Generating Application Development Environments for Java Frameworks

Markku Hakala[1], Juha Hautamäki[1], Kai Koskimies[1],
Jukka Paakki[2], Antti Viljamaa[2], Jukka Viljamaa[2]

[1]Software Systems Laboratory, Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
{markku.hakala, csjuha, kk}@cs.tut.fi
[2]Department of Computer Science, Universirty of Helsinki
P.O. Box 26, FIN-00014 University of Helsinki, Finland
{antti.viljamaa, jukka.viljamaa, jukka.paakki}@cs.helsinki.fi

An application framework is a collection of classes implementing the shared architecture of a family of applications. A technique is proposed for defining the specialization interface of a framework in such a way that it can be used to automatically produce a task-driven programming environment for guiding the application development process. Using the environment, the application developer can incrementally construct an application that follows the conventions implied by the framework architecture. The environment provides specialization instructions adapting automatically to the application-specific context, and an integrated source code editor which responds to actions that conflict with the given specialization interface. The main characteristics and implementation principles of the tool are explained.

1 Introduction

Product line architecture is a collection of patterns, rules and conventions for creating members of a given family of software products [4, 14, 17]. *Object-oriented frameworks* are a popular means to implement product line architectures [10]. An individual application is developed by *specializing* a framework with application-specific code, e.g., as subclasses of the framework base classes. The specialization interface of a framework defines how the application-specific code should be written and attached to the framework.

Typically, the documentation provided together with a framework describes informally the specialization interface of the framework. Usually this is done simply by giving examples of possible specializations. Unfortunately, such descriptions cannot be used as the basis of building systematic support for the specialization process. An attractive approach to solve this problem is to define the specialization process of a framework as a "cookbook" [8, 18, 22, 23, 25]. Related approaches include also motifs [19] and hooks [9]. The support offered by these approaches ranges from improving the understanding of frameworks to providing algorithmic recipes for separate specialization tasks. Our work continues this line of research, but

we focus on issues that we feel are not adequately addressed so far. In particular, these issues include:

1. *Support for incremental, iterative and interactive specialization process*. We strongly believe that the specialization of a framework, or even its single hot spot, should not be regarded as a predefined sequence of steps, far less an atomic, parameterized action. The application developer should be able to execute the specialization tasks in small portions, see their effect in the produced source code, and go back to change something, if needed. This kind of working is inherent to software engineering, and the tool should support it. Therefore, specialization should be guided by a dynamically adjusting list of specialization steps that gradually evolves based on the choices made in the preceding steps. In this way, the application developer has better control and understanding of the process and of the produced system.
2. *Specialized specialization instructions*. The problem with traditional framework documentation is that it has to be written before the specialization takes place. Therefore the documentation has to be given either with artificial examples or in terms of the general, abstract concepts of the framework, not with the concrete concepts of the specialization at hand. In an incremental specialization process the tool can gather application-specific information (e.g., names of classes, methods and fields) and gradually "specialize" the documentation as well. This makes the specialization instructions much easier to follow.
3. *Architecture-sensitive source-code editing*. In our view, the architectural rules that must be followed in the specialization can be seen much like a higher level typing system. In the same sense as the specialization code must conform to the typing rules of the implementation language, it must conform to the architectural rules implied by the framework design. A framework-specific programming environment should therefore enforce not only the static typing rules of the programming language but also the architectural rules of the framework.
4. *Open-ended specialization process*. The specialization process should be open-ended in the sense that it can be resumed even for an already completed application. We feel that this is important for the future maintenance and extension of the application.

In this paper we propose a technique to define the specialization interface of a framework in such a way that it can be used to generate a task-driven application development environment for framework specialization. We demonstrate our tool prototype called *FRED* (*FRamework EDitor*) that has been implemented in Java and currently supports frameworks written in Java. The approach is not however tied to a particular language.

Different techniques to find and define the specialization interfaces for Java frameworks using FRED have been discussed in [12], summarizing our experiences with FRED so far. We have applied FRED to two major frameworks: a public domain graphical editor framework (JHotDraw [15]) and an industrial framework by Nokia intended for creating GUI components for a family of network management systems. This paper focuses on the characteristics of the FRED tool and its implementation principles.

In the next section we will present an overview of the FRED approach. In Section 3 we will discuss the underlying implementation principles of FRED. Related work is discussed in Section 4. Finally, some concluding remarks are presented in Section 5.

The FRED project has been funded by the National Technology Agency of Finland and several companies. FRED is freely available at http://practise.cs.tut.fi/fred.

2 Basic Concepts in FRED

A basic concept for defining the specialization interface in FRED is a *specialization pattern*. A specialization pattern is an abstract structural description of an extension point (a hot spot) of a framework. Specialization pattern is typically of the same granularity as a recipe or hook [9].

In principle, a specialization pattern can be given without referring to a particular framework; for example, most of the GoF design patterns [11] can be presented as specialization patterns. However, we have noted that this is usually less profitable for our purposes: a framework-specific specialization pattern can be often written in a way that provides much stronger support for the specialization process, even though the specialization pattern followed one or more general design patterns. This is due to the fact that the way a design pattern is implemented in a framework affects the exact specialization rules and instructions associated with that pattern. Hence, for the purposes of this paper we can assume that a specialization pattern is given for a particular framework.

A specialization pattern is a specification of a recurring program structure. It can be instantiated in several contexts to get different kinds of concrete structures. A specialization pattern is given in terms of *roles*, to be played by structural elements of a program, such as classes or methods. We call the commitment of a program element to play a particular role a *contract*. Some role is played by exactly one program element, some can be played by several program elements. Thus, a role can have multiple contracts. This is indicated by the *multiplicity* of the role; it defines the minimum and maximum number of contracts that may be created for the role. Combinations are from one to one (1), from zero to one (?), from one to infinity (+), and from zero to infinity (*). E.g., a specialization pattern may define two roles; a base class and a derived class, where the base class role must have a single contract, but the derived class role may have an arbitrary number of contracts. Respectively, a single program element can have multiple contracts and 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*. 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.

When a specialization pattern is framework-specific, certain roles are played by fixed, unique program elements of the framework. We say that such roles are *bound*; otherwise a role is *unbound*. Hence, a bound role is a constant that denotes the same program element in every instantiation of the pattern, while unbound roles are variables that allow a pattern to be applied in different contexts.

Specialization patterns, together with the contracts for the bound roles and the framework itself, constitute a developer's kit delivered for application programmers. We call the process of creating the rest of the contracts *casting*. As each contract acts as a bridge between a role and a suitable program element, casting essentially requires the specializer to produce specialization-specific code for the contracts. The set of contracts for a given software system is called a *cast*. It consists of the contracts defined by bound roles as well as the contracts established by the framework specializer. Together, the contracts convey the architectural correspondence between the source-code and the framework specialization interface. If a pattern defines relationships between roles, these relationships must manifest in the program elements that are contracted to the roles. Thus, the connection between framework and specialization-specific code are made explicit. It is also equally necessary to define mutual relationships between the different parts of the specialization, an important aspect often overlooked.

Casting is the central activity of framework specialization. Each contract is a step required for developing an application as a specialization of a framework. In a sense, casting can be regarded as the instantiation of specialization patterns. The main purpose of FRED is to support the programmer in the casting process. This is achieved by presenting missing and breached contracts as programming tasks that usually ask the user either to provide or correct some piece of code. Based on the relationships encoded in the pattern and the contracts already made, the tool is able to suggest new contracts as the specialization proceeds, leading to an incremental and interactive process which follows no single predetermined path.

Let us illustrate the concept of a specialization pattern with a simple example. Suppose there is a graphical framework which can be extended with new graphical shapes. The framework is designed in such a way that a new shape class must inherit the framework class Shape and override its draw method. In addition, the new class must provide a default constructor, and an instance of the new class must be created and registered for the application in the main method of the application-specific class.

The required specialization pattern is given Table 1. FRED provides a dedicated tool for defining the specialization patterns. However, we use here an equivalent textual representation format to facilitate the presentation. In the example, we have followed the naming convention: if a role is assumed to be played by a unique program element of the framework (it is bound), it has the same name as that element.

In Table 1, the creator of the pattern has specified some properties for the roles. Some properties, when not specified, have a default value provided by the tool. Properties description and task title are exploited in the user interface for a general description of the role and for the task of creating a contract, respectively (see Figure 1). Properties return type, inheritance and overriding are constraints specifying the required return type of a method, the required base class of a class, and the method required to be redefined by a method. Property source gives a default implementation for a method or for a code fragment, while Insertion tag specifies the tag used in the source to mark the location where this code fragment should be inserted. Tags are written inside comments, in the form "#tag". Tags are used only in inserting new code to an existing method.

Note that the definitions of properties may refer to other roles; such references are of the form <#r>, where r is the identification of a role. By convention, if <#r> appears within string-valued property specification (e.g., task title), it is replaced by the name of the program element playing the role. This facility is used for producing

adaptable textual specialization instructions. In constraints, references to other roles imply relationships that must be satisfied by the program elements playing the roles. For example, the class playing the role of SpecificShape must inherit the class playing the role of Shape. The role SpecificShape is also associated with a multiplicity symbol "+", meaning that there can be one or more contracts for this role for each contract of Shape. However, as Shape is bound, it has actually only a single contract.

Table 1. Textual representation of a specialization pattern

NewShape		
Bound roles	*Properties*	
Shape : class	description	Base class for all graphical figures.
draw : method	description	The drawing method.
Unbound roles	*Properties*	
ApplicationMain : class	description	The application root class that defines the entry point for the application.
main : method	description	The method that starts the application.
	type	void

	source	Canvas c = new Canvas();
		/* #CanvasInitialization */
		c.run();

args : parameter	type	String[]
	position	1
creation : code	insertion tag	CanvasInitialization
	description	Code creating a prototype instance of <#SpecificShape> by invoking constructor <#SpecificShape.defaultConstructor>.
	task title	Provide creation code for <#SpecificShape>
	source	c.add(new <#SpecificShape>());
SpecificShape+ : class	description	Defines a graphical figure by extending <#Shape>.
	task title	Provide a new concrete subclass for <#Shape>
	inheritance	<#Shape>
	default name	My<#Shape>
defaultConstructor : constructor	task title	Provide a constructor for <#SpecificShape>
draw : method	task title	Override <#Shape.draw> to draw <#SpecificShape>
	overriding	<#Shape.draw>

Nesting of roles in Table 1 specifies a containment relationship between the roles, which is an implicit constraint: if role *r* contains role *s*, the program element playing role *r* must contain the program element playing role *s*. This makes the specialization pattern structurally similar to the program it describes.

During casting, new contracts are created for the roles and associated with program elements. This process is driven by the mutual dependencies of the roles and the actions of the program developer, including the direct editing of the source code. The framework cast consists of contracts which bind roles Shape and draw to their counterparts in the framework. Given this information, FRED is able start by displaying two mandatory tasks for the specializer. These are based on the roles SpecificShape and ApplicationMain, since these roles do not depend on other application-specific roles. The user can carry out the framework specialization by executing these tasks and further tasks implied by their execution. Eventually there will be no mandatory tasks to be done, and the specialization is (at least formally) complete with respect to this extension point.

Roughly speaking, FRED generates a task for any contract that can be created at that point, given the contracts made so far. For example, it is not possible to create a contract for draw unless there is already a contract for SpecificShape, because draw depends on SpecificShape. A task prompting the creation of a contract is mandatory if the lower bound of the multiplicity of the corresponding role is 1, and there are no previous contracts for the role; otherwise the task is optional. FRED generates a task prompt also for an existing contract that has been broken (e.g., by editing actions). We will discuss the process of creating contracts in more detail in Section 3.

The organization of the graphical user interface is essential for the usability of this kind of tool, and the current form is the result of rather long evolution. We have found it important that the user can see the entire cast in one glance, and that a task is shown in its context, rather than as an item in a flat task list. For these reasons the central part of the user interface shows the current cast structured according to the containment relationship of the associated 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. The tasks are shown with respect to this view: for each contract selected from the cast view, a separate task pane shows those tasks that produce or correct contracts under the selected contract, according to the containment relationship of the corresponding roles. For example, if a contract has been created for SpecificShape, and this contract is selected, the task pane displays a (mandatory) task for creating a contract for the draw role.

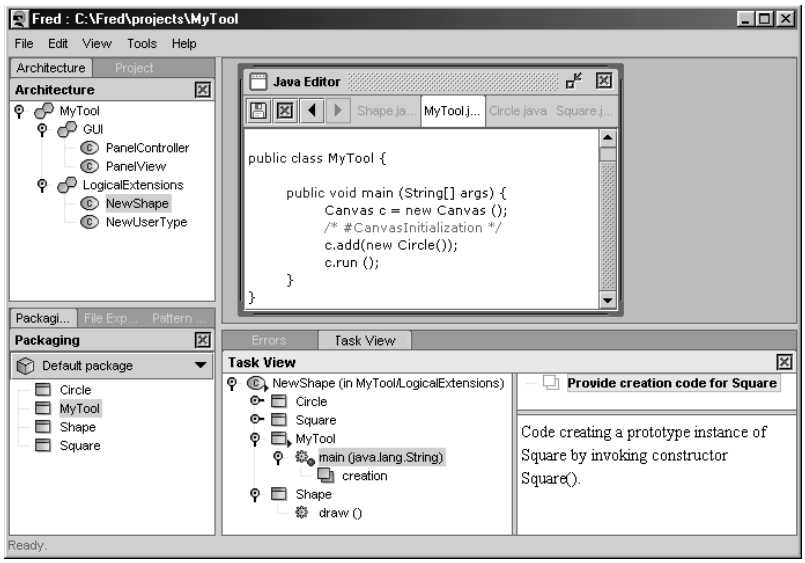

Fig. 1. User interface of FRED

The user interface of FRED is shown in Figure 1. It contains a number of views to manage Java projects and the casting process. In the figure, the application developer has opened the Architecture View, which shows the project in terms of subsystems and instantiated specialization patterns. The Task View shows the existing contracts

in the left pane. Tasks related to a selected contract are shown in the right pane of the Task View. A small red circle in the left pane indicates that there are mandatory tasks related to that contract, a white circle indicates an optional task. The lower part of the right pane shows the instructions associated with the role (that is, given by property description).

Figure 1 shows the FRED user interface in a situation where the application developer has already carried out the necessary tasks related to a new subclass Circle. In addition, the developer has done an optional task for creating yet another sublass named Square, and the resulting mandatory task for providing its draw method. The remaining mandatory task is indicated by a red circle. This task is selected in the figure, and the user is about to let the tool generate the creation code at the appropriate position.

To carry out the specialization the developer needs to complete all the rest of the mandatory tasks, and the mandatory tasks resulting from the completion of these tasks. However, this process need not be a linear one. A mechanism is provided to undo contracts, providing the means to backtrack the instantiation process and reconsider the decisions made.

Although the example is very simple, it demonstrates our main objectives. The specialization of the framework is an interactive, open-ended process where the application developer gets fine-grained guidance on the necessary specialization tasks and their implications in the source code. The specialization instructions are adapted to the application context (see the task title and instructions in Figure 1). The source editor is tightly integrated with the casting process: for example, if the user accidently changes the base class of Circle by editing, the tool generates a new task prompting the user to correct the base class. Therefore, much like a compiler is able to check language-specific typing, FRED enforces architecture-specific typing rules. If the user then re-edits the source and fixes the base class, the task automatically disappears.

3 Implementation

To understand how the tool fulfils its responsibilities we have to investigate the specialization patterns and their interpretation little deeper. A specialization pattern, as presented in previous chapters, is given as a collection of roles, each defined by its properties. The approach permits quite arbitrary properties and kinds of roles, and indeed we consider the independence of exact semantics (provided by these primitives) as one of the principal strengths of our approach. The current FRED implementation offers one alternative set of primitives tailored for Java. Changing the set of primitives it is possible to turn FRED into a development environment for a different language, a different paradigm or even a different field of engineering.

The properties supported by the current FRED implementation can be roughly categorized into constraints and templates. A constraint attaches a requirement on a role or a relationship between two roles. The constraints must be satisfied by the program elements playing a role, and can be statically verified by FRED. A template in turn is used for generating text, mostly code, instructions or documentation. Templates support a form of macro expansion that makes it possible to generate context-specific text.

Properties can refer to other roles of the pattern. Whenever the definition of role *r* refers to role *s* (at least once) or role *r* is enclosed in role *s*, we say there is *dependency* from *r* to *s*, or that the role *r* depends on *s* or has a dependency to *s*. From a pattern specification it is possible to construct a directed graph, whose nodes and edges correspond to roles and dependencies, respectively. In addition, each node of the graph carries the multiplicity of the associated role. The resulted graph describes declaratively the process of casting, and is interpreted by the tool to maintain a list of tasks. Actually, the bound roles and dependencies to them can be omitted from this graph, as being constant bound roles do not change the course of the casting process. Likewise, the dependencies that can be deduced from other dependencies can be discarded from the graph, i.e., a dependency from *r* to *s* can be removed if there is directed path from *r* to *s* in the graph.

A graph based on the specialization pattern NewShape, from Chapter 2, is presented in Figure 2. In this diagram, the boxes denote roles. The label of a node is made up of the role name and a multiplicity symbol. A dependency is presented by an arc, or nesting in case the role is nested in the original specification. In addition to denoting an edge, nesting works as a name scope, as in the original pattern specification. Different kinds of visual decorations are used on the nodes to denote their kind. A class role is presented with a thick border and a method role with a thinner one. A parameter role is circular and a code snippet is denoted with bent corner. Bound roles are absent from the diagram. Nesting, decorations and omitted nodes are all just means of compacting the graph and carry no specific semantics in the discussion to follow.

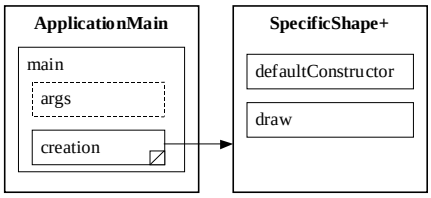

Fig. 2. A diagram of the NewShape specialization pattern

The pattern graph is the basis of casting. The process starts by selecting a pattern and creating a cast for it. Initially, the cast consists of contracts for bound roles. For each unbound role, a number of contracts must be eventually established in the cast. The state of the cast at any point during the casting can be presented as a graph of contracts. The edges of also this graph are called dependencies, and are implied by the dependencies of the pattern. To be more precise, if a role *r* depends on role *s*, each contract of role *r* depends on some unique contract of role *s*, determined unambiguously during the casting. In the cast graph, we need to include only contracts established by the specializer and can thus ignore the contracts for bound roles and the related dependencies. Likewise, as with pattern graphs we can omit redundant dependencies.

Figure 3 presents a diagram of an example cast graph (on the left), and its relation to some specialization-specific source code (on the right). The diagram presents some point in the middle of casting of NewShape pattern. We use a notation similar to presenting pattern graphs. In the diagram, the boxes denote contracts, and the arcs and

nesting denote the dependencies. The label of a node refers to the role associated with the contract. A colon is used before the label to mark that the node doesn't represent a role but a contract of the role. Similar to pattern graphs, we use border decorations on the nodes, depending on the kind of the role the contract stands for. It is easy to read from the figure which parts of code play which roles in the pattern. The figure also shows that the dependencies between roles (e.g. from creation role to SpecificShape role) have implied dependencies between contracts. This is also evident in the nesting of contracts.

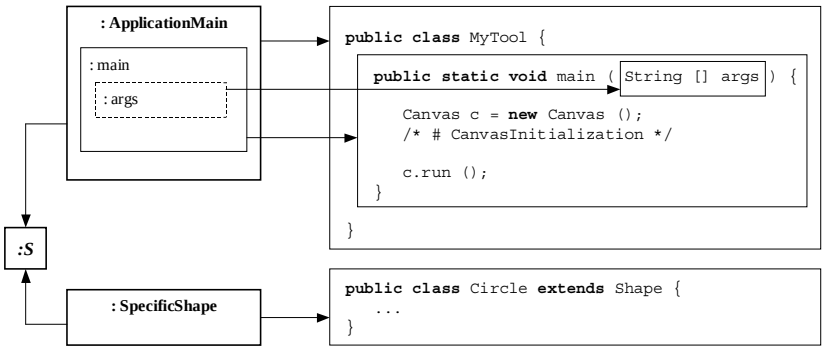

Fig. 3. An example of cast that relates specialization code to the roles of the pattern

The function of the development tool can be defined in terms of the pattern graph and the cast graph. The exact process of casting can be reduced to nondestructive graph transformations on the cast graph, based on the pattern graph. In fact, the pattern graph can be seen as a relatively restricted, but compact way of specifying a graph grammar. This representation can be derived systematically to a more conventional presentation of a graph grammar [6], a set of transformation rules. We shall now describe the process of casting more accurately.

A graph grammar can be defined with a start graph and a set of graph transformation rules. The start graph of a grammar produced from a pattern graph contains a single node, start role *S*, that besides acting as a starting point of graph transformations carries no special meaning. The transformation rules in turn, are generated by the algorithm in Figure 4.

For each role *r* in the pattern graph:
  R := A graph that contains *r* and all roles and dependencies on every directed path that goes from *r* to a sink of the pattern graph.
  If *R* contains only *r* then add start role *S* to *R*
  L := *R* - *r* and all dependencies from *r*.
  Add transformation rule *L* ::= *R* to the grammar.
End

Fig. 4. An algorithm that generates the transformation rules from a pattern specification

This results in a simple grammar, consisting of a single non-destructive transformation rule for each role of the original pattern. The rules are expressed in terms of roles

and are responsible in generating a network of contracts, the cast. Moreover, due to the regularity of the generated rules, an application of any of the rules results in a single new contract and its dependencies.

In Figure 5 we see a graph grammar that has been produced from the pattern graph presented in Figure 2. As there were seven roles in the pattern graph, there are seven numbered rules. The full name of the associated role, along with the multiplicity symbol is placed above each rule.

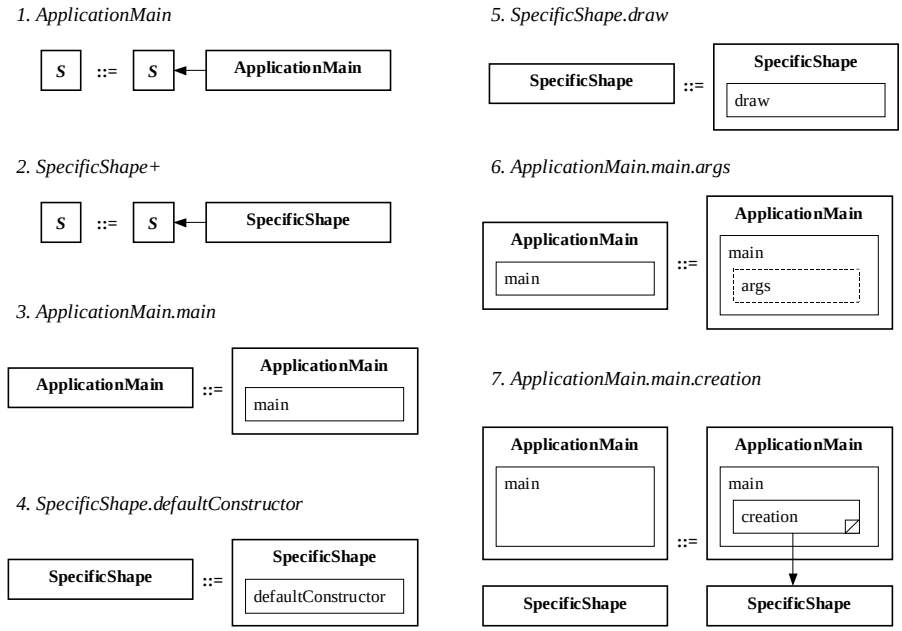

Fig. 5. The graph grammar of NewShape, derived from its pattern graph

Casting starts by creating a cast with a special start contract, a contract of start role *S*. It's only purpose is to start the casting process and is not bound to any program element. The transformation rules, whose left hand sides contain only *S*, are first applicable. In general, the left hand side of the transformation rule is matched against the current cast, and the rule is applicable for each found match, i.e., for each suitable sub-graph of the cast. Then, the matched sub-graph is substituted with the right hand side of the rule, resulting in a new contract and a set of dependencies in the cast graph. The multiplicity of a role constrains the number of times the rule can be applied for each different sub-graph. E.g., the rule 2 above is matched always, rule 5 is matched only once for each contract of SpecificShape, and rule 7 matched for each pair of contracts of main and SpecificShape.

Whenever a transformation rule is applicable for some match, the tool applies the rule to produce a new contract for that match. This contract is *incomplete* as it is not bound to any program element at that time. An incomplete contract corresponds to a task in the user interface, shown to the developer as a request to provide a new program element to complete the contract. The task is either mandatory or optional,

depending on the multiplicity and number of contracts already created for the same match. Once the contract is completed by a suitable program element, it is added to the cast making new transformation rules applicable.

As an example, look at Figure 3. At that point the user has already created a class for SpecificShape, as well as the main class with the main method. At this point, the user may apply rule 2 to create a new SpecificShape, or rules 4 or 5 to continue with the existing SpecificShape – the Circle, or with rule 7 to add the intialization code within the main method. These choices are presented as programming tasks, from which only the task for rule 2 is optional. Figure 6 presents the situation after application of transformation rule 7. A new contract has been added to the cast and made available for matching.

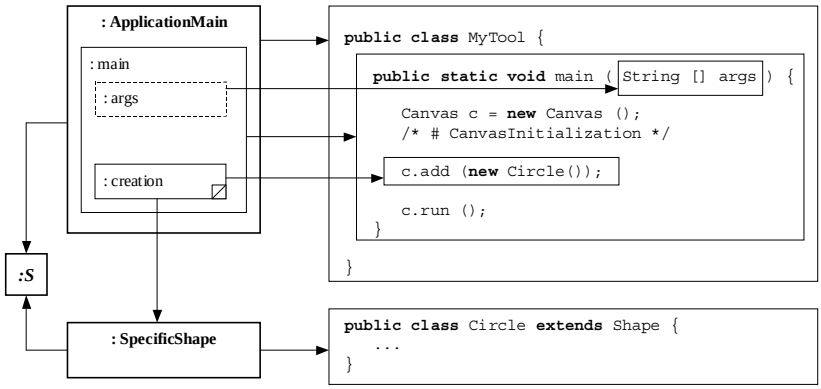

Fig. 6. Result of an application of a grammatical rule to the cast graph of Figure 3

Code generation, adaptive specialization instructions, constraints and other properties are evaluated in the context of a single contract, always linked to a graph of contracts in a way determined by the piecemeal application of grammatical rules. This means that whenever a property refers to role *r*, this reference can be unambiguously substituted by a contract of *r* obtained by following the dependencies in the cast graph. Furthermore, this can be substituted by a reference to the associated program element. E.g., in the case of the contract of for the role creation in Figure 6, all references to SpecificShape can be substituted with references to the class Circle. Thus, the constraints can be evaluated separately for each contract and it is possible to provide contract-specific instructions and default implementation, like the line of code in this case.

Most contracts are not automatically determined based on the source code, but instead explicitly established by the developer by carrying out tasks. As a side effect, some code can be generated, but a contract can also be established for an existing piece of code, thus allowing a single program element to play several roles. Once a contract is established for a piece of code, the environment can use this binding for ensuring that the code corresponds to the constraints of the role. For this purpose, FRED uses incremental parsing techniques to constantly maintain an abstract syntax tree of the source code and can thus provide immediate response for any inappropriate changes to the code.

4 Related Work

To tackle the complexities related to framework development and adaptation we need means to document, specify, and organize them. The key question in framework documentation is how to produce adequate information dealing with a specific specialization problem and how to present this information to the application developer. A number of solutions have been suggested, including *framework cookbooks* [18, 25], *smartbooks* [23], and *patterns* [16].

As shown in this paper, an application framework's usage cannot be adequately expressed as a static and linear step-by-step task list, because a choice made during the specialization process may change the rest of the list completely. That is why cookbooks [18, 25], although a step to the right direction, are not enough. Our model can be seen as an extension of the notion of framework cookbooks.

Another advanced version of cookbooks is the SmartBooks method [23]. It extends traditional framework documentation with instantiation rules describing the necessary tasks to be executed in order to specialize the framework. Using these rules, a tool can be used to generate a sequence of tasks that guide the application developer through the framework specialization process [22]. This reminds our model, but whereas they provide a rule-based, feature-driven, and functionality-oriented system, our approach is pattern-based, architecture-driven and more implementation-oriented.

Froehlich, Hoover, Liu and Sorenson suggest semiformal template on describing specialization points of frameworks [9] in the form of *hooks*. A hook presents a recipe in a form of a semiformal, imperative algorithm. This algorithm is intended to be read, interpreted and carried out by the framework specializer.

Fontoura, Pree, and Rumpe present a UML extension *UML-F* to explicitly describe framework variation points [8]. They use a UML *tagged value* (a name-value-pair that can be attached to a modeling element to extend its properties) to identify and document the hot spots such that each of the variation point types has its own tag.

Framework adaptation is considered to be a very straightforward process in [8]. UML-F descriptions are viewed as a structured cookbook, which can be executed with a wizard-like framework instantiation tool. This vision resembles closely that of ours, but we see the framework specialization problem to be more complex. The proposed implementation technique is based on adapting standard UML case tools, which does not directly support FRED-like interactivity in framework specialization.

The specification of an architectural unit of a software system as a pattern with roles bound to actual program elements is not a new idea. One of the earliest works in this direction is Holland's thesis [13] where he proposed the notion of a contract. Like UML's collaborations, and unlike our patterns, Holland's contracts aimed to describe run-time collaboration. After the introduction of design patterns [11], various formalizations have been given to design patterns resembling our pattern concept (for example, [7, 20, 21, 26]), often in the context of specifying the hot spots of frameworks. Our contribution is a pragmatic, static interpretation of the pattern concept and the infrastructure built to support its piecemeal application in realistic software development. In fact, our patterns can be seen as small pattern languages [2] for writing software.

In [5] Eden, Hirshfeld, and Lundqvist present LePUS, a symbolic logic language for the specification of recurring motifs (structural solution aspect of patterns) in object-oriented architectures. They have implemented a PROLOG based prototype

tool and show how the tool can utilize LePUS formulas to locate pattern instances, to verify source code structures' compliance with patterns, and even to apply patterns to generate new code.

In [1] Alencar, Cowan, and Lucena propose another logic-based formalization of patterns to describe *Abstract Data Views* (a generalization of the MVC concept). Their model resembles ours in that they identify the possibility to have (sub)tasks as a way to define functions needed to implement a pattern. They also define parameterized *product texts* corresponding to our code snippets.

We recognize the need for a rigor formal basis for pattern tools, especially for code validation. We emphasize support for adaptive documentation and automatic code generation instead of code validation.

5 Conclusions

We have presented a new tool-supported approach to architecture-oriented programming based on Java frameworks. We anticipate that application development is increasingly founded on existing platforms like OO frameworks. This development paradigm differs essentially from conventional software development: the central problem is to build software according to the rules and mechanisms of the framework. So far there is relatively little systematic tool support for this kind of software development. FRED represents a possible approach to produce adequate environments for framework-centric programming. A framework can be regarded, in a broad sense, as an application-oriented language, and FRED is a counterpart of a language-specific programming environment. Our experiences with real frameworks confirm our belief that the fairly pragmatic approach of FRED matches well with the practical needs. Our future work includes integration of FRED with contemporary IDEs and building FRED-based support for standard architectures like Enterprise Java Beans.

References

1. Alencar P., Cowan C., Lucena C., A Formal Approach to Architectural Design Patterns. In Proc. *3rd International Symposium of Formal Methods Europe*, 1996, 576-594.
2. Alexander C., *The Timeless Way of Building*, Oxford University Press, New York, 1979.
3. Boris Bokowski, CoffeeStrainer - Statically-Checked Constraints on the Definition and Use of Types in Java, Proceedings of *ESEC/FSE '99*, Springer-Verlag.
4. Bosch J., *Design & Use of Software Architectures — Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.
5. Eden A., Hirshfeld Y., Lundqvist K., LePUS — Symbolic Logic Modeling of Object Oriented Architectures: A Case Study. *NOSA '99 Second Nordic Workshop on Software Architecture*, University of Karlskrona/Ronneby, Ronneby, Sweden, 1999.
6. Ehrig H., Taentzer G., Computing by Graph Transformation: A Survey and Annotated Bibliography, Bulletin of the EATCS, 59, June 1996, 182-226.
7. Florijn G., Meijers M., van Winsen P., Tool Support for Object-Oriented Patterns. In: Proc. *ECOOP '97 (LNCS 1241)*, 1997, 472-496.
8. Fontoura M., Pree W., Rumpe B., UML-F: A Modeling Language for Object-Oriented Frameworks. In: Proc. *ECOOP '00 (LNCS 1850)*, 2000, 63-83.

9. Froehlich G., Hoover H., Liu L., Sorenson P., Hooking into Object-Oriented Application Frameworks. In: Proc. *ICSE '97*, Boston, Mass., 1997, 491-501.
10. Fayad M., Schmidt D., Johnson R., (eds.), *Building Application Frameworks — Object-Oriented Foundations of Framework Design*. Wiley 1999.
11. Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns — Elements of Object-Oriented Software Architecture*. Addison-Wesley, 1995.
12. Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Annotating Reusable Software Architectures with Specialization Patterns. In: Proc. *WICSA '01*, Springer 2001, to appear.
13. Holland I., The Design and Representation of Object-Oriented Components. Ph.D. thesis, Northeastern University, 1993.
14. Jacobson I., Griss M., Jonsson P., *Software Reuse — Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.
15. JHotDraw 5.1 source code and documentation. http:// members.pingnet.ch/gamma/JHD-5.1.zip, 2001.
16. Johnson R.: Documenting Frameworks Using Patterns. In: Proc. *OOPSLA '92*, Vancouver, Canada, October 1992, 63-76.
17. Jazayeri M., Ran A., van der Linden F., *Software Architecture for Product Families.* Addison-Wesley, 2000.
18. Krasner G., Pope S., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Object-Oriented Programming*, 1988.
19. Lajoire R., Keller R., Design and Reuse in Object Oriented Frameworks: Patterns, Contracts and Motis in Concert. In: *Object Oriented Technology for Database and Software Systems*, Alagar V., Missaoui R. (eds.), World Scientific Publishing, Singapore, 1995, 295-312.
20. Meijler T., Demeyer S., Engel R., Making Design Patterns Explicit in FACE — A Framework Adaptive Composition Environment. In: Proc. *ESEC '97 (LNCS 1301)*, 94-111.
21. Mikkonen T., Formalizing Design Patterns. In: Proc. *20th International Conference on Software Engineering (ICSE '98)*, *IEEE Press*, 1998, 115-124.
22. Ortigosa A., Campo M., Salomon R., Towards Agent-Oriented Assistance for Framework Instantiation. In Proc. *OOPSLA '00, Minneapolis, Minnesota USA, ACM SIGPLAN Notices*, 35, 10, 2000, 253-263.
23. Ortigosa A., Campo M., SmartBooks: A Step Beyond Active-Cookbooks to Aid in Framework Instantiation. *Technology of Object-Oriented Languages and Systems* 25, June 1999, IEEE Press. ISBN 0-7695-0275-X
24. Pasetti A., Pree W., Two Novel Concepts for Systematic Product Line Development. In: Donohoe P. (ed.), *Software Product Lines: Experience and Research Directions (First Software Product Lines Conference, Denver, Colorado)*, Kluwer Academic Publishers, 2000.
25. Pree W., *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.
26. Riehle R., Framework Design — A Role Modeling Approach. Ph.D. thesis, ETH Zürich, Institute of Computer Systems, February 2000.