

A Tool Environment for Aspectual Patterns in UML

Imed Hammouda, Mika Katara, and Kai Koskimies
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
{imed.hammouda, mika.katara, kai.koskimies}@tut.fi

Abstract

An aspectual pattern is a pattern that captures a generic aspect. It is argued that the realization of an aspect in the form of a pattern is beneficial because various mechanisms available in generic pattern-based environments become available for aspects as well. These benefits follow mostly from the fact that the structural information concerning an aspect is clearly defined and preserved (as a pattern) separately from the actual system description. We demonstrate the concept of an aspectual pattern and its application in the case of the JUnit testing framework. A prototype tool environment supporting aspectual patterns in UML has been developed.

1 Introduction

Modeling has become an essential practice in software development, allowing complex systems to be understood at a high level of abstraction. UML (Unified Modeling Language [24]) has been widely adopted as de facto standard notation for expressing software models. However, software systems are inherently multi-dimensional in the sense that no single viewpoint or structuring can fully explain a system, even at a high level of abstraction. In general, the need for overlapping viewpoints in system descriptions has been widely acknowledged (e.g., [23]). Thus, we can understand and manage software systems best in terms of model slices, each covering a particular viewpoint only. This has been the basic motivation for aspect-oriented development [4], which strives for describing the system properties relevant for a particular viewpoint separately, and merge these descriptions by automated means into a comprehensive system description. Since we are here interested in modeling, we assume static merging (weaving).

On the other hand, patterns have emerged in software engineering as a concept for expressing solutions to recurring problems. The general aim of the pattern movement is to

raise the quality level of software systems by documenting solutions that are known to yield certain desired quality attributes in many existing systems. Depending on the nature of the problem, we may speak of analysis patterns [8], architectural patterns [2], design patterns [2], coding patterns [2] etc. Essentially, a pattern describes a collection of software entities (like, say, UML modeling elements) which collaborate in a certain way to solve a stated problem. A pattern is described independently of any particular system, in terms of generic roles which are replaced by actual software elements when the pattern is applied. If we detach the pure structural character of a pattern from the purpose of the pattern, a pattern can be simply understood as an arrangement of interrelated roles of software elements, crosscutting any other structuring of a system in which the pattern is applied.

Since an aspect can be viewed as a collection of software elements relevant for a particular concern, from a structural point of view the concepts of an aspect and a pattern come close to each other: both capture a crosscutting slice of a system that is logically meaningful for the understanding of the system. The weaving of an aspect into a full system description corresponds to the binding of the roles of a pattern to the concrete elements of a system. Thus, the idea of unifying the concepts of an aspect and a pattern seems in many ways attractive. In particular, using the tool technology developed for patterns we can achieve a number of advantages in the context of aspects. These include:

- Aspects can be weaved interactively and incrementally, under the guidance of the designer. This makes the weaving process open to the designer, and allows for customizable weaving.
- Weaving is not a one-shot action but it can be done partially if desired. This is useful if an aspect needs to be introduced but all the participants of the aspect are not available yet.
- An individual aspect can be easily viewed or highlighted in a system. This is useful for generating views

that help to understand the system.

- The information about an aspect appearing in a system is preserved and maintained. If the system is later changed so that an aspect is affected, the tool keeps track of the properties required by the aspect and shows possible violations of these properties.

Briefly, an aspectual pattern is a pattern that represents an aspect. We have built a prototype tool environment which supports aspectual patterns in UML modeling, using a generic pattern engine [10] as the core component of the environment. In this paper, we demonstrate how such a tool environment can be exploited for aspects.

The remaining of the paper is organized as follows. In Section 2 we briefly discuss the main characteristics of patterns, aspects, and aspectual patterns. In Section 3 we present in more detail the UML-based pattern concept we have used in this work. In Section 4 we apply aspectual patterns in developing the design model of the JUnit testing framework. In Section 5, we discuss our prototype tool environment and show how it is used in applying aspectual patterns. Related work is discussed in Section 6. Finally, in Section 7 conclusions are drawn and possible future work is highlighted.

2 Basic Concepts

In this section, we review the main technologies used in this paper: patterns and aspects. We show how the two concepts can be merged into so-called aspectual patterns.

2.1 Patterns

A pattern is an arrangement of software elements for solving a particular problem. In the sequel we will give a simple structural characterization of a generic pattern concept. To be able to define a pattern independently of any particular system, a pattern is defined in terms of element roles rather than concrete elements; a pattern is instantiated in a particular context by binding the roles to concrete elements. A role has a type, which determines the kind of software elements that can be bound to the role; the set of all the role types is called the domain of the pattern. Here we assume that the domain of a pattern is UML; that is, the roles are bound to UML model elements.

Each role may have a set of constraints. Constraints are structural conditions that must be satisfied by the model element bound to a role. For example, a constraint of association role P may require that the association bound to P must appear between the classes bound to certain other roles Q and R.

A cardinality is defined for each role. The cardinality of a role gives the lower and upper limits for the number of

the instances of the role in the pattern. For example, if an operation role has cardinality 0..1, the operation is optional in the pattern, because the lower limit is 0.

2.2 Aspects

In software engineering, separation of concerns refers to the ability to identify those parts of software artifacts that are relevant to a particular concept, goal, task, or purpose. Concerns are the primary motivation for organizing and decomposing software into smaller, more manageable and comprehensible parts. Aspect-oriented software development (AOSD) [4], which is a direct implication of the separation of concerns principle, has been proposed as a solution to cope with the characteristics of software that are difficult to capture with other development approaches such as object-oriented development. These characteristics are basically the different concerns cutting across several classes or other units of decomposition.

Aspect-oriented programming (AOP) [18] is a programming paradigm implementing the ideas of aspect orientation. AOP organizes the crosscutting concerns into separate modules called aspects. AspectJ [1] has been the most popular language for AOP. AspectJ is a general-purpose aspect-oriented extension to Java that provides support for modular implementation of a range of crosscutting concerns.

At the design level, there have been many proposals on how to model aspects in UML, for instance [3, 17]. Using current modeling languages, such as UML, it is often hard to identify the model elements that are relevant to certain concerns only. Similar problems arise when superimposing existing models with new model elements. In this regard, it is argued that AOSD techniques can be useful for model development, in general.

Traditionally, AOP has been applied to weave new functionality into programs statically by instrumenting the source code. Recently, dynamic weaving during runtime has become a more flexible option supported by several tools. At the modeling level, however, static weaving remains a useful approach since we usually do not want to get into details of specific implementation mechanisms, such as the moment of actual code weaving.

2.3 Aspectual Patterns

An aspectual pattern is a pattern that captures an aspect. When implemented as patterns, aspects are represented using a role structure that can be instantiated and weaved into base models (applications). The weaving corresponds to the binding of the roles: each role stores the information of a joint point. The constraints associated with a role determine the context where the aspect may appear, and the constraints can be used to check whether the aspect, implemented by

Constraint Role type	Stereotype	Abstract	Visibility	Inheritance	Multiplicity	Aggregation	Return Type	Parameter	Overriding	Type	Navigability	Participant
UML Package	X											
UML Class	X	X	X	X								
UML Operation	X	X	X				X	X	X			
UML Attribute	X		X							X		
UML Association	X											
UML Association End					X	X					X	X
UML Realization	X											X
UML Dependency	X											X

Figure 1. Pattern roles and constraints

the pattern, is correctly weaved. In contrast to traditional weaving, however, the weaving of aspectual patterns is considered as an interactive, incremental process where the join points are located under the guidance of a tool, rather than in a fully automated fashion. Aspect overlapping can be represented and implemented in a straightforward way using role-based pattern composition techniques: a model element can play different roles in different aspectual patterns. Another important benefit of bridging patterns and aspects for AOSD is the readily available tool technology for pattern-oriented development.

3 Aspectual Patterns for UML

In order to apply aspectual patterns at the design level for model development, we have defined pattern roles to represent a subset of the UML metamodel. Therefore, the domain of the patterns is UML. In this work, however, we will restrict the application of aspectual patterns to UML class diagrams. For this, we have specified the roles shown in Figure 1. Roles are used to represent different kinds of model elements in UML class diagrams. For example, a 'UML Package' role stands for model elements of type UML package.

In addition, Figure 1 shows the constraints that can be associated with the roles and which constraints can be applied on which roles. Constraints are used to enforce certain properties in the constructed UML class diagrams. For instance, the 'Inheritance' constraint can be used to enforce a generalization-specialization relationship between two elements bound to UML class roles. A role can be associated with various kinds of constraints. However, a constraint makes sense only when applied to a proper role kind. Attaching a constraint to a role is optional and should be used only if we want to enforce certain model properties. For example, the 'Stereotype' constraint, which stands for

a stereotype value on a model element, can be attached to almost any role; whereas the 'Multiplicity' constraint, which specifies the value of a multiplicity, applies only to 'UML Association End' roles since it does not make sense in the case of other role types.

Example:

In order to elevate the comprehensiveness of pattern structures, several visual specification techniques, like [6, 22], have been suggested. Figure 2 shows our notation for visual pattern specification. The figure depicts a role diagram of the Command design pattern [9]. The nodes, marked with white color, depict the pattern roles. The Invoker role, for example, stands for any concrete element that may play the class role Invoker, the type of the role is specified on top of the role name. The edges in the upper part of the figure denote the dependencies between the roles. There are two kinds of dependencies: 1) the dependency from role execute to the role Command, which is marked with a diamond-ended line, represents the containment relationship between the elements that may play these two roles, 2) the dependency from role execute to role action, which is marked with a light-arrow-ended line, stands for a logical relationship. In this case, any element that plays the role execute should call the corresponding element that plays the role action. The cardinality symbol ('1' for exactly one, '?' for optional, '*' for zero or more, '+' for at least one) that comes along with the role name indicates the allowed number of concrete elements that may play that role. For instance, there should be at least one element that plays the ConcreteCommand role. If not otherwise indicated, the cardinality of the role is 1.

In order to show how the Command pattern can be used, the bottom part of the figure gives a concrete example binding (weaving). The concrete element Application, repre-

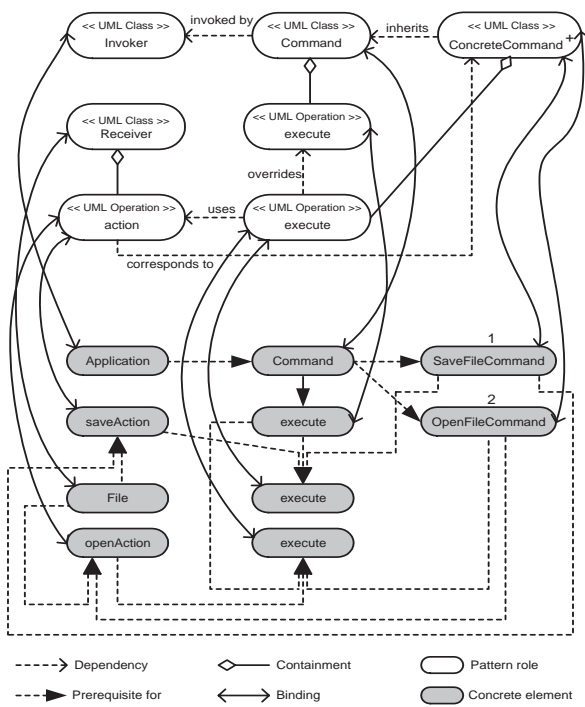


Figure 2. Role diagram of the Command pattern

sented by a dark-grey node, plays the role of Invoker, this is marked by the double-headed line between Application and Invoker. There are two elements that play the role ConcreteCommand, this is a direct implication of the '+' cardinality symbol associated with this role. As a next weaving step, the user might want to provide a third ConcreteCommand element, named NewFileCommand, for creating new files. In case several concrete elements play the same pattern role, the order of the binding is indicated by an integer index. Moreover, the dark-headed arrows in this part of the figure denote the order how the bindings should be performed. For instance, the binding between the concrete element Application and the role Invoker is a prerequisite for the binding of the concrete element Command to its role.

4 Applying Aspectual Patterns

4.1 Developing JUnit Design Model

Aspectual patterns can be used to superimpose models. Assuming that each pattern represents a specific model part, it is possible to apply the patterns one after another to form a larger model out of the parts. Usually, the final model is formed by accumulating the desired parts only. Because every pattern encapsulates a well-defined aspect, a desired

aspect can be weaved into the existing model by applying the pattern it encapsulates. The undesired aspects are left out simply by ignoring the patterns they represent. Aspectual patterns may have overlapping roles. In this situation, a pattern role may be bound to a concrete model element that has previously been bound to another role. The overlapping roles define how the individual aspects relate to each other.

As an example, let us consider the case of the JUnit [16] design model. JUnit is a popular open-source framework for implementing unit testing of Java programs. The design of the framework reflects three different concerns: creating tests, defining a generic test interface, and the ability to run multiple tests. Some of these concerns are defined in terms of smaller goals. These goals have been discussed in [16]. We refer to each of these concerns as a separate aspect. Therefore, in this work, the terms concern and aspect are used interchangeably.

The first concern, named 'Creating tests' is defined by three goals: representing a test case as an object, giving the tester a convenient place to put her fixture code and her test code, and reporting the test results. The first goal is achieved by applying the Command design pattern which encapsulates the test request as an object (test case) and uses the method execute (called run) to execute tests. For achieving the second goal, the Template Method design pattern is used. The pattern lets concrete tests redefine certain steps of the testing algorithm. The third goal uses the Collecting Parameter idiom to store the test results into an object that is passed to the run method as a parameter.

The second concern, called 'Generic test interface', consists of two smaller goals: making all the test cases look the same from the point of view of the invoker of the test; and avoiding the creation of a subclass (of the test case) for each testing method. For handling the first problem, the class version of the Adapter design pattern is used. The pattern adapts the testing method to the command interface. The second goal is realized through the Pluggable Selector idiom. This solution uses Java reflection API to invoke the testing method from a string representing the method's name.

The last concern, which we call 'Supporting test suites', is implemented using the Composite design pattern. The pattern treats single or multiple test cases uniformly. The run method is therefore used to execute either single test cases or collections of them.

Based on the discussion above, the design model of the JUnit can be described in terms of three aspects. Each of these aspects can be separately implemented as an aspectual pattern. In this way, every aspectual pattern corresponds to a separate feature in the design model of the framework. Generally, one feature may be composed of a set of smaller sub-features. However, we want to consider the larger features since often a sub-feature alone does not make much

sense. In the case of JUnit, the Collecting Parameter solution is better understood in the context of the bigger concern.

In the case of JUnit, the aspectual patterns that we have identified consisted of design patterns and idioms. In typical situations, however, aspectual patterns consist of any other kind of solutions. In other words, an aspectual pattern can consist of any arrangement of roles that is used to represent a given aspect in a system.

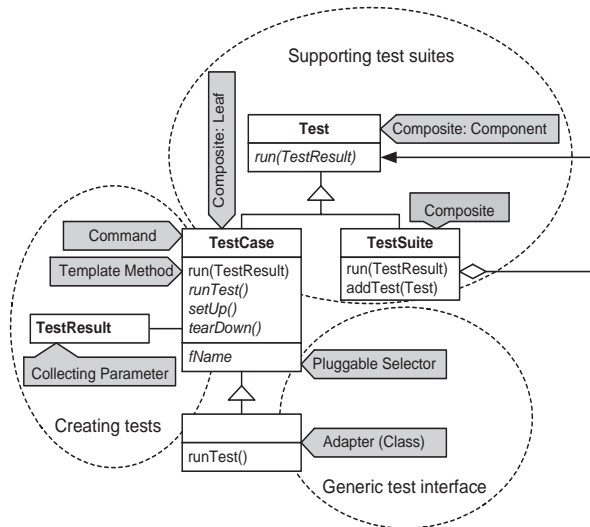


Figure 3. Aspectual patterns in JUnit

Figure 3 shows the overall architecture of the JUnit framework. According to the original documentation [16], the design is achieved by applying four design patterns and two idioms. From an AOP perspective, some of these solutions can be regrouped under the same concern. The two design patterns Command and Template Method, and the idiom Collecting Parameter, for instance, form a larger concern called 'Creating tests'. The overall JUnit design is achieved by starting a design from scratch. Aspectual patterns are then applied one after another, until the final architecture of the system is formed.

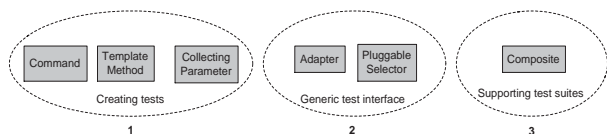


Figure 4. Steps in JUnit model design

Figure 4 shows a typical order of applying JUnit aspectual patterns. Firstly, the 'Creating tests' pattern is applied in order to create and structure the framework TestCase class. Pattern 'Generic test interface' is then applied to provide a generic interface for using the TestCase class.

Finally, by applying the 'Supporting test suites' aspectual pattern, support for test suites is added. Each time a pattern is applied, new model elements are weaved into the existing design.

Using aspectual patterns, it is possible to generate specialized views of a model. By constructing a specialized view of a model, we mean slicing the model into parts that correspond to the patterns applied. A model can be sliced in various ways. Ideally, each slice represents one or more aspects in the original design model. Therefore, a slice of a model can be regarded as a combination of the set of features defining that model. An aspect in the original model may or may not be included in the slice. In the case of JUnit, it is possible to highlight certain slices in the design that correspond to specific features. This helps in understanding the model.

As we have seen earlier, design models evolve as new aspects are added and other are dropped out. In the case of adaptive maintenance, models can be extended to adapt to new platforms or support new features. In many cases, such maintenance activities can be anticipated during the design phase. In such situations, aspectual patterns can be used to encapsulate the maintenance interface. Each pattern models a separate maintenance aspect. Considering the JUnit case, aspectual patterns may be used to describe how the design model of the framework can be extended. More specifically, there are pattern roles bound to the base model elements presented in Figure 3. These roles are used to define join points for other roles used to annotate the extension. The architects of the JUnit framework, for example, may document the extension points and the maintenance tasks required for supporting other types of testing as well.

4.2 Other Usage Scenarios

In this work, we have shown how aspectual patterns can be used to develop the design model of the JUnit testing framework. However, the approach tends to be more beneficial when applied to more complex case studies.

Slicing a model can be used to solve various kinds of problems. When modeling complex systems, design models can be too complex and may become difficult to understand. Model slicing can be used in this case to group related features into smaller submodels. In [11], we have shown how specialized views of models may enhance system comprehensibility.

Aspectual patterns can be applied to encapsulate the maintenance interface of design models. Each pattern is used to represent a separate maintenance aspect. In this regard, pattern roles are utilized to document the model extension points and the way models can evolve. In [12], we have shown how patterns can be used to document maintenance tasks. In this work, we show how the idea of patterns can

be applied early in the design phase in order to document model maintenance tasks.

Preserving the bindings between pattern roles and concrete elements represents an important advantage. This feature can be used to review which model elements have been weaved to the base model. In the case of model maintenance, for example, it is possible to control the way models have been updated. This can be used as a basis for supporting undo operations of maintenance actions.

5 Implementation

5.1 Tool Platform

MADE [13] is an experimental platform for pattern-driven UML modeling. The platform is the result of the integration of a number of different tools. JavaFrames [10] and Rational Rose [21] represent the key components of the integration. JavaFrames is a pattern-oriented task-based development tool built on top of the Eclipse [5] platform. Rational Rose is mainly used for designing and processing UML models. The communication between JavaFrames and Rational Rose is achieved through a UML model processing platform, xUMLi [20], providing a tool-independent API for accessing the UML models.

The MADE environment realizes aspectual patterns in the UML domain, as explained in Section 3. MADE supports the specification of patterns, and the interactive binding of the roles of a pattern to UML model elements residing in Rose. A key functionality of the environment is provided by JavaFrames which transforms a (possibly partially bound) pattern specification into a task list: every unbound role which can be bound in the present situation, taking into account the dependencies between the roles, becomes a task. Such a task can be performed in two ways by the designer: either she points out an existing model element to be bound to the role, or she asks the tool to generate one before binding it to the role. For the latter purpose, a role specification can be associated with a default element description, used in the generation. Typically, the default element descriptions refer to the elements bound to other roles in the pattern. For example, a class role in an aspectual pattern can have a default element description consisting of the name specification of the class element generated by default.

When a task is executed, other tasks become doable. The tool maintains the task list, and checks that the role constraints are satisfied by the elements bound to the roles. In the case of constraint violations, new corrective tasks are shown in the task list. In many cases the tool can provide an option to correct the model automatically. Since the UML modeling tool, Rose, is tightly integrated with the pattern

tool, free model editing actions which result in constraint violations are responded to by new corrective tasks, too.

In principle, any role-based pattern concept and its tool support could be used as a platform for aspectual patterns: a pattern concept is so generic that it can cover almost any kind of logical slice of a model, assuming that the role types and constraints are defined in an appropriate way. In this respect our pattern platform does not essentially differ from other pattern tools (like [7]). However, the task-driven interactive support for binding the roles, provided by our environment, brings the additional benefits for aspectual patterns, mentioned in the introduction. In particular, the weaving (i.e., binding) process becomes open: the designer performs simple tasks in a context she understands, rather than a large black-box operation. In addition, the weaving process can be easily customized: the designer can choose between different alternative tasks, leading to different design solutions.

5.2 Applying Aspectual Patterns in MADE

MADE can be used to specify the various properties of aspectual patterns discussed earlier. Figure 5 shows a textual representation of the 'Creating Tests' aspectual patterns discussed earlier. The pattern defines two UML class roles. The Command UML class role is an element of the Command design pattern and stands for the class which declares an interface for executing a request. The CollectingParameter role is part of the Collecting Parameter idiom. The purpose of the role is specified by the description property. The Command role has a UML operation role called execute. The execute role is associated with a parameter constraint that refers to the concrete name of the instance playing the CollectingParameter role. The primitiveOperation role, contained in the Command role, is a UML operation role that belongs to the Template Method design pattern. There can be multiple concrete elements playing this role. This is indicated by the '+' cardinality. The role is associated with an 'abstract' constraint stating that every concrete element playing this role must be abstract.

Figure 6 shows an overall view of MADE environment when applying aspectual patterns to develop the design model of JUnit. The top most part is the Rose tool. The bottom part is the JavaFrames tool. JavaFrames itself is composed of multiple integrated views. Aspectual Patterns are shown in the bottom left view. The 'Creating tests' pattern is fully bound whereas 'Generic test interface' is unbound and 'Supporting test suites' is half bound. In fact, the bottom right view displays a task defined by the latter pattern. The task is to provide a UML operation that adds a child to the composite component. Bindings are shown in the bottom middle part of the figure. This view reflects the tasks that have been already carried out.

Aspectual Pattern: Creating Tests	
Roles	Properties
Command : UML Class (1) execute : UML Operation (1) parameter : Constraint commandName : UML Attribute (1) type : Constraint	defaultClassName : AbstractCommand description : Declares the interface how to execute the operation. description : The execute method. value : <#:/CollectingParameter.i.shortName> description : Name of the command. value : String.
Template Method primitiveOperation : UML Operation (+) abstract : Constraint	description : A primitive method to be overridden by subclasses. value : true
Collecting Parameter CollectingParameter : UML Class (1)	description : An object passed to methods in order to collect information from those methods.

Figure 5. Textual representation of the 'Creating Tests' pattern

It is possible to highlight different features in the design by selecting the corresponding aspectual pattern. In Figure 6, the feature for creating tests is highlighted. The model elements corresponding to this feature, at the class level, are marked with a darker color. This can enhance the understandability of the model since it decomposes it into multiple views. The different relationships between features are also exposed. In the example, the relationship is expressed by the fact that the TestCase UML class inherits from the Test class, which belongs to another feature. It should be noted, however, that the TestCase class represents an overlapping model element since it treats two different features. On the one hand, it plays the role of Command in 'Creating Test' pattern. On the other hand, it stands for the leaf component in the Composite design pattern which implements the 'Supporting test suites' aspectual pattern.

6 Discussion and Related Work

The term aspectual patterns used in this paper is inspired by the work on aspectual components [19]. The constructs in both approaches are represented in terms of a graph of nodes. In the case of [19], a graph node, called a participant, is a class in the participant graph that should be bound to classes in other participant graphs or to a concrete class graph. In our methodology, the graph nodes represent the pattern roles. Roles may overlap with other roles and need to be bound to concrete elements. The key difference between the two approaches is that aspectual components operate at the programming level whereas aspectual patterns, in this work, are used for processing static design models.

The relationship between patterns and aspects have been identified in earlier works. An AspectJ implementation of

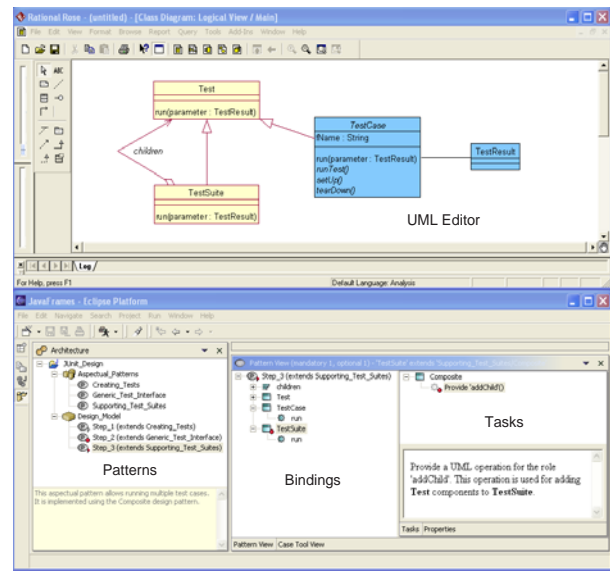


Figure 6. Composing models in MADE: the JUnit case

design patterns by Hannemann et al. [14] shows modularity improvements in 17 of the 23 GOF patterns. It is argued that the patterns with crosscutting nature between roles and the concrete elements they are bound to see the most improvement. Rather than implementing patterns as aspects, in this work we implement aspects using the role-based pattern concept at the design level.

A similar approach to aspect-oriented modeling is presented in [3]; the author presents a subject-oriented model design. Composition patterns, which are used to model crosscutting behavior, can be applied to supplement the behavior of base model operations with pattern behavior defined in the composition pattern. Compared to composition patterns, aspectual patterns augments the base model with new structural elements. However, we believe that MADE could provide tool support for the ideas of [3] provided that aspectual patterns cover behavioral information.

A tool for building and manipulating UML models with aspects, know as UMLAUT, is presented in [15]. The tool can be used to weave model elements into existing design. Every weaving step is a transformation step applied to a UML model. The weaving process is done by applying a set of transformation rules expressed in terms of combinations of predefined operators. Once applied, the effect of the transformation rules cannot be recalled after the transformation is done. In our methodology, the transformation rules are expressed in terms of pattern roles and constraints. The effect of the transformations rules is permanent and can be recalled after the transformation is applied. This is done through the persistent binding between the pattern role and

the concrete model element it is bound to.

Furthermore, in [17] an aspect-oriented view on software architecture has been defined. The work resembles the current work in the way which aspects are superimposed on top of each other, or alternatively on top of an existing base design, in a certain order. However, patterns enable capturing reusable designs at a higher level of abstraction than UML packages and can be customized for different domains. Similar architectural views can although be used for visualizing pattern orderings and the concerns they capture collectively, as was done in [13].

7 Conclusions

In this paper, we have presented our approach to aspect-oriented model development. The approach is based on capturing development steps in so called aspectual patterns which combine a generalized pattern concept with aspect orientation.

Aspectual patterns are aspect-oriented not only in the sense that they cut across several classes or components, but also in the way they are designed. More specifically, we aim at modularizing development steps in a way that the design decisions treating a particular concern are grouped into one aspectual pattern. This aspectual pattern can then be applied to the system under development which can be completely oblivious of those design decisions.

As discussed in Section 5, our approach is supported by tools that are currently under development. Other possible future work directions include integrating our ideas with existing work on stepwise development and maintenance support.

Acknowledgments

This work is supported financially by the Academy of Finland (project 51528) and by the National Technology Agency of Finland (project 40183/03).

References

- [1] AspectJ WWW site. Available at <http://eclipse.org/aspectj/>.
- [2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.
- [3] S. Clarke. Extending standard uml with model composition semantics. *Science of Computer Programming*, 44(1):71–100, July 2002.
- [4] Communications of the ACM. Special issue on Aspect-Oriented Programming, 44:10, October 2001.
- [5] Eclipse WWW site. Available at <http://www.eclipse.org>.
- [6] A. H. Eden. LePUS: a visual formalism for object-oriented architecture. In *Proceedings of IDPT'02*, pages 22–28, Pasadena, California, USA, June 2002.
- [7] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In *Proceedings of ECOOP '97*, volume 1241 of *LNCS*, pages 472–495, Jyväskylä, Finland, June 1997.
- [8] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading MA, 1997.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [10] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa. Generating application development environments for Java frameworks. In *Proceedings of GCSE'01*, pages 163–176, Erfurt, Germany, September 2001.
- [11] I. Hammouda, O. Guldogan, K. Koskimies, and T. Systä. Tool-supported customization of UML class diagrams for learning complex system models. To appear in *Proceedings of IWPC 2004*, June 2004. Bari, Italy.
- [12] I. Hammouda and M. Harsu. Documenting maintenance tasks using maintenance patterns. In *Proceedings of CSMR'04*, pages 37–47, Tampere, Finland, March 2004.
- [13] I. Hammouda, M. Pussinen, M. Katara, and T. Mikkonen. UML-based approach for documenting and specializing frameworks using patterns and concern architectures. In the 4th workshop of AOSD Modeling with UML, October 2003. San Francisco, California, USA.
- [14] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of OOPSLA '02*, pages 161–173, Seattle, Washington, USA, November 2002.
- [15] W. M. Ho, F. Pennaneach, and N. Plouzeau. UMLAUT: A framework for weaving UML-based aspect-oriented designs. In *Proceedings of TOOLS 33*, pages 324–334, Mont-Saint-Michel, France, June 2000. IEEE Computer Society.
- [16] JUnit WWW site. Available at <http://www.junit.org>.
- [17] M. Katara and S. Katz. Architectural views of aspects. In *Proceedings of AOSD'03*, pages 1–10, Boston, Massachusetts, USA, March 2003.
- [18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, June 1997.
- [19] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. Technical report, NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.
- [20] J. Peltonen and P. Selonen. An approach and a platform for building UML processing tools. To appear in *Proceedings of WoDiSEE 2004*, May 2004. Edinburgh, Scotland.
- [21] Rational Rose WWW site. Available at <http://www.rational.com/products/rose/index.jsp>.
- [22] D. Riehle. Composite design patterns. In *Proceedings of OOPSLA'97*, pages 218–228, Atlanta, Georgia, USA, October 1997.
- [23] The IEEE Standards Board. Recommended practice for architectural description of software-intensive systems. ANSI/IEEE-Std-1471, September 2000.
- [24] Unified Modeling Language WWW site. Available at <http://www.uml.org/>.