as a framework specialization tool. The framework developer describes the specialization interface of the framework in the form of patterns. These patterns include templates for code and documentation. They define a specialization path that results in a structurally correct framework specialization. When applied under the guidance of Fred, the patterns result in code and documentation for a specialized application. The patterns are usually applied by the application developer, but the framework developer itself can apply them as well to create simple example specializations to show the critical points of the framework. Most of the code and documentation of the examples can be generated based on the templates included in the patterns, which lessens the effort of creating various examples of the framework usage.

It is widely accepted that framework documentation benefits from concrete specialization examples. Hence, it would be beneficial to store the actions made during the specialization process and browse this information afterwards as part of framework documentation.

The pattern-based specialization process proceeds by carrying out the tasks generated by the tool, possibly typing in some code in between the tasks. Each executed task results in a binding between some piece of code and a role in the original pattern. What results from this process, is the source code for the specialization, partially generated, partially augmented by hand, and the bindings that describe the relationship of each code fragment to the specialization interface of the framework. Moreover, documentation of the application could have been produced based on templates just like the code. This corresponds to the situation introduced at the bottom of Figure 1.

The information that has been gathered during the specialization process effectively documents the resulting application as a specialization of the given framework. However, optimally we would also record the order the specialization steps are carried out. The specialization steps include the tasks generated by Fred, as well as any modifications made to the source code by hand. Recording all these actions and their ordering would document the whole stepwise specialization process, not just the end product. Thus, Fred environment would act as a specialization recorder.

The amount of information that needs to be recorded is relatively small, because pattern instantiation in Fred is a non-destructive process. The hand-made changes to the source code may indeed be destructive, but the amount of destructive hand-written code for an example specialization tends to be small. Thus, it is not necessary to store the complete snapshots of the system after each specialization step. It will suffice to store only the hand-made changes and the bindings that result from the specialization process.

The information gathered this way would allow replay of the whole specialization process. The player program would be able to show the execution of programming tasks, one by one. Associated with each task, the player could show the generated code, and how the developer modified it by hand to meet the needs of the particular example. The recorded textual tasks would give the rationale of each action just like they provided instructions for the developer when the specialization process was being recorded.

This form of framework documentation resembles the way the tutorials are conventionally written. New things are introduced gently one thing at a time in a practical order. This includes any exemplar code, which is typically provided in fragments instead of a revealing it all at once. Hence, using Fred to store the documentation process could be characterized as automatic generation of a tutorial for the framework. Using Fred for several specializations would result in tutorials for different kinds of example applications.

To support adoption of open-source frameworks, the documentation needs to be accessible on the web even prior to downloading the framework itself. This could be implemented using Java technologies. E.g., the documentation could be accessed by a Java applet contacting a Java servlet. The applet would thus provide the user interface for the documentation player application, whereas the servlet would have access to the specialization data exported from Fred.

The user interface of the possible player application is presented in Figure 3. It imitates the essential parts of the original Fred user interface, but allows only playback of the development process. Forward-button brings the user to the next specialization step. The selection is changed in the binding view to the binding that has been created in the step, or to the manual change that took place. Similarly, the description pane on the right is updated to show the documentation for the step, and the source code view highlights the piece of code that was generated or modified. Hence, it is possible to observe the specialization of an example application step by step. With each button click, the documentation grows incrementally, allowing the user to inspect all the previous steps. Similarly, the user is able to browse all the source code that has been produced at that point of specialization.

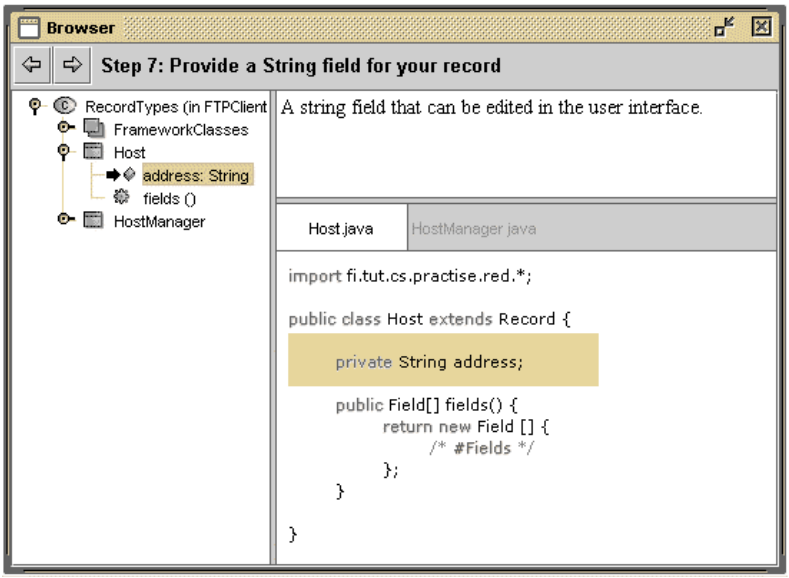

Figure 3. User interface of the player application

The steps are displayed as a tree that promotes the understanding of the recurring form that takes place in specialization of the particular framework. Our intention is that the user understands the framework specialization as an instantiation of a well-defined pattern, instead of an unrelated set service method invocations. The tool does not explicitly teach the abstract form that takes place between the framework-application-boundary, but builds the big picture by using examples. This makes use of the developer's inherent ability to abstract – a way that has proven to be an easiest way to adopt complex structures.

5 Example: a list-box framework

Framelets are small frameworks consisting of handful of classes and used as reusable, customisable building blocks for creating components [PrK99]. The Red framework is a simple framelet used to demonstrate Fred [Hak01c]. Here we will use this framework to illustrate our ideas. The Red framelet is also included as an example in the Fred release.

Red provides user interface facilities to maintain a list of Record-objects and to edit their fields. Typically, the Red framelet is used by deriving a new Record subclass with some application-specific fields. Once the application developer has created this new record type, the framelet provides facilities to automatically generate dialogs to set the values of the instantiated Record-objects. Figure 4 illustrates the use of Red; in the figure an application is started that uses the Red framelet and defines a new record type for ftp hosts. In the figure, the framelet has provided a dialog to update the fields of the selected host.

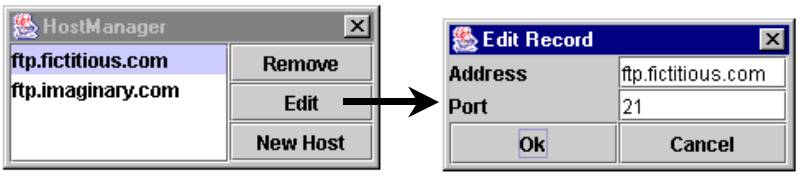

Figure 4. The Red framelet provides views to manage application-specific Record-objects.

The structure of the example specialization is shown in Figure 5. The most essential concept of the Red framework is Record. A record is an arbitrary class with some member variables exposed to the framework as editable fields. In the example application, new Host records are created by the HostManager class that implements the RecordFactory interface. Each record object must implement the fields() method to create an adapter object for each field that it wants to expose to the framework. Although not assumed by the framework, the adapters are most conveniently implemented as anonymous inner classes, declared directly in the fields() method.

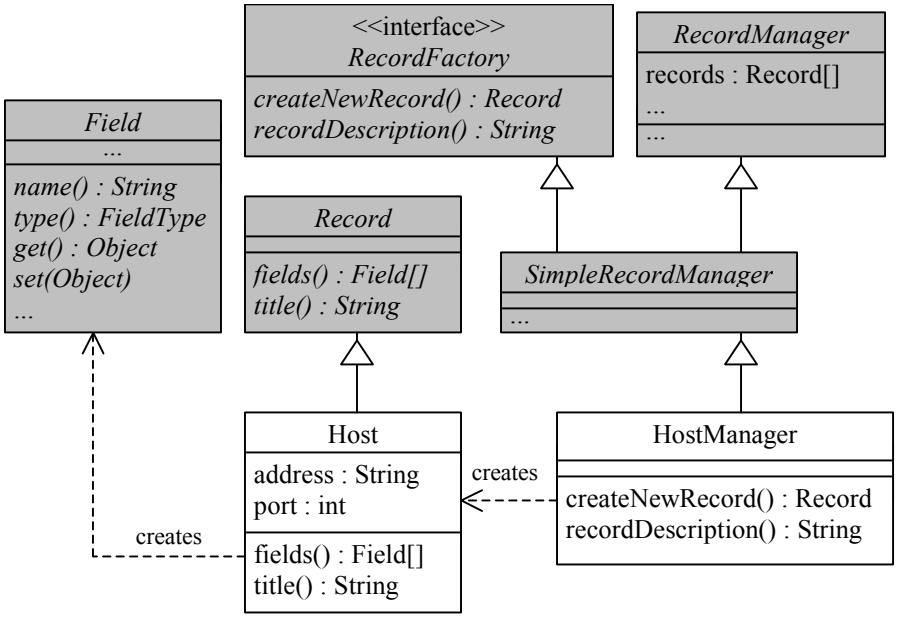

Figure 5. Example application is derived from Red

From the application developer's standpoint, one of the specialization problems is how to create a new record type that complies with the framework system. The framework expert, in turn, has identified this request as a specialization goal pursued by the application developer. Here it is assumed that the framework expert has used the Fred environment to create a specialization pattern called RecordTypes to guide the application developer through the specialization process. This pattern specifies the required subclasses and methods as discussed in Section 3.

Now, as discussed in the previous section, when the application developer uses the RecordTypes pattern to create his or her application, pattern-based steps that lead to the end product could be recorded. This information could then be browsed by another application developer as a concrete and illustrative specialization example. For instance, if someone wants to learn about field adapters of the Red framelet, he or she could use a dedicated tool to examine the created specialization. A screenshot of such browser tool was outlined in Figure 3.

The use of the player tool continues in Figure 6; the student has browsed one step forward in the recorded pattern to see how to create an adapter object for the address field illustrated in the previous step (recall Figure 3). On the left, a small black arrow indicates the associated Java element for each step while the role-specific documentation and source code are shown on the right. The documentation can contain links to more detailed Java documentation. It uses application-specific terms, like "address", which were obtained when the pattern was used and recorded. Also, the associated source code was given or generated during the original pattern usage.

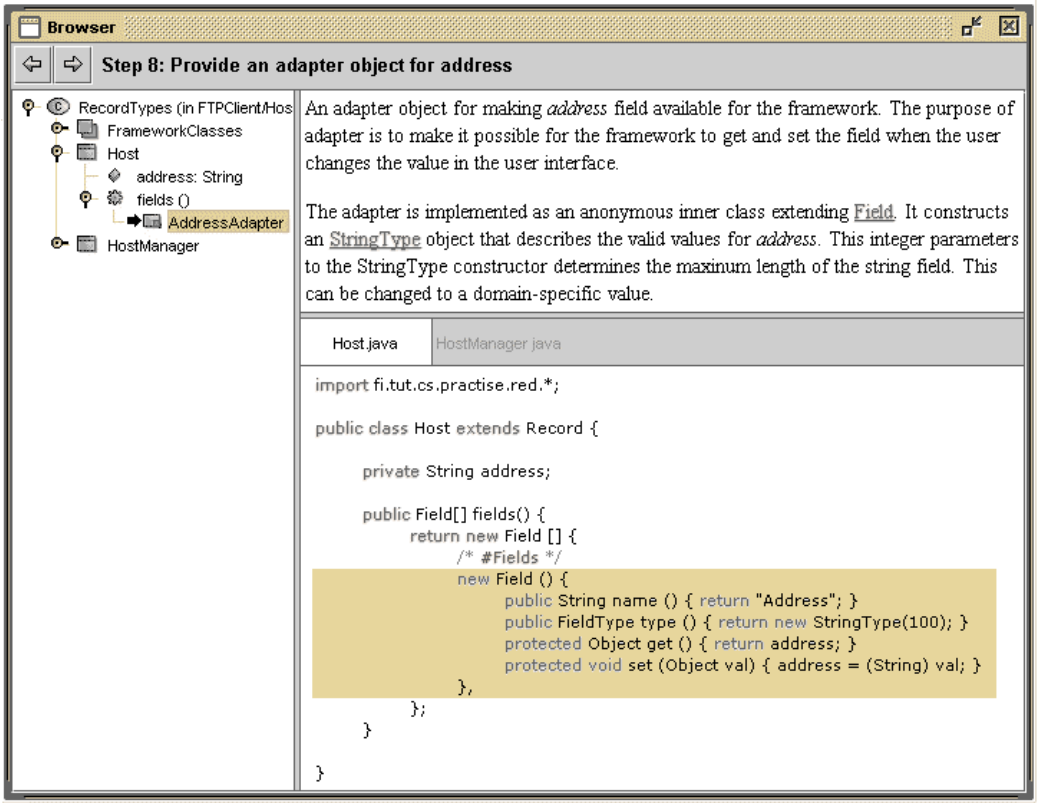

Figure 6. Illustrating that each field must have an adapter-object.

6 Related Work

We have discussed a technique to produce automatically the necessary programming interface documentation for application frameworks, based on the concept of a pattern. Our vision is to provide a facility which allows the potential user of an open source framework to browse through the essential parts of the system, imitating the actual use of the framework. This corresponds to a guided construction tour described by Kasper Østerbye et al. [Øst00]. When available on a web site offering the downloading of the framework as well, this kind of facility can significantly support the distribution and acceptance of open source systems.

Thomas Vestdam suggests a system using elucidative programming, a variation of literate programming to provide program tutorials [Ves02]. He describes a set of tools for creating and maintaining these tutorials. Created program tutorials can be viewed in a web browser and they provide hyperlink navigation between documentation, source code extracts and the actual program code. A special-purpose language called Source Code Extraction Language is used in describing source code fragments that should be included in the documentation. Whereas our approach provides dynamic visualization of actions recorded during the framework specialization process, Vestdam's system presents a technique to produce and maintain hyperlinked, static documentation.

7 Discussion

A possible weakness of our approach is that the pattern-based documentation easily becomes too implementation-oriented: the patterns relate the source structures nicely together, but they do not relate application requirements to code which would be more relevant for the application developer. To some extent this problem can be taken care of by associating patterns with textual explanations referring to the possible requirements of applications, but in general this approach gives no support for writing such explanations. We are currently investigating feature modeling techniques as a possible bridge between requirements and patterns, aiming at more systematic development of patterns from requirements.

Acknowledgements. This work is funded by the National Technology Agency of Finland (TEKES) and Nokia, Necsom, Sensor SC, and SysOpen.

References

[Bos00] Bosch J.: *Design and Use of Software Architectures - Adopting and evolving a product-line approach*. Addsion-Wesley 2000.

[BRJ99] Booch G., Rumbaugh J., Jacobsen I.: *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[ClN02] Clements P., Northrop L.: Software Product Lines - Practices and Patterns. Addison-Wesley 2002.

[FSJ00] Fayad M.E., Schmidt D.C., Johnson R.E.: *Building Application Frameworks — Object-Oriented Foundations of Framework Design*. John Wiley & Sons, 2000.

[Gam94] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley 1994.

[Hak01a] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Annotating Reusable Software Architectures with Specialization Patterns. In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), Amsterdam, August 2001, 171-180.*

[Hak01b] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Generating application development environments for Java frameworks. In: *Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering (GCSE'01), Erfurt, Germany, September 2001, Springer, LNCS2186, 163-176.*

[Hak01c] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Task-Driven Specialization Support for Object-Oriented Frameworks. *Tampere University of Technology, Software Systems Laboratory, Report 22, February 2001*.

[Hau02] Hautamäki J.: *Task-Driven Framework Specialization - Goal-Oriented Approach.* Licentiate thesis, Report A-2002-9, Department of Computer and Information Sciences, University of Tampere, 2002.

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

[MCK97] Meusel M., Czarnecki K., Köpf W.: A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and Hypertext. In: *Proc. of ECOOP '97, LNCS 1241, 496-510.*

[Mey88] Meyer B.: *Object-Oriented Software Construction*. Prentice-Hall 1988.

[Øst00] Østerbye K., Madsen O.L., Sandvad. E., Bjerring C., Kanmeyer O., Skov S.H., Hansen F.O.: Hansen F., *Documentation of Object-Oriented Systems and Frameworks*, COT/2-42-V2.4, Centre for Object Technology, Danmark, 2000.

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

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

[Ves02] Vestdam T.: Generating Consistent Program Tutorials, NWPER '02, Copenhagen, August 2002.

[Vil01] Viljamaa A.: *Pattern-Based Framework Annotation and Adaptation - A Systematic Approach*. Licentiate thesis, Report C-2001-52, Department of Computer Science, University of Helsinki, 2001.