Creating Framework Specialization Instructions for Tool Environments

Antti Viljamaa and Jukka Viljamaa
Department of Computer Science, University of Helsinki
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland
E-mail: {antti, jukka}.viljamaa@cs.helsinki.fi

Object-oriented application frameworks provide an established way of reusing the design and implementation of applications in a specific domain. Using a framework for creating applications is not a trivial task, however, and special tools are needed for supporting the process. Tool support, in turn, requires explicit annotations of the reuse interfaces of frameworks. Unfortunately these annotations typically become quite extensive and complex for non-trivial frameworks. In this paper we focus on describing techniques for minimizing the work needed for creating framework annotations. We discuss the possibility of generating annotations based on frameworks' and example applications' source code, automating annotation creation with dedicated wizards, and introducing coding conventions and advanced language features, such as inheritance, for framework annotations languages. We also introduce a programming environment that supports framework annotation and specialization. In our environment we have incorporated many of the techniques described in this paper.

1. Introduction

For a long time *reuse* has been acknowledged as a major factor in enhancing software production [1, 2]. At the moment *object-oriented application frameworks* [3, 4, 5] represent the state of the art in reuse and they are quickly becoming the most established reuse technique. A framework defines the main concepts of its application domain as abstract interfaces. It nails down the overall architecture of the applications derived from it by defining the relationships between the main concepts and the default functionality and algorithms associated with them.

A framework provides a *reuse interface* that enables an application developer to specialize the framework and call its services. The application developer specializes the framework by defining subclasses for the abstract framework classes (*white-box reuse*) or by combining and customizing the ready-made components provided by the framework (*black-box reuse*). The *hot spots* of the framework are an important part of the reuse interface, especially for white-box reuse [6]. They are the variation points where the application developer can plug in her application-specific code that can be called from within the framework, e.g., by using dynamic binding. In practice, this usually involves overriding the *abstract hook methods* defined in the framework with the *concrete hook methods* defined in the application.

In addition to reuse, proper tool support is also important in software development. A modern *integrated development environment* (IDE) provides a set of seamlessly incorporated tools for editing, compiling, and debugging. Advanced environments even support refactoring and various kinds of source code structure visualizations.

The support for utilizing frameworks in IDEs is still mostly limited to black-box reuse. There are graphical tools for composing a user interface from a set of customizable components, and there are also general-purpose composition tools for standard component architectures such as JavaBeans (see http://java.sun.com/products/javabeans/). These kinds of tools have already proven useful and they are becoming a standard part

of IDEs. The essence of these tools is that they provide an easy-to-use interface to a library of ready-made components.

Tool support for white-box reuse is just taking its first steps, however, even though most frameworks offer white-box reuse because it allows for more flexibility than mere black-box reuse. Obviously, defining subclasses and overriding methods is far more demanding than just instantiating and providing parameters for default components. It usually requires a thorough understanding of the framework's concepts and the dependencies between them. That is why we argue that there is a need for tools supporting the white-box reuse of frameworks.

Framework specialization is a process that is controlled on one hand by the restrictions posed by the framework's architecture and on the other hand by the application developer's demands. From the framework user's point of view there should be support for specializing frameworks by adding features to the produced application in a step-by-step manner. In our vision, a framework should be accompanied with a set of tools that guide the user through the specialization process and provide automatic code generation, dynamic and context-sensitive user documentation, as well as continuous validation of application-specific code against the requirements posed by the framework's architecture.

In this paper we discuss ways to use an *annotation language* for describing the framework's reuse interface in order to implement the framework specialization support functionality in a generic way for an arbitrary framework. First, in section 2 we introduce our model for specifying *role-based annotations* for supporting framework specialization. The rest of the paper is devoted to techniques that are needed to handle large and complex framework annotations. Section 3 gives general guidelines for writing effective annotations. In section 4 we discuss advanced annotation language features and tools that make framework annotation easier. Section 5 sketches a method for extracting parts of annotations automatically from source code. Our prototype framework engineering environment, in which we have implemented many of the techniques introduced in this paper, is briefly described in section 6. Section 7 concludes the paper with a discussion on the importance of the framework specialization assistance tools and requirements for them.

2. Role-Based Framework Specialization Instructions

The tools in a traditional IDE work with plain source code. Plain source code is not enough to support framework specialization, however. This is because framework reuse interfaces typically involve complex requirements and restrictions among multiple *program elements* (e.g. classes, methods, or data fields). Such relationships are difficult or even impossible to express using the current implementation languages[1]. Thus, we have two choices: we can either design a new language, or we can construct a tool that takes care of enforcing those restrictions that are not directly supported by the constructs of the framework's implementation language.

Even though a new language might be considered a more elegant solution to this problem, there are also quite a few advantages to the tool based solution that utilizes a separate annotation. First of all, when the annotation language is separate from the actual implementation language, it is possible to use an existing (popular) implementation lan-

[1] There are mechanisms that enable the prevention of some simple framework misuses. In Java, for example, it is possible to declare classes *final*, which can be used to some extent to distinguish the framework's *frozen spots* from the hot spots.

guage. It is also relatively easy to adjust the method to fit multiple programming languages. Furthermore, the tool itself can be integrated into an existing IDE so as to get full advantage of an established environment and the standard tools it provides.

Using a separate annotation makes it possible to annotate existing frameworks without modifying their implementations. It is also feasible to make multiple annotations for different audiences. For example, novice users might have a simplified and restricted view to a framework, whereas an annotation for experts might reveal more details and advanced features. Also, from a conceptual point of view, separate framework annotations provide a more abstract and high-level view over the framework source code than possible language extensions do.

Roles and Constraints

We argue that framework specialization instructions can most conveniently and intuitively be described in terms of role-based annotations. Similar views on the suitability of role-based models to describe reusable software systems can be found, for example, in Riehle's work on *role modeling* [7] and Gamma et al.'s work on *design patterns* [8]. Also van Gurp et al. emphasize the role-oriented nature of framework interfaces [9]. In the following we use our own syntax and semantics for a role-based framework annotation. However, we argue that similar basic principles are found in almost all role-based languages.

In our model, the annotations of framework reuse interfaces are specified using *specialization patterns*. A specialization pattern is a specification of a program structure, which can be instantiated in several contexts to get different concrete variants of the structure. A specialization pattern is given in terms of *roles* to be played by structural elements of a program. We call the commitment of a program element to play a particular role a *contract*. A role may stand for a single element or a set of elements. Thus, a role can have multiple contracts, and a program element can play many roles through a number of contracts. *Cardinality* of a role bounds the number of its contracts.

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

Unlike constraints, some properties are only meant to affect code generation or dynamic user guidance. For instance, most role kinds support a default name property for specifying the name of the program element used when, e.g., a tool generates a default implementation for the element. Those properties are called *templates*. They are used when textual information needs to be generated, but they are not checked afterwards.

A specialization pattern can be expressed as a *pattern diagram*. Figure 2.1 shows an example of a simplified role-based annotation for a small organization framework. The framework contains three classes[2] describing the workers and tools of an organization (the classes in the *fw* package). Each worker has a salary. A worker can be a tool expert, in which case she holds an association to a tool she masters. The salary of a tool expert depends on her basic salary (the value of the *salary* field) as well as on the complexity factor of the tool she knows. Applications are derived from the framework by defining

[2] The names of the abstract classes and methods are written in italics.

appropriate subclasses (the classes in the *mycompany* package in this example) for the framework classes.

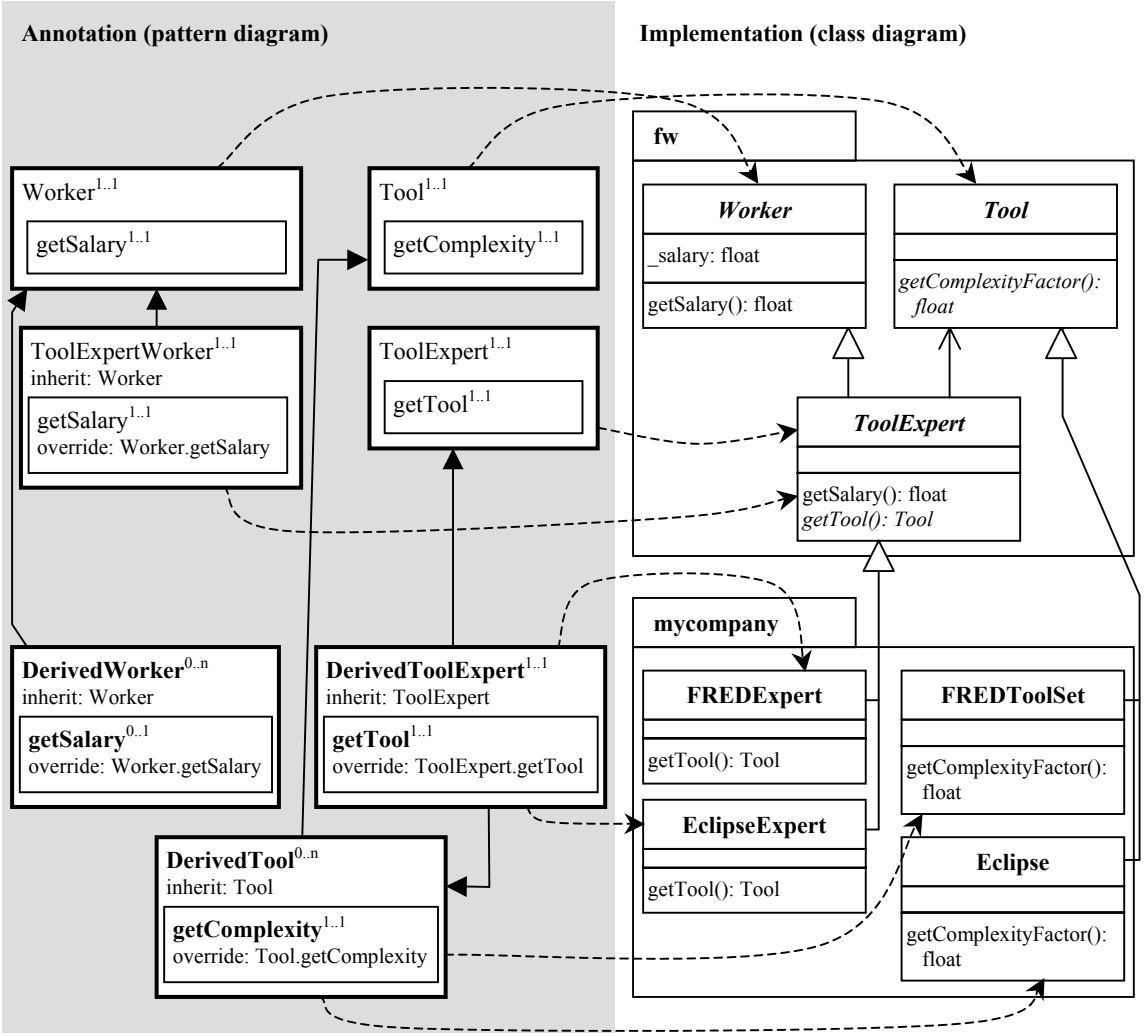


Figure 2.1: An example of a simplified role-based annotation for a framework

The example annotation consists of only one specialization pattern[3] that models the inheritance relationships between the framework classes and the application classes as well as overriding of some important hook methods (e.g. *Tool.getComplexityFactor* and *ToolExpert.getTool*). The rectangles in the pattern diagram represent roles. The class roles (e.g. *Worker*, *Tool*, and *DerivedWorker*) are denoted with thick borders and the method roles (*getSalary*, *getComplexity*, and *getTool*) with thin borders. The relative placement of roles within other roles reflects the declaration hierarchy of roles. Figure 2.1 shows, for instance, that a *getTool* role is declared within the *DerivedToolExpert* role, which means that also the methods playing that *getTool* role (e.g. *FREDExpert.getTool*) must be declared within a class that plays the *DerivedToolExpert* role.

Figure 2.1 differentiates the roles from their *instances* (i.e. the programming language elements playing the roles)[4]. The dashed arrows in the figure show the program elements that play each role. For example, the *ToolExpert* class plays two roles: *Tool-*

[3] In practice, there would be multiples specialization patterns in an annotation for any non-trivial framework, e.g., one for each hot spot.

[4] Note that in [8], for example, both pattern roles and their instances are described with similar class diagrams. This may cause confusion if the roles and their instances are depicted in the same diagram.

ExpertWorker and *ToolExpert*. The former role describes *ToolExpert* as a subclass of the *Worker* class and the latter role describes *ToolExpert* as a super class for the application classes playing the *DerivedToolExpert* role.

The names of the roles for the application classes and methods in figure 2.1 are written in bold to distinguish them from the *framework roles* that stand for the program elements defined in the framework. The *application roles*, on the other hand, describe the program elements, which the application developer must supply. The requirements for those program elements are specified with constraints associated with the corresponding roles.

The cardinality of a role is given as superscript after the name of the role. The cardinality can be either exactly one (*1..1*), from zero to one (*0..1*), from one to infinity (*1..n*), or from zero to infinity (*0..n*) with respect to the other roles the role depends on. The *dependencies* are denoted with arrows between roles (e.g. from *DerivedToolExpert* to *ToolExpert* and to *DerivedTool*)[5].

The main properties of a role can be optionally listed below the role name. In figure 2.1, there are *inheritance constraints* declared for the subclass roles (*ToolExpertWorker*, *DerivedWorker*, *DerivedTool*, and *DerivedToolExpert*) stating that the program elements playing the subclass roles must inherit the class playing the corresponding base class role. There are also *overriding constraints* in the method roles declared within subclass roles describing the overriding of the hook methods of the framework.

Note that most properties of the roles have been left out from the figure to keep it readable. For example, an important aspect of this framework is that each concrete tool expert (e.g. *EclipseExpert*) returns a reference to an appropriate concrete tool (e.g. *Eclipse*) as a result of its *getTool* method (see figure 6.1). The dependency arrow from *DerivedToolExpert* to *DerivedTool* is a hint of that. Field roles and *code snippet roles*[6] would be needed to fully describe that relationship.

Using Framework Annotations in Tools

A framework annotation language like the one introduced above can be used for various purposes. First of all, the whole framework specialization process can be seen as a sequence of steps in which the roles of the framework annotation are bound to implementation entities. The dependencies between the roles form a natural ordering for these binding steps; a role can be bound only if there already are program elements playing the roles the particular role depends on. We say that there is a *potential binding* for a role, if that role could be bound to a program element at a given moment. Furthermore, based on each potential role binding, we can generate a note that suggests the framework specializer to make the binding. This mechanism forms a basis to the implementation of a dynamic version of *framework cookbooks* [10] that have traditionally been used for framework documentation.

Typically the application roles depend on the framework roles, so the framework roles are bound first[7]. In our annotation example, there would first be potential bindings for the roles *Tool*, *Worker*, and *ToolExpert*. After binding the role *Worker*, there would be a potential binding for the remaining framework role *ToolExpertWorker*, too. For the

---

[5] Actually, there should be dependencies between method roles, too (e.g. from the roles describing the overriding methods to the roles of the methods to be overridden), but we have left them out from the figure for simplicity.

[6] Code snippets can be used to describe method bodies and field initialization clauses.

[7] This binding can be done by the framework developer, for instance. We call this process *framework initialization*.

application developer there will be potential bindings for *DerivedWorker* and *DerivedTool*. Potential bindings for *DerivedToolExpert* will be available once the application developer has first defined some concrete tool classes.

The making of role bindings could be automated to some extent. A framework specialization tool could actively look for program elements that conform to the constraints specified in roles. If such elements were found contracts for binding them to the appropriate roles could be made automatically. In practice, automatic role bindings must probably be limited to some subset of potential bindings. For example, let us assume that there is a class role for certain classes and a method role for the methods inside these classes. Once the class role is bound, the tool could automatically try to bind the method role to some methods inside the implementation class.

Framework roles typically have one-to-one relationship with the framework implementation entities, and it is known beforehand which roles should be bound to which program elements. Thus, automatic role binding would be especially useful when binding the framework roles during framework initialization.

Besides the order in which the role bindings are made, our annotation language also supports the validation of the program element playing a role. The validation is made against the constraints specified in the role. For example, we can check that an application class playing the role *DerivedToolExpert* actually inherits the framework class *ToolExpert*.

We mentioned earlier the possibility to generate notes for the framework specializer about the potential bindings that can be made. The possibility to add *templates* into the roles enhances this functionality. Templates are evaluated to text whenever necessary. Besides freeform text, templates may include references to other roles and to the program elements playing those roles. For example, in the *DerivedToolExpert* role we could have a template whose contents would be "*Provide '</DerivedTool>Expert' by inheriting '</ToolExpert>*"'. This template could be used to generate a note for the framework specializer whenever there exists a potential binding for the role *DerivedToolExpert* (i.e. after she has first identified the appropriate concrete tool).

The same template mechanism can also be utilized for generating code based on a role. The easiest approach is to support simple template-based code generation where the resulting code is not checked afterwards for constraint violations. This *one-way* code generation obviously has significant drawbacks; we cannot verify the validity of source code after the user modifies it nor can we say anything about the validity of any non-generated source code that is bound to some role.

We suggest the separation of code generation and constraint checking. Code generation can be conveniently implemented with a template-based mechanism, but we should also be able to separately verify the properties of program elements whether they were generated or not. Code generation templates must of course conform to the constraints as well, but in addition they typically include lots of example-like default values that can be altered by the user.

The constraints can be implemented as dynamically checked run-time invariants or as static source code validations. In the former approach it is possible to create more detailed constraints based on, for example, the dynamic types of variables. The drawback with the latter approach is that the constraints may only utilize static source code information. However, the feedback about possible constraint violations can be given to the framework specializer sooner than with dynamic constraints.

Static constraint checking can be implemented in a compiler-like manner, where the implementation source code is analyzed and all the constraints are validated once the user asks the tool to do so. Constraint checking can also be implemented in a more interactive way by parsing the source code incrementally as the user edits it and by (re)evaluating the constraints of a role every time the program elements bound to that role are changed.

The Problem of Creating Framework Annotations

In this section we have briefly introduced a role-based method for annotating the reuse interfaces of application frameworks. The description given here is only meant to provide the reader with some background knowledge and motivation for the following discussions. For a complete definition of our annotation language and its relation to other similar methods as well as for detailed large-scale examples of its usage refer to [11] and [12].

However, as can be seen even in the simplified annotation example above, framework annotations typically become quite extensive and complex. Based on our experiences the ratio of a framework annotation size to the framework size typically ranges from 15 % to 50 %. The ratio naturally depends on the level of annotation detail. In the worst case the annotation size can even exceed the size of the framework.

Our main focus in this paper is to represent techniques that can be used to support the creation of framework annotations. These techniques include introducing systematic coding instructions for framework annotations as well as code reuse possibilities into the annotation language (e.g. inheritance and composition of annotation structures). We also describe tools that enable the generation of roles and their properties based on other roles, specific (often recurring) role types, or source code.

3. Standards for Framework Specialization Instructions

Clearly, any non-trivial framework can be used in various ways and thus there are numerous options on how to annotate a framework. In practice, the framework annotator must decide what kind of assistance she wants to give for the framework users. Adding constraints will give the users better guidance, but at the same time they will loose some of their freedom. We argue that it is better to first provide annotations for quite a narrow framework reuse interface, and later modify the annotations and add new ones to enable more advanced ways of using the framework. This means that it is advisable to first make annotations for the most relevant framework hot spots.

Even after choosing the relevant hot spots there still remains a number of possibilities in making the actual annotations. However, we argue that is possible to introduce systematic coding conventions and good practices for annotating frameworks and that these *annotation standards* are crucial when developing the process of framework specialization annotation. By systemizing framework annotation we ease the overall burden of making large annotations and thus leverage the problems discussed in section 2.

Many *patterns* that are used to introduce flexibility in hot spots occur over and over again in different frameworks [8]. Consequently, there are also recurring schemas for framework specialization. We argue that these recurring schemas can be polished into a catalogue of best practices for framework annotation. In general, the annotations should guide the framework specialization process in a clear and intuitive manner. The quality of the annotations should be evaluated with empirical usability analyses.

Determining Framework's Hot Spots

In order to develop a framework annotation, we must have enough information about the framework's structure and its intended use. In an ideal situation the framework developer herself does the annotation, which makes the annotation process smooth and the results accurate. However, if the framework developer is not available for making the annotation, we must rely on interviewing current framework users, studying textual and graphical documents of the framework, and examining the framework source code and example applications derived from the framework.

During framework analysis we should first gain a basic understanding of the framework's hot spots. According to Pree, *template* and *hook methods* are obvious candidates when trying to locate the hot spots [6]. Demeyer claims that almost all hot spots can be found by analyzing overridden methods, because polymorphism needed in hook methods is usually implemented using method overriding [13].

There are usually hundreds of overriding relationships between methods in any non-trivial application framework. That is why we need an effective way to separate those important hook methods that actually constitute the framework's reuse interface from other overriding methods that are only used internally and are thus irrelevant to the framework user. For this purpose we propose locating the framework's *abstract concepts* that correspond to a set of selected abstract root classes of the inheritance trees in the framework's implementation. From these abstract classes we rule out framework's internal interfaces (as which we classify interfaces that don't have several concrete implementations) and interfaces that only define additional behavior that can be implemented in arbitrary classes (as which we classify interfaces that don't describe the main purpose of any class, e.g. listener interfaces). The methods that are left (optionally) overridable in the abstract concepts constitute the relevant hook methods of the hot spots in the framework.

Structuring Framework Annotations

Before actually implementing an annotation for a framework, the overall structure of the annotation must be planned. Instead of using a huge monolith annotation to model the framework's entire reuse interface, we suggest that separate partial annotations are defined at least for each individual hot spot. We argue that by splitting the reuse interface description we make framework annotation more manageable and framework adaptation more intuitive. It is far easier to develop annotations as smaller entities, because we typically have to make adjustments to them even after framework initialization. If we have to modify an annotation after it has been taken into use, we always have to reinitialize it. Therefore it is beneficial, if we don't have to reinitialize everything after a small adjustment.

There are different types of partial framework annotations that constitute the whole annotation of the framework reuse interface. These include *concept annotations* that model the refinement of the framework's abstract concepts, *interface annotations* that are used to add interface implementations into existing classes, and *connection annotations* that provide classes with new methods describing interactions between two or more other abstract concepts. In addition, there are *initialization annotations* that provide assistance for setting up the main class (and a minimal working application) and all the necessary machinery for the application initialization as well as *coding convention annotations* that are used to provide existing classes with code that conforms to specific conventions.

In practice, annotations are usually combinations of these annotation archetypes. We suggest that annotation development is started out by examining the framework source code and locating the interfaces and classes that result in concept annotations, an initialization annotation, and possibly some interface annotations. The need for separate connection annotations typically arises when modeling the internal class structure, for example when describing algorithms defined in method bodies, modeling data fields that are needed to implement the algorithms, and specifying constructors required to initialize the fields.

Organizing Dependencies

Special attention should be paid to organizing the dependencies between the roles in annotations, since they affect the order of framework specialization steps. The dependencies should produce sequences of *programming tasks* that force the user to complete clearly separated parts in the whole framework adaptation process. The dependencies must be designed so that following the resulting sequence of tasks is logical. Tasks should not force the developer to jump from making one thing to another that seems to have no logical connection to the previous task.

A good design principle is to develop framework annotations so that each time the user has done all the mandatory tasks covered by the roles she has instantiated (i.e. there are enough valid contracts for all roles with respect to their cardinalities), she has a working version of her application that she can compile, run and test. With optional tasks she can add new features to the application, and the optional tasks may, in turn, cause new mandatory tasks to appear.

There are of course trivial dependencies like the dependency between a subclass role and a super class role. Determining these dependencies can be considered mechanical and their construction can even be automated, as we will see in section 5.

However, when creating templates for method bodies, for example, we typically encounter more complex dependencies. For instance, consider a framework hot spot that involves overriding a method that returns the instances of framework subclasses that are defined in the application. Should we organize the role dependencies so that the framework specializer is first asked to implement the subclasses and after that the instantiation method or vice versa? We argue that a good rule of thumb in organizing dependencies is to reflect the order in which the resulting application code gets executed. According to this rule, we would organize the dependencies in such a manner that the skeleton for the instantiation method gets created first and it is supplemented with a new instantiation statement every time the application developer defines a new subclass.

Documenting the Annotations

Before an annotation is ready to be used it must be documented for the application developer. First of all, each part of the annotation must have a general description. The description should clearly state the purpose of the hot spot and its dependencies to other hot spots, so that the application developer can determine whether the part of the annotation at hand offers help for accomplishing the desired task. Especially, the initialization hot spot should be clearly identified as being the suggested starting point for application development.

Whenever possible the documentation template should be made context sensitive by referring to the names of the program elements already bound to other roles. We also

suggest linking the annotation documentation to other documentation, for example to the external reference documentation for the framework implementation.

4. Reuse Features and Wizards for the Annotation Language

In order to make the framework annotations themselves more compact we can naturally develop the language and incorporate special reuse features in it. Such features could include *annotation inheritance* and *annotation composition*.

Annotation inheritance is exemplified in figure 4.1. The figure represents how our example framework annotation in figure 2.1 could have been based on a general annotation describing class inheritance. In figure 4.1 the roles *Base* and *Sub* describe the properties of a super class and a subclass. The dotted arrows from the roles in the framework specific annotation to *Base* and *Sub* show how the generic inheritance situation that they describe is reused, e.g., in *ToolExpert* and *DerivedToolExpert*. The benefit is that we don't have to duplicate the inheritance constraint or any other role properties that are related to a situation where one role describes a base class and another describes a subclass[8].

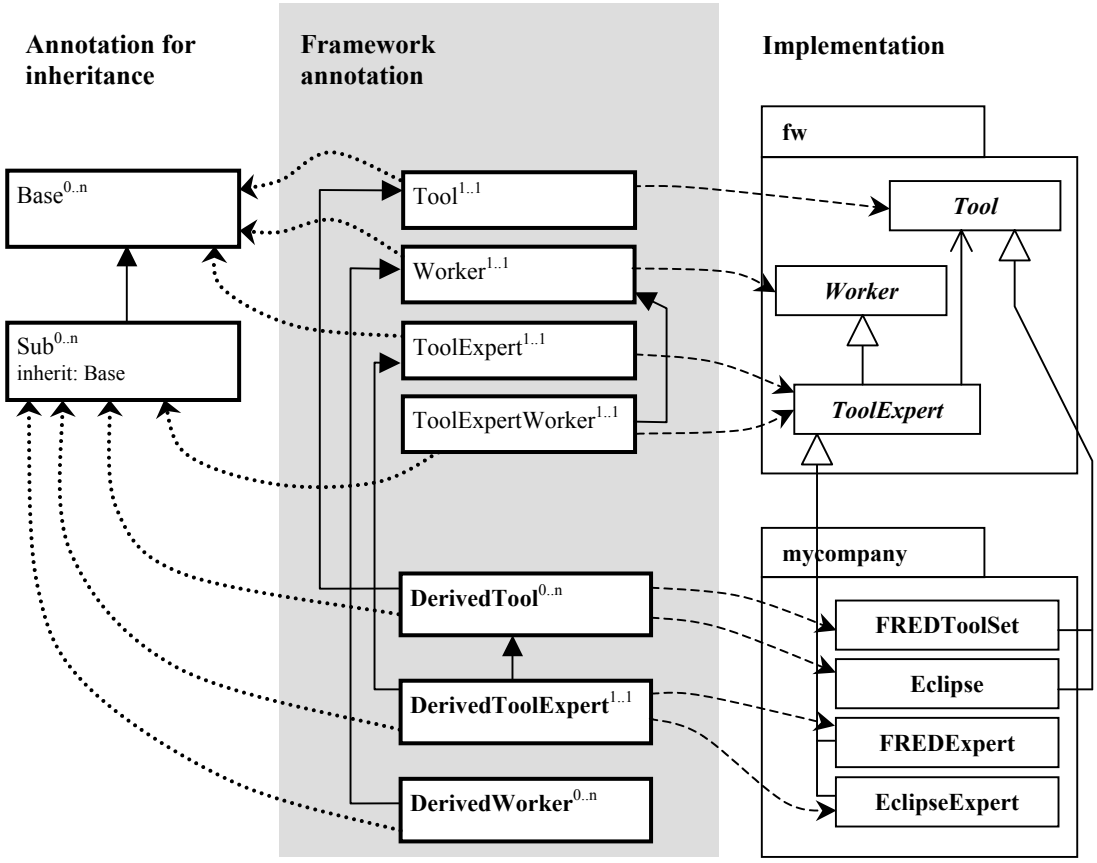


Figure 4.1: Using inheritance in framework annotations

The core of our annotation language doesn't fix the target language for which annotations are made. Actually, the targets of the roles don't even have to be programming language structures. However, when an annotation language is constructed for a particular domain, we must introduce special types of roles, constraints, and templates according to the needs of the target language. In the context of object-oriented frame-

[8] We have left the methods and method roles out from the figure to keep it readable. The generic inheritance annotation enables also the reuse of constraints and templates related to method overriding even though that is not visible in the figure.

works, e.g., the target language is an object-oriented programming language and there should be roles for classes, methods, and fields, for example.

In an annotation language that is fixed for a certain domain (target language), it is possible to create dedicated graphical wizards that help in specifying recurring roles and their properties. A typical example of this is making roles for subclassing or for overriding a method. Similar constraints (inheritance and overriding relationships) recur in these roles. In order to facilitate the introduction of these roles, we may provide wizards that help in creating the constraint and the templates. In order to make a subclass role, for example, the annotation maker only has to choose the super class role for which the subclass role is made. The subclass role wizard can generate sensible implementations for all the other properties of the role (e.g. a suitable name, an inheritance constraint, and templates for code generation and role binding) based on that one selection.

As mentioned earlier, we suggest separating code generation and constraint checking in a framework annotation language. This naturally causes a little more work for the annotation maker, since code generation must reflect the constraints specified in the role. In order to ease the burden of making code templates we suggest code template generation based on the role constraints. Typically a meaningful code generation template (and also a documentation template) can be made based on the constraints declared for a role. This also helps in keeping the constraints and templates consistent with each other.

5. Concept Analysis for Generating Roles from Source Code

When we want to annotate the reuse interface of a framework with roles we have two objectives that must be brought into balance. First, we try to manage with as few roles as possible. This means that the size of the set of program elements represented by any particular role should be maximized. On the other hand, the cohesion of the set should be maximized also. This problem resembles the general modularity problem of software engineering where the goal is to divide a software system into modules with loose coupling between modules and strong cohesion within them.

Concept analysis is a method for identifying commonalities within systems. It has been successfully applied for semi-automatic modularization of software systems [14, 15] as well as for detecting instances of design patterns without a predefined pattern library [16]. We argue that concept analysis can also be used to extract role-based specialization instructions from various kinds of framework descriptions or source code.

In general, concept analysis provides a way to discover sensible groupings of *objects* that have common *attributes* in a certain *context* [14]. Informally, a *concept* can be defined as a collection of all the objects that share a set of attributes in a given context. The set of common attributes of the concept is called the concept's *intent* and the set of objects belonging to the concept is called the concept's *extent*.

In order to extract a role-based annotation that defines the reuse interface of a framework we can select relevant source code entities and their properties from both the framework itself and its available specializations and use them as concept analysis objects and attributes. For example, consider the framework given in figure 2.1. If we were to extract class roles to annotate that framework, we could locate the classes in the framework's and its example applications' source code and use them as concept analysis objects. Selected characteristics of those classes could then be defined as attributes, and the context could be expressed as a table showing which characteristics each class has.

The appropriate selection of attributes is crucial for getting accurate results from the analysis. In figure 5.1 we have decided to use the inheritance and association relationships visible in figure 2.1 as attributes. In addition to the abstract framework classes in figure 2.1, we assume that we have knowledge of some concrete applications-specific subclasses for *Worker* (*Consultant*, not visible in figure 2.1), *Tool* (*FREDToolSet*, *Eclipse*), and *ToolExpert* (*FREDExpert*, *EclipseExpert*). We use also the possible concreteness of classes as an attribute. Note that *Worker* and *Tool* don't have any attributes. We compensate that in the analysis by introducing two additional attributes (*name Worker*, *name Tool*).

		attributes				
		is concrete	***inherits Worker***	***inherits Tool***	***inherits ToolExpert***	***knows Tool***
objects	***Worker***					
	Tool					
	ToolExpert		√			√
	Consultant	√	√			
	FREDToolSet	√		√		
	Eclipse	√		√		
	FREDExpert	√			√	√
	EclipseExpert	√			√	√

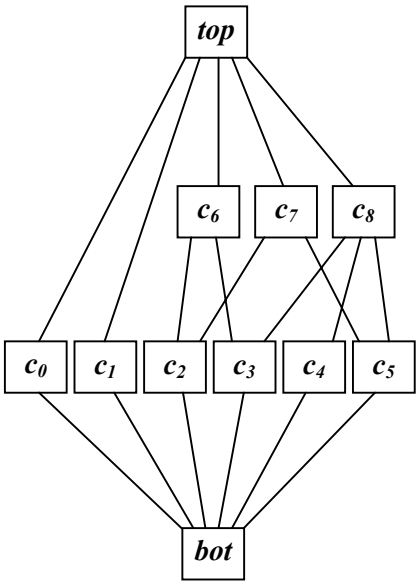
Figure 5.1: A context for determining class roles

It is possible to mechanically determine the concepts of a given context [15] and to form a *concept lattice* that shows the *subconcept relationships*[9] between the concepts (see figure 5.2 for an example based on the context given in figure 5.1). After the concept lattice has been formed, a suitable set of concepts can be chosen as a blueprint for the annotation to be generated. The extent of each selected concept will be translated to a role and the intent of the concept to a set of constraints for that role.

In the translation process we are looking for a set of roles where each program element (i.e. each concept analysis object) plays exactly one role (i.e. belongs to exactly one extent). Unfortunately, in a general case an object may belong to a number of extents in a concept lattice (for instance, *ToolExpert* belongs to the extents of c_2, c_6, c_7, and *top* in figure 5.2). However, we can form a *concept partition* (i.e. a set of concepts where each object takes part in exactly one concept) from any concept lattice [15]. The concepts in the partition can then be translated to a set of roles that forms an annotation for that piece of code of which the extents of the concepts were originally derived from. In figure 5.2 concepts from c_0 to c_5 form a partition that corresponds to the annotation given in figure 2.1 (the only exception is that there is just one concept for *ToolExpert*).

As this example shows, it is fairly straightforward to use concept analysis to translate a set of selected source code structures to a role-based description of their properties and relationships. Our preliminary experiments with this method suggest that it is possible to automatically extract up to 50 % of a framework reuse interface annotation from the source code of the framework itself and a set of representative example applications derived from the framework [17].

[9] A concept is a subconcept of another concept if its extent is a subset of that other concept's extent.



name	extent & intent
top	{*Worker*, *Tool*, *ToolExpert*, *Consultant*, *FREDToolSet*, *Eclipse*, *FREDExpert*, *EclipseExpert*}, ∅
c_8	{*Consultant*, *FREDToolSet*, *Eclipse*, *FREDExpert*, *EclipseExpert*}, {*is concrete*}
c_7	{*ToolExpert*, *FREDExpert*, *EclipseExpert*}, {*knows Tool*}
c_6	{*ToolExpert*, *Consultant*}, {*inherits Worker*}
c_5	{*FREDExpert*, *EclipseExpert*}, {*is concrete*, *inherits ToolExpert, knows Tool*}
c_4	{*FREDToolSet*, *Eclipse*}, {*is concrete*, *inherits Tool*}
c_3	{*Consultant*}, {*is concrete*, *inherits Worker*}
c_2	{*ToolExpert*}, {*inherits Worker*, *knows Tool*}
c_1	{*Tool*}, {*name Tool*}
c_0	{*Worker*}, {*name Worker*}
bot	∅, {*name Worker*, *name Tool*, *is concrete*, *inherits Worker*, *inherits Tool*, *inherits ToolExpert*, *knows Tool*}

Figure 5.2: A concept lattice and an accompanying key

6. The FRED Environment

FRED (Framework Editor for Java) is a prototype of a programming environment intended for aiding framework-based software development [11, 12, 17, 18, 19]. FRED implements an annotation language that can be used to specify the hot spots of a framework. The language defines the format of the annotation in terms of role-based specialization patterns as described in section 2. The tool uses the specialization patterns to provide goal-oriented assistance for application developers using the framework. This assistance includes dynamically changing context-sensitive documentation, automatic code generation, and validation of existing code against the requirements posed for the applications using the framework.

The user interface of FRED is shown in figure 6.1. In the figure, the user has created a project (*MyCompany*) where she has three pattern instances: *Inheritance* (the pattern describing generic inheritance relationships), *Organization* (the annotation for the organization framework introduced in section 2), and *MyCompany* (the contracts related to her particular specialization of the framework). The pattern instances as well as their relationships to one another are visible in the *Pattern Assembly* view in the top left corner. *Pattern View* on the bottom left displays the contracts (i.e. those roles that are already bound) of the selected pattern together with the programming tasks associated with it. A description of the selected task is shown below the task list.

In general, the tasks can either guide the user in providing program elements to be bound to roles or instruct on how to fix possible constraint violations the already bound elements cause. Some of the tasks are mandatory, while some of them are optional. Also, some tasks are mutually ordered and must be solved in a certain sequence.

All the tasks in figure 6.1 are grouped under the *Organization* node, which reflects the fact that all the tasks are related to the specialization of the organization framework. There are optional tasks for providing more concrete tool classes (the user has already defined two tools: *Eclipse* and *FREDToolSet*) and worker classes (there is only one sub-

class for worker, namely *ToolExpert*, which is defined on the framework's side). There is also a mandatory task to provide a *ToolExpert* subclass that would correspond to the *FREDToolSet* class since the annotation requires there to be exactly one concrete tool expert class for each concrete tool. Note, however, that there isn't a similar task for identifying a tool expert class for *Eclipse*. This is because there already exists a valid contract that binds *EclipseExpert* for that purpose.

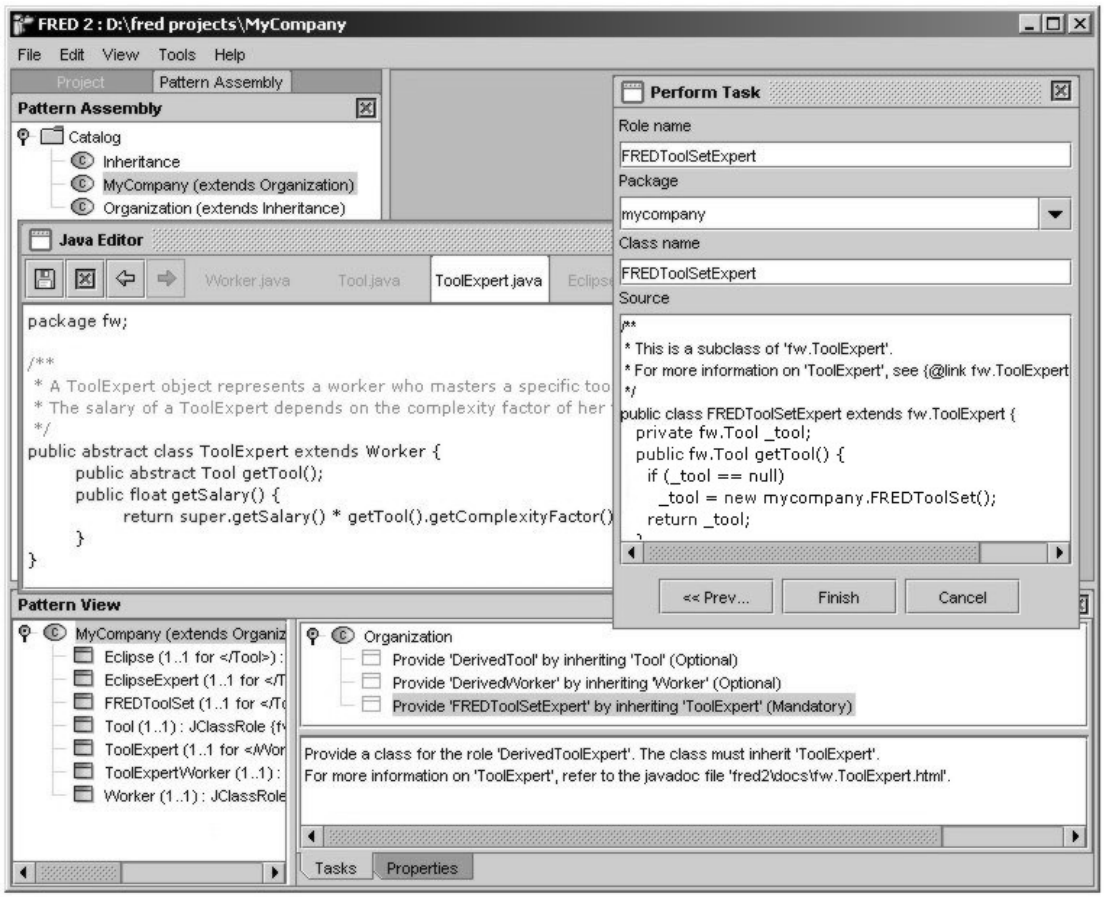


Figure 6.1 The FRED framework engineering environment

Source code for realizing the tasks can be provided in several ways: by coding from scratch, by introducing a binding to a suitable class or method that already exists, or by tailoring a default code snippet generated by FRED. Figure 6.1 shows how the user has selected to use automatic code generation to perform the selected task. The dialog on the right shows a code snippet that has been adapted to this situation based on the templates defined for the role corresponding to the current task as well as the selections the user has made previously. Once the user pushes the *Finish* button, the code will be inserted into a file for further editing.

FRED provides a dedicated *Java Editor* (visible in the middle in figure 6.1) that allows the user to edit her source files. *Java Editor* parses the source incrementally enabling interactive constraint checks and accurate insertion of generated code. The editor not only checks the static typing rules of the Java language, but also the architectural rules of the framework. Changes in the source code are monitored as the user types it in, and possible violations of constraints immediately result in new repairing tasks. Hence, the proper use of the framework is constantly validated and supervised by the system.

7. Conclusion

It is already widely accepted that tool support is practical and useful for aiding the specialization of certain kinds of frameworks. Visual builders for constructing user interfaces have been standard parts of commercial IDEs for quite some time. There have even been attempts to generalize the idea of visual builders to allow the composition of applications from arbitrary components as long as they conform to a standardized component interface or protocol. In our view, these experiences together with our own experiments show that tool support can become a significant factor in framework engineering; not perhaps so much in designing and constructing them, but in documenting them and guiding their usage.

We see framework specialization as a challenging task for which tool support is of vital importance. According to our vision, future frameworks are accompanied with framework-specific programming environments that both guide and control application programmers in creating applications according to the conventions of the framework.

In practice, a framework usage environment should offer at least context-sensitive documentation that dynamically adjusts to the choices the developer makes as well as code generation to automate the production of skeletal implementations and trivial details, so that the developer can concentrate on those parts of the application that are genuinely application-specific. It is also important to maintain an explicit connection between the constraint annotations and source code, in order to be able to validate code against constraints even if it hasn't been generated by the tool or it has been modified later on.

We believe that it is possible to describe the intended rules governing the framework's specializations with a precise role-based formalism. It is clear that a thorough and systematic annotation of a framework's specialization interface raises the development costs. However, we argue that these costs are relatively low when compared to the framework development costs on the whole and, furthermore, the savings gained in training and mentoring will be considerable, especially if many users are going to specialize the same framework.

Moreover, our early experiences indicate that it is possible to reduce the framework annotation costs substantially by using the techniques presented in this paper. For example, a significant portion of a framework annotation can be extracted from the source code of the framework and the example applications using the framework. In addition, it is possible to introduce advanced reuse features (e.g. inheritance and composition) into the annotation language itself and thus make annotations more compact and more manageable.

References

[1] Basili V., Briand L., Melo W., How Reuse Influences Productivity in Object-Oriented Systems. *Communications of the ACM* 39, 10, 1996, 104-116.

[2] Rine D., Nada N., Three Empirical Studies of a Software Reuse Reference Model. *Software — Practice and Experience* 30, 6, 2000, 685-722.

[3] Johnson R., Foote B., Designing Reusable Classes. *Journal of Object-Oriented Programming* 1, 5, 1988, 22-35.

[4] Deutsch L., Design Reuse and Frameworks in the Smalltalk-80 System. In: Biggerstaff T., Perlis A. (eds.), *Software Reusability Vol. II*, ACM Press, 1989, 57-71.

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

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

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

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

[9] van Gurp J., Bosch J., Design, Implementation, and Evolution of Object Oriented Frameworks: Concepts and Guidelines. *Software — Practice & Experience* 31, 3, 2001, 277-300.

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

[11] 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.

[12] 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.

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

[14] Siff M., Reps T., Identifying Modules via Concept Analysis. In: *Proceedings of International Conference on Software Maintenance (ICSM'97)*, Bari, Italy, October 1997, 170-178.

[15] Siff M., Reps T., Identifying Modules via Concept Analysis. TR-1337, Computer Sciences Department, University of Wisconsin, Madison, WI, 1998.

[16] Tonella P., Antoniol G., Object Oriented Design Pattern Inference. In: *Proceedings of International Conference on Software Maintenance (ICSM'99)*, Oxford, England, August-September 1999, IEEE Computer Society Press, 1999, 230-239.

[17] Viljamaa J., Automatic Extraction of Framework Specialization Patterns. Licentiate thesis, Submitted for review and publication, Department of Computer Science, University of Helsinki, 2002.

[18] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J., Generating Application Development Environments for Java Frameworks. In: *Proceedings of 3rd International Conference on Generative and Component-Based Software Engineering (GCSE'01)*, Erfurt, Germany, September 2001, Springer LNCS 2186, 163-176.

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