

**Practical Support for Frameworks:
The Story of Fred**

by
Markku Hakala

University of Tampere
Department of Computer Sciences
Master's Thesis
June 2008

University of Tampere

Department of Computer Sciences

HAKALA, MARKKU: Practical Support for Frameworks : The Story of Fred

Master's Thesis

June 2008

Application frameworks are a popular way of implementing product line architectures. The Fred project was launched in 1997 to tackle the complexity of using frameworks by experimenting with the possibilities of programming tool support. During the years a novel method of task-based programming was developed and demonstrated by a tool prototype. The tool was able to assist the programmer in applying architectural conventions and design patterns by giving detailed context-sensitive programming instructions. Since the original Fred project, the tool has also been adapted to various unanticipated domains such as software maintenance and documentation. This thesis is presents a chronological walkthrough of the project and its results in the form of six publications.

Keywords: Software frameworks, design patterns, programming tools

Foreword

It has taken too many years for me to complete this master's thesis. I would like to thank professor Kai Koskimies for the tremendous amount of support he has given me during my years in the university. It would not have been without him I could have pulled through.

– Markku

Contents

- 1. Introduction**
- 2. Pattern-Oriented Framework Engineering Using FRED**
- 3. Managing Object-Oriented Frameworks with Specialization Templates**
- 4. Task-Based Tool-Support for Framework Specialization**
- 5. Generating Application Development Environments for Java Frameworks**
- 6. Annotating Reusable Software Architectures with Specialization Patterns**
- 7. Feature Models, Pattern Languages and Software Patterns: Towards a Unified Approach**

Chapter 1.

Introduction

Introduction

This thesis is composed on six papers published during the Fred- and JavaFrames-projects between 1998 and 2002. The publications communicate the core ideas of these projects to the academic community. They have been presented in various conferences and workshops in the Europe and U.S.

The Fred project was set set to investigate the possibilities of tool support for object-oriented application frameworks [FSJ00] based on the concept of design patterns [Gam95], and the JavaFrames project was a straightforward continuation for it. The research was mainly constructive. During these projects a programming environment was developed. This tool is able to assist the programmer in recurring tasks, such as using a design pattern, a software component or framework. This tool was first named Fred, but the name was later changed to JavaFrames.

The tool works by taking in specifications and providing practical programming assistance based on them. The specifications describe recurring programming jobs, and they are specified for the tool using a special-purpose editor. Thereafter, the tool is able to assist the programmer in these jobs hand-in-hand with the actual programming. The tool does this by generating code, giving programming instructions in plain English, and checking the hand-written code against the specification. The instructions are given to the programmer in the form of a cookbook-like [Pre95] step-list. When the programmer carries out the steps, the tool is able to refine both the generated code and any further instructions accordingly.

The Fred tool demonstrates a method for specifying crosscutting software structures as reusable components. It is complementary to other means of reuse, such as software components, object-orientation, genericity or aspects. The first experiments were carried out to support Java programming, but since then the tool has been applied to various other domains, including UML and C++.

All the publications in this thesis are aimed on the same thing - introducing Fred to the academic audience. They have been presented in various forums, each of which speaks their own tongue. Thus, each the authors have had to relate their work against many different backgrounds. Originally, the Fred project started in building on top of the concepts of frameworks and design patterns, but since then the work has been related with various approaches, including pattern languages and feature models [Hak02a], graph grammars [Hak01a], software architectures [Hak01b], aspects

[HKK04], genericity [Hak99], software maintenance [HaH02], concept lattices [Vil04], and even framework documentation [Hak02b]. Each of these domains have borrowed some of their terminology for describing the central concepts of Fred, but never without misunderstandings. Thus, throughout the years the Fred terminology has been in constant change. The specifications fed to the Fred tool, first called patterns, have also been referred to as templates, contracts, frames, recipes, graphs, programs, features, pattern languages and forms. This suggests that Fred has succeeded in capturing something central to many approaches, but it also tells about the confusion and the lack of a stable viewpoint that has bothered the project group along the way.

The publications of this thesis are organized chronologically. Hence, it is possible to follow the evolution of both the ideas and the terminology.

Chapter 2. Pattern-Oriented Framework Engineering Using FRED [Hak98] introduced the Fred approach for the first time. The design patterns and application frameworks serve as the conceptual framework for the discussion. The ideas of this paper are sketchy and it is questionable whether the paper was able to communicate the early thoughts of the project group. Moreover, at that time there was strong resistance against tool support for design patterns, which made the project group rethink about the terminology for the first time.

Chapter 3. Managing Object-Oriented Frameworks with Specialization Templates [Hak99] switches the vocabulary from patterns to templates to avoid conflicts with the design pattern community. The ideas have matured a lot from the first position paper, but basically it is the same story for another audience.

Chapter 4 Task-Based Tool-Support for Framework Specialization [Hak00] describes the incremental algorithm that is used by the Fred tool. The paper speaks about generative patterns in an attempt to emphasize the creational aspects of the adopted pattern concept in contrast with pattern matching and design-patterns in general. This was the first attempt to put down something more formal from the inner workings of the tool. At the same time the paper goes through the user experience of the tool. This proved to be significantly better in sharing the vision than trying to describe the tool solely in a conceptual level.

Chapter 5. Generating Application Development Environments for Java Frameworks [Hak01a] goes through a step-by-step case where the developer is using the tool in specializing a small example framework. Thereafter the paper continues in describing the underlying model and

the required algorithm using the notion of graph grammars [EhT96].

Chapter 6. Annotating Reusable Software Architectures with Specialization Patterns

[Hak01b] communicates pretty much the same issues as the previous one but relating them with a different background. It was published soon after, but it emphasized different aspects of the approach mainly due to the different audience. The concepts and terminology hadn't changed much since the last paper, and it seemed the model was finally passed its early infantry.

The last two of these papers constitute the final results of the original Fred project. Thereafter, the project continued under a different name, refining the tool to incorporate more flexibility. This was desperately required in order to apply the tool in real-life situations.

Chapter 7. Feature Models, Pattern Languages and Software Patterns: Towards a Unified

Approach [Hak02a] presents the author's view on the possibilities of the Fred approach, some of which has not yet been harnessed. It relates the Fred model with feature models [Kan90] and pattern languages [Ale79, MRB98], and discusses how the model needs to be extended in order to bridge the approaches.

Unfortunately, at the time of writing, the model of reuse introduced by the Fred tool is yet unfinished and immature. The underlying model works on an abstract level and it has been applied over and over to several unanticipated domains. It has been a huge effort to come this far, but the model is still lacking flexibility. Most importantly, since it's infantry the model has been plagued with strong ties with a particular tool implementation, and the lack of formal background. The best formalization of the model is in the form of Java code making up the tool itself.

The second question about Fred is its practicality. In theory there shouldn't be a problem with this as the tool has been developed by a group of researchers with very practical orientation, aimed to solve problems of their own everyday lives as programmers. However, the tool has never been actually used by these people in their own work, the very thing the tool was supposed to improve. The biggest mistake of Fred may be in its basic unspoken assumption of the world as a clean, organized and static environment. In reality, the world is messy, unorganized and dynamic, quite likely to be too alive and evolving for the rigid reusability models of Fred to handle. Even more so with the emergence of agile processes that take change and code refactoring as the core premises of the contemporary software development. It remains to be seen whether the Fred model can be improved to better incorporate change.

All this is not to say that there weren't a significant contribution in what has been done within the Fred and JavaFrames projects. It is very likely that the Fred model has succeeded in capturing essential aspects of forms that keep recurring over and over in the software. However, the attempts in introducing the core ideas to the academic community can't be characterized as overly successful. Lacking solid foundation, Fred hasn't provided much for the other research groups to build on. It is evident that what this line of research urgently needs is a formal model and better evidence of the usability of the tool. A formal model would create a well-defined workspace enabling researchers to build on top of each others work, debate, argue and relate their findings. Relating this to reality with concrete experiments is secondary, and will remain a question of faith unless clear statistics can be provided, or the tool will become practical and polished enough to gain wide acceptance amongst the developer community.

Several Ph.D. thesis have been published around the subject [Hau05, Vil04, Vil05]. The author's own work on Fred has ended, but there are several ongoing projects applying the tool for various originally unanticipated domains [Hak02b, HaH02, HKK04].

References

- Ale79 Alexander C.: *The Timeless Way of Building*. Oxford University Press, New York, 1979.
- EhT96 Ehrig H., Taentzer G.: *Computing by Graph Transformation: Survey and Annotated Bibliography*, *Bulletin of the EATCS*, 59, June 1996, 182-226.
- FSJ00 Fayad M.E., Schmidt D.C, Johnson R.E.: *Building Application Frameworks - Object-Oriented Foundations of Framework Design*. John Wiley & Sons, 2000.
- Gam95 Gamma E., Helm R., Johnson R., Vlissides L.: *Design Patterns - Elements of Object Oriented Software Architecture*. Addison-Wesley, 1995.
- HaH02 Hammouda, I., Harsu, M.: *Documenting Maintenance Tasks Using Maintenance Patterns*. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pp. 37-47, Tampere, Finland, March 2004.
- Hak98 Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J.: *Pattern-Oriented Framework Engineering Using FRED*. In: *Object-Oriented Technology, Springer LNCS 1543*, 1998, 105-109.
- Hak99 Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J.: *Managing Object-Oriented Frameworks with Specialization Templates*. In: *Workshop on Object Technology for Product-Line Architectures*. European Software Institute, Spain, 1999,

87-98.

- Hak00 Hakala M.: Task-Based Tool Support for Framework Specialization. In: Proceedings of OOPSLA'00 Workshop on Methods and Tools for Framework Development and Specialization, Tampere University of Technology, Software Systems Laboratory, Report 21, October 2000.
- Hak01a Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Generating application development environments for Java frameworks. In: Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering (GCSE'01), Erfurt, Germany, September 2001, Springer, LNCS2186, 163-176.
- Hak01b Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J.: Annotating Reusable Software Architectures with Specialization Patterns. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), Amsterdam, August 2001, 171-180.
- Hak02a Hakala M.: Feature Models, Pattern Languages and Software Patterns - Towards a Unified Approach. In: Proceedings of the Nordic Workshop on Software Development Tools and Techniques (K. Osterbye, ed.), NWPER'2002, IT University of Copenhagen, 2002.
- Hak02b Hakala M., Hautamäki J., Koskimies K., Savolainen P.: Generating Pattern-Based Documentation for Application Frameworks. In: Proceedings of the Nordic Workshop on Software Development Tools and Techniques (K. Osterbye, ed.), NWPER'2002, IT University of Copenhagen, 2002.
- Hau05 Hautamäki J.: Pattern-Based Tool Support for Frameworks: Towards Architecture-Oriented Software Development Environment. Ph.D. Thesis, Tampere University of Technology, Publication 521, 2005.
- HKK04 Hammouda I., Katara M., Koskimies K.: A Tool Environment for Aspectual Patterns in UML. In Workshop on Directions in Software Engineering Environments (WoDiSEE 2004), Edinburgh, Scotland, UK, May 2004.
- Kan90 Kang K., Cohen S., Hess J., Nowak W., Peterson S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990.
- MRB98 Martin R., Riehle D., Buschmann F.: Pattern Languages of Program Design. Addison-Wesley, 1998, 163-185.
- Pre95 Pree W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.

- Vil04 Viljamaa J., Applying Formal Concept Analysis to Extract Framework Reuse Interface Specifications from Source Code. PhD Thesis, Report A-2004-5, Department of Computer Science, University of Helsinki, 2004.
- Vil05 Viljamaa A., Specifying Reuse Interfaces for Task-Oriented Framework Specialization. PhD Thesis, University of Helsinki, Department of Computer Science, Series of Publications A, Report A-2005-??, 2005.

Chapter 2.

Pattern-Oriented Framework Engineering Using FRED

Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J.: Pattern-Oriented Framework Engineering Using FRED. In: Object-Oriented Technology, Springer LNCS 1543, 1998, 105-109.

PATTERN - ORIENTED FRAMEWORK ENGINEERING USING FRED

Markku Hakala^{*}, mh@cs.uta.fi
Juha Hautamäki^{*}, csjuha@uta.fi
Jyrki Tuomi^{*}, jjt@cs.uta.fi
Antti Viljamaa[†], antti.viljamaa@cs.helsinki.fi
Jukka Viljamaa[†], jukka.viljamaa@cs.helsinki.fi

Abstract

Application frameworks are reusable architectures used to improve the software development process. Although design patterns may be used in designing, implementing, and documenting architectural constructs, lacking a systematic approach frameworks are still hard to design and reuse. This paper presents the methodology and the set of supporting tools that provide a methodical practice for applying design patterns in software engineering, especially when constructing reusable designs such as application frameworks.

Keywords

Framework, Design Pattern, FRED, CASE Tool, Software Engineering, Java

INTRODUCTION

An application framework is a set of objects that captures the special expertise in some application domain to a reusable form [Lew95]. An application can be specialized from this skeleton by adding new features to it.

Frameworks are hard to design and reuse. However, design patterns [GHJ95, CoS95] can be used to capture the knowledge of framework experts in an easily accessible way. Design patterns name and explain architectural constructs that may be found in number of architectures. A design pattern is an abstract solution to a general design problem, and may be reused by individual developers as building blocks for new frameworks and applications. [Joh92, BMR96]

FRED (Framework Editor) is a development environment, especially designed for framework development and specialization. In addition of being a development tool, FRED introduces a uniform model of a software architecture and software development that makes heavy use of generalization of design patterns.

The FRED environment has been implemented with Java programming language [ArG98] and is used in developing Java applications and frameworks. However, the model behind the environment is not tied to any specific programming language or tool set.

FRED is an ongoing project between the departments of Computer Science at the University of Tampere and University of Helsinki, supported by TEKES (Technol-

ogy Development Centre, Finland) and several Finnish industrial partners.

FRED MODEL

Both frameworks and applications are software architectures. FRED, as a development environment, is a tool for creating such architectures. In FRED, an architecture is always created based on another architecture or architectures. A typical example is an application that is derived from an application framework.

Designated by the object-oriented domain, each architecture eventually consists of classes and interfaces, which in turn contain fields (Java synonym for attributes) and methods. Also, the term *data type* is used to refer to both classes and interfaces.

Data types alone are insufficient to represent architectural constructs when reusability is essentially required. They do not provide enough documentation for the architecture, nor control the specialization of the architecture. To meet these two requirements, *pattern* is hereby defined as a description of an arbitrary relationship between number of classes and interfaces. Patterns range from generic design patterns to domain and even application specific patterns. A pattern is an architectural description, but needs not to be general. In this context, general constructs such as those listed by Gamma *et al.* [GHJ95] are called *design* patterns. No distinction between patterns on the basis of their generality is made in FRED.

Patterns are used to couple together arbitrary data types that participate in a particular design decision or

^{*} University of Tampere, Department of Computer Science, P.O. Box 607, FIN-33101 Tampere, Finland

[†] University of Helsinki, Department of Computer Science, P.O. Box 26, FIN-00014 University of Helsinki, Finland

architectural feature. This kind of coupling of data types provides structural documentation for the architecture. Any data type may participate in more than one pattern, in which case it plays several roles in the architecture.

Structures

An architecture is a complex construction of patterns, data types and both their static and dynamic relations. Structural elements of an architecture, such as patterns, data types, methods and fields are called *structures*. Also, the architecture itself is a structure. Adopting the general concept of structure simplifies both the discussion and internals of the FRED environment.

Structures are divided into composite and leaf structures. Architectures, patterns and data types are composite structures, which may contain other structures. An architecture contains patterns, patterns contain data types, and data types in turn contain methods and fields, which are leaf structures in the sense that they do not contain other structures.

Based on these relations, a directed acyclic graph can be presented for an architecture. Structures of the architecture are nodes of this graph, and edges range from composite structures towards the leaf structures. The graph has single source, namely the architecture node, and multiple sinks representing the leaf structures. As a data type can belong to a number of patterns, the graph is not a tree. In FRED development environment, the graph is however visualized with a tree-like notation. In this notation, the root equals the source of the graph. Furthermore, a data type contained in multiple patterns appears in the tree at each containing pattern, but only methods and fields relevant to a particular pattern are shown. Based on this notation, a structure containing another structure is called a *parent* of the latter. The parent of a structure is naturally not necessarily unambiguous.

An example architecture is shown in figure 1 with both graph representation and the corresponding FRED notation.

Templates

All structures may be classified as *implementations* or *templates*.

An implementation is a structure that is actually implemented in the architecture. In a conventional application, all structures are essentially implementa-

tions. From the developer's point of a view, an implementation is a "normal" structure, in contrast to the concept of template.

A template defines a set of possible implementations, but does not specify the actual implementation. In a way a template is a blueprint of an implementation. Providing a template in an architecture means defining a gap that must be filled in by the developer deriving from the architecture.

Templates are structures just like implementations. An architecture template contains patterns, a pattern template contains data types, and data type templates contain methods and fields. Architecture and pattern templates may contain both templates and implementations, but data type template contains only templates. If a structure contains a template, it is itself a template.

In FRED, a structure (including architecture) is always based on other structures, namely templates; every structure is based on corresponding meta-structure and in addition to some other templates. There is a meta-structure for each type of structure. Meta-structures are based on themselves.

Templates are used in creating new structures. This is called *instantiating* the template. The instantiated template is called a *model* in relation to its *instance*. For instantiation purposes a template provides the following properties:

1. Free-form hyperlinked documentation that guides in creating an instance for the template.
2. Parameterized default implementation that automatically adjusts to the instantiation environment.
3. Tools for instantiating the template.
4. Constraints that all its instances must conform to.

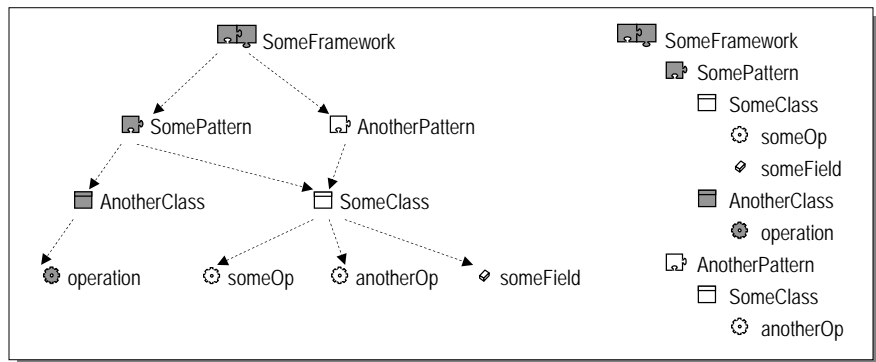


Figure 1. An example architecture as a directed graph and using FRED tree-like notation.

Template is instantiated by introducing a structure (an implementation or a new template) that is conformable to the model. This instance can be created using the documentation, default implementation, and tools provided by the template. By default the FRED environment includes tools for creating and editing structures, but the architecture developer may also provide specialized domain-specific tools.

Once a structure is created the default tool or tools suggested by the template can be used to modify the structure. Constraints are used to ensure that the instance conforms to the template. For example, a constraint may state how many times the template must be instantiated with respect to its parent template (cardinality constraint). See appendix A for more examples on constraints.

An *automatic* template is a template that is instantiated automatically whenever its parent structure is instantiated. Instantiating an automatic template may however require user interaction, e.g., when all the parameters of the default implementation cannot be resolved. Automatic or tool-generated structures are updated according to the modifications in surrounding code. The developer cannot access the automatically maintained code fragments unless explicitly requested.

Patterns Using Templates

By the earlier definition, a pattern describes a relationship between participating data types. In FRED, a pattern is described using templates.

A pattern template couples together data type templates and data type implementations. The constraints of the contained templates define the required relationships between collaborating structures. The default implementation makes it easy to instantiate a pattern in a software architecture. Hyperlinked documentation attached to the templates provides the required documentation for the pattern and its collaborations. In addition, specialized tools may be provided. For example a template representing the Interpreter design pattern [GHJ95] may be accompanied with a tool that takes a grammar as input and parameterizes the default implementation with it.

Instantiating a (design) pattern means binding the domain-specific vocabulary and implementation. Frameworks usually provide only partial implementations for design patterns and leave specific parts to be supplemented by the specialist. In FRED this means providing templates that instantiate the original templates of the

pattern. This instantiation chain may be arbitrary long for any structure. Constraints of a template apply to all following instances in the chain. Thus constraints cumulate and the set of possible implementations becomes smaller in every instantiation. This implies kind of inheritance hierarchies for frameworks and design patterns. Layered frameworks are discussed for example by Koskimies and Mössenböck [KoM95].

A BRIEF EXAMPLE

To illustrate how the FRED model works in practice, a simple framework is implemented and an application is derived from it. The example is based on the sample *framelet* presented by Pree and Koskimies [PrK98]. In their paper they define framelets as small architectural building blocks that can be easily understood, modified, and combined. Here the example framelet appears somewhat modified. The reader may consult appendix B when going through the following example.

Many graphical applications provide list boxes together with buttons to add items to the list box, and to modify and remove them. Typically a separate dialog is used to add and modify items. The associated programming task can be packed into a small self-contained framework, called List Box Framelet. The framelet class diagram and user interface are shown in figure 2.

The framelet provides a window (ListBoxFrame) to display and maintain a list of items. An application developer provides item and dialog classes. Item classes must implement the Item interface and dialog classes must extend the Display base class. Transformer class is used internally by the framelet to manage the data transfer between FieldAccessor instances, namely items and displays.

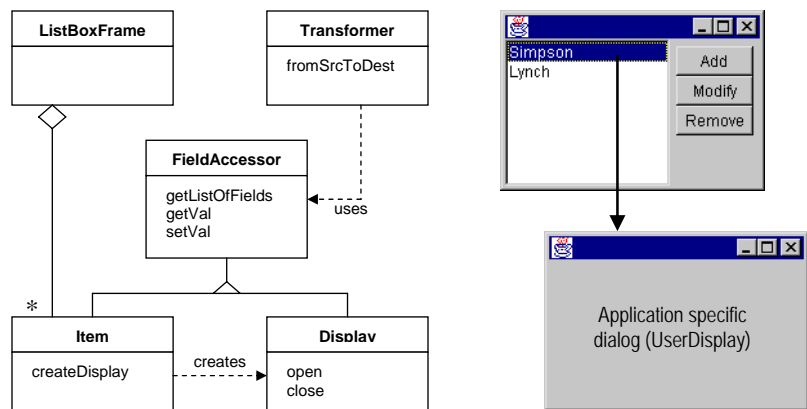


Figure 2. Class diagram and user interface for List Box Framelet.

Creating the Framelet

In FRED List Box Framelet must be presented with structures. In the following example, names of structures are printed with sans serif font (e.g. ListBoxFramelet).

As an architecture, List Box Framelet must be based on another architecture. FRED environment provides a special architecture called PatternCatalog, which collects arbitrary design patterns by several authors. PatternCatalog can be expanded by individual developers. The root structure of the framelet, an architecture called ListBoxFramelet, is based on PatternCatalog.

Among other things the catalog contains patterns AbstractFactoryPattern, SingletonPattern, and BeanComposition. The first two are representations of design patterns Abstract Factory and Singleton, which are discussed by Gamma *et al.* [GHJ95]. The BeanComposition in turn stands for the Bean Composition pattern, which represents a composition of Java bean components [Sun97] and is developed within FRED project.

After browsing the documentation of the catalog, the developer chooses to instantiate AbstractFactoryPattern to implement the framelet's behavior. The developer names the instance as ItemPattern. To implement the created pattern, all of the templates contained in the model pattern must be instantiated according to their cardinality. AbstractFactoryPattern contains data type templates AbstractFactory, ConcreteFactory, AbstractProduct and ConcreteProduct. The developer creates the Item interface and the Display class based on the AbstractFactory and AbstractProduct templates. Implementation of ConcreteFactory and ConcreteProduct the developer wants to leave to the specializer. For this purpose, the developer creates templates UserItem and UserDisplay, respectively.

All structures are also based on the corresponding meta-templates. Similarly, ItemPattern is based on meta-pattern. For this reason, the developer can augment ItemPattern by introducing FieldAccessor interface by instantiating it from meta-datatype. Likewise, a number

of methods and fields can be created that are not based on method and field templates contained in AbstractFactoryPattern.

Data types that are not involved in any specific pattern, are contained in a special OtherClasses pattern, which is automatically generated for an architecture. The developer creates the Transformer class in that pattern, but soon realizes that SingletonPattern may be used here. The developer creates a new SingletonPattern instance called TransformerPattern and associates the existing Transformer class as an instance of the Singleton template. Members required by the Singleton template may be pointed out similarly from the existing code. The created structures and their relations to their models are shown in figure 3.

After creating most of the framelet logic, the developer begins to implement the user interface. This can be done by instantiating BeanComposition pattern, which is equipped with a FRED Java Beans builder tool that generates appropriate code for the user interface.

Because the framelet contains a template (ItemPattern) it is itself a template which can be instantiated. The developer chooses to make ItemPattern automatic implying that once the framelet is instantiated, the ItemPattern template is also instantiated. Finally, the developer provides guidance for the framelet specialization by documenting the contained templates.

Specializing the Framelet

Any architecture that is a template can be derived from by another architecture. E.g., List Box Framelet can be used to create an application to create, modify, and remove customer contact entries. The Contact Manager application may be based on multiple architectures, but for simplicity, only the List Box Framelet-based portion is discussed here.

The developer names the root structure of the application as ContactManager. At first the application developer makes it an instance of ListBoxFramelet. As ItemPattern is an automatic template, the environment auto-

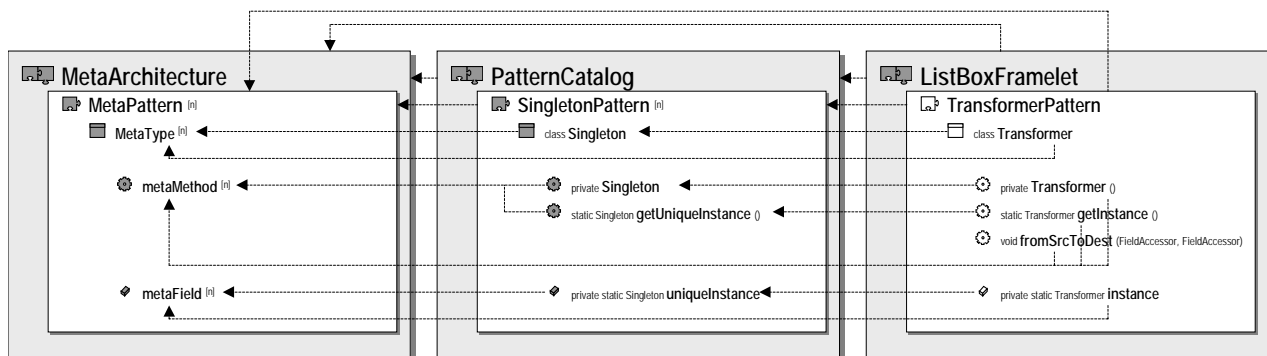


Figure 3. TransformerPattern is based on SingletonPattern. All structures are based on the corresponding meta-structures.

matically creates an instance of it. The developer changes the name of the instance to ContactPattern.

ItemPattern contains template classes UserItem and UserDisplay. These are not instantiated automatically. Hence, the application developer creates the Contact class based on UserItem and ContactDisplay based on UserDisplay. Substructures (methods and fields) of the templates are instantiated similarly. Template specific documentation and constraints guide the developer during this instantiation process.

To implement the user interface, the application developer may use the BeanComposition pattern. To do this, the developer will need to point out the application as an instance of PatternCatalog and instantiate the pattern. The developer uses the pattern to create user interface for ContactDisplay. For this purpose ContactDisplay is made an instance of the Composition template. Thereafter ContactDisplay is an instance of two templates. The Java Beans builder tool is used to generate the user interface –specific code for the ContactDisplay class.

As a result, the Contact class represents a contact entry and the ContactDisplay dialog is used by the framelet to add and modify contacts. Because there are no violated constraints, the developer chooses to test Contact Manager by implementing a class with the required main method. The developer creates the class in the OtherClasses pattern.

FRED Tool

Although the FRED model is not tied to any specific tool, the model requires a sophisticated user interface in order to lessen cognitive load on the developer. The user interface should adjust to the level of abstraction of the architecture by hiding the internals of the archi-

itecture and using more specific vocabulary when necessary. For this reason, FRED is as much of a user interface issue as it is a conceptual model, though only the latter has been discussed here.

However, part of the user interface for FRED development environment is shown in figure 4. The window is divided into three parts. Hierarchies on the left are used to browse the architecture from different aspects. Structures are modified on the desktop area with the template tools, and the lower part displays documentation for the selected hierarchy item. The same environment can be used in developing architectures ranging from applications to frameworks and even abstract pattern catalogs.

In figure 4 the developer is implementing the ContactManager architecture. The ContactDisplay class has been selected on the left. Models of the selected structure are listed below with their associated tasks. *Tasks* are user interface representations of unimplemented templates and broken constraints. UserDisplay is listed as a model because ContactDisplay is based on it. The associated tasks state that the developer has to instantiate the getVal and setVal method templates.

CONCLUSION AND RELATED WORK

Many design pattern tools (see, e.g., [BFV96], [ACL96], and [Wil96]) use macro expansion mechanisms to generate implementation code. This implies design – implementation gap [MDE97]: changing generated code breaks the connection between design patterns and the implementation and any changes that involve regeneration of code will force the user to manually reintegrate the pattern code to the rest of the system. We think that a better way is to use explicit representation of (design) patterns that stays at the background all the way from design to implementation.

Furthermore, mere code generation is not enough. It is essential to be able to combine multiple pattern instances and to annotate existing code with patterns. Our implementation of this role binding functionality is influenced by Kim and Benner’s Pattern-Oriented Environment (POE) where one can manage and validate pattern instances and their role mappings [KiB96].

Another similar tool is represented by Florijn, Meijers, and van Winsen [FMW97]. Their model is based on cloneable fragments that represent design elements (design patterns, charts), implementation entities (classes, methods), or other objects (comments, arbitrary

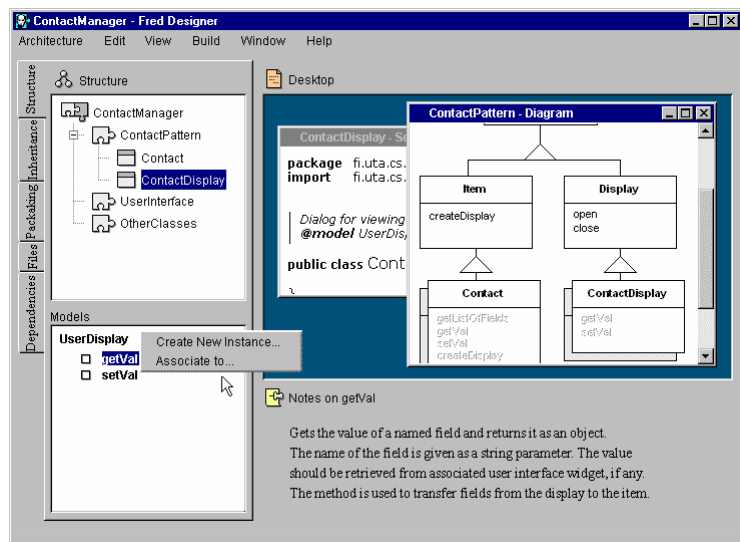


Figure 4. The user interface of the FRED development environment in action.

text). The system being developed is represented as a graph of interrelated fragments of different types.

Besides supporting development of frameworks, the FRED environment also aids framework users. Instantiating any structure in FRED involves customization of default implementations within the bounds of constraints associated with the structure (in a specialized *visual* or *active text* [MöK95] editor) guided by the associated documentation. This makes, e.g., deriving an application from a framework a systematic process.

One major shortcoming of the adopted model is that constraints defined in the templates apply only to method and field signatures, data types, and larger constructs of the architecture. E.g., FRED environment cannot verify that the code within a method body is valid while it can ascertain that the method's return type is correct.

As simplicity for the end-user was one of the main concerns of FRED, the model is more than adequate. Extensive systems involving formalisms like predicate logic tend to chase developers away. FRED model hopefully provides a relatively easy approach to pattern-oriented software engineering.

Some of the functionality referred to in this paper is not implemented in the current version of the FRED environment. In addition to making the FRED environment fully operational, future directions of project include:

- Pattern selection wizard to assist the user in finding a solution to a design problem.
- Support for pattern mining by associating an appropriate tool for a pattern template.
- An automatic mechanism for creating customized wizards to assist the user in framework specialization.
- Importing Java standard extensions and similar popular frameworks (by JavaSoft) into the FRED model.
- Visualization of both static and dynamic relations within an architecture.

ACKNOWLEDGEMENTS

FRED project is supported by TEKES (Technology Development Centre, Finland) and Finnish industrial partners: Dycom, ICL Data, Major Blue Company, Nokia Telecommunications, Nokia Research Center, Novo Group, Profit, Stonesoft, Sun Microsystems, Telecom Finland, TT Tieto, and Valmet Automation.

REFERENCES

- ACL96 Alencar P., Cowan D., Lichtner K., Lucena C., Nova L.: Tool Support for Design Patterns. Internet: <ftp://csg.uwaterloo.ca/pub/ADV/theory/fmsp96.ps.gz>.
- ArG98 Arnold K., Gosling J.: *The Java Programming Language*, 2nd ed. Addison-Wesley, 1998.
- BFV96 Budinsky F., Finnie M., Vlissides J., Yu P.: Automatic Code Generation from Design Patterns. *IBM Systems Journal* 35, 2, 1996, 151-171.
- BMR96 Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: *Pattern – Oriented Software Architecture, A System of Patterns*, John Wiley & Sons Ltd, 1996.
- CoS95 Coplien J., Schmidt D. (eds.): *Pattern Languages of Program Design*. Addison-Wesley, 1995.
- FMW97 Florijn G., Meijers M., van Winsen P.: Tool Support for Object-Oriented Patterns. *Proc. ECOOP '97 European Conference on Object-Oriented Programming*, Jyväskylä, Finland, June 1997, Lecture Notes in Computer Science 1241, Springer-Verlag, 1997, 472-495.
- GHJ95 Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- Joh92 Johnson R.: Documenting Frameworks with Patterns. In: *Proc. OOPSLA '92, ACM SIGPLAN Notices* 27, 10, 1992, 63-76.
- KiB96 Kim J., Benner K.: An Experience Using Design Patterns: Lessons Learned and Tool Support. *Theory and Practice of Object Systems (TAPOS)* 2, 1, 1996, 61-74.
- KoM95 Koskimies K., Mössenböck H.: Designing a Framework by Stepwise Generalization. In: *Proc. of ESEC'95, Lecture Notes in Computer Science* 989, Springer-Verlag, 1995, 479-497.
- Lew95 Lewis T. (ed.): *Object-Oriented Application Frameworks*, Manning Publications Co., 1995.
- MDE97 Meijler T., Demeyer S., Engel R.: Making Design Patterns Explicit in FACE — A Framework Adaptive Composition Environment. In: *Proc. 6th European Software Engineering Conference*, Zurich, Switzerland, September 1997, Lecture Notes in Computer Science 1301, Springer-Verlag, 1997, 94-110.
- MöK95 Mössenböck H., Koskimies K.: Active Text for Structuring and Understanding Source Code. *Software Practice & Experience* 26(7), July 1996, 833-850.
- PrK98 Pree W., Koskimies K.: *Framelets - Small and Loosely Coupled Frameworks*. Manuscript, submitted for publication, 1998.
- Sun97 Sun Microsystems Inc.: *JavaBeans Documentation*. Internet: <http://java.sun.com/beans/docs>
- Wil96 Wild F.: Instantiating Code Patterns — Patterns Applied to Software Development. *Dr. Dobbs' Journal* 21, 6, 1996, 72-76.






APPENDIX A – FRED NOTATION AND CONSTRAINTS

In FRED, the structural graph of an architecture is presented with a tree-like notation consisting architecture, pattern, data type, method, and field nodes. Each node contains an image indicating the type of the structure and the name. Templates come with dark gray images and implementations with white ones. Other information, such as implementation information or constraints may also be presented with a node.

In the following table, notation for template nodes and some of their most important constraints are presented. Notation for implementations follows closely the syntax of Java language and is not discussed. Also, more specific constraints and parameterized default imple-

mentations are bypassed as they are not included in this notation.

Some of the constraints are interpreted differently in an overriding situation. For this purpose, *constraint signature* for a method is defined as collection of constraints for return type, parameters, and exceptions. When a constraint signature of a method template equals a signature or constraint signature of another method in any of the supertypes, the method is said to override the other method. The signature of its instance should be exactly the same as that of the overridden method (or its instance).

	NOTATION	ALLOWED MODIFIERS
Architecture	 name	
Pattern	 modifiers name [cardinality]	automatic
Data Type	 modifiers kind name [cardinality] supertypes	automatic, public, pkgprivate, abstract, final
Method	 modifiers return_type name [cardinality] (parameters) exceptions	automatic, public, pkgprivate, protected, private, static, abstract, final, synchronized, native
Field	 modifiers return_type name [cardinality]	automatic, public, pkgprivate, protected, private, static, final, transient, volatile

cardinality

Cardinality states how many times a template must be instantiated with respect to its parent template. By default, cardinality is one and need not to be shown. The notation [n] is a shortcut for [0..n] and means that the template can be instantiated any number of times.

modifiers

Modifiers are equal to Java-modifiers, supplemented by a few additional keywords. However, the meaning of a modifier for a template is different from the meaning of a modifier for an implementation; the existence of a modifier in template sets a constraint on the structure. E.g., *automatic* keyword means that the structure is instantiated automatically once its parent is instantiated, and *private* means that the instantiated structure must be private. If there are no modifiers present, no associated constraints are applied.

kind

A special modifier for a data type is its kind. A kind may be *class* or *interface*. It defines whether the instance data type should be class or interface, respectively. If missing, the instance can be either one, assuming that other constraints are not violated.

supertypes

Required supertypes for an instance may be indicated by *extends*, *implements*, and *inherits* clauses. By using the latter alternative, the type of inheritance is not fixed and subtypes of the mentioned data type are also accepted.

return_type

Both methods and fields can be associated with a return type. Existence of a return type states that the instance should return the mentioned type or any of its subtypes.

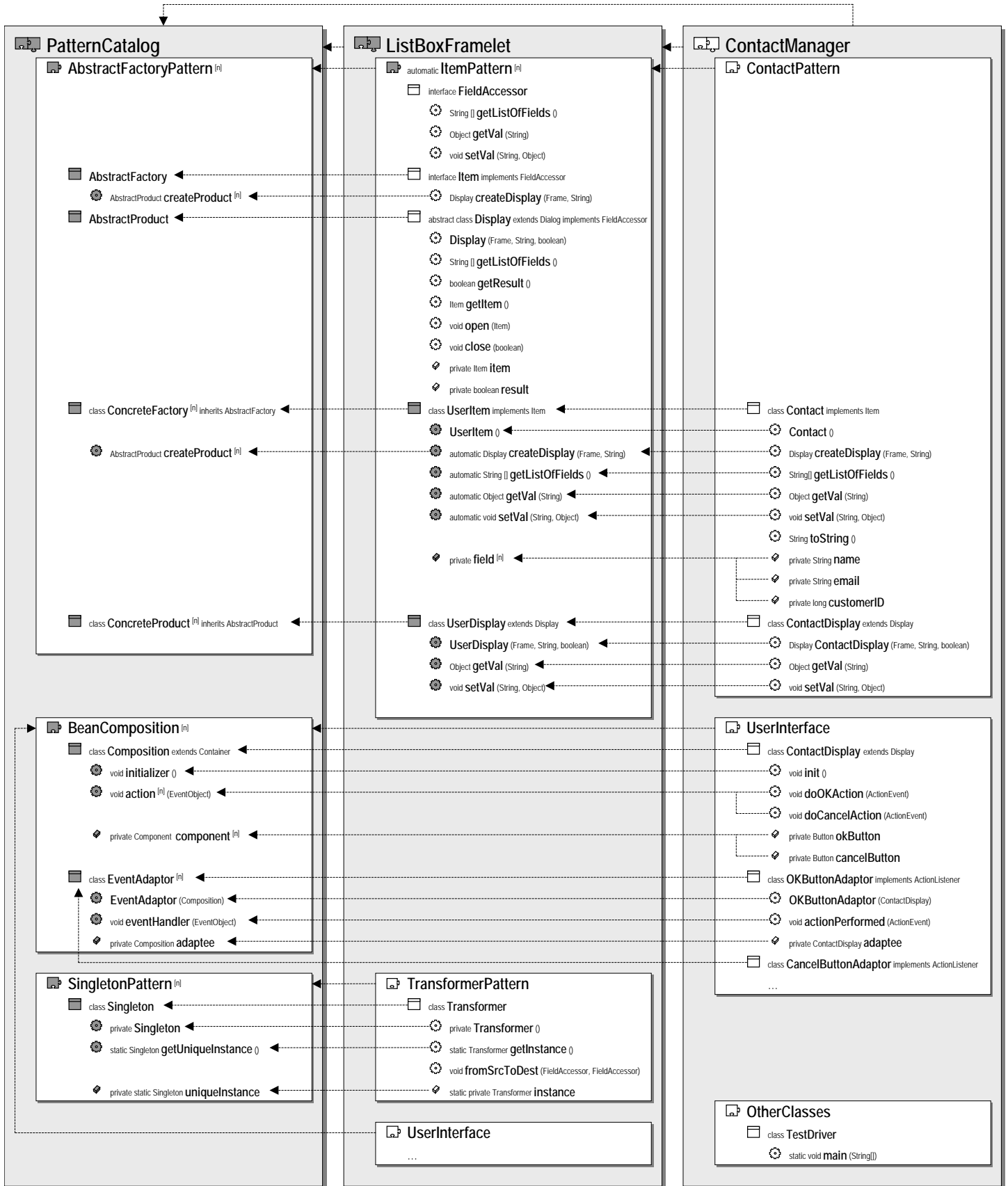
parameters

Formal parameters may be left unconstrained for a method template. Similar to the return type constraint, existence of a formal parameter states that the method should accept a parameter of the mentioned type or any of its subtypes. At the same time the order of parameters is fixed.

exceptions

Like parameters, a method can be constrained for the exceptions it must introduce in its throws clause. Exceptions-constraint follows the syntax of throws-clause in Java-language.

APPENDIX B – EXAMPLE ARCHITECTURES



For simplicity, meta-structures are not shown in this example.

Chapter 3.

Managing Object-Oriented Frameworks with Specialization Templates

Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J.: Managing Object-Oriented Frameworks with Specialization Templates. In: Workshop on Object Technology for Product-Line Architectures. European Software Institute, Spain, 1999, 87-98.

Managing Object-Oriented Frameworks with Specialization Templates

Markku Hakala¹, Juha Hautamäki¹, Jyrki Tuomi¹, Antti Viljamaa², Jukka Viljamaa²,
Kai Koskimies³, Jukka Paakki²

Abstract. The concept of a specialization template is introduced. It is shown that specialization templates are useful in the design and documentation of object-oriented frameworks. A prototype tool for Java frameworks is implemented based on a formalization of a specialization template. The tool helps a framework developer to manage the hot spots and the specialization rules of the framework, and an application developer to specialize the framework. The approach has been applied to a realistic case study, a Java framework for developing graphical editors.

1 Introduction

Within the object-oriented paradigm, *framework* has become a popular catchword for product-line architecture. Nonetheless, frameworks are generally not well understood as software products. The term framework is used for a wide spectrum of architectures without always recognizing the features that actually characterize a framework. The design process of a framework is currently not mastered by software professionals, a fact harshly revealed by the long, unpredictable and iterative development projects of frameworks. The use of a framework for producing an application is often problematic due to the mere size and complexity of the framework, making it hard to understand exactly which parts must be specialized and how. Tool support for making the development and use of frameworks easier is just taking its first steps ([FMW97], [MDE97]). Taking into account the lack of understanding of the framework concept itself, it is hardly surprising that the tool support is inadequate so far.

There are both impressive success stories and (less published) crushing catastrophes in making use of frameworks in the industry. For mainly positive experiences, see e.g. [Cas95], [SBF96], [Joh92], [Lew95] and [CACM97]. We feel that frameworks have the potential for truly powerful software reuse, but as everything incompletely understood they also contain a significant risk. In this paper we will propose a new concept, *specialization template*, that helps to understand a framework and its design process. We also show that this concept can be used as the basis of tool support for frameworks.

¹ University of Tampere, Box 607, 33101 Tampere, Finland (email {mh, csjuha, jjt}@cs.uta.fi)

² University of Helsinki, Box 26, 00014 University of Helsinki, Finland (email {antti.viljamaa, jukka.viljamaa, jukka.paakki}@cs.helsinki.fi)

³ Tampere University of Technology, Box 553, FIN-33101 Tampere, Finland (email: kk@cs.tut.fi)

In the next section we will introduce the concepts of design contract and specialization template as basic design artifacts and documentation means for a framework. In section 3 we discuss the role of specialization templates in the design of a framework. The basic design of a practical tool that builds on the idea of a specialization template is presented in section 4. Finally, we summarize the benefits of the approach in section 5.

The tool presented here is FRED (Framework Editor), a recent result of research carried out at the universities of Helsinki and Tampere. An early, significantly different version of FRED was reported in [Hak98]. As a realistic case study, FRED has been applied to JHotDraw, a Java framework for graphical editors [Gam98]. FRED and the JHotDraw application (along with a tutorial) are freely available at <http://www.cs.helsinki.fi/research/fred>.

2 Design contracts and specialization templates

Basically, a *design contract*⁴ is a variant of the contract concept introduced by Holland [Hol93]. It is also closely related to a design pattern [Gam95]. A design contract is an application-independent description of a certain OO design aspect. In contrast to a traditional design pattern, it does not necessarily describe a solution to a frequently occurring problem (or to any "problem" for that matter). In a sense, a design contract is a design pattern released from its semantic burden. It describes simply a set of classes, together with certain attributes, methods and relations. Further, a design contract specifies the (structural) conditions that any realization of the contract must fulfill. Naturally any design pattern can be easily converted to a design contract, but not vice versa.

The idea of a design contract is close to Holland's notion of a contract. Since we want eventually to use design contracts as the basis of our tool, we must formalize them as a language, resembling Holland's Contract language. The differences stem from our aim to support the specialization process using the information in a design contract, rather than to specify general reusable program fragments. When compared to Holland's contracts, design contracts are less specific about the method bodies (algorithms). Further, we introduce static constraints that can be used as guidance in the specialization phase, rather than run-time invariants. All this means that we consider contracts as static, structural specifications rather than as semantically meaningful abstract program fragments.

Because of its character, the informal description of a design contract is more compact than that of a design pattern. The central parts of a design contract are:

⁴ This concept is not directly related to "design-by-contract" as proposed by Meyer.

- *Name*: the name that identifies the design contract.
- *Base contract*: the possible design contract, which this design contract is derived from.
- *Structure*: the static structure of the participants of the contract, in terms of a UML class diagram. The names appearing in the structure are *roles*, i.e. placeholders that can be bound to actual code items.
- *Constraints*: the constraints that must be followed in any realization of the structure, when the roles of the structure are bound to application specific items.

Often design contracts can be organized into hierarchies in which a more specific contract extends a more general contract, adding certain classes, attributes, methods (roles) or constraints. For example, there might be a simple design contract consisting of two classes with inheritance relationship; this contract could then be extended by another contract with a structure more specific to it. The possible design contract serving as the base of extension is given in the "base contract" part. This is equivalent to Holland's notion of "refinement".

We describe design contracts formally using a textual language called Cola (Contract language). To get some idea of the flavor of Cola, consider the following (somewhat simplified) Cola specification of the Composite design contract:

```
contract Composite {
  single type Component { method operation ; }
  class Leaf inherits <Component C> {
    named "Leaf" + C ;
    single method operation overrides <C~operation O> ;
  }
  single class Composite inherits <Component C> {
    named "Composite" + C ;
    single method operation overrides <C~operation O> {
    }
    single method add {
      takes C child ;
    }
    single method remove {
      takes C child ;
    }
    single method get {
      returns Component ;
      takes int i ;
    }
    single field components {
      returns java.util.Vector ;
    }
  }
}
```

Since Java is the underlying implementation language in FRED, Cola deliberately follows Java's syntactic style. The contract includes three top-level roles: `Component`, `Leaf`, and `Composite`. A main constraint of a role is its *cardinality*. Cardinality denotes the number of Java entities that must or can be bound to the role. The modifier `single` states the role's cardinality: one suitable Java type (an interface or a class) must be bound to the role in every instance of `CompositeContract`. `Component` in turn includes a method role called `operation`. Its cardinality is relative to the parent role.

The role has no cardinality modifiers, implying that there must be $1 - n$ Java methods in each Java type bound to the role `Component`.

For example, the role `Leaf` represents Java classes whose objects cannot have children. It defines a *parameter* `C` of type `Component`. If a role has a parameter, its cardinality is relative to the number of Java entities bound to the role referred to by the parameter. In this case, there must be one Java class bound to the role `Leaf` for every Java type bound to the role `Component`. The parameter can also be used in the constraints associated with the role as well as in its subroles. The `inherits` constraint in `Leaf` specifies that each Java class playing the role of `Leaf` must inherit (extend or implement either directly or indirectly) a Java type bound to the role referred to by the parameter `C`, i.e. the role `Component`. The named constraint says that by default the name begins with the string constant “Leaf” followed by the name of the Java type bound to the parameter. These default values can be used e.g. in code generation.

Design contracts are an essential part of *specialization templates*. A specialization template is a framework-specific design solution to a particular flexibility requirement. Using the terminology of Pree [Pre95], a specialization template specifies a *hot spot* in a framework. Logically a specialization template corresponds to Holland's concept of "conformance", although we emphasize different aspects. A specialization template is based on a design contract: essentially it binds a design contract to a particular framework and its flexibility requirements. The central parts of a specialization template are:

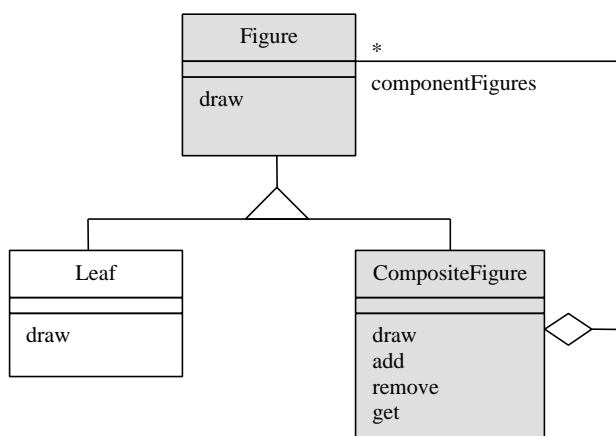
- *Name*: the name of the specialization template.
- *Flexibility requirement*: the flexibility requirement of the framework, in terms of application variance, that gives rise to this specialization template.
- *Design contract*: the design contract this specialization template is based on.
- *Structure*: the structure of the contract, presented so that the classes that are bound to actual (Java) classes are shaded. Unbound roles provide placeholders for new classes and methods.
- *Bindings*: The bindings between Java items and roles.
- *Constraints*: the constraints related to the roles.
- *Specialization hints*: additional hints to guide the application developer in writing the specialization code (e.g. method bodies).
- *Example*: a representative example of a particular specialization.

Note that a design contract is framework-independent whereas a specialization template is framework-dependent. A specialization template binds a part of the contract structure to certain entities (classes, attributes, and methods) in the Java framework. The more a specialization template binds, the less freedom is left to the application developer. On the other hand, if the application developer leaves certain roles unbound, the result is a new, less general specialization template, associated with a less general framework.

Indeed, we consider framework development as a continuum of specialization, terminating in an executable application (see next section).

As an example, we present a simplified specialization template associated with the JHotDraw framework. We omit the design contract this template is based on: the contract is a fairly straightforward reformulation of the Composite design pattern [GHJV95].

- *Name*: CompositeFigure.
- *Flexibility requirement*: It must be possible to define and add new types of figures appearing as nodes in the graphs manipulated by the editor.
- *Design contract*: Composite.
- *Structure*:



- *Bindings*:

<u>Specialization Template</u>	<u>Design Contract</u>
Figure	→ Component
Figure.draw	→ Component.operation
CompositeFigure	→ Composite
CompositeFigure.add	→ Composite.add
CompositeFigure.remove	→ Composite.remove
CompositeFigure.get	→ Composite.get
CompositeFigure.draw	→ Composite.operation
CompositeFigure.componentFigures	→ Composite.components

- *Constraints*:
 - any number of Java classes can be bound to Leaf
 - any class bound to Leaf must override the method bound to operation
 - the method bound to operation must implement the drawing of the figure.
- *Specialization hints*: The figure can be defined also as an icon as follows: ...
- *Example*: Typically, the drawing operation is given as follows: ...

Specialization templates are not formally defined using a textual language, but they are instead constructed interactively using the FRED interface.

3 Specialization templates in framework design

The crucial part of the analysis phase in framework development is stepwise generalization ([KoM95], [Sch97]). The process begins with the specification of an example application in the intended domain of the framework. In each generalization step, some concepts in the requirement specification are generalized, transforming the application specification into a framework specification. When this process is continued, more and more general framework specifications emerge, until eventually a suitable level of abstraction is reached.

Each generalization of a concept gives rise to a flexibility requirement. Essentially, a flexibility requirement stores the information concerning a single concept generalization by specifying the concept that can have variations. Roughly, a generalization of concept *A* to concept *B* gives rise to flexibility requirement "It must be possible to define and add variants of *B*" and to example specialization *A*. Each generalization step in the analysis stage typically comprises several flexibility requirements.

In the design phase, the most abstract framework specification is first implemented ([KoM95], [Sch97]). Each flexibility requirement associated with the specification is transformed into a specialization template. Note that the flexibility requirement itself will be part of the template. An example specialization is also directly available from analysis. The framework developer then designs an architecture that fulfills the flexibility requirement, either using an existing design contract or creating a new one. Possible specialization hints and example specializations are written down. The framework classes involved in the template are implemented and bound to its roles.

After implementing in this way the most abstract framework, the next more concrete framework specification is implemented in the same way. In the ideal case the next framework can be obtained as a specialization of the previous one, exploiting its specialization templates. This results in a layered framework, with increasing level of concreteness. The design process is depicted in Figure 1.

This design scheme for frameworks is idealistic in many ways, but we feel that it gives a useful guideline that the framework developer should have in mind, even though the scheme were not strictly followed. In particular, we argue that the concepts of a flexibility requirement and a specialization template are instrumental in framework design and documentation. However, note that the FRED tool described below does not care about how the specialization templates are produced.

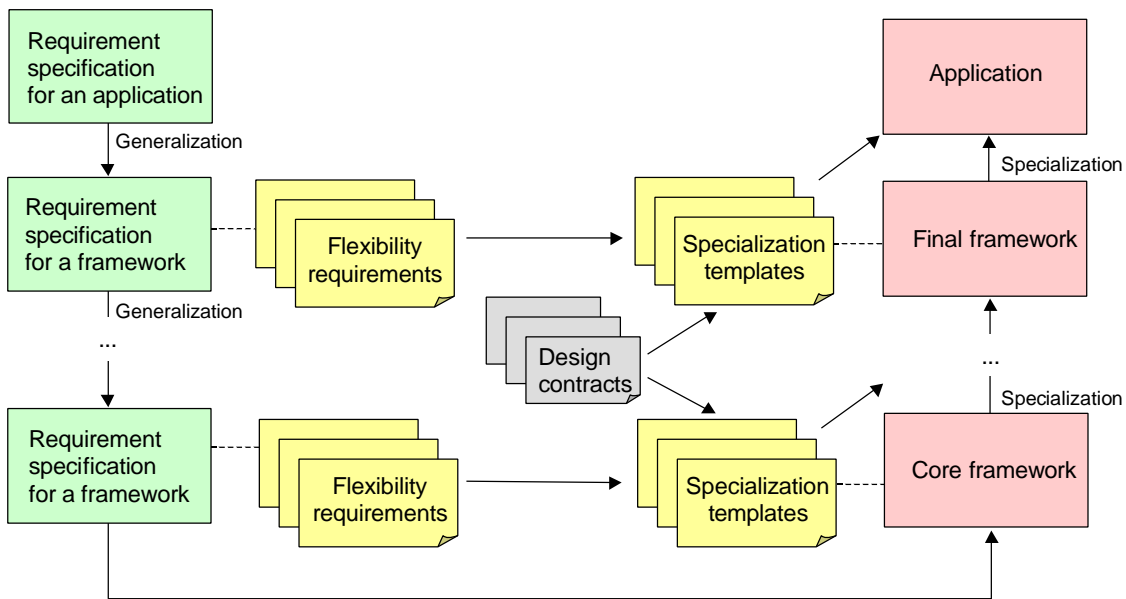


Figure 1. Flexibility requirements, design contracts and specialization templates in framework design based on stepwise generalization.

4 FRED: Tool support based on specialization templates

FRED is a tool for managing framework-based software development. The basic idea is to use specialization templates (and design contracts) for active documentation of a framework. Our vision is that a framework can be specialized under the guidance of FRED, in the same sense as various kinds of wizards assist the programmer in the case of, say, GUI frameworks. However, FRED is completely domain-independent: it can be used to construct a "wizard" for any OO (Java) framework. The basic mechanisms of FRED take care of managing the bindings of specialization templates to actual source code, generating default implementations on the basis of example specializations, keeping track of the remaining tasks that have to be done, displaying informal instructions for specialization (e.g. specialization hints), enforcing the constraints defined in the specialization templates, and editing source code. A dedicated source code editor is integrated with the system so that all modifications of the source text are immediately checked against the specialization templates (and design contracts they are based on). A violation automatically generates a new task for the user in which this violation has to be removed.

The first stage in using FRED is to transform the design contracts into formal Cola specifications. The Cola compiler is currently under development; so far the Cola specifications have been manually transformed into the internal representation format of FRED.

The specialization templates are created through the graphical user interface of FRED. Within FRED, the specialization templates based on the known design contracts are

instantiated, and some of the roles are bound to existing Java entities. FRED generates and automatically maintains a task list for the remaining unbound roles and checks that the bindings are legal. As a result, a collection of partially bound specialization templates is created, representing the specialization interface of the framework.

Figure 2 gives an overview of using FRED. The user interface of FRED consists of structural views of design contracts (left below) and specialization templates with (partial) bindings (left above), a task list, an info sheet (right below) and a work area. The icon of an item (role) in a specialization template indicates the type of the item and whether it is bound or not. The standard tool used in the work area is a source editor. However, various other tools for more high-level (and possibly domain-oriented) tools can replace the source editor, using a general tool interface.

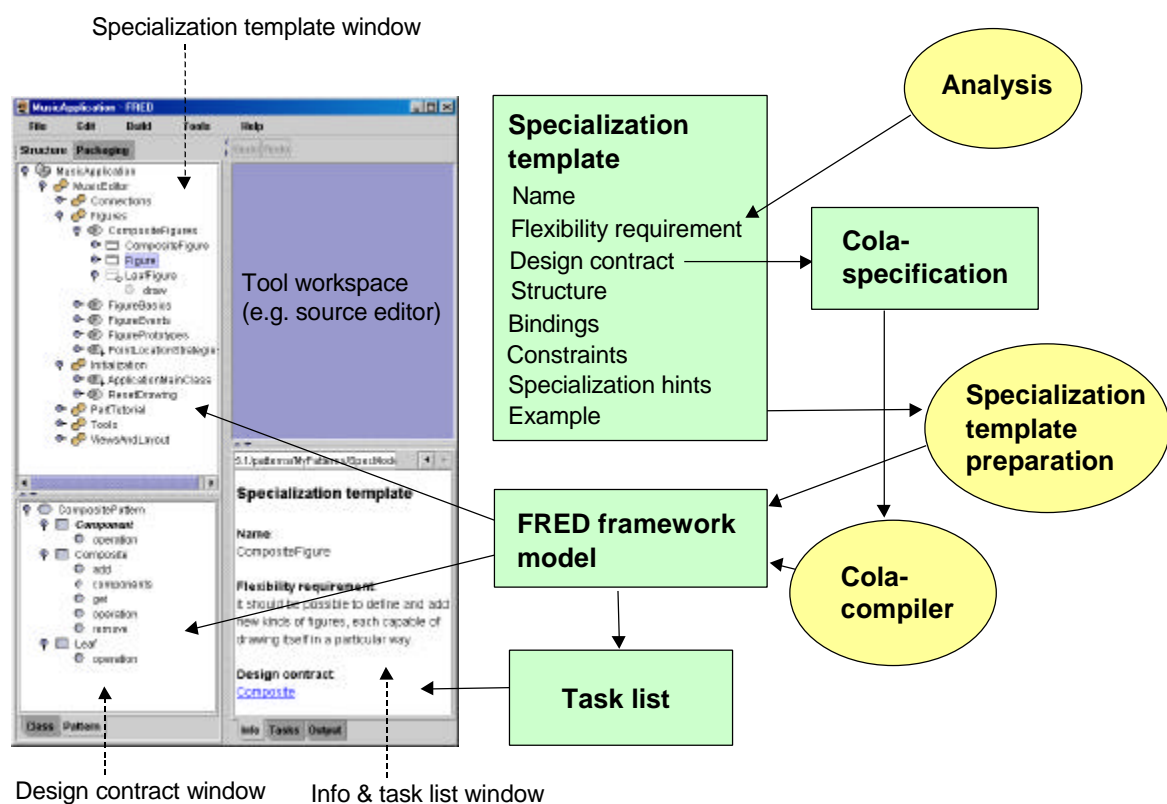


Figure 2. FRED in the framework specialization process.

In the user interface snapshot of Figure 2, the specialization templates have been created for JHotDraw, representing the specialization interface for this framework. One can immediately see that the hot spots in JHotDraw are related with connections (of figures), figures, initialization, tools (for manipulating figures and connections), and views/layout. For each template item in the specialization template view, the user can ask to see the (hierarchical) list of remaining specialization tasks. Each task item can be executed either by generating a default implementation or by binding some unbound role in the template to an actual Java entity. In the latter case the system checks whether

the given Java entity conforms to the structural constraints given in the template (or actually in the design contract). One can also ask for the information sheet associated with the template; this is an informal hyperlinked (HTML) description of the template. When the task list is empty, the user can compile and run the resulting application through FRED as well.

5 Conclusions

Wrapping a framework up inside FRED yields several benefits. Constructing a framework becomes more systematic and manageable since FRED forces the designer to explicitly specify the specialization interface of the framework as specialization templates. This is particularly useful for white-box frameworks. Common design contracts can be specified and stored as reusable foundations of specialization templates. On the other hand, an application developer need not understand the framework as a whole: the specialization templates give condensed information about the relevant parts of the framework. The amount of information the user must perceive is relatively small⁵, and FRED guides her through the specialization tasks by maintaining a things-to-do list. Further, FRED guarantees that the specializations follow the assumptions made by the framework developer, as described in the specialization templates based on design contracts.

So far FRED is a fairly "syntactic" tool: it takes care of structural aspects of hot spots, rather than their semantics. A possible future approach is to allow the specification of methods in the design contracts, for example using pre- and post-conditions. However, introducing a full-fledged program verification system is beyond the scope of FRED: we wish to keep FRED as simple as possible. An attractive compromise might be to apply the idea of "grey-box components" introduced in [BW97], in which a certain pattern is specified for a method, rather than the black-box semantics (pre- and post-conditions).

References

- [BW97] Büchi M., Weck W.: A Plea for Grey-Box Components. TUCS Technical Report 122, Turku Centre for Computer Science, August 1997.
- [CACM97] *Communications of the ACM* 40, 10 (October 1997). Special issue on Object-Oriented Application Frameworks.
- [Cas95] Casais E.: An Experiment in Framework Development. *Theory and Practice of Object Systems* 1, 4 (1995), 269-280.
- [FMW97] Florijn G., Meijers M., van Winsen P.: Tool Support for Object-Oriented Patterns. In: *ECOOP 97*, LNCS 1241, Springer-Verlag 1997, 472-495.
- [Gam98] Gamma E.: The JHotDraw framework. Download: <http://members.pingnet.ch/gamma/JHD-5.1.zip>.

⁵ For example, for JHotDraw we needed 16 specialization templates.

- [GHJV95] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley 1995.
- [Hak98] Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J.; Pattern-Oriented Framework Engineering with FRED. In: *Object-Oriented Technology, ECOOP 98 Workshop Reader*, Springer-Verlag 1998, 105-109.
- [Hol93] Holland I.: *The Design and Representation of Object-Oriented Components*. PhD thesis, Northeastern University, 1993.
- [Joh92] Johnson R.: Documenting Frameworks using Patterns. In: *OOPSLA '92, SIGPLAN Notices* 27,10 (Oct 92), 63-76.
- [KoM95] Koskimies K., Mössenböck H.: Designing a Framework by Stepwise Generalization. In: *Proc. of 5th European Software Engineering Conference (ESEC '95)*, Sitges, Spain. LNCS 989, Springer-Verlag 1995, 479-498.
- [Lew95] Lewis T., Rosenstein L., Pree W., Weinand A., Gamma E., Calder P., Andert G., Vlissides J., Schmucker K.: *Object-Oriented Application Frameworks*. Manning Publications/Prentice Hall 1995.
- [MDE97] Meijler T.D., Demeyer S., Engel R.: Making Design Patterns Explicit in FACE - A Framework Adaptive Composition Environment. In: *Proc. of 6th European Software Engineering Conference (ESEC '97)*. LNCS 1301, Springer-Verlag 1997, 94-111.
- [Pre95] Pree W.: *Design Patterns for Object-Oriented Software Development*. Addison-Wesley 1995.
- [Sch97] Schmid H.A.: Systematic Framework Design by Generalization. In: [CACM97], 48-51.
- [SBF96] Sparks S., Benner K., Faris C.: Managing Object-Oriented Framework Reuse. *Computer* 29,9 (Sept 96), 52-62.

Chapter 4.

Task-Based Tool-Support for Framework Specialization

Hakala M.: Task-Based Tool Support for Framework Specialization. In: Proceedings of OOPSLA'00 Workshop on Methods and Tools for Framework Development and Specialization, Tampere University of Technology, Software Systems Laboratory, Report 21, October 2000.

Task-Based Tool-Support for Framework Specialization

Markku Hakala

Tampere University of Technology

P.O.Box 553

FIN-33101 Tampere

Finland

markku.hakala@cs.tut.fi

ABSTRACT

An idea of providing assistance for framework-specialization by the means of interactive task list is presented. A model based on the notion of generative patterns is described, that provides basis for task-based tool support for framework specialization.

1 INTRODUCTION

Product line architecture is a system of rules and conventions for creating software products for a given domain. The broader the domain, the more complex rules the architecture embodies.

This growing complexity leads to a usability problem that can seriously hinder the reuse of the architecture. Thus, there is a need to promote to the comprehensibility of these rules, as well as validate the application of these rules.

Object oriented frameworks are an established way to implement product line architectures. The framework implements the invariant part of the architecture and defines a specialization interface. A new product can be derived from the framework by providing the application-specific part using the provided interface.

In frameworks, the features of the implementation language are used to express the architectural rules and conventions. This most often incorporates inheritance.

However, only fraction of the rules of the framework can be expressed adequately with language constructs. Every framework contains rules implied by the implementation but not statically checkable by the compiler.

This emphasizes the importance of the documentation. Thus, much effort has been placed on the research on framework documentation, as well as technologies and tools supporting the framework specialization.

This paper proposes a task-driven approach to framework specialization. The model embodies an algorithm that maintains a dynamic “things-to-do”-list in co-operation with the framework developer. The idea is to provide interactive specialization instructions that adapt to the current specialization problem.

In addition, code generation, and to some extent, verification, can be implemented on top of the model.

Proving the correctness of the specialization is however,

beyond the scope of this research. Our approach aims to improve specialization process by emphasizing the education of the developer beforehand or concurrently with the process, rather than validation that usually takes place after the code has been produced.

This model can also be seen as an extension of the notion of Framework Cookbooks [Pre95]. However, the task driven approach has appeared to be useful also outside the scope of framework specialization, e.g. in formalizing architectural standards and conventions such as Java Beans.

The suggested model has been implemented in a development tool called FRED (Framework Editor). The tool has been evaluated in a realistic environment in the Finnish industry.

Chapter 2 describes the task driven programming model. Chapter 3 discusses the current implementation of the model along with a minimal case study.

2 TASK DRIVEN MODEL

Overview

Nowadays software development rarely starts from scratch. Each piece of software is produced against some standards, conventions and underlying systems and architectures. Most software structures manifest patterns that have been invented before. Everyone can agree that more than inventing ideas, software engineering is about applying ideas.

Applying an idea means following a plan to manifest that idea. The idea of task-based tool support is to model this plan as a list of tasks.

When dealing with complicated structures, which are typical for software, this task list cannot be adequately expressed by a linear step-by-step list. Making software is conceptually harder than making supper. A choice made during the process may change the rest of the plan completely. That is why cookbooks [Pre95], although a step to a right direction, are not enough.

By doing a task, the developer continues with the plan, but also makes choices. There might be several ways to do the task, each of which will generate succeeding tasks.

Technically, there could be two ways to do tasks. The developer may either explicitly mark a task as done, or the tool may implicitly consider a task as done.

The problem with traditional documentation is that it has to be written before the specialization takes place. Therefore the documentation has to communicate with the abstract concepts of the framework, not with the concrete concepts of the specialization. By providing tasks incrementally, the tool can gather information of the specialization and parameterize the documentation with the specialized concepts. The same information can be used to provide more active assistance in for task, e.g. code generation that adapts to the specialization context.

Pattern Based Model

In order to provide task-based tool support, we suggest a model that builds on the notion of patterns as generative descriptions that can be used systematically to produce a number of co-similar structures.

This fits a more traditional way to see a pattern as a description of a recurring problem along with a reusable solution to that problem within a certain context. However, our notion of patterns should not be confused with design patterns [Gam95], often associated with strict rules on their known uses and the domain of applicability. With the goal of providing programming assistance using patterns in mind, everything is a considered a pattern that contains a structure definable in a form described in this chapter. Most design patterns seem however to fall into this category. Nevertheless, considering our current model, tool support benefits most the instantiation of less abstract implementation-oriented variants of design patterns.

One essential part of a pattern definition is the description of the pattern structure. Within the scope of this research, we concentrate on the structure and leave the other parts open. However, a pattern is not seen as a purely declarative description of the solution, but rather a description of the process to instantiate the abstract solution. A pattern can therefore be seen as an algorithm that can be applied in several environments to build the same structure.

Given this informal description, the structure of a pattern can be presented as a directed acyclic graph (DAG) that can be formalized by the following 4-tuple:

$$P = (R, D, c, S)$$

This is called the *definition graph*. The vertices R of the graph are called *roles*, and the directed edges D are called dependencies. A dependency from r to s is an ordered pair (r, s) , where role r is called the *depender*, and role s is called the *dependee*.

Function c is called the *cardinality function*, and is defined as a mapping from a role to an integer range. The range is defined by its end points and is called the *cardinality constraint* of the role. The cardinality function is defined as follows:

$$c : R \rightarrow \{0, 1, \dots, n\} \times \{1, 2, \dots, \infty\}$$

$$c(r) = (l, u), u \geq l$$

The definition graph represents a reusable abstract structure. The roles define all the abstract components of this structure, whereas the dependencies and cardinality constraints define the abstract relationships between roles.

The fourth part of the pattern graph definition is the *structural constraint set* S , which is a set of constraints that are used to restrict the instantiation process. Different kind of formalisms could be adopted for such constraints. E.g. predicate logic could be used to state the required properties. We are currently researching different alternatives in expressing structural constraints. One that has proved to be very useful in practice is what we call the *path constraint*. A path constraint C is defined as a set of paths¹, which all begin at the same role and end to the same role, i.e.:

$$C = \{ (r_1, \dots, r_n) \mid (r_i, r_{i+1}) \in D \}$$

$$\forall (r_1, \dots, r_n), (s_1, \dots, s_m) \in C : r_1 = s_1, r_n = s_m$$

Figure 1 presents an example pattern structure with a graphical graph notation, where nodes represent roles and edges represent dependencies. The roles are labeled, and the cardinality constraint is placed after each role label. Below the graph there is a path constraint between roles Adapter and Record. The same pattern is examined in detail by going through the specialization example in Chapter 3.

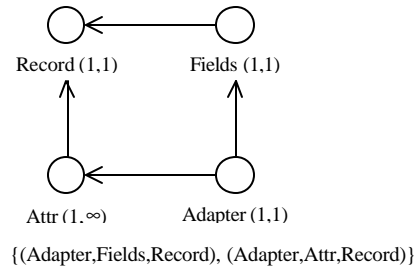


Figure 1. An example pattern definition graph

Applying a pattern is called *instantiation*. It results in a *pattern instance*, which describes a concrete structure as an instance of the abstract structure defined by the pattern. In practice, this correlation is described by binding concrete *program elements*, like classes, methods, formal parameters and so on, to the abstract roles defined by the pattern.

At any time during the instantiation, the structure of the instance of pattern $P = (R, D, c)$ can be described as a DAG as well. It can be presented by a 3-tuple as follows:

$$P' = (R', D', s)$$

¹ A path within a DAG from vertex v_1 to vertex v_n connected with number of edges is defined unambiguously as an ordered list of vertices (v_1, \dots, v_n) on that path.

In this *instance graph*, the vertices R' are called *role instances*. Each role instance is a manifestation of some role in the definition graph. Function m called *meta-function* defines this relationship as a mapping from a role instance to a role:

$$m : R' \rightarrow R$$

Similarly, the directed edges D' between role instances are called *dependency instances*. Each dependency instance (x, y) manifests a dependency defined between the two roles that the role instances x and y are instances of. In other words:

$$\forall (x, y) \in D' : \exists (m(x), m(y)) \in D$$

The last component of the instance graph, function s , called *state function*, maps each role instance to a *state*. The function is defined as follows:

$$s : R' \rightarrow \{\text{mandatory, optional, valid, invalid}\}$$

To understand the state function, we must look at the instantiation process.

The instantiation process is incremental and guided by the definition graph of the pattern. The appendix lists an algorithm that takes a definition graph P as an input, and gradually constructs an instance graph P' and function m . The algorithm assumes a development tool environment that works in the interaction with the developer.

The algorithm works by gradually augmenting the instance graph with new role instances. New instances are generated based on the dependency and cardinality constraint definitions. Each new role instance is considered a task to be carried out by the developer. A state, either *mandatory* or *optional* (with obvious semantics), is assigned to the task. Once the user does the task, the state of the role instance is changed. The state becomes *invalid*, if the tool is able to judge that the task is not properly executed, or *valid* in other cases.

In addition, whenever a task becomes done, the algorithm may augment the pattern instance with new role instances.

Mandatory, *optional* and *invalid* role instances are called as *tasks*. *Valid* role instances constitute the concrete structure that has been instantiated from the abstract structure defined by the pattern. If the definition graph is well formed, and the user follows the ever-changing task list, a point is reached where all role instances are either *valid* or *optional*. As a result, an abstract structure defined by the pattern has been specialized in a user-defined context.

Figure 2 presents a partial pattern instance and its relation to a pattern. The state is indicated by a superscript after role instance label and the meta-function is indicated by edges between role instances and roles. Path constraints are omitted from this figure.

The definition graph provides syntax for describing

patterns. To provide useful tool support, the graph has to be decorated with tool-specific semantics. Therefore, a practical application of this model extends it in many ways. E.g., tool-specific attributes could be defined for the roles. Such attributes could include the default implementation for the required program element, documentation, programming-language specific constraints, such as the return type of the method or required inheritance, and so on. Provided sufficient information exists, the tool can generate context-specific instructions for the task, provide automation such as code generation, and validate the user actions.

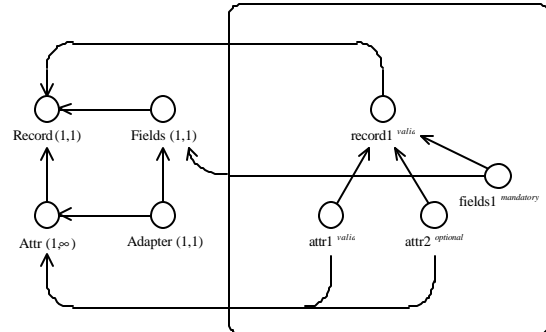


Figure 2. An example pattern instance

The tool-specific attributes may also include rules and heuristics to determine automatically when the user completes a certain task. In practice however, the tool cannot rely on such heuristics alone. Instead, the user should explicitly *acknowledge* some tasks. By requiring explicit commitment from the user, the tool may behave in a more predictable way. E.g., if the user is converting one of his classes to a Singleton [Gam95], the tool may provide assistance much earlier and reliably if the user explicitly states his or her intentions.

Completing a task may require "evidence" to be provided by the user. In a development tool, such evidence could be part of the produced source code. This program element can then be checked against the rules imposed by the pattern and any inconsistencies may be reported. This equals to enhancing the compiler with architecture-specific typing checks. Ideally, the tool provides an incremental development environment where these checks could be re-evaluated whenever the user manipulates the source code, thus making it possible for the task-list to evolve by itself concurrently with the development process.

3 FRED AND FRAMEWORKS SPECIALIZATION

FRED is a prototype tool implementing the discussed model. FRED is implemented in Java and intended for providing task-driven assistance for Java programming, especially to support specialization of Java frameworks. The tool currently being tried out in Finnish software industry. A small snapshot of the user interface is shown in Figure 3.

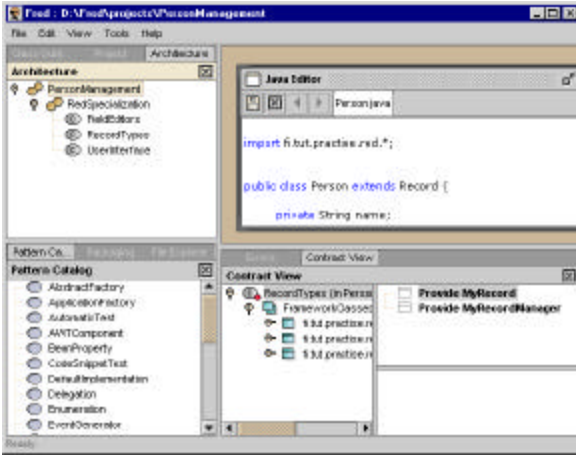


Figure 3. User interface of FRED tool

As an example of task-driven framework specialization, a framelet called Red is used. It comes with FRED 1.1 release. Framelets [Prk98] are small frameworks consisting of a handful of classes and used as reusable building blocks for creating components. The Red framelet is an evolved version of a framelet discussed in [Prk98].

Red provides user interface facilities to maintain a list of Record-objects and edit their fields. A specialization of Red typically defines a new Record subclass with some application domain –specific fields. Once the user has defined this new record type and derived some other classes, the framelet can generate the user interface automatically. Typical user interface windows provided by Red are seen in Figure 4.

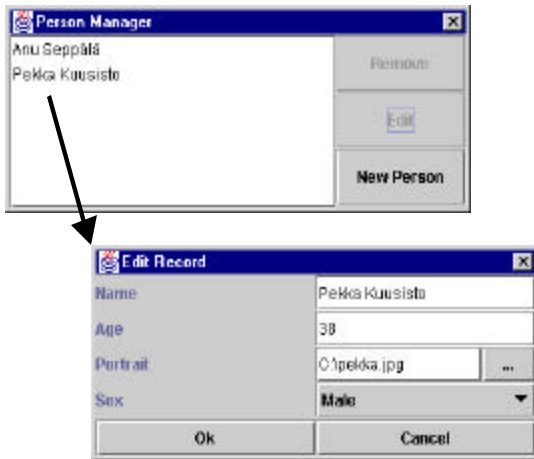


Figure 4. Typical views provided by Red framelet

Through this example, a development tool similar to FRED environment (see acknowledgements) is assumed, that guides in the instantiation process.

Figure 1 presented a simple pattern definition graph, without any tool-specific attributes and semantic constraints. The graph describes a pattern with roles

named *Record*, *Attr*, *Adapter* and *Fields*. Let us assume that sufficient information, although suppressed from our simplified notation, is provided for the tool to interpret *Record* as a description of a class role, *Attr* as a role for class member variable, *Fields* as a method role, and *Adapter* as a role that describes some piece of code to be written within a method. This minimal pattern is a simplification of a pattern enclosed in the FRED release, to describe creation of a new Record subclass using Red framelet.

Given this description of the pattern, the tool can provide a task list for the user. The tool begins by instantiating each role that has no dependencies. Therefore, an instance of role *Record* is created. Let us name it *record1*. The state of the instance is set to *mandatory*, which denotes a mandatory task. This task tells the developer to provide a class to play the role *Record*. Provided the proper semantic constraints are included, this task means that the developer should subclass the Record-class provided by the framework. Based on the tool-specific attributes associated with the role, most notably the free-form documentation, the tool can generate textual description for the task as well as suggest a skeletal implementation.

This is the first and only task the tool is able to generate at this point. This instance graph at this point is presented visually in step 1 of Figure 5.

To complete the task, the user has to point out a suitable class for the tool. This may involve creation of a new class, modification of an existing one, or letting the tool generate the skeletal default implementation. Suppose the developer is specializing Red to store information on personnel. For that purpose, the developer creates a new Record subclass named Person. If heuristics are provided for the pattern to determine every subclass of Record to play the role *Record* in the pattern, the task may be considered done automatically. If such heuristics are unavailable, the user must point out the Person class explicitly to adhere the task.

The Person class may be left empty for the moment. If the inheritance constraint is satisfied, the tool changes the state of the role from *mandatory* to *valid*. The binding between the class and the role instance is maintained, so that the state can be re-evaluated whenever there are any changes in the associated source code.

After completion of the first task, the tool re-evaluates the pattern instance against the definition. No more instances of role *Record* are created because the upper bound of the associated cardinality constraint is 1, and an instance of *Record* already exists. However, an optional task named *attr1* to create a member variable to play role *Attr* is created, along with a dependency instance (*attr1*, *record1*). The task is mandatory, as the lower bound of the associated cardinality constraint is 1, requiring there to be at least one instance of *Attr* for each instance of *Record*. Similarly, an instance of *Fields* is created. This

task corresponds to overriding a method declared in the Record-class. Step 2 in Figure 5 presents the instance graph at this point.

Given these two tasks, the user may continue in his or her preferred order. Note that the graphical representations do not describe any semantics on the structure. Such semantics, including the requirement for the expected fields and methods to be declared in the class for the associated instance of *Record*, has to be defined by tool-specific extensions to the model.

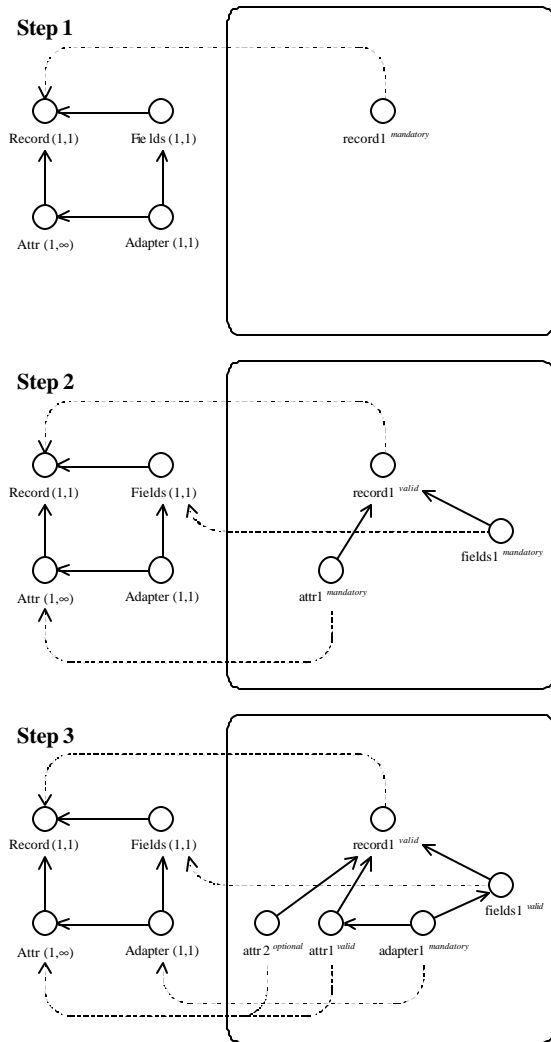


Figure 5. Some specialization steps for specializing Red

For task *attr1*, the developer introduces a new member variable in the Person class, and denotes this variable as the required program element. For *fields1*, the developer

asks the tool to generate the default implementation. Step 3 of the same figure presents the situation where the developer has done both these tasks and the tool has re-evaluated the pattern instance once again. Two new tasks have been created. One concerns creating another member variable and is optional. The other instructs the developer to type in some adapter code in the overridden method. In Red specialization, this adapter code provides access to read and write the member variable through the Red user interface. Our pattern definition states that such adapter code must exist for each member variable declared to play the role *Attr*.

A mechanism can be provided to undo tasks, providing the means to backtrack the instantiation process and reconsider the decisions made. In FRED, the source code is modified under supervision of the tool, thus earlier decisions are refined automatically based on the modified source code. Whenever the code no longer complies with constraints of the pattern, the associated role instances become invalid, reminding the user of the architectural rules and conventions.

ACKNOWLEDGEMENTS

The model on task-driven programming has been developed within FRED project, between the Department of Computer Science at the University of Helsinki and the Department of Computer and Information Sciences at the University of Tampere, funded by TEKES and several industrial partners. The work is being continued in the Department of Information Technology at the Tampere University of Technology, Finland. Earlier work is reported in [Hak99]. FRED tool can be downloaded from the project web site at <http://practise.cs.tut.fi/fred>.

REFERENCES

- [Gam95] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns – Elements of Reusable Object-Oriented Architecture*, Addison-Wesley, 1995.
- [PrK98] Pree W., Koskimies K.: *Framelets – Small and Loosely Coupled Frameworks*. Manuscript, submitted for publication, 1998.
- [Pre95] Pree W.: *Design Patterns for Object-Oriented Software Development*. Addison-Wesley 1995.
- [Hak99] Hakala M., Hautamäki J., Tuomi J., Viljamaa A., Viljamaa J., Koskimies K., Paakki J.: *Managing Object-Oriented Frameworks with Specialization Templates*. *ECOOP'99 Workshop on Object Technology for Product-line Architectures*, 1999.

APPENDIX: PATTERN IN STANTIATION ALGORITHM

The following algorithm *UPDATE* expects a pattern $P = (R, D, c, S)$ and a pattern instance $P' = (R', D', s)$. In addition, it is assumed that all structural constraints in set S are path constraints. The algorithm makes also use of the following definitions:

$$I(r) = \{ x \mid m(x) = r \}$$
$$\text{dependees}(r) = \{ s \mid (r, s) \in D \}, r \in R$$
$$x \rightarrow r = y, y \in I(r), r \in R, (x, y) \in D'$$

The algorithm should be evaluated by a development tool to update P' and the meta-function m , that defines the mapping between P' and P . Provided the state function returns *valid* or *invalid* for tasks that have been done at that time, the algorithm may construct new tasks with either *mandatory* or *optional* state. Modification of a function or set is denoted by a left-pointing-arrow " \leftarrow ".

UPDATE is

For each $r \in R$ Do

$\{ s_1, \dots, s_n \} \leftarrow \text{dependees}(r)$
 $(l, u) \leftarrow c(r)$

For each $(d_1, \dots, d_n) \in I(s_1) \times \dots \times I(s_n)$

where $\forall C \in S : \forall (r, a_1, \dots, a_p), (r, b_1, \dots, b_q) \in C : \exists i, j :$

$d_i \in I(a_1)$ and $d_j \in I(b_1)$ and $d_i \rightarrow a_2 \rightarrow \dots \rightarrow a_p = d_j \rightarrow b_2 \rightarrow \dots \rightarrow b_q$ Do

$X \leftarrow \{ x \in I(r) \mid \forall i \exists (x, d_i) \in D' \}$

If $|X| < l$ and not $(\exists x \in X : s(x) = \text{mandatory} \text{ or } s(x) = \text{optional})$ Then

Let $x = \text{new role instance}$

$m(x) \leftarrow r$

$X \leftarrow X \cup \{ x \}$

$R' \leftarrow R' \cup \{ x \}$

$D' \leftarrow D' \cup \{ (x, d_1), \dots, (x, d_n) \}$

End

If $\exists x \in X : s(x) = \text{mandatory} \vee s(x) = \text{optional}$ Do

If $|X| \leq u$ Then $s(x) \leftarrow \text{mandatory}$

Else $s(x) \leftarrow \text{optional}$

End

End

End

Chapter 5.

Generating Application Development Environments for Java Frameworks

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

Generating Application Development Environments for Java Frameworks

Markku Hakala¹, Juha Hautamäki¹, Kai Koskimies¹,
Jukka Paakki², Antti Viljamaa², Jukka Viljamaa²

¹Software Systems Laboratory, Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland

{markku.hakala, csjuha, kk}@cs.tut.fi

²Department of Computer Science, University 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	
<i>Bound roles</i>	<i>Properties</i>
Shape : class	description Base class for all graphical figures.
draw : method	description The drawing method.
<i>Unbound roles</i>	<i>Properties</i>
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.

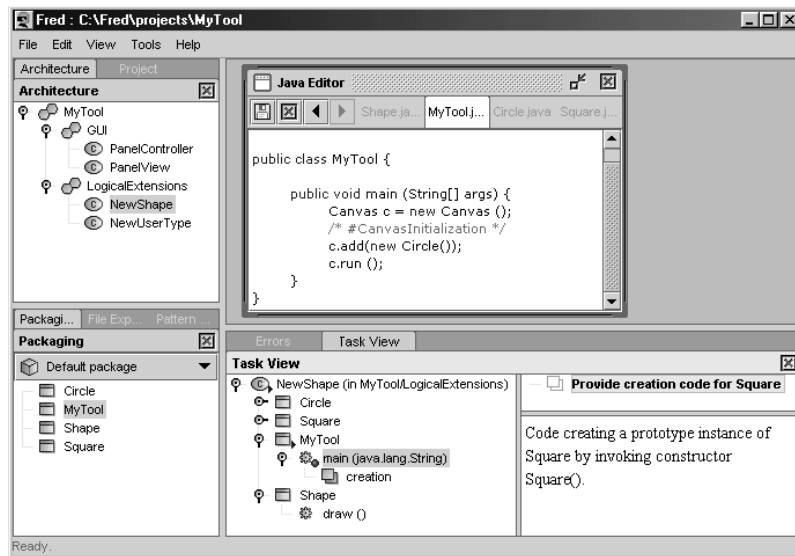


Fig. 1. User interface of FRED

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 subclass 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 accidentally 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.

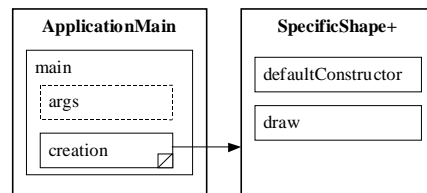


Fig. 2. A diagram of the `NewShape` specialization pattern

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.

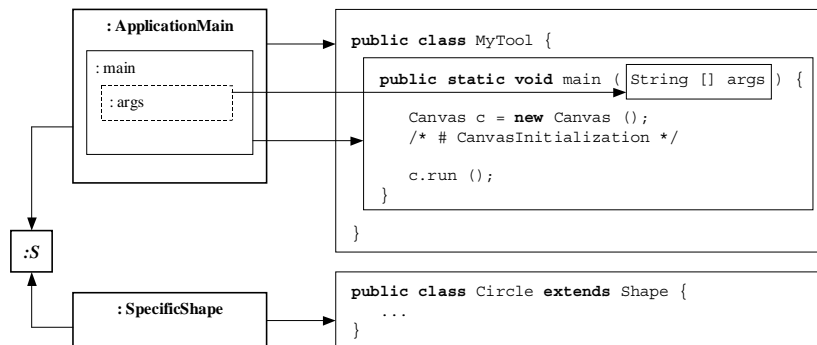


Fig. 3. An example of cast that relates specialization code to the roles of the pattern

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.

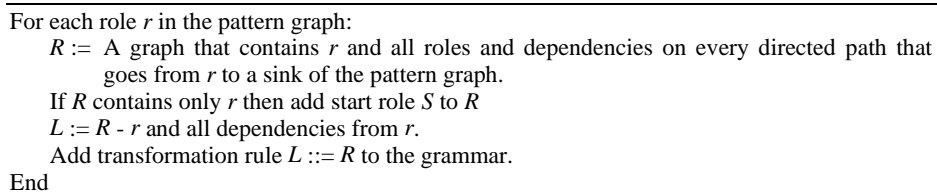


Fig. 4. An algorithm that generates the transformation rules from a pattern specification

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.

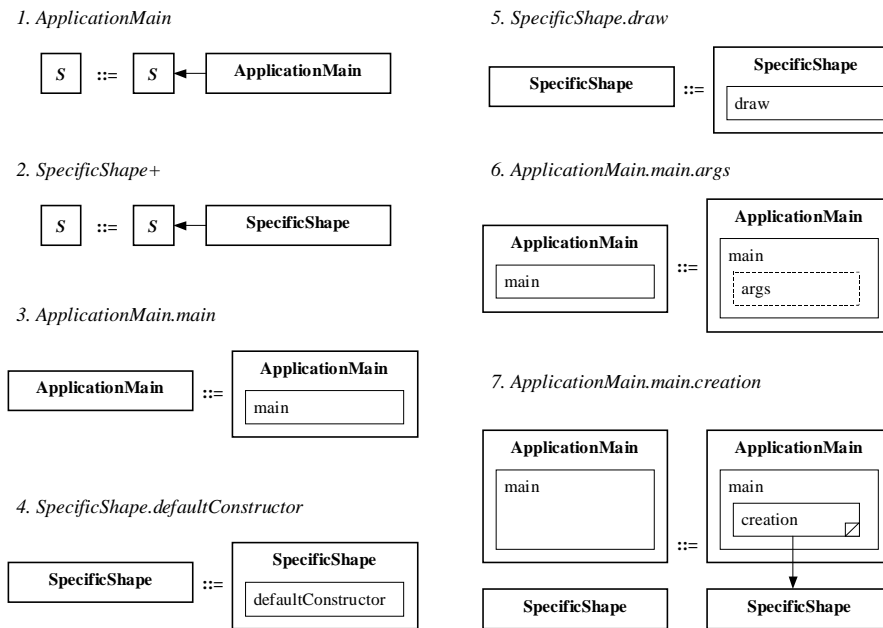


Fig. 5. The graph grammar of NewShape, derived from its pattern graph

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 initialization 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.

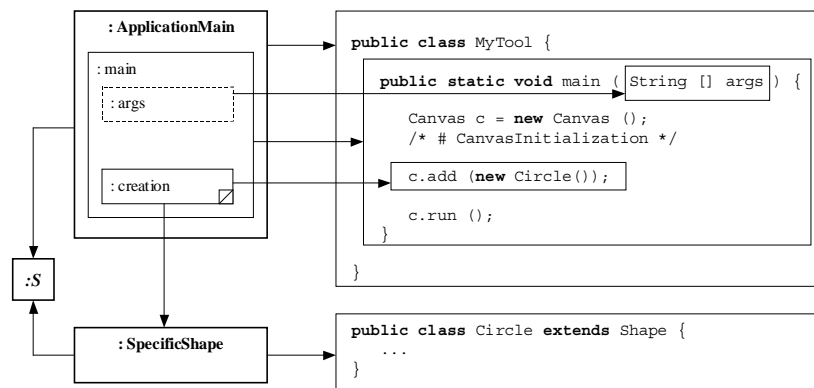


Fig. 6. Result of an application of a grammatical rule to the cast graph of Figure 3

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.

Chapter 6.

Annotating Reusable Software Architectures with Specialization Patterns

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

Annotating Reusable Software Architectures with Specialization Patterns

Markku Hakala, Juha Hautamäki, Kai Koskimies

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

Jukka Paakki, Antti Viljamaa, Jukka Viljamaa

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

Abstract

An application framework is a collection of classes implementing the shared architecture of a family of applications. It is shown how the specialization interface (“hot spots”) of a framework can be annotated with specialization patterns to provide task-based guidance for the framework specialization process. The specialization patterns define various structural, semantic, and coding constraints over the applications derived from the framework. We also present a tool that supports both the framework development process and the framework specialization process, based on the notion of specialization patterns. We will outline the basic concepts of the tool and discuss techniques to identify and specify specialization patterns as required by the tool. These techniques have been applied in realistic case studies for creating programming environments for application frameworks.

1. Introduction

An extensible architecture is the cornerstone of large-scale software reuse. Such an architecture can be implemented as an (object-oriented) *application framework* [1]. Application frameworks are steadily growing in industrial importance since they provide a suitable technology for implementing large-scale architecture-centric reusable assets such as *product lines* [2].

An object-oriented framework is a collection of classes that implement the shared architecture and the common functionality of a family of applications. The interface between the common, reusable framework and the application-specific parts built on top of it is realized as a set of extension points, or *hot spots* [3]. The architecture and specialization interface of a framework can be documented with (*design*) *patterns* [4]. Understanding and

specializing a framework using patterns and hot spots is a challenging task for which tool support is of vital importance (see, e.g., [5] and [6]). According to our vision, future frameworks are accompanied by framework-specific programming environments that both guide and control application programmers in creating applications according to the conventions of the framework.

FRED (FRamework EDitor) is a prototype tool intended for generating programming environments for Java frameworks [7]. The FRED approach is based on a vision of architecture-oriented programming where the specialization interface of a framework is defined by *specialization patterns*. FRED supports both a framework developer in creating the specialization patterns for the framework, and an application developer in specializing the framework by following a task list generated by the tool, as implied by the patterns. The tool guides the application developer through the task list, dynamically adjusts the list according to the choices made by the adapter, and verifies that the syntactic and semantic constraints of the framework are not violated.

FRED provides thus an interactive environment in which specialization tasks can be executed incrementally in small pieces, allowing the application programmer to observe the effect in the source code, to cancel the tasks if needed etc. Ideologically, FRED is a descendant of the “*cookbook*” concept [8, 3, 9, 10, 11]. Related approaches include also *motifs* [12] and *hooks* [13]. The current implementation of FRED aims at supporting frameworks written in Java, but in principle the approach is not tied to any particular language. FRED is freely available at <http://practise.cs.tut.fi/fred>.

In this paper we will present the conceptual basis of FRED and discuss various techniques to apply this approach in practice. These techniques have been developed and evaluated in the context of two major case studies where we have applied the FRED methodology to realistic

Java frameworks. One of them is *JHotDraw* [14], a framework for implementing graphical editors. JHotDraw was chosen as an example framework because it is commonly known, mature, well structured, relatively large (about 150 classes), implemented in Java, and freely available.

Our second case study is a network management GUI framework with about 300 classes. This framework was developed by Nokia, and the application programming environment produced as a result of the case study is currently being used within Nokia.

We proceed as follows. The specialization pattern concept is introduced in Section 2. In Section 3 we discuss techniques for identifying specialization patterns and using the FRED approach in practice. Related work is discussed in Section 4. Finally, concluding remarks summarizing our experiences and future work topics are presented in Section 5.

2. Specialization patterns

Traditionally, software architecting has been understood as a part of the software design phase, and architectures have been mainly described with standard modeling languages (such as UML). To some degree, these abstract descriptions make it possible to assess the quality of the system at the architectural level and at design time, but they fall short in supporting the construction of the actual executable system based on the architecture.

Especially the construction of product families calls for systematic architecture-centric methodologies that support the implementation of both the reusable core of the family and the products derived from it. In particular, we need an environment that guarantees that the application-specific code conforms to the underlying architecture.

As the basis of architecture-oriented programming, we propose the notion of a *specialization pattern*. It is a specification of a recurring program structure, which 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. 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. *Multiplicity* 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* etc. 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 affect code generation or user instructions. For instance, most role kinds support a default name property for specifying the name of the program element used when the tool generates a default implementation for the element.

2.1. Pattern definition graph

A specialization pattern can be expressed as a directed acyclic graph called a *pattern definition graph*. Figure 1 shows the definition graph of a pattern representing a reusable structure for a class having a number of fields and an accessor method for each of them. The nodes in the graph represent roles. Class roles are denoted with circles, method roles with white squares, and field roles with black squares in the figure.

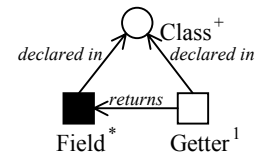


Figure 1. A pattern definition graph

The directed edges between roles are called *dependencies* (e.g. the edge from *Field* to *Class*). If there is a dependency from role *r* to role *s* then *s* is called the *dependee* of *r*. The multiplicity constraint for a role, in relation to its dependees, is placed in parentheses after its name in Figure 1. Multiplicity can be either exactly one (1), from zero to one (?), from one to infinity (+), or from zero to infinity (*).

Some of the properties of the roles are given as labels of the associated dependency arrows. They state that program elements playing the *Field* or *Getter* roles must be declared inside the class playing the corresponding *Class* role, and that each getter must return an object whose type is compatible with the type playing the *Field* role.

2.2. Casting

Applying a pattern is called *casting*, and the resulting structure is called a *cast*. Casting means incremental and interactive binding of suitable program elements to the unbound roles of the pattern. The tool provides the developer with a sequence of tasks that guide the developer in adapting the generic solution proposed by the pattern. *Production tasks* instruct the developer to

instantiate and bind a role. *Refactoring tasks* assist the user in modifying a program element bound to a role to adhere to the constraints imposed by the role.

Since a cast is an instance of a specialization pattern it can be presented as a directed acyclic graph as well. Figure 2 shows a *cast graph* based on the pattern in Figure 1. The nodes in the cast graph represent contracts. Each contract is a manifestation of some role in the associated definition graph. Contracts are of form r_i where r is that role and subscript i , as a positive integer, identifies the contract amongst all contracts of role r . The directed edges between contracts are called *dependency instances*. Each dependency instance between two contracts manifests a dependency defined between the two corresponding roles.

Contracts are visible as production tasks within a tool environment. That is why each contract has a *state*. As a task, a contract can be *done* or *undone*. Furthermore, an undone task may be considered as *mandatory* or *optional*, depending on the multiplicity of the associated role and the number of its instances (i.e. contracts). The state of each contract is written in superscript after its label.

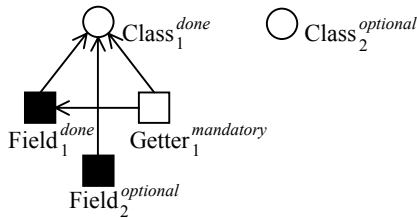


Figure 2. A cast graph

The *casting algorithm* implemented in the tool generates the tasks. The algorithm assumes a definition graph and a cast graph, which it augments with new contracts whenever possible. The newly created contracts imply mandatory or optional production tasks to be carried out by the developer. As the user completes a task, the state of the corresponding contract is changed to done, and the tool re-evaluates the algorithm to determine whether it is possible to create new contracts.

The algorithm processes each role and decides whether it is necessary to create new contracts for that role. This is determined by first constructing all possible combinations of the contracts of the dependee roles. Then the algorithm checks if a correct amount of contracts exists for each of these combinations. If not, a new contract is created, with state set to optional or mandatory, depending on whether the lower bound denoted by the multiplicity constraint has been exceeded or not.

Figure 2 above portrays a *partial* cast, i.e. a pattern instance that is in the middle of instantiation. Some tasks have been done, resulting in a graph of contracts. The

interpretation of the graph can be based on the semantic outline sketched for the graph in Figure 1. The developer has created a class and a field inside it. An unbound contract $Getter_1$ is shown to the user as a task to provide a getter method for that field. As this is a mandatory task, the cast is not yet considered *complete*. The developer has also a choice of continuing with optional tasks, some of which may lead to new tasks, even mandatory.

Figure 3 gives an overview of the whole situation at this point. The dashed arrows show how the contracts of the current cast are related to the roles in the pattern definition graph on the left, and how the bound (done) contracts, on the other hand, are also associated to the pieces of code on the right.

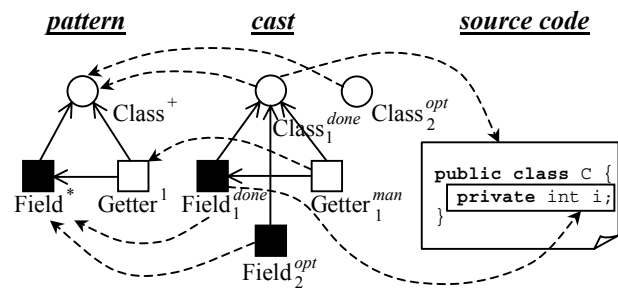


Figure 3. Pieces of code cast to roles of a pattern

3. Working with specialization patterns

FRED is a prototype tool employing the model discussed in Section 2. It is implemented in Java and provides task-driven assistance for architecture-oriented Java programming. Our original motivation was to support specialization of Java frameworks, but it has later turned out that the approach can be used as well to guide programming according to various kinds of other architectural or coding conventions. As an example, we have modeled parts of the *JavaBeans* architecture as patterns, obtaining thus an environment for JavaBeans programming.

In this section we illustrate how FRED can be used to annotate a specialization interface of an object-oriented application framework with specialization patterns. We use JHotDraw [14] to demonstrate our approach. We will not go into the details of FRED or JHotDraw. Instead, we give an overview of the framework annotation process.

The user interface of FRED is shown in Figure 4. It contains a number of views to manage Java projects and specialization patterns. In the figure, the user is specifying specialization patterns for JHotDraw. She writes the patterns with *Pattern Editor* by creating roles and defining their properties and dependencies.

Pattern definitions are typically based on the analysis

of the framework source code and its documentation. The FRED environment provides a dedicated *Java Editor* for browsing and editing source code.

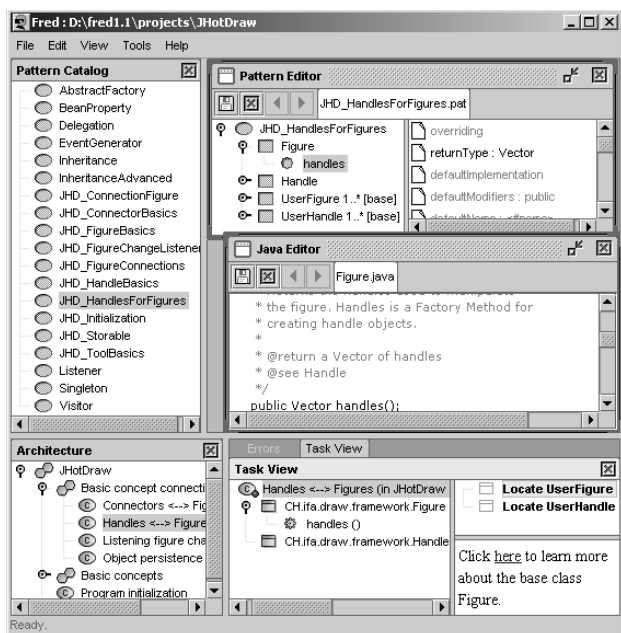


Figure 4. The user interface of FRED

The framework developer can also utilize existing general patterns from FRED’s *Pattern Catalog* as templates for her framework-specific patterns. The catalog contains, for example, versions of many generally applicable patterns like the design patterns presented in [4]. When the user wants to see her patterns in action, she can instantiate them and then examine the instantiated patterns in the *Architecture View*. The *Task View*, in turn, shows the task list for the selected pattern instance (cast).

3.1. Identifying framework hot spots

There are a number of useful heuristics that help in identifying and specifying a framework’s hot spots. The heuristics we discuss first are most relevant to *white-box frameworks*, which use inheritance as the main specialization technique. An overview of a more general technique is given later on in Section 3.4.

Here we assume that the framework has a layered structure and that its basic concepts are implemented on the highest layer as abstract interfaces. In addition, we assume that we are annotating a fairly mature framework and that we have enough information about the framework’s structure and its intended use.

Since any framework can be annotated in numerous different ways, the framework annotator must decide what kind of assistance she wants to give for the framework

users. Adding constraints will give the user better guidance. However, at the same time she will lose some of her freedom. The formalization of a framework’s specialization interface always reveals only a subset of possible implementation variations. We argue that it is better first to provide patterns for a quite narrow specialization interface, and later modify the patterns and add new ones to enable more advanced ways to use the framework.

Template and *hook methods* are obvious candidates when trying to locate the important hot spots of an object-oriented framework [3, 15]. Most of the hot spots can often be found by analyzing overridden methods, because polymorphism needed in hook methods is usually implemented using method overriding [16].

There are typically hundreds of overriding relationships between methods in any non-trivial application framework. That is why it is best to concentrate first on the main concepts of the framework and their relationships. The main concepts of the framework usually map fairly consistently with the top-level interfaces in the framework implementation.

Figure 5 represents the highest-level interfaces of the JHotDraw framework as a UML class diagram. By implementing these interfaces directly (or indirectly by adapting the default implementations provided with the framework) the user can have different kinds of figures, handles to grab them, connectors to link them together, and tools to manipulate them in her drawing application.

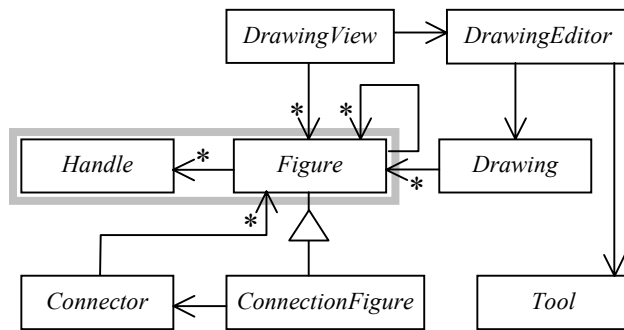


Figure 5. A hot spot in the JHotDraw framework

In our case study we needed 10 specialization patterns to annotate the main parts of the specialization interface of JHotDraw. As an example of a real, although somewhat simplified specialization pattern, we look at *HandlesForFigures*. As its name suggests, its purpose is to assist the application developer to define handles for her figure objects. This hot spot involves two framework interfaces highlighted in Figure 5.

Before explaining the details of this specialization pattern, it is important to distinguish between framework roles and application roles. A *framework role* is a role that

will be bound to a framework source code entity by the framework developer. Framework roles can be deduced from the source code directly. For example, there usually exists a one-to-one mapping between a framework interface representing a certain framework's concept and a framework role in a pattern describing ways to implement that interface.

An *application role*, on the other hand, is a role that will be bound to an application source code entity later on. Application roles typically depend on the framework roles and contain constraints that guide the framework adapter as she derives her application from the framework. The structure and constraints of application roles should condense the available information on the expected framework adaptations. This information can be gathered from the ready-made default components incorporated in the framework itself as well as from the existing applications already utilizing the framework.

3.2. Specifying class and method roles

Figure 6 represents a more detailed UML class diagram describing the hot spot related to the relationship between *Figure* objects and *Handle* objects in JHotDraw. The diagram shows that there can be a number of handles for each figure, and that each handle object knows its owner figure. The methods, which are relevant to that particular relationship and actually implement the association, are highlighted in the diagram.

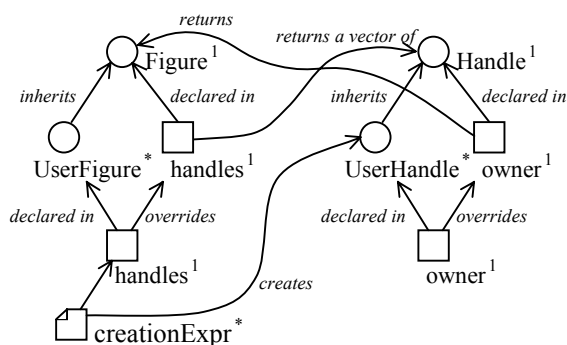
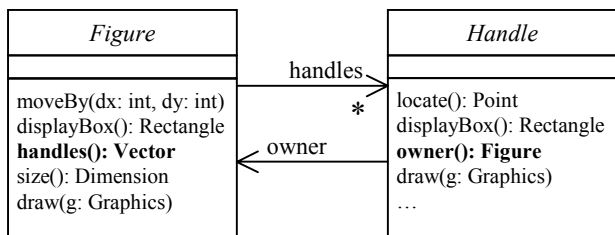


Figure 6. A pattern for defining handles for figures

Below the class diagram in Figure 6 there is a graph representation of the *HandlesForFigures* pattern. A part

of it is also visible in *Pattern Editor* in Figure 4. The pattern specifies that if the user wishes to have handles in her figures she must bind her *Figure* subclasses to the *UserFigure* role and then provide or generate a method, which overrides the *handles* method declared in *Figure* interface. Similarly, she must override the *owner* method in her *Handle* subclasses.

There are two class roles for both basic concepts involved in this hot spot: a framework role and a related application role. The framework role is named after the framework class it has been derived from (e.g. *Figure*). The corresponding application role (identified with the "User" prefix in Figure 6) represents the set of possible application-specific subclasses to be derived from the framework class. Thus, it has a dependency to the framework role and an associated inheritance constraint.

There is one method role for each class role in Figure 6. Those method roles that denote methods that are to be declared within framework classes have constraints restricting the types of their return values (e.g. *Handle*'s *owner* method must return a *Figure*). The application method roles, on the other hand, have overriding constraints referring to the corresponding framework method roles.

3.3. Roles for method implementation

So far we have discussed only class and method roles that guide the user to a particular hot spot of the framework. They indicate the classes she must inherit and the methods she must override, in order to adapt the framework. Specialization patterns can, however, be used also for representing various ways to code the actual implementation. For instance, we can use patterns to describe algorithms defined in method bodies, to show data fields that are needed to implement the algorithms, and to describe constructors required to initialize the fields.

In FRED, we describe method bodies and field initialization clauses with *code snippet roles*. Figure 6 shows an example of a code snippet (*creationExpr*), which has a dependency (*creates*) to the *UserHandle* class role. The snippet is used here to give the user a possibility to generate code for creating a number of concrete *Handle* objects that she can then return in a vector from her *handles* method.

In general, code snippets can be created under application roles as required. There can be many code snippets for one role, e.g. to describe algorithmic options. The framework developer should specify snippets as generalizations of the most representative examples among the existing implementations. The guiding roles and constraints for method bodies should be specified as suggestions, not as mandatory constraints. The same applies for field and constructor roles. On the other hand, selecting

the implementation strategy for a hook method to be overridden usually fixes the variation possibilities for the related fields and constructors, too.

3.4. Goal-oriented identification of hot spots

JHotDraw is a typical example of a framework, which uses inheritance and method overriding as a means to provide extensibility. FRED can be used also with frameworks that involve more advanced reuse techniques, such as *reflection* or *dynamic object composition*. For these cases we have come up with a more general way of identifying the hot spots of a framework. It is based on an analysis of the expected behavior of the framework specializer.

When starting to use a framework, one usually has a particular objective in mind or at least a hint of the desired outcome. We call such objectives pursued by the framework user as *specialization goals*. The solution to achieve a specialization goal may be well known and documented, or it can be found by examining existing applications based on the framework and by interviewing the framework's users. This resembles an *implementation case* [17] that describes how functionality for an application in the framework domain can be implemented using the constructs offered by the framework.

Achieving a specialization goal means that some of the framework's extension points must be satisfied. In case of specialization goals, these hot spots are not isolated from each other; instead, when pursuing the goal, the user may struggle with a number of hot spots and their complex interactions. The informal framework documentation does not necessarily describe these steps precisely, but has a more general view about the framework and its use.

To create goal-oriented specialization patterns the framework expert must recognize typical specialization goals, analyze the architectural aspects involved in these goals, and find the required tasks expected to be carried out by the application developer. Typically, from the standpoint of the framework user, specialization goals constitute a linked structure where achieving one goal leads to another.

As an example, Figure 7 presents specialization goals of a framework that is used to derive MVC (Model-View-Controller) applications. The MVC paradigm was first used in Smalltalk environment, and it aims at making a standardized separation between the graphical user interface and the rest of the application [8]. It divides the user interface into three kinds of components: models, views, and controllers. A view manages a region of the display and keeps it consistent with the state of the model. A controller converts user actions into operations between the view and the associated model. The example framework provides a skeleton to create such a system and the

framework expert has identified the specialization goals that most probably will interest the framework user. Note that the example is slightly simplified; new goals may be identified and the goals shown in the figure may be further divided into more specific sub goals.

One method to construct specialization patterns for a specific goal is to first derive an example specialization that achieves the goal. This example specialization helps the pattern modeler to identify the required program elements and their interactions. This process is similar to object-oriented analysis on the architecture level: central concepts of a specialization pattern are identified and associated with roles. In this way it is usually easy to find class and method roles. However, other aspects of the specialization pattern like constraints may be more implicit in an example specialization.

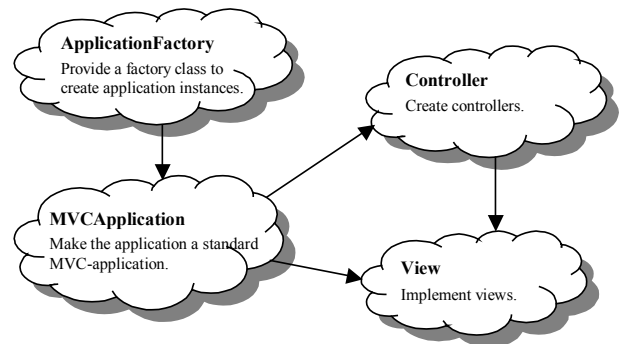


Figure 7. A set of specialization goals

Thus, the pattern modeler creates a set of specialization patterns to describe solutions for the identified specialization goals while the specializer utilizes this knowledge by doing tasks generated by FRED. This approach for finding specialization patterns was used in the network management GUI case study. In this way, we identified 13 specialization goals for the framework, each giving rise to a specialization pattern.

3.5. Pattern initialization

Once the framework developer has specified the framework hot spots as specialization patterns, she must make the framework annotation ready to be used. This *pattern initialization* means instantiating the specialization patterns and associating the framework roles with the corresponding framework source code elements. The rest of the roles will be left for the application developer to bind.

The *Architecture View* of FRED displays the instantiated patterns in the current project. In Figure 4, the user has selected an instance of the *HandlesForFigures* pattern described above. On the left hand side of the *Task View*,

one can see the current cast with bindings to classes *Figure* and *Handle*, as well as to method *handles*. The contracts for the application-specific subclasses are not bound yet, so they are represented on the right as tasks for the user to provide the missing classes. An HTML description of the selected task is shown below the task list.

3.6. Task-driven framework specialization

After the framework developer has initialized her patterns the framework annotation is finally ready to be used. At this point, FRED will create a task list for the framework adapter to systematically derive an application from the framework. Each task in the task list represents a certain contract and its associated role and constraints. The task list is inherently dynamic: solving a task may generate additional tasks. For instance, the creation of a class for an application role typically implies that some (abstract) method inherited from a framework class must be overridden and tuned for the specific application. In such a case, the creation of the application class is a task, which implicitly generates another task for method overriding as described in Section 2.

The generated tasks together with the cast representing the already bound roles are shown in FRED's *Task View*. 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. For example, from the pattern shown in Figure 6 FRED generates a list of tasks to attach handles for figures in an editor application. The list contains, e.g., an optional task for providing a class for the role *UserFigure* (see Figure 4) and, if the user carries out this task, a mandatory task for providing a method that overrides the *handles* method inherited from the framework class *Figure*.

Executable 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. For these, FRED provides a dedicated *Java Editor*, which parses the source code incrementally as the user types it in. Changes in the source code are monitored and their validity is continuously checked against the constraints specified in the patterns. Possible violations of constraints immediately result in new refactoring tasks. Hence, the proper use of the framework is constantly validated and supervised by the system. Besides the standard *Java Editor*, also more high-level (framework-specific) tools can be provided by extending FRED's general tool API.

When walking through the task list, the application can be developed step-by-step under the interactive guidance and documentation provided by FRED. This disciplined process makes sure that all the core hot spots of the framework are traversed and that the framework is extended by concrete code that is necessary to make the application complete. FRED keeps track of the status of the tasks, and the application is considered complete and executable when the user has done all the mandatory tasks.

In addition to following the tasks generated from the patterns defined for the framework, the application developer can herself instantiate general patterns that are suitable for her application and use them for producing application code. Typically these patterns would involve coding convention patterns like JavaBeans component patterns or *Singleton* design pattern [4].

4. Related work

4.1. Tool support for framework specialization

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* [8, 3] and *patterns* [5].

As emphasized in this paper, instructions for adapting a framework 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, 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* [9]. 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 [10]. This reminds our model, but while SmartBooks method provides 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 semi-formal templates for describing specialization points of frameworks [13] in the form of *hooks*. A hook presents a recipe as an imperative algorithm. This algorithm is intended to be read, interpreted, and carried out by the

framework adapter. Tool support has been suggested, but as the solution description within a hook is given in procedural form it may be hard to support the non-linearity of software engineering process.

Fontoura, Pree, and Rumpe present a UML extension *UML-F* to explicitly describe various kinds of framework variation points [11]. 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. Each variation point type has a dedicated tag. In addition, there are tags for differentiating between static and dynamic variation points (i.e., whether or not the variable information is available at compile time) as well as for identifying application-specific classes as opposed to classes belonging to the framework.

Fontoura et al. view UML-F descriptions as a structured cookbook, which can be executed with a wizard-like framework instantiation tool. This vision resembles closely that of ours. We see the framework specialization problem to be more complex than what is implied in [11], however. The proposed implementation technique is based on adapting standard UML case tools. This does not directly support interactivity in framework specialization.

To manage the complexity of large frameworks they should be organized into smaller and more manageable units. *Framelets* provide a way to do exactly that [18]. A framelet is a small framework with a clearly defined simple interface used for structuring new software architectures and especially for reorganizing legacy code. We have gained good experiences with annotating framelets with FRED patterns to make it easy to adapt and combine them into software systems.

4.2. Pattern-based tool support

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 [19] 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 [4], various formalizations have been given to design patterns resembling our pattern concept (for example, [6], [20], [21], and [22]), 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 [23] Eden, Hirshfeld, and Lundqvist present *LePUS*, a symbolic logic language for the specification of recurring *motifs* (structural solution aspects 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 [24] 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. Our model, however, is more analogous with programming languages and attribute grammars than with logic formalisms. In addition, we emphasize 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 envisage application development shifting towards using platforms like object-oriented frameworks, which support extensive reuse. So far there is relatively little tool support for this kind of software development, where the central problem is to build software according to the rules and mechanisms of the framework.

FRED represents a possible approach to produce adequate environments for framework-centric programming. It supports architecture-oriented programming by providing tasks, which guide the adaptation of reusable architectures realized as object-oriented application frameworks. The tasks are generated dynamically based on specialization patterns that specify the specialization interface of the framework.

We are aware of some restrictions in our current specialization pattern model. For instance, it does not allow dependencies between patterns, and it does not provide enough modularity within patterns. Also, currently in FRED there is no way to check that the user has defined a method body as intended by the framework designer. To allow more control, FRED could be augmented with a richer set of statically verifiable constraints like, for example, in *CoffeeStrainer* [25].

In order to further validate and enhance our methodology of using specialization patterns, we are going to apply it to a range of frameworks of varying sizes and characteristics. At the same time we will investigate ways to make it easier to import existing code into FRED for systematic management.

In the current version of FRED, annotating a framework with specialization patterns includes many trivial details that could well be automated. For example, many roles, dependencies, constraints, and default values could

be deduced directly from the framework source code and existing example applications using various kinds of heuristics. We could also apply the techniques developed for automatic discovery of design patterns from source code (see, e.g., [26]).

A possibility to automate trivial (one-to-one) role bindings would also greatly ease pattern initialization, i.e. the process of binding the specified patterns to the framework entities so as to provide an initial set of annotations for the user.

Also, since new ways of adapting a framework are found even in the application development process, the tool should make it possible to easily modify the patterns during the specialization process. Currently this is not possible. A potential solution to this problem is to make pattern instances more dynamic, modifiable entities.

Despite the restrictions mentioned above, 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 development environments and building FRED-based support for standard architectures like *Enterprise Java Beans*.

Acknowledgements

The FRED methodology and programming environment have been developed in a joint research project between University of Tampere, Tampere University of Technology, and University of Helsinki. The project has been funded by the National Technology Agency of Finland (Tekes) and by several software companies.

References

- [1] Fayad M., Schmidt D., Johnson R., (eds.), *Building Application Frameworks — Object-Oriented Foundations of Framework Design*. Wiley, 1999.
- [2] Bosch J., *Design & Use of Software Architectures — Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.
- [3] Pree W., *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.
- [4] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns — Elements of Object-Oriented Software Architecture*. Addison-Wesley, 1995.
- [5] Johnson R., Documenting Frameworks Using Patterns. In: Proc. *OOPSLA '92*, Vancouver, Canada, 1992, pp. 63-76.
- [6] Florijn G., Meijers M., van Winsen P., Tool Support for Object-Oriented Patterns. In: Proc. *ECOOP '97*, Jyväskylä, Finland, 1997, LNCS 1241, pp. 472-496.
- [7] FRED Site. <http://practise.cs.tut.fi/fred>, 2001.
- [8] 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, pp. 26-49.
- [9] Ortigosa A., Campo M., SmartBooks: A Step Beyond Active-Cookbooks to Aid in Framework Instantiation. In: Proc. *TOOLS '99 Europe (Technology of Object-Oriented Languages and Systems)*, Nancy, France, 1999, IEEE Press, pp. 131-140.
- [10] Ortigosa A., Campo M., Salomon R., Towards Agent-Oriented Assistance for Framework Instantiation. In: Proc. *OOPSLA '00*, Minneapolis, Minnesota, ACM SIGPLAN Notices, 35, 10, 2000, pp. 253-263.
- [11] Fontoura M., Pree W., Rumpe B., UML-F: A Modeling Language for Object-Oriented Frameworks. In: Proc. *ECOOP '00*, Sophia Antipolis and Cannes, France, 2000, LNCS 1850, pp. 63-83.
- [12] Lajoire R., Keller R., Design and Reuse in Object Oriented Frameworks: Patterns, Contracts and Motifs in Concert. In: *Object Oriented Technology for Database and Software Systems*, Alagar V., Missaoui R. (eds.), World Scientific Publishing, Singapore, 1995, pp. 295-312.
- [13] Froehlich G., Hoover H., Liu L., Sorenson P., Hooking into Object-Oriented Application Frameworks. In: Proc. *ICSE '97*, Boston, Massachusetts, 1997, pp. 491-501.
- [14] JHotDraw 5.1 source code and documentation. <http://members.pingnet.ch/gamma/JHD-5.1.zip>, 2001.
- [15] Schauer R., Robitaille S., Martel F., Keller R., Hot Spot Recovery in Object-Oriented Software with Inheritance and Composition Template Methods. In: Proc. *IEEE Int. Conf. on Software Maintenance 1999 (ICSM '99)*, Keble College, Oxford, England, 1999, IEEE Press, pp. 220-229.
- [16] Demeyer S., Analysis of Overridden Methods to Infer Hot Spots. In: Proc. *ECOOP '98 Workshops, Demos, and Posters (Workshop Reader)*, Brussels, Belgium, 1998, LNCS 1543, pp. 66-67.
- [17] Pasetti A., Pree W., Two Novel Concepts for Systematic Product Line Development. In: *Software Product Lines: Experience and Research Directions (First Software Product Lines Conference, Denver, Colorado)* (Donohoe P. ed.), Kluwer Academic Publishers, 2000.
- [18] Pree W., Koskimies K., Framelets — Small is Beautiful. In: *Building Application Frameworks — Object-Oriented Foundations of Framework Design* (Fayad M., Schmidt D., Johnson R., eds.), Wiley, 1999, pp. 411-414.
- [19] Holland I., The Design and Representation of Object-Oriented Components. Ph.D. thesis, Northeastern University, 1993.
- [20] Meijler T., Demeyer S., Engel R., Making Design Patterns Explicit in FACE — A Framework Adaptive Composition Environment. In: Proc. *ESEC/FSE '97*, Zurich, Switzerland, 1997, LNCS 1301, pp. 94-110.
- [21] Mikkonen T., Formalizing Design Patterns. In: Proc. *ICSE '98*, Kyoto, Japan, 1998, IEEE Press, pp. 115-124.
- [22] Riehle R., Framework Design — A Role Modeling Approach. Ph.D. thesis, ETH Zürich, Institute of Computer Systems, 2000.

- [23] Eden A., Hirshfeld Y., Lundqvist K., LePUS — Symbolic Logic Modeling of Object Oriented Architectures: A Case Study. In: Proc. *Second Nordic Workshop on Software Architecture (NOSA '99)*, University of Karlskrona/Ronneby, Ronneby, Sweden, 1999.
- [24] Alencar P., Cowan C., Lucena C., A Formal Approach to Architectural Design Patterns. In: Proc. *3rd International Symposium of Formal Methods Europe (FME '96)*, St Hugh's College, Oxford University, England, 1996, pp. 576-594.
- [25] Bokowski B., CoffeeStrainer — Statically-Checked Constraints on the Definition and Use of Types in Java. In: Proc. *ESEC/FSE '99*, Toulouse, France, 1999, ACM Software Engineering Notes 21, 6, 1999, pp. 355-374.
- [26] Krämer C., Prechelt L., Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: Proc. Working Conference on Reverse Engineering (WCRE '96), Monterey, California, 1996. IEEE Press, pp. 208-215.

Chapter 7.

Feature Models, Pattern Languages and Software Patterns: Towards a Unified Approach

Hakala M.: Feature Models, Pattern Languages and Software Patterns - Towards a Unified Approach. In: Proceedings of the Nordic Workshop on Software Development Tools and Techniques (K. Osterbye, ed.), NWPER'2002, IT University of Copenhagen, 2002.

FEATURE MODELS, PATTERN LANGUAGES AND SOFTWARE PATTERNS: TOWARDS A UNIFIED APPROACH

Markku Hakala
Tampere University of Technology
markku.hakala@cs.tut.fi

1. INTRODUCTION

Feature models [Kan90] and pattern languages [Ale79] are similar ways of specifying domain specific languages (DSL). Whereas feature models may be used in deriving a matrix computation library from the family of matrix computation libraries [CzE00], a pattern language could be used in deriving a shopping mall from a family of shopping malls [Ale75]. Both approaches are ways of specifying a domain as a language, so that given the requirements of a specific application it is possible to derive a sentence of that language which leads to a concrete solution.

Fred [Hak01a, Hak01b] is a programming environment that allows the specification of recurring program implementation in a reusable form. Fred does not provide full automation in code generation, but it supports domains where full automation is unachievable or undesirable, such as in the implementation of design patterns and specialization of white-box frameworks.

The way of specifying recurrence in Fred resembles the way DSLs are specified by feature models and pattern languages. Whereas feature models model the problem space of some domain, Fred captures patterns in the solution space. Despite the differing levels of abstraction, the approaches are syntactically similar, and thus it might be reasonable to investigate the possibility of combining the approaches. Given the code generation facilities offered by Fred, a unified approach would allow modeling of both the problem and solution spaces of a given domain as a single DSL, permitting a seamless tool-supported transition from requirements to code.

We aim in a system that would provide high degree of code generation and structural validation, still maintaining its ease of use and applicability to a wide range of technologies such object-oriented framework specialization, instantiation of design patterns and coding idioms, and application of generic architectural styles. The approach in itself is not tied to any particular problem or solution domain, although the implementation is likely support only a restricted set of programming languages.

Chapter 2 provides motivation for this paper by discussing the role of code generation in software engineering and in particular, arguing that full automation is not always enough. Chapter 3 briefly outlines the three existing approaches. A sketch for a unified model is constructed in Chapter 4. Chapter 5 concludes the paper.

2. AUTOMATION IN SOFTWARE CONSTRUCTION

Generative programming aims at full automation in software construction [CzE00]. A generative programming tool assumes a high-level program specification, and is able to generate a software component based on that specification. Typically, the process of writing the input specification is separate from generating the output. Thus, generative programming tools are black boxes that perform input-output transformations. Like compilers, they do not support any interference in between. Preprocessors, template programming and program generators all fall into this category.

The classical generative approach results in total isolation from the produced code, which is excellent, assuming we get full automation. However, sometimes it's not possible – or even desirable – to achieve full automation. This is especially true with software patterns and white-box frameworks; it would be nice to get generative support for design patterns, but patterns cannot be generated in isolation. Instead, they intertwine with the rest of the code. Similarly, white-box frameworks are open by their nature. We can't specialize such a framework by giving a featural description, simply because being white-box it supports unanticipated features. Still, there is a desperate need for code generation for these domains too fuzzy for full automation. Moreover, because of the human component, partial code generation must be complemented with code validation, and is even harder to achieve than full automation.

3. EXISTING TECHNOLOGIES

Fred

Fred (Framework Editor) is a software development environment prototype for Java being developed since 1997 at the Tampere University of Technology and University of Helsinki. Fred helps to capture recurring textures within software in a form that supports systematic generative tool support. Suitable areas of application range from implementation patterns to architectural conventions and white-box framework specialization [Hau02, Vilo1]. The tool has been discussed in [Hako1a, Hako1b] and demonstrated in [Hako1c]. It is available for download with a tutorial at <http://practise.cs.tut.fi/fred>.

In Fred, recurrence is stored in hierarchical structures called *patterns*. Being very implementation-oriented, these should not be confused with design patterns, although implementation-oriented variants of design patterns can be represented in Fred.

A Fred-pattern is essentially a very fine-grained implementation-oriented DSL, presented as a tree with cross-references. During software development, the user instantiates the pattern by walking through the tree under the guidance of the tool, resulting in generated code and documentation. Moreover, instructions on instantiating the pattern are generated on the fly. As a result, Fred releases the user from tedious programming tasks, but at the same time promotes learning by doing.

A minimal Fred-pattern is depicted in Figure 1. The pattern is used in Fred to generate accessor methods for member variables. The pattern consists of *roles* that will be bound to concrete implementation elements during the pattern instantiation. Roles have cardinalities and syntactic constraints in the form of cross-references.

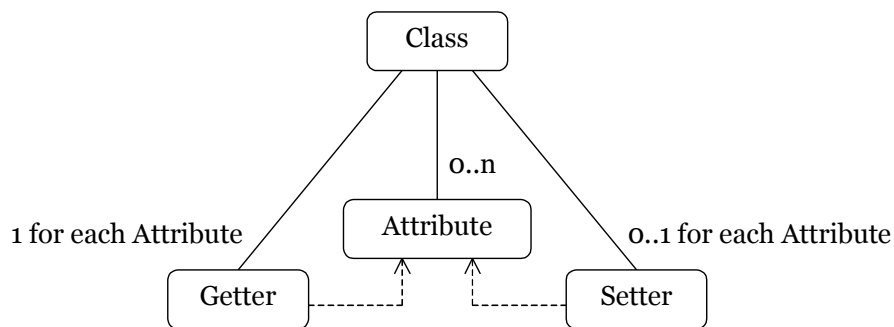


Figure 1. A minimal Fred-pattern

This diagram presents a grammar for a small DSL. Although suppressed from the diagram, semantic information is attached to the nodes to provide code generation, constraint checking and adaptive documentation facilities. E.g. after typing in a string field “name” in a class, Fred is able to provide suggestions like “Provide a setter method for the field *name*” to the developer. Like the instructions, source code can be generated on demand. Following the tasks provided by Fred eventually results in a complete application of the pattern.

Feature Models

A feature model represents and classifies instances of a particular concept providing a structural representation of the possible properties (i.e. features) of the concept instances. Feature models are used in describing variability and commonality of software components within a family of software artefacts. However, although they model variability within software they don’t imply any specific means of implementing that variability. Thus, feature models present a convenient way of describing software product or system families at a high level of abstraction.

A feature model is represented as a hierarchically structured diagram. The nodes of this diagram are called *features*, and they present individual configuration points within the family. Features range from high-level requirements to implementation-specific configuration details.

Feature diagrams are used in deriving *featural descriptions* of instances of the family. This process involves traversing through the feature diagram in a well-defined manner binding each traversed feature to some specific configuration. The traversal must proceed according to the syntactic rules of the diagram and each binding must adhere to the semantic description of the particular feature. The result is a set of bindings that uniquely describes a certain piece of software.

Feature Models, Fred and Pattern Languages

Feature diagrams and Fred-patterns are both used in deriving concrete software artefacts from generic descriptions. They differ in their level of abstraction however; where Fred concerns the recurring implementation strategies, feature models attempt to be abstract in this sense.

Despite this, the methods are syntactically very similar. They provide similar means of presenting a hierarchically structured DSL; the tree of features effectively corresponds to the tree of roles. Both approaches also incorporate grammatical constraints – hard dependencies between the elements of the language.

There are syntactic differences too, most importantly the notion of cardinality. This concept essential to Fred is lacking from the feature models in an attempt to maintain them independent of any structural information of the solution space.

All in all, the approaches differ in two aspects; 1) their intended use (level of abstraction), and 2) the expressive power (i.e. the set of languages they can represent).

There is yet another related approach, however. It has turned out that the way patterns are applied in Fred resembles the way pattern languages [Ale79] are applied, although Fred-patterns are of much smaller scope than pattern languages. In essence, Fred-patterns are fine-grained pattern languages, the same way they can be conceived as fine-grained feature models.

The three approaches are far from being identical, but the similarities suggest that the works are characterizing a similar phenomenon. Table 1 outlines these approaches.

Table 1. Terminology and features of the three related approaches

	<i>Feature model</i>	<i>Fred pattern</i>	<i>Pattern language</i>
Primary elements	features	roles	patterns
Primary structure	tree	tree	graph
Grammatical constraints	requires and mutual-exclusion constraints, default-dependency rules.	cardinality and positive, negative, hard and soft dependencies.	informal annotations

	alternative and or-features	alternative roles*	
Language sentence	featural description	pattern instance	sentence
Example domain	Matrix computation components [CzE00]	Java MVC-framework [Hau02]	Shopping malls [Ale75]

*) Some of the grammatical constraints are not yet implemented in the latest Fred release.

All the approaches are ways of presenting a grammar for a DSL as a graph. Feature models and Fred model some primary structure within the graph as a tree, augmenting it with cross-references (Fred-patterns have also been described using graphs instead of trees, see [Hako1b]). The cross-references are called constraints in feature models and dependencies in Fred.

4. UNIFIED APPROACH

Fred has been promising in supporting the reuse of software patterns, white-box frameworks and generic architectures such as EJB [HaKo2]. However, Fred-patterns mostly model recurrence in the solution space (the code), instead of the problem space (the requirements). This results in the inability to support high-level specifications that would disregard the actual implementation.

Feature models suggest the notion of vertical constraints to map high-level specification features onto implementation-level features. During the Fred project the notion of cardinality has proven to be valuable in modeling recurring patterns in the solution space, even though cardinality may not play an important role (or it may be circumvented) in modeling the problem space. Hence, lacking the concept of cardinality, feature models themselves are inadequate in providing support for code generation.

Our vision is to use Fred machinery to support specifications of various levels of abstraction. Vertical constraints binding the featural model of the problem space to the patterns of the solution space, could serve as a foundation for the attempt.

We are currently carrying out a project that continues the development of Fred by generalizing its implementation to support patterns, feature models and pattern languages of arbitrary domains and granularity. The terminology for the new model is adopted mostly from the field of pattern languages, reusing existing Fred-terminology whenever possible. Thus, a DSL will be called a pattern language, being composed of patterns.

The main problem in combining the approaches rises from the fact that creating a language capturing a given domain all the way from high-level requirements to variable implementation details would be an overwhelming effort. The problem could be sized down considerably however, by making it possible to reuse existing patterns and pattern languages in creating new pattern languages. The reuse mechanisms planned for Applause are specialization and dynamic composition of pattern languages.

Pattern Language Specialization

A pattern language is a description of a certain domain, such as a system or product family. Typically, it is applied to get a description of a specific instance of that family. However, we consider this as a special case. In general, application of a pattern language corresponds to refining the original language to a narrower domain, and results in a new pattern language. This language may indeed cover only a single instance of the family, but that is not necessary. Thus, applying a pattern language is called *specialization*. This is the fundamental reuse mechanism for pattern languages. E.g. a pattern language describing the structure of a design pattern may be specialized to get a pattern language for a specific implementation pattern.

Foundations of Approach

Pattern language is a structure of patterns. The patterns are organized in a parent-child hierarchy with cross-references. Figure 2 depicts a pattern language of four patterns, with pattern Server as the root, and a cross-reference from Handler to Request.

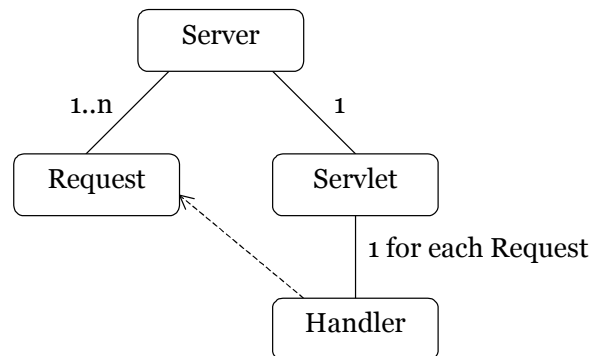


Figure 2. A pattern language for creating Java servlets

This pattern language is an abbreviated description on how to create server applications that handle multiple types of requests from clients. Although suppressed from the diagram, each pattern encapsulates a description of certain part of this domain. E.g., the pattern Request stands for the requests the server should be able to handle. Servlet-pattern describes how to implement the server using Java Servlet technology. It is supplemented by Handler, which describes the handler methods that should be provided for each request.

In general, the patterns provide a vocabulary for the domain, and the relationships between patterns form a grammar for composing sentences describing sub-domains or specific instances of the original domain. In other words, pattern language is always applied to create a new language. Creating a new pattern language corresponds to refining patterns of the original language according to the semantics of the original patterns and the grammatical rules of the original language.

Figure 3 presents another pattern language. It describes a servlet that is able to respond to two types of requests (log in and fetch). Clearly, this language describes a sub-domain of the previous pattern. It has been created according to the grammatical rules of the previous language. Patterns LogIn and Fetch refine the pattern Request of the previous pattern language. Similarly, the original Handler-pattern is refined for the two types of requests.

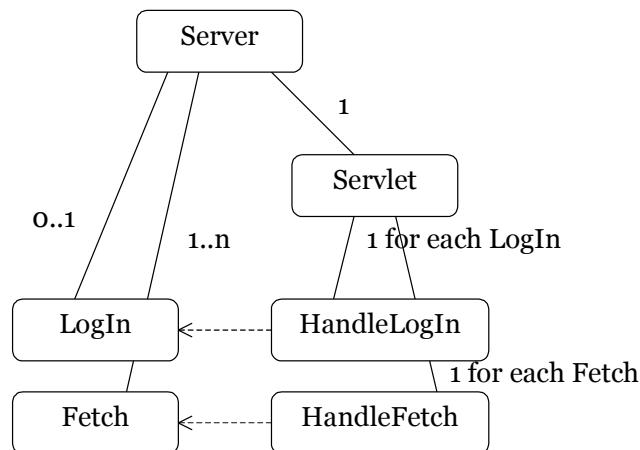


Figure 3. A refined pattern language for Java servlets

This language can be further refined, however. The logging functionality is marked as optional, and there can be several kinds of fetch-requests. A new language may introduce a fixed set of requests, or even dictate the exact implementation of the handler methods. In case the domain is completely fixed and thus cannot be further refined, the pattern language effectively describes a single application. In other cases, the language encapsulates architectural configuration knowledge of a certain domain.

If we look at three patterns on the left side of the previous figure, we see something that looks like a feature model with cardinalities. A more specialized language with no cardinalities would correspond exactly to a conventional feature model. Application of a pattern would then correspond to selecting the particular feature. E.g., the logging functionality could be selected by applying the LogIn pattern.

The other patterns in the figure describe low-level implementation features. As we have dependencies between the sides, the high-level decisions are effectively mapped to low-level implementation strategies. With similar bindings, it is possible to bridge problem space to solution space, and eventually produce code based on the low-level pattern descriptions.

The approach does not impose any particular requirements on the patterns. Our focus has been in working on the mechanical interpretation of pattern languages and thus we're trying to leave the semantics of patterns as open as possible, allowing arbitrary extensions to the

approach. However, three important kinds of patterns can be identified based on what kind of information they contain and how they should be applied. These are features, roles and constraints. They are all considered patterns in our approach; they all describe a domain, but with different kind of content.

FEATURES

Features correspond closely to the features of feature models. They encapsulate some configuration knowledge. Refining a feature means narrowing the variability captured by the feature. Traditionally a feature in a feature model describes a point and bounds of variation, and a featural description (the result of the application of a feature model) would select a unique value within the range of variation. In general, the feature in a refined language must define a sub-domain of the original feature, but it does not need to be a single value.

In the previous example, the patterns `LogIn` and `Fetch` are examples of features, refining the `Request` feature.

ROLES

The second important category of patterns are roles. They are descriptions of program elements such as classes, methods, fields and so on. A pattern language may leave the actual program element open, in which case the pattern may be further refined in subsequent pattern languages, or bind the role to a specific piece of code.

The description of a role should encapsulate the knowledge on writing the required piece of code, or generating the code automatically. The description can include templates and scripts that make use of the dependencies within the pattern language, allowing code and instructions to be generated according to the other generated elements and selected featural configuration. E.g., the code template for the `Getter` method role in Figure 1 could be “`return #Attribute#;`”, expanding to something like “`return name;`” during the development time.

In our previous example, the pattern `Servlet` is a class role, and patterns `HandleLogIn` and `HandleFetch` are method roles.

CONSTRAINTS

The third class of patterns is constraints. They describe conditions on the program elements bound to roles.

There are no constraint patterns in the previous example, but it could be augmented with constraints. E.g., it would be natural to

add a constraint demanding the servlet class (bound to role Servlet) to extend the HttpServlet base class declared by Java Servlet API. This requirement could be imposed as a constraint pattern, attached as a child of the Servlet role.

To achieve generative tool-support, there must be a tool-supported format for defining the semantics of patterns. The tool could then use a pattern language as the basis of guiding the developer through developing a new application for the domain, or defining a more restricted sub-domain. The developer is responsible of applying the pattern language, but the tool makes sure that the pattern language application is syntactically correct, and enforces the semantic constraints embedded in pattern descriptions. Scripts provide automatic and semi-automatic code generation, and architecture-specific violations could be reported at compile-time based on constraint descriptions.

At the heart of this are the features that are used in gathering the featural description of the system under construction. Depending on the nature of the pattern language the system could be generated automatically based on the choice of features, or the features could be used as the basis of co-operative software development between the tool and the developer.

Pattern Language Development

The approach makes no distinction between deriving software systems based on pattern languages and deriving new pattern languages from old ones. An important and immediate consequence is that pattern languages itself can be developed exactly the same way as the applications are derived from the pattern languages. Thus, if we have a generative tool for software, we also have a generative tool for describing feature models, design pattern variants, architectures, and everything that can be captured by the kinds of pattern languages proposed.

The discussion has also discovered why cardinalities do play an important role in after all. Even though cardinalities do not play an important role in conventional feature models, a generic DSL would be hard to write without them. E.g., without cardinalities, the language in Figure 3 could not be considered a specialization of language in Figure 2, and thus no tool support could be provided in that language.

A continuation to our previous example is depicted in the Figure 4. Patterns Server, LogIn, Search and Download essentially define a simple feature model with no cardinalities. The roles Servlet, HandleLogIn, HandleSearch and HandleDownload can be used to generate code for the application based on the selected choice of features. The pattern language is a refinement of a more general language, the one we saw before, and Search and Download are actually refinements of Fetch-feature. Thus, a feature model can exist without cardinalities, but they are central to general-purpose pattern languages.

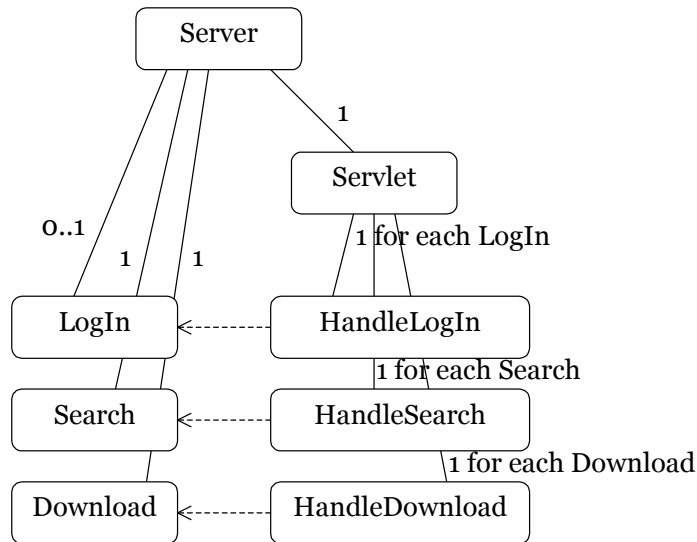


Figure 4. A pattern language with limited cardinalities only

It is possible for a pattern language to combine multiple existing pattern languages. E.g., continuing with our example, it would have been possible to create the original example as a refinement of two languages – a pattern language for servers in general, and a pattern language for servlets. The figure below depicts these primordial languages, however omitting many details as in previous examples.

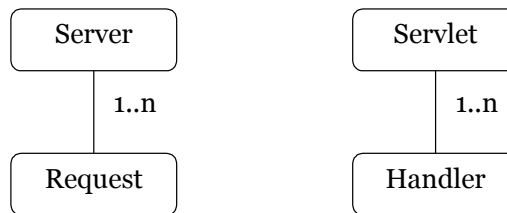


Figure 5. Simple pattern languages that could have been used as the basis of composing the language in Figure 2

Dynamic Composition of Pattern Languages

Pattern language specialization alone is not sufficient mechanism for reusing pattern languages, however. Consider we were supposed to describe a pattern language for web servers, as in previous examples, but wanted not to restrict the language on any particular implementation technology, like Java servlets. In general, sometimes we would like to leave a part of the pattern language open, to be refined later, in the subsequent specialization of the language. For this purpose, another mechanism is required.

Dynamic composition of pattern languages means that in certain pattern language we leave some parts unspecified by introducing a placeholder so that another pattern language can augment the original specification at later time. This requires that we can somehow specify the boundary

between two pattern languages, and the required characteristics of these parts. Thus, a typing system for pattern languages is required.

Figure 6 repeats our original example, but this time using dynamic composition. The pattern marked with dashed border indicates a slot to be fulfilled by some other pattern language. A dashed arc in the diagram indicates that this pattern language is able refer to the Request pattern of the host language by the name R.

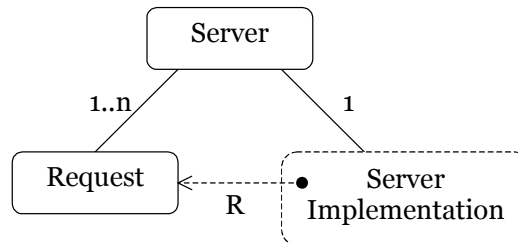


Figure 6. A pattern language that makes use of the dynamic composition

Another pattern language is presented in Figure 7 that may be used in place of the server implementation of the previous pattern language. This language describes a server implementation using Java servlets.

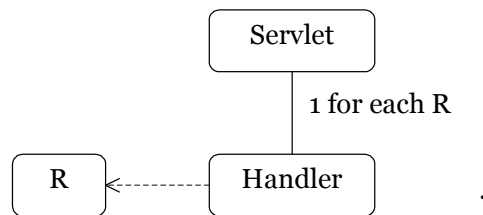


Figure 7. A pattern language that can be used to fulfill the slot of the language in Figure 6

The pattern language defines an external pattern R. This is used in gluing the two pattern languages together. When using this pattern language to fulfill the slot in the previous language, Request-pattern stands for R. As a result, we get a language that looks like our original example, but has resulted from the dynamic composition of pattern languages. This allows us to specify multiple variations of server implementation independently of the original pattern language. We can even apply more fine-grained decomposition by specifying the contents of the Handler-pattern as a separate pattern language. This would allow different kinds of handler implementations according to differing needs. E.g., handlers could be implemented as ordinary methods, or utilizing the Command design pattern [Gam94] (which in turn could be presented as another pattern language).

5. CONCLUSION

Feature models and pattern languages have much in common. This paper has outlined our attempt in capturing the essence of these approaches to

provide systematic architectural tool-support. The approach is based on the experiences of Fred-project, extending the DSL-processing machinery already present in the Fred-tool. The resulted model itself is independent of any particular problem or solution domain in the same sense feature models or design patterns can be applied in variety of domains.

The promise of the unified approach is in bridging languages of problem space to the languages of solution space, providing seamless, tool-supported and traceable transition from requirements to code. However, the problem in combining the approaches lies in the complexity and size of the resulted DSLs, and must be confronted by DSL-level reuse mechanisms.

We have briefly outlined ideas on pattern language specialization and dynamic composition, as mechanisms for specifying domain specific languages by reusing others. It is our aim to provide a framework that would allow development of reusable pattern language libraries capturing design abstractions in the same way object-oriented frameworks are used in capturing architectures at implementation level.

The approach itself is not tied to any particular programming language. At the time of writing, we have a tool prototype supporting pattern language specialization, and are continuing to investigate dynamic composition. Fred 2.0, incorporating these features, as well as a Java-framework for extending support for different programming languages and solution domains, is to be released in the first quarter of 2003. The release incorporates support for Java and XML.

REFERENCES

[Ale75] Alexander, C., Silverstein M., Angel S., Ishikawa S., Abrams D., The Oregon Experiment, Oxford University Press, 1975.

[Ale79] Alexander C., The Timeless Way of Building, Oxford University Press, New York, 1979.

[CzE00] Czarnecki K., Eisenecker U. W., Generative Programming: Methods, Tools and Applications, Addison-Wesley, 2000.

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

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

[Hak01b] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J., Generating Application Development Environments for Java Frameworks. In: Proceedings of the 3rd International Conference of

Generative and Component-Based Software Engineering (GCSE'01), Erfurt, Germany, September 2001, Springer, LNCS2186, 163-176.

[Hak01c] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., Viljamaa J., Architecture-Oriented Programming Using Fred. In: Proceedings of International Conference of Software Engineering (ICSE'01), Toronto, May 2001, 823-824. (Formal Research Demo)

[HaKo2] Hammouda I., Koskimies K., Generating a Pattern-Based Application Development Environment for Enterprise JavaBeans. Accepted for presentation in Computer Software and Applications Conference (COMPSAC'02) Oxford, August 2002.

[Hau02] Hautamäki J., Task-Driven Framework Specialization – Goal-Oriented Approach. Licentiate thesis, Department of Computer and Information Sciences, University of Tampere, January 2002.

[Kan90] Kang K., Cohen S., Hess J., Nowak W., Peterson S., Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990.

[Vilo1] Viljamaa A., Patter-Based Framework Annotation and Adaptation – A Systematic Approach. Licentiate thesis, Department of Computer Science, University of Helsinki, June 2001.