Imed Hammouda

**Multi-Dimensional Structuring of Software Systems**
Tools and Applications

Thesis for the degree of Doctor of Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 4th of November 2005, at 12 noon.
Software development is no longer about creating simple monolithic programs, but it has become a complex process involving many phases, stakeholders, tools, and artifacts. Consequently, software systems are viewed and managed along multiple dimensions covering different artifact types, languages, and stakeholders’ interests. Multi-dimensionality has thus become an inherent property of software systems. In addition to complexity, multi-dimensionality raises various development challenges mainly related to conformity, changeability, and invisibility. Therefore, one way of addressing such sources of difficulty is to provide solutions for managing the different dimensions. Despite the latest advancements in software research and technology, managing the multi-dimensional nature of software systems is still misunderstood and is in particular challenged by the lack of well-established formalisms and tool support.

We argue that software systems can be structured along two main decomposition criteria: the concern dimension and the artifact dimension. These two dimensions are themselves multi-dimensional. Software concerns can be understood in terms of certain concern types, stakeholders, and development needs whereas artifacts can be associated with artifact types, notations, and tools. These dimensions are not independent of each other, and decisions regarding any one will be influenced by the decisions taken for the other dimensions. One can define various relations between the different dimensions.

We discuss the proposed dimensions, show their impact on the overall software life cycle, and provide an approach for managing them. Moving towards this goal, we propose a tool concept, known as a heterogeneous aspectual pattern, to solve the problem of managing multi-dimensional software systems. The pattern-based approach is used to express and manage different concern categories in a software system. The concerns themselves can be represented across heterogeneous software artifacts. The idea is demonstrated by showing how a pattern mechanism can be used as a unified tool concept for various development activities. Yet, the approach is fundamentally tool, language, and notation independent.

The thesis is validated with the help of an industrial case study. The case study is split up into eight experiments, each of which is associated with a given artifact and concern dimension. The experiments are further classified into four software development activities, which are (framework) specialization, feature variability, software maintenance, and model comprehension. It is argued that the discussed concepts and tools will bring a number of benefits to the development of industry-level software systems.

The main contributions of this dissertation are two-fold. First, we generalize a pattern concept, which allows the management of heterogeneous software artifact types. Second, we generalize the application of a pattern-based tool concept covering different concern types and allowing the management of different software development activities.

**Keywords:** multi-dimensional structure of software systems, variability management, separation of concerns, pattern-oriented engineering, software development environments, software development activities, framework specialization, feature variability, software maintenance, model comprehension.
Preface

Research work for this thesis has been carried out during the years 2001-2005 in the research projects "JavaFrames", "Archimedes", and "INARI" at the Institute of Software Systems, Tampere University of Technology, Finland. This thesis was financially supported by the Academy of Finland, National Technology Agency of Finland, Nokia Foundation, and several industrial partners: Nokia, Plustech, TietoEnator, Almare, and Geracap.

First, I would like to express my sincere and deep gratitude to my supervisor Professor Kai Koskimies for his invaluable guidance, continuous support, and infinite tolerance during the course of this work.

I would like to thank Professor John Grundy from University of Auckland and Professor Lech Krzanik from University of Oulu for reviewing my thesis, and for their constructive feedback and comments on the manuscript.

Special thanks are due to Assistant Professor Pertti Kellomäki, my former M.Sc. advisor, to Professor Moncef Gabbouj, my second M.Sc. advisor, for his encouragement as well as for his close friendship, to Professor Ilkka Haikala for his kind advice, to Professor Tommi Mikkonen and Tarja Systä for creating an enthusiastic work atmosphere, to Professor Jaakko Astola, and to Dr. Pertti Koivistom, coordinator of TISE.

I am indebted to all my colleagues and friends at the Institute of Software Systems for the pleasant work environment and for the help I have received during my work. I owe special thanks to my co-authors: Anna Ruokonen, Jari Peltonen, Johannes Koskinen, Juha Hautamäki, Olcay Guldogan, Maarit Harsu, Markku Hakala, Mika Pussinen, Mika Katara, and Samuel Lahtinen for their cooperation, suggestions, and for the fruitful technical discussions. Special thanks are also due to Milja Järvinen, Aila Vallin, Outi Soiniaho, and Elina Orava.

Special thanks to all my friends and relatives in Finland, Tunisia, and Turkey for their support and care. I am very obliged to families Hammouda, Gabbouj, Maala, Hamila, Alaya Cheikh, Bouazizi, Ekim, Niiranen, Heikkilä, and Juutila. I also would like to thank Vladimir Guzma, Ali Hazmi, Monaeem Lakhzouri, Mejadi Trimeche, and Fehmi Chebil.

I wish to express my deepest gratitude to my beloved wife Yasmin, my sweet son Emin, my devoted parents Zohra and Tahar, my brother and sisters, my closest school friends Brahim, Khames, and Nejib for their endless support and encouragement all these years.

Most of all, I thank God for his mercy and protection. May God cure my son and bless him with good health.

IMED HAMMOUDA

Tampere, September 29, 2005
# Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>iii</td>
<td></td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
<td></td>
</tr>
<tr>
<td>List of Publications</td>
<td>ix</td>
<td></td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1.1 On the Difficulty of Software Development</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1.2 Multi-Dimensional Structure of Software Systems</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1.3 Organization of the Dissertation</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1.4 Author’s Contribution</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2 Basic Approach - Aspectual Patterns</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2.1 Structuring of Software Systems</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2.2 Aspectual Patterns</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>2.2.1 Definition</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>2.2.2 Example</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2.2.3 Characteristics</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>2.3 Exploiting Aspectual Patterns</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>2.3.1 Basic Tool Functionalities</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>2.3.2 Towards a Pattern-Oriented Prototype Environment</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2.3.3 Tool Requirements</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>3 Implementation - The MADE Tool Environment</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>3.2 User Interface</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>3.3 Presenting Aspectual Patterns in MADE</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>3.4 Tool Features</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>3.5 Using Aspectual Patterns in MADE</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>3.6 Documenting MADE Patterns</td>
<td>34</td>
<td></td>
</tr>
</tbody>
</table>
CONTENTS

4 Case Study - Nokia GUI Platform 37
   4.1 Introduction ........................................... 37
   4.2 Experiment ........................................... 38
   4.3 Specialization ............................................ 41
      4.3.1 Building New Platform Applications ................. 41
      4.3.2 Providing a New Service Implementation .......... 44
   4.4 Feature Variability Management .......................... 46
      4.4.1 Managing the Use of Platform Services .............. 46
      4.4.2 Varying the Settings of a Service .................. 49
   4.5 Maintenance ............................................. 51
      4.5.1 Modifying a Service Interface ....................... 51
      4.5.2 Maintaining Platform Service Implementations ....... 53
   4.6 Comprehension .......................................... 55
      4.6.1 Learning Product Configuration Facilities ........... 56
      4.6.2 Learning Platform Libraries ......................... 58
   4.7 Experiences and Evaluation ............................ 61
      4.7.1 Dealing with Essential Difficulty ................... 61
      4.7.2 Identifying and Building the Patterns ............... 63
      4.7.3 Using the Patterns ................................. 64

5 Related Work .............................................. 67
   5.1 Concern-Oriented Structuring of Software Systems ....... 67
   5.2 Managing Heterogeneous Software Artifacts ............... 69
   5.3 Other Related Development Approaches .................. 71
      5.3.1 Software Product-Lines ............................. 71
      5.3.2 Model-Driven Development ........................... 72
      5.3.3 Generative Programming ............................. 72

6 Introduction to the Included Publications .................. 75
   6.1 Summary of the Publications ............................ 75
   6.2 Author’s Contributions ................................ 77

7 Conclusions ................................................ 79
   7.1 Summary .................................................. 79
   7.2 Limitations ............................................. 80
   7.3 Future Work ............................................ 81

References .................................................. 83

Publications ................................................ 89
List of Figures

2.1 Structuring of software systems ........................................ 10
2.2 Conceptual model for software dimensions .......................... 12
2.3 Conceptual model for heterogeneous aspectual patterns .......... 14
2.4 Role diagram for pattern ’Security’ ..................................... 16
2.5 Pattern-oriented tool support ............................................. 20
3.1 The MADE environment architecture ................................. 24
3.2 JavaFrames components .................................................. 25
3.3 The MADE environment user interface ............................... 27
3.4 Simplified role diagram for pattern ’Security’ ...................... 28
3.5 A partial representation of pattern ’Security’ ....................... 30
3.6 Using patterns in MADE .................................................. 33
3.7 Usage scenario for pattern ’Security’ ................................. 36
4.1 Logical view of Nokia GUI platform .................................. 38
4.2 Role diagram for patterns ’Creation’, ’Controller’, and ’View’ . 42
4.3 Usage scenario for patterns ’Creation’, ’View’, and ’Controller’ . 45
4.4 Role diagram for pattern ’LoggingServiceImpl’ .................... 46
4.5 Usage scenario for pattern ’LoggingServiceImpl’ .................. 47
4.6 Feature model for the GUI platform services ...................... 48
4.7 Role diagram for pattern ’Services’ ................................. 48
4.8 Usage scenario for pattern ’Services’ ............................... 49
4.9 Feature model for online help service ........................................ 50
4.10 Role diagram for pattern 'OnlineHelp' ...................................... 51
4.11 Usage scenario for pattern 'OnlineHelp' ................................... 52
4.12 Role diagram for pattern 'AuthenticationInterface' .................... 53
4.13 Usage scenario for pattern 'AuthenticationInterface' ................. 54
4.14 Role diagram for pattern 'ServiceImpl' .................................... 55
4.15 Usage scenario for pattern 'ServiceImpl' .................................. 56
4.16 Role diagram for pattern 'DeploymentDescriptor' ....................... 57
4.17 Usage scenario for pattern 'DeploymentDescriptor' ..................... 59
4.18 Customized deployment descriptor file .................................... 59
4.19 Original model for platform application management .................. 60
4.20 Role diagram for pattern 'ApplicationManagement' .................... 60
4.21 Usage scenario for pattern 'ApplicationManagement' ................. 62
4.22 Customized model for platform application management ............. 62
List of Publications


The permissions of the copyright holders of the original publications to reprint them in this thesis are hereby acknowledged.
Chapter 1

Introduction

Of all the monsters that fill the nightmares of our folklore, none terrify more than were-wolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest. The familiar software project, at least as seen by the nontechnical manager, has something of this character; it is usually innocent and straightforward, but is capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a silver bullet—something to make software costs drop as rapidly as computer hardware costs do.

- Frederick P. Brooks, Jr.

1.1 ON THE DIFFICULTY OF SOFTWARE DEVELOPMENT

Software systems have rapidly spread throughout the main sectors of our modern societies ranging from medical services to telecommunications, and military applications. As computation tasks grew more complex, software development has evolved from creating simple monolithic programs to a more complex process involving many phases, stakeholders, tools, and artifacts.

The complexity issue of software development has been addressed in all popular software life-cycle models including the waterfall model [Roy87], the incremental model [Gil88], and the spiral model [Boe88]. All these development models include typical phases such as requirements specification, analysis, architectural design, detailed design, implementation, testing, and maintenance. These phases are usually addressed by different stakeholders. For example, clients, managers, and analysts are typically involved in the requirements specification phase. Software architects are concerned with architectural design and design engineers cope with detailed system designs closing the gap between abstract requirements and concrete implementation. GUI (Graphical User Interface) developers and system users are particularly interested in user interface issues. Testers have the role of identifying possible errors, whereas the mission of maintainers is to ensure evolution and adaptation of the system.
In order to support the needs of different stakeholders, various CASE (Computer Aided Software Engineering) tools and IDE’s (Integrated Development Environments) are utilized. The use of these tools ranges from modeling and design to code generation and testing. Some tools, such as testing tools, are specific to a particular development phase while others, such as documentation tools, address multiple phases and may cover various kinds of artifacts. Some tools are used early in the development process while others are used at a later phase. Often, different tools are used by different kinds of developers during the lifetime of a project. In smaller projects, however, the same developer may play multiple roles, such as being both a designer and a programmer. In this case, a single developer may interact with multiple development tools.

The primary concern of development tools is to cope with the difficulties of software engineering. Essential difficulties in software development have been associated with four main sources [Bro87]. The first source of essential difficulty is complexity: it is a function of the number of features and the number of relationships among them that must be considered simultaneously to decompose the problem being solved [GSCK04]. Here, a feature can be defined as a logical unit of behavior that is specified by a set of functional and quality requirements, from the perspective of one or several stakeholders of a software product [Bos00]. The next source is conformity: software has to conform to real-world constraints and limitations such as existing technology, architectural constraints, legacy structures, and legal issues. The third source is changeability: due to changing requirements and technology, software is expected to cover new functionality, domains, and platforms not originally intended for. The last source is invisibility: software cannot be fully understood by only studying its static structure or one kind of its various artifacts. In addition, representations of software are still complicated and not easy to grasp, despite latest advancements in development environments and language technologies.

The level of complexity gets higher as software systems involve more features and more variability in the way the features can be combined [Weg78]. Individual features, as they reflect the needs of different stakeholders, can sometimes be conflicting. Conflicts should be properly managed and variability among features should be conveniently expressed in the artifacts of the solution. As features are represented across multiple solution artifacts, it is essential to keep the different solution artifacts aligned and consistent with each other. Complexity can be considered as a measure for the other sources of difficulty. High levels of complexity often leads to communication problems among the stakeholders, thus causing conformity problems. Furthermore, the more complex a software system is, the more difficult it is to comprehend and to extend.

Considering conformity, software, for instance, has to implement strict business rules requested by certain stakeholders taking into account the needs of other stakeholders. A key challenge is to represent these business rules in all solution artifacts (when appropriate) used to describe the software system. Otherwise, the artifacts may become inconsistent. Another example of how complexity can influence conformity is the fact that application developers should conform to the architectural abstractions, among other design rules, defined by systems architects or designers. In some situations, however, programmers tend to break certain architectural abstractions due to limitations of the used development tools and programming languages, due to performance and other non-functional requirements [RSSX04], or simply because the software has to be quickly implemented and delivered to the client. In case architectural rules are broken, it might become hard for maintain-
ers to comprehend and maintain certain parts of the source code, unless the reasons are well-documented.

Changeability often requires the propagation of a particular change to all the affected software artifacts. An architecture-level change, such as the reconfiguration of components, may result in a detailed change in the components’ code, or changes in system documentation. In addition, maintaining a software system based on the request of a given client might break the needs of other clients in the sense that changes made to certain features may cause problems in other features. A widely used solution to cope with changeability is to create multiple products on top of the same platform to satisfy the needs of different clients. This however requires careful study of the dependencies, variabilities, and commonalities of product features to avoid new kinds of problems such as reimplementations of the same features in different platform products.

Invisibility, in turn, may become a serious problem as software systems get more complex. Typically, studying a software system through its static structure might involve a variety of textual and graphical notations such as informal documents, UML (Unified Modeling Language) [UML05] diagrams, and programming language constructs. As these notations are supported by different tools, it is often essential to learn how to use the tools themselves before browsing through the artifacts. In addition, the problem of invisibility might involve other issues such as the learner’s specific interests (i.e. the parts or the features of the software to be comprehended). Studying large complex software systems is often an incremental process where the overall system is comprehended through its parts. It is well-known that the comprehensibility of a system is mainly determined by the possibility of studying the system one part at a time [Par72].

The notion of difficulty of software development is not only a technical challenge that we can provide definite solutions for, or simply ignore whenever possible. Developing concepts and tools managing difficulty, in the sense discussed above, is now central to the research and development in the field of software engineering. In this dissertation, we argue that essential difficulty in software development stems mainly from the multi-dimensional structure of software systems. Thus, one way of addressing such sources of difficulty is to provide solutions for managing the different dimensions. We also argue that the issue of multi-dimensionality is not sufficiently understood and supported in software development environments.

1.2 MULTI-DIMENSIONAL STRUCTURE OF SOFTWARE SYSTEMS

Software systems are inherently multi-dimensional [Bro87]. A software system does not consist of source code only. Instead, it is composed of various artifact types expressed using different formalisms. The artifacts are derived as the software life-cycle progresses from the problem domain to the solution domain. Examples of artifact types include, but are not restricted to requirement specifications, design models, source files, XML (eXtensible Markup Language) [XML05] files, scripts, make files, etc. These artifacts can be expressed using different formalisms, but they all represent properties of the same software system, realized eventually in machine instructions. The languages and notations software artifacts are expressed in vary from graphical notations like UML to textual representations such as Java or XML. Generally, different kinds of artifacts are managed using different tools.
A clear benefit of this diversity is that each form may serve the need of different stakeholders in the software development cycle. Business managers, for example, are mostly comfortable with informal textual information and high-level visual diagrams. System designers, however, prefer to analyze systems using more detailed design diagrams whereas programmers express their software solutions in terms of source code. This diversity in the used formalisms has lead to the situation where a software system cannot be fully understood in terms of a single notation or a single artifact type.

Furthermore, the structure of a software system may be imposed by the interests and the needs of different stakeholders. The simplest and mostly practiced decomposition has been to structure a system into concerns reflecting the various functional requirements. However, a system may be structured, for example, to reflect quality attributes, architectural solutions (like design patterns), maintenance needs, features, reuse needs, product configurations, or just to support the comprehension of the system. Each kind of structuring defines a certain concern category. In its general meaning, a concern is any matter of interest in a software system. In the case of maintenance needs, for instance, we may speak of maintenance concerns.

These concern-based structures may overlap and span the whole system or some parts of it. Besides, a structure may span multiple phases in the development process or it may be addressed in one phase only. For example, structuring a system into features might affect several artifacts including requirements documents, design models, and source code, whereas other kinds of concerns such as maintenance concerns might involve implementation code only. In addition, concern categories may be considered at any phase in the development process, separately or in combination with other concern categories. In summary, we can identify two major dimensions used to structure a software system, the concern dimension and the artifact dimension.

Multi-dimensional structure of software systems has been the subject of various research areas in the field of software engineering. Recent research in aspect-oriented software development [FECA04] suggests that the main structural decomposition of a system should be complemented by software entities (aspects) cross-cutting the primary structure of the system. Typically, aspects correspond to specific concerns identified in the problem domain. Multi-dimensional separation of concerns [TOSH04] extends beyond the ideas of aspect-orientation by broadening the decomposition of a system to cover multiple dimensions. In practice, the idea is that a program (or other representation of a software system) can be decomposed using multiple modularization techniques such as by classes, by objects, by features, by aspects, by functions, etc. No modularization technique is predominant, but all decompositions should be treated equally.

In software evolution, synchronization and consistency of software models (e.g. architecture, design, and source code) during their evolution and re-engineering process has been recognized as a central issue [IK04, Rei02, TK04]. Model-driven development [MCF03, MDA05] promotes an approach where software models, corresponding to different artifact types, are usually derived from each other leading to a better alignment between the models. Model alignment has been considered as an important challenge in several development approaches such as the area of software product-lines [Bos00, CN01, JRvdL00], it has been recognized that variability should be addressed at different model levels [SB00].

Nevertheless, the multidimensional structure of software systems is still insufficiently understood and supported in software development environments. While software developers
need to cope with multiple artifact types, they often face the problem of missing or limited inter-tool communication. Some tools do not even provide a proper API for accessing the content of the artifacts. Poor inter-tool communication may lead to problems of misalignment, lack of traceability, and inconsistencies of models. This may result in problems with comprehensibility and maintainability. The situation is worsened if the development process itself does not take into account the multidimensional structure of a software system. Software development in many software companies still has the bad practice of being code-centric, thus causing problems of maintainability, low reuse, high impact of changes, and reduced concurrency in development. These problems, however, are generally encountered if an ad-hoc development process is used. Popular code-centric development processes such as XP (eXtreme Programming) [Bec99] avoid such problems by defining strict development rules and practices.

In addition, most tools provide support only for specific development tasks, thus being unable to support transition from one phase to another. For instance, source code is often developed based on well-defined architectural and detailed design models. Hence, it would be beneficial for programmers, for example, to use an environment where software development is guided by architectural and design models. Furthermore, several development-process activities can be valuable for other activities. For example, maintenance can be made easier if maintainers are provided with proper model comprehension tools and strategies. Often, related activities involve different notations, which may impede the development process. For example, model comprehension tools may use graph-based notations whereas tools involved in other related activities such as design and re-engineering may use different notations like UML. It would be more convenient if the same notation were used for both processes, which would increase understandability as the notation is familiar and the structures are compatible. Otherwise, conversion between the heterogeneous notations may be required.

Furthermore, as more and more software companies are basing their development strategies on using common platforms (i.e. software product-lines), tools are required to address different aspects of such development settings. This includes, for example, providing support for both platform-level and product-level activities, aligning product development to platform-level conventions, and facilitating the communication between platform developers and product developers.

Research and tool environments addressing the problem of managing multi-dimensional structure of software systems face the following key challenges.

- What kind of dimensions could be used in developing and analyzing a software system?
- How do the different dimensions manifest during the development process?
- How can we cope with the difficulty of software development through making these dimensions explicit?
- What kind of methodologies and tool concepts can be used to manage the dimensions?
- To what extent can we apply these methodologies in industrial-level settings?

In this dissertation, we discuss a tool-supported approach developed to model the multi-dimensional structure of software systems. We discuss the possible dimensions and show
their impact on the overall development process by considering a number of development scenarios in the context of an industrial case study.

1.3 ORGANIZATION OF THE DISSERTATION

In addition to the included publications, the dissertation consists of seven chapters. In Chapter 2, we introduce our approach to analyzing the multi-dimensional structure of software systems and we propose a set of tool concepts, functionality, and requirements for managing the multi-dimensional structure of software development. In Chapter 3, a concrete tool environment implementing the approach is presented. Chapter 4 presents a case study evaluating the methodology and the tool. Chapter 5 presents an overview of the different approaches and tool environments related to the topic. Chapter 6 gives an introduction to the included publications. Finally conclusions are drawn in Chapter 7.

The dissertation comprises eight publications. Paper [P1] introduces the fundamental concepts of an approach for managing multi-dimensionality. The approach is based on structuring a software system into separate concerns and representing these concerns as role-based patterns. It further presents an overview of the MADE tool, a pattern-oriented development environment. Papers [P2], [P3], and [P4] apply the approach to the problem of framework specialization. In Paper [P2], the approach is used to generate a pattern-based application development environment for Enterprise JavaBeans. Paper [P3] extends beyond the results of paper [P2] covering other parts of the J2EE framework. In both papers [P2] and [P3], an earlier version of the MADE environment is used. Paper [P4] discusses the task of framework specialization in UML considering the case of the Symbian platform. Paper [P5] argues that the approach can be used to document anticipated maintenance tasks. In Paper [P6], the approach is applied to support learning of complex system models through model customization. Paper [P7] considers the problem of feature-driven variability as an instance of multidimensional separation of concerns. Finally, paper [P8] discusses the problem of multi-dimensional structure of software systems in the context of model-driven development.

1.4 AUTHOR’S CONTRIBUTION

In summary, the main contributions of this dissertation are two-fold. First, we generalize a pattern concept, which allows the management of heterogeneous software artifact types. Second, we generalize the application of a pattern-based tool concept covering different concern types and allowing the management of different software development activities. A more detailed list of contributions include the following.

- A conceptual model for multi-dimensional structure of software systems. The author discusses an approach to decompose software systems into several dimensions. The model makes explicit the dimensions clarifying their relationships and giving a framework for tool development.

- The concept of a heterogeneous aspectual pattern for managing the various dimensions of software systems. Based on the proposed dimensions, the dissertation presents a
generic pattern-based approach for representing system concerns across various artifact types.

- A tool concept exploiting aspectual patterns. The author suggests a set of key operations and requirements for a tool concept that can be used to manage the various dimensions of software systems.

- A pattern-oriented task-driven development environment. The dissertation presents a concrete implementation of a tool environment addressing the problem of managing multi-dimensional software systems.

- Validation of the approach using an industrial case study. The author analyses the multi-dimensional nature of a real-life software system and demonstrates the applicability of the concept of aspectual pattern and the tool environment.

The author’s contribution is presented in Chapters 2, 3 and 4 in addition to the included publications. A detailed account of the author’s contribution to the individual publications is presented in Chapter 6.
Chapter 2

Basic Approach - Aspectual Patterns

In this chapter, we first present our approach to analyzing the multi-dimensional structure of software systems and show that the main sources of difficulty stem mainly from the tensions between the dimensions. Next, we present a pattern-based tool concept and a generic tool architecture for managing the issue of multi-dimensionality.

2.1 STRUCTURING OF SOFTWARE SYSTEMS

In software engineering, a model of a system is a description or specification of that system and its environment for a certain purpose [MDA05]. Models represent the basic building blocks of software systems. Depending on the degree of abstraction of the subject matter under study, models are expressed and utilized differently [MCF03]. This observation can be generalized and illustrated as a multi-dimensional matrix, shown in Figure 2.1. A model of a software system can be viewed as a collection of model fragments. Here, a model fragment can be any system model or portion of a model. A model fragment is associated with specific artifact types, is expressed using certain notations, and is managed by certain tools. At the elementary level, a model fragment (represented by a single cube in the upper right part of Figure 2.1) is associated with exactly one artifact type, notation, and tool. We refer to such fragments as elementary artifact fragments.

Furthermore, a model fragment (including the special case of an elementary artifact fragment) might consist of smaller model fragments, each of which comprises different concern types, addresses different stakeholders, and involves different development needs. To give an example, a detailed design model of a banking system, which is expressed in UML and managed by Rational Rose [Ros05], may consist of smaller model fragments of which some are used by senior developers to produce application source code while other fragments are used by novices for comprehending functional properties of the system.

As illustrated in Figure 2.1, we define two main dimensions for structuring a software system, considering all the artifacts relevant to that system. Firstly, a software system involves multiple artifact types ranging from requirements descriptions to actual implementation.
Secondly, a software system can be regarded as a combination of possibly overlapping concerns, which may be addressed during different development phases. In the following, we discuss these two dimensions in more detail.

**Artifact dimension** Software artifacts are documents describing software systems. The descriptions are often presented as a combination of drawings and text. The text may be in a formal language or in a natural language [MDA05]. The artifact dimension can, in turn, be analyzed with respect to three sub-dimensions: artifact type, notation, and tool.

- **Artifact type.** In addition to executable code, a software system consists of other kinds of artifacts residing at different abstraction levels. Software artifacts include requirement specification documents, architecture descriptions, detailed design models, deployment descriptors, test code, user manuals, etc. Often one artifact is considered as an input to the development of other artifacts. For example, architecture descriptions are based on requirements specification documents and source code is implied by detailed designs.

- **Notation.** Software artifacts are expressed using different kinds of representations, formalisms and notations. Some are graphical (e.g. UML) while others are textual (e.g. XML); some are formal (e.g. Java) while others are informal (natural languages). Notations have various abstraction, expressiveness, and extension capabilities.

- **Tool.** In order to support the software development process, software artifacts are typically managed using a variety of tools. Some tools are dedicated to software development (CASE-tools) while others are multi-purpose (e.g. Microsoft Word used
to specify system requirements and user manuals). Some tools are developed in-house while others are provided by external sources. Different tools comprise diverse capabilities such as support for project management, revision control, round-trip engineering, etc.

**Concern dimension** In software engineering, separation of concerns (SoC) [Dij76] is the process of decomposing a system into manageable parts that are relevant to a particular concept, goal, task, or purpose, and then integrating these parts to make up the whole system. As we have seen previously, a concern is any matter of interest in a software system. Similar to artifacts, concerns can be discussed in the context of three separate sub-dimensions: stakeholder, concern type, and development need.

- **Stakeholder.** Software development involves different kinds of stakeholders. External stakeholders are actors that are not directly involved in the development process. Their role could be, for example, to capture the business drivers and the requirements that a software system should achieve or to send feedback to system developers concerning a system prototype. Internal stakeholders are actors that are concerned with the daily development and evolution of the system. Examples of such stakeholders include system analysts, architects, developers, and testers. Different stakeholders have different kinds of skills including business, managerial, and technical skills, thus having different influence on the development process.

- **Concern type.** Stakeholders’ interests can be structured to reflect different properties of a system including features, quality attributes, functional requirements, architectural abstractions, design rules, coding conventions, and extension points. A more general classification of concerns has categorized them into architectural, non-architectural, reifiable, and non-reifiable [Kan03]. Architectural concerns are concerns that globally affect a whole system. Examples of such concerns include security issues. Non-architectural concerns, on the other hand, are those concerns affecting localizable parts of a software system. Examples of these include single responsibilities. Reifiable concerns have explicit representation in the software system. A feature, for example, can be addressed in the implementation using a number of class methods. Non-reifiable concerns, on the opposite, cannot be represented in the system artifacts. Examples of such kinds of concerns include usability attributes of a system.

- **Development need.** Software development process encompasses many different phases and tasks, each of which is related to a specific abstraction level and addresses the needs of specific stakeholders. Stakeholders’ needs depend mainly on what role(s) the stakeholders play in the development process. Examples of development needs include analyzing, developing, testing, comprehending, and maintaining software artifacts. Some of the needs such as maintenance are specific to a particular phase in the development life-cycle, whereas others such as artifact generation are relevant to multiple phases.

Figure 2.2 depicts a conceptual model for our approach to structuring software systems, highlighting the main structuring dimensions and their various relationships. A software
A system is typically described by several kinds of artifacts. Each artifact type can be specified using a number of notations and can be managed by a number of dedicated tools. The same notation can be used to describe different kinds of artifacts and the same tool can support multiple notations. Thus, one tool can be used to manage more than one artifact type. A system can furthermore be structured according to the concerns it addresses. A concern type can be addressed in several artifact types where an artifact type typically addresses more than one concern type. Besides, a concern type can be relevant to multiple systems and can be considered for multiple development needs. The same kind of concern can be considered for various development needs and can be important for different stakeholders in the development process. The same stakeholder can be associated with various kinds of concerns. Typically, stakeholders interact with the software system through the various tools where the artifacts of that system are managed. Stakeholders exhibit various kinds of development needs while the same need could be relevant to different stakeholders.

So far, we have discussed the decomposition of system models into smaller model fragments. At the elementary level, each model element appearing in an artifact can be categorized using six properties. Considering an arbitrary model element \( M \), the properties are:
- the type of the artifact containing \( M \);
- the notation in which \( M \) is expressed;
- the possible tools through which \( M \) is managed;
- the list of stakeholders addressed by \( M \);
- the set of concern types associated with \( M \);
- the development needs where \( M \) is involved.

Obviously, model elements forming a single model fragment may exhibit different property values. However, such elements should share at least one property value. For example, a detailed design model of a software system, which is expressed as a set of UML class
diagrams, may be managed using different CASE tools. Nevertheless, all the model elements of the diagrams refer to the same artifact type. This idea has been illustrated in Figure 2.1, where a model fragment may consist of smaller model fragments (and thus model elements), each of which is associated with a different concern dimension. According to the same figure, a model fragment groups together those model elements that share at least one property value. Otherwise, the model fragment would not have been considered as a logical model slice. Another interesting point is that two model elements having the same property values should logically be part of the same model fragment. Otherwise, the decomposition scheme becomes non-orthogonal in the sense that such elements would be shared by two separate elementary model fragments.

Based on Figures 2.1 and 2.2, we explain the reasons why we consider the multi-dimensional structure of software systems as a major cause for the difficulty of software development, as discussed in this dissertation. Complexity stems from the fact that a software development process has to consider the different dimensions of software systems, no dimension should be ignored. The problems of conformity, changeability, and invisibility are major challenges in software development since each of these sources of difficulty is at the heart of typical development needs in the development process. Conformity is often required during the generation of artifacts. Detailed designs and source code, for example, should conform to the architectural models. Changeability is an important aspect of maintainability, especially in a software system that is subject to frequent changes. It has been recognized that a truly successful software system is one that is continually modified and enhanced [SD00] offering a high level of reuse through the customization of its components. Invisibility is an important issue for comprehension, as it is often hard to comprehend those parts of a system that cannot be made visible and visualizable.

As mentioned earlier, conformity, changeability, and invisibility become even more challenging as system complexity is increased. For example, it is sometimes required to manage conformity at different model levels: design, source code, and deployment artifacts. Similarly, changeability in the case of complex systems needs to be handled at all appropriate system and abstraction levels. Invisibility, in turn, becomes challenging as software systems involve multiple artifact types and addresses a wide area of issues. We think that the essential sources of difficulty can be addressed through understanding and making explicit the various dimensions identified above. In the next section we present a generic pattern concept as a basis for representing the various dimensions of software systems.

### 2.2 ASPECTUAL PATTERNS

In order to manage the multi-dimensional structure of software systems, we use a structural entity called *aspectual pattern* [P1]. Aspectual patterns, which are based on the pattern concept introduced in [HHK+01a, HHK+01b], have little to do with the notion of patterns (such as design patterns [GHJV95] or analysis patterns [Fow97]) widely used in the software community. For simplicity, the term pattern will be used in this presentation to refer to aspectual patterns. For other types of patterns, explicit terms will be used.
2.2.1 Definition

An aspectual pattern is an organized collection of software elements capturing a concern that is relevant for some stakeholder of a software system. Patterns are used to represent concerns, which have explicit representation in the artifacts of a software system (reifiable concerns). Here, software element refers to problem or solution-level concepts expressed as model elements following well-defined metamodels.

Figure 2.3 depicts a conceptual model in UML for aspectual patterns. A pattern is a collection of hierarchically organized roles rather than concrete elements. A pattern is instantiated in a particular context by binding the roles to certain elements of concrete artifacts. Each role can be associated with a set of constraints expressing conditions that must be satisfied by the element(s) bound to a role.

Artifacts contain models. As discussed earlier, a model is any formal presentation describing some system properties, including, say, design models and source code. Models can be expressed in different textual or visual notations following well-defined metamodels (for example, a metamodel for UML or a metamodel for Java). Among other properties, a metamodel is assumed to define a containment relationship between the model elements. For example, a UML class may contain a UML operation. In any binding of roles to concrete elements, the containment relationships of the bound elements must respect the hierarchy of the roles.

The metamodels of the notations used in a model define properties for the model elements that can be checked by constraints. Constraints may refer to the elements bound to other roles, implying dependencies between the roles. For example, in a pattern that
covers UML class diagrams, a constraint of an association role may require that the association bound to this role must appear between the classes bound to certain class roles, thus implying a dependency from the association role to the two class roles.

A role is associated with a type, which determines the kind of model elements that can be bound to the role. A role type typically corresponds to a metaclass in the metamodel of a given notation. For example, there is a role of type UML class that corresponds to the UML class metaclass in the metamodel for UML. As indicated in the lower part of Figure 2.3, a pattern can be associated with multiple sets of role types (for example UML, Java, etc). Each set groups together related role types. For example, there is a set of role types for representing UML model elements. Each set of role types can be associated with specialized constraints applicable only for the roles in that set. For example, a Visibility constraint checks the visibility option of classes and their members in Java.

A cardinality (i.e. multiplicity) is defined for each role. The cardinality of a role gives the lower and upper limits for the number of elements playing the role in an instantiation of the pattern. For example, if a class role has cardinality [0..1], the class is optional in the pattern, because the lower limit is 0. The other possible cardinality values are [1..1] for exactly one concrete element, [0..n] for any number of concrete elements including zero, and [1..n] for at least one concrete element. The default cardinality is [1..1].

### 2.2.2 Example

Our example aspectual pattern has roles associated with a set of role types (and constraints) corresponding to a subset of metaclasses in the UML metamodel. We will first present this set of role types before presenting the actual pattern specification.

Table 2.1 depicts a set of role types that are used to represent model elements in UML class diagrams [P1]. For example, a 'UML Package' role type stands for model elements of type UML package. In addition, Table 2.1 shows which constraints can be associated with which role types. 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 type 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 type; whereas the 'Multiplicity' constraint, which specifies the value of a multiplicity, applies only to 'UML Association End' role types since it does not make sense in the case of other role types.

In order to specify aspectual patterns, we use a notation for visual pattern specification illustrated in Figure 2.4. We refer to such a figure as a pattern role diagram. The figure, in this case, defines a simple aspectual pattern for managing system security at the design level. Security is recognized as a critical concern, for example when developing business applications. The pattern structure is given in terms of role types representing UML elements, the concern here is modeling security in a system design model. A concrete security manager class ('ConcreteSecurityManager') provides an operation ('checkPermission') for checking the security policy of a custom permission ('ConcretePermission').
Table 2.1 Pattern roles and constraints for UML class diagrams

<table>
<thead>
<tr>
<th>Constraint Role Type</th>
<th>Stereotype</th>
<th>Abstract</th>
<th>Visibility</th>
<th>Inheritance</th>
<th>Multiplicity</th>
<th>Aggregation</th>
<th>Return Type</th>
<th>Parameter</th>
<th>Overriding</th>
<th>Type</th>
<th>Navigability</th>
<th>Participant</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML Package</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Class</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Operation</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Attribute</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Association</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Association End</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Realization</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Dependency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.4 Role diagram for pattern 'Security'
In Figure 2.4, the white nodes depict the pattern roles. The type of the role is specified on top of the role name. For example, there is a UML class role named 'ConcreteSecurityManager' that stands for any concrete element (in this case a UML class) implementing a custom security policy. Role 'SecurityManager' represents a UML class for the system abstract security manager class. Role 'ConcreteSecurityManager' is associated with an 'inheritance' constraint, referring to role 'SecurityManager', thus implying a dependency from 'ConcreteSecurityManager' to 'SecurityManager'. Dependencies are marked with a light-arrow-headed dashed line. In this case, any UML class bound to 'ConcreteSecurityManager' should inherit from the UML class bound to 'SecurityManager'.

The cardinality of roles is given after role names. If no cardinality value is specified, then the default cardinality (i.e [1..1]) is used. In pattern role diagrams, the following cardinality symbols are used: '?' for [0..1], '*' for [0..n], and '+' for [1..n]. For example, role 'SecurityManager' comes with no cardinality symbol meaning that there is exactly one UML class bound to 'SecurityManager'; whereas the cardinality symbol of 'ConcreteSecurityManager' is '+' meaning that there should be at least one UML class element bound to this role.

A containment relationship between two roles is marked by a diamond-headed line. For example, 'SecurityManager', similarly to 'ConcreteSecurityManager', defines a UML operation role named 'checkPermission'. The cardinality of both roles is [1..1]. An overriding constraint is defined between the two roles. This constraint states that any UML operation bound to 'checkPermission' in 'ConcreteSecurityManager' should override the UML operation bound to 'checkPermission' in 'SecurityManager'. Consequently, there is a dependency from the former role to the latter. Furthermore, 'checkPermission' in 'ConcreteSecurityManager' is associated with a parameter constraint stating that an operation bound to this role should take an instance of 'ConcretePermission' as a parameter. It is possible to define multiple concrete permission classes ('ConcretePermission').

Even though the cardinality of 'checkPermission' in 'ConcreteSecurityManager' is [1..1], this role might be bound to several concrete elements. The reason is that the dependency (implicitly implied by the containment relationship in this case) between 'checkPermission' and 'ConcreteSecurityManager' implies that there is a role instance of 'checkPermission' for every role instance of 'ConcreteSecurityManager'. Thus, there are as many concrete elements bound to 'checkPermission' as there are bound to 'ConcreteSecurityManager'. Dependencies indicate the order of role bindings, a role might be bound only if the roles it depends on have already been bound.

In order to show how the pattern can be applied, the bottom part of Figure 2.4 gives a possible concrete binding. The concrete element 'FilePermission', represented by a dark-grey node, plays role 'ConcretePermission'. This is marked by the double-arrowed line between the two nodes. There are two elements that play role 'ConcreteSecurityManager', 'WorkingDaysSecurityManager' and 'WeekEndsSecurityManager'. This is allowed by the '+' cardinality symbol associated with the role. 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 in which the bindings should be performed. For instance, the binding of role 'SecurityManager' to concrete element 'SecurityManager' is a prerequisite for the binding of 'ConcreteSecurityManager' to 'WorkingDaysSecurityManager' as there is a dependency from the latter role to the former.
2.2.3 Characteristics

The original motivation for the concept of a pattern, as used in this thesis, has been to model the solution part of software (design) patterns. Later, the concept has been extended to represent other kinds of concerns (aspects). The term "aspectual pattern" refers to this generalization. Compared to design patterns, an aspectual pattern can be regarded as a low-level mechanism that can be used as a development tool concept. With this difference in mind, aspectual patterns exhibit the following characteristics.

**System-specific.** Generally, aspectual patterns are applied in the context of given systems to solve specific problems but not as solutions to more general problems. For example, an aspectual pattern can be used to structure and specify the extension interface of a given platform, which would make the pattern not applicable to other systems.

**Notation and artifact neutral.** An aspectual pattern is a generic mechanism that can be used to model the representation of different kinds of concerns across different abstraction levels. Depending on the abstraction level, concern representations can be expressed using different notations and addressed in different artifact types.

**Heterogeneous.** The same pattern can have roles associated with heterogeneous role types. For example, a pattern could have roles covering both UML models and Java source code. In addition to model elements, pattern roles could be bound to other non-software artifacts such as informal documents.

From the viewpoint of aspects, binding of pattern roles corresponds to the weaving process: each role stores the information of a joint point. The constraints that are associated with a given role can be used to determine the context where the aspect may appear, and the constraints can be used to check whether the aspect, implemented by the pattern, is correctly weaved. There are a number of advantages in representing aspects as role-based patterns.

**Flexible weaving process.** In contrast to traditional weaving, the weaving of aspectual patterns is considered as an interactive, incremental process where the join points are located under the guidance of a pattern tool, rather than in a fully automated fashion. The weaving process is performed as simple tasks in the context of the developer, rather than as a large black-box operation. A task stands for binding an unbound element or enforcing a role constraint.

**Addressing several key challenges in aspect orientation.** First, aspect overlapping can be represented and implemented in a straightforward way using role-based pattern composition techniques where a model element can play different roles in different patterns. Second, aspectual patterns offer a symmetric model where there is no explicit distinction between "base-level" elements and "aspect" elements. Yet, using binding information it is still possible to distinguish between the two. Another desired property of aspectual patterns is the ability to support various phases of the development process as roles can be bound to software and non-software elements covering various kinds of system artifacts. Finally, aspectual patterns can be reused in multiple contexts. Roles are attached with parameterized
properties whose values are calculated and adapted to the context where the patterns are applied.

2.3 EXPLOITING ASPECTUAL PATTERNS

Aspectual patterns have potential characteristics that can be exploited to achieve a tool concept for managing the various sources of difficulty of software development, as presented earlier. Towards this aim, this section proposes a set of tool functionalities using aspectual patterns. The subsequent sections introduce a general architecture for pattern-oriented tool support and present a number of essential tool requirements.

2.3.1 Basic Tool Functionalities

Aspectual patterns are applied by binding roles to concrete elements. In the context of tool support, the binding information can be utilized in different ways offering a variety of key tool features. We define four kinds of basic tool functionalities for coping with the four sources of difficulty of software development, as described in this dissertation. For each tool functionality, we discuss the main source of difficulty that is addressed.

Synthesis. Binding a role to a concrete element can be carried out by first generating the concrete element and then binding the element to this role. The properties of the generated element are defined by the constraints associated with the role. Synthesis is a basic operation for artifact development that is mainly utilized to manage conformity and changeability. Based on the available role bindings and the rules expressed in the role specification, new system artifacts can be developed and existing artifacts can be maintained conforming to the current context.

Checking. Binding a role to a concrete element can be carried out by binding the role to an existing model element. When the binding is performed, the pattern specification can be used to check whether the bound elements conform to role properties. This operation can be used to typically manage changeability and conformity. When the bound artifacts are modified due to a maintenance action, broken bindings can be detected and constraint violations can be monitored. In the context of a concrete tool support, the pattern tool could suggest repair options. Furthermore, the pattern could be used to check whether all required maintenance tasks have been carried out correctly.

Visualization. Binding information can be used to highlight the model elements participating in the pattern, i.e. the concrete elements bound to pattern roles. This can be used to identify relevant parts of system artifacts. This functionality can be used, in particular, to cope with invisibility. Model elements corresponding to a certain concern are made explicit, which would allow to visualize the parts relevant for learning a certain aspect of the software system. In addition, visualization can be used to address complexity, for example by highlighting the representation of a system feature across multiple artifact types.
2.3.2 Towards a Pattern-Oriented Prototype Environment

Figure 2.5 depicts a pattern-oriented development environment; the goal of the tool environment is to exploit aspectual patterns, realizing the general tool operations defined above. A software system is modeled using multiple artifacts at different abstraction levels. Aspectual patterns are used to document model elements addressing specific system concerns. Artifacts are managed using their own tools. Generally, a concern is addressed in multiple artifact documents (for example multiple Java files). Furthermore, a single concern might be represented in artifacts across different levels of abstraction (e.g., source code and deployment documents). Patterns, therefore, might have roles supporting heterogeneous sets of role types (e.g., Java and XML).

A single tool is used to manage the patterns, communicating with artifact-specific tools through their APIs. Using the same pattern, the pattern tool might communicate with different artifact-specific tools, thus managing different levels of abstraction. Aspectual patterns can be used by the pattern tool in different scenarios. They can be used to generate software artifacts in their specific tools (synthesis), to check the content of those artifacts against the pattern specification (checking), to highlight the pattern instances in the models (visualiza-

Figure 2.5 Pattern-oriented tool support

**Tracing.** Binding information can be used to trace between model elements (bound to pattern roles) across the same or multiple artifacts. Tracing is an operation that is usually relevant to complexity, invisibility, and conformity. It is often hard to identify which solution features correspond to which problem features and whether changes made in one end are consistent with changes made at the other end [GSCK04]. In practice, this means the possibility to navigate through multiple artifact types (e.g., tracing back source code to design model), to turn implicit information in models into explicit information and knowledge, and maintain the consistency across different artifact types.
tion), and to navigate through the different artifacts based on the pattern binding information (tracing).

The pattern mechanism presented in Figure 2.5 represents a unified tool concept that is language and notation independent. It is a conservative approach that does not make assumptions about used languages or supported artifact types. The only thing assumed here is that the tools used to process the relevant system artifacts offer an API, which allows the pattern tool to access the elements of those artifacts and to catch certain events (e.g., when an artifact has been modified). It is possible to support new notations (and artifact types). This is achieved by constructing new pattern role types representing the newly introduced notations and implementing proper mechanisms for communicating with the tool where the new artifact type is managed. It is also possible to manage new artifact properties by constructing new kinds of constraints for the role types.

2.3.3 Tool Requirements

In addition to the tool properties discussed in the previous section, the following major tool requirements can be identified. Throughout the discussion, we give the abstract tool operations each requirement addresses.

1. **Stepwise development process** (for synthesis): Generally, stepwise development offers better control and understanding of the development process and of the developed system artifacts. By considering role bindings as development tasks, artifact synthesis could be decomposed into a number of smaller, simple, and incremental steps, where each each role binding represents a step.

2. **Model monitoring** (for checking): In addition to artifact generation, the tool is assumed to monitor the artifacts for undesired model changes, such as broken architectural rules. The broken rules could be identified when role bindings or role properties become invalidated by changes in the model.

3. **Explicit representation of concerns** (for visualization and tracing): The tool is required to explicitly identify and relate together different matters of interests in system models. When an aspectual pattern is applied to a model, the pattern instance could be viewed as a gluing mechanism that defines implicit links between the model elements bound to that pattern.

4. **Adaptive development** (for synthesis): Development effort can be reduced by using developer’s specific context, for example when describing development tasks. Since aspectual patterns are system-specific, development steps corresponding to role bindings could be made more explicit by using the actual system context captured by other role bindings.

5. **Openness to new rules** (for synthesis): Using the pattern tool, it should be possible during the software development process to incorporate new business rules and design issues. The new rules could be modeled as newly introduced aspectual patterns or changes made to existing pattern specifications.

6. **Tool independence** (for all tool operations): The tool architecture presented in Figure 2.5 suggests that it should be possible to integrate new CASE-tools and IDEs to the
pattern tool for managing new artifact types as well as achieving inter-tool communication and cross-artifact navigation. This can reduce the complexity of using several separate tools.

7. Extensibility (for all tool operations): A key requirement is the ability to support new artifact types. Since aspectual patterns are language independent, it should be possible to support new formalisms and notations.

We argue that aspectual patterns provide support for most of the tool requirements. In the tool implementation chapter, we discuss the extent of such support.
This chapter introduces a concrete tool environment for heterogeneous aspectual patterns built according to the architecture presented in Figure 2.5. The first section discusses the main components of the tool. The subsequent sections show how aspectual patterns are specified and utilized in the tool, outlining the main features of the environment. The discussion uses the example pattern 'Security' introduced in the previous chapter.

3.1 INTRODUCTION

In order to demonstrate the tool concepts and operations discussed in Chapter 2, a prototype tool environment called MADE (Modeling and Architecting Development Environment) [P4] has been developed. The MADE platform is an extension of an existing pattern-oriented development environment, called JavaFrames [HHK+01b]. In fact, the MADE toolset is based on a number of different commercial, non-commercial, and research tools: JavaFrames, xUMLi [AKK+02, PS04], Eclipse [Ecl05], and Rational Rose [Ros05]. JavaFrames, which is a research tool, is a pattern-oriented task-based development environment for Java built on top of Eclipse, a free development environment. Rational Rose is a commercial tool that is used as the UML editor. The third component, xUMLi, is a CASE-tool-independent research platform for processing UML models. It is, however, assumed to be integrated with a tool supporting UML modeling (for real and practical use). It is used for integrating JavaFrames and Rational Rose. The remainder of this section discusses these components in detail.

The layered architecture of MADE is depicted in Figure 3.1. JavaFrames, which is implemented as a set of Eclipse plugins, represents the core component of the toolset. The MADE extension to JavaFrames includes support for file, UML, and XML role types. Figure 3.2, in turn, shows the main components of JavaFrames.

1The MADE toolset can be downloaded from http://practise.cs.tut.fi/~mda/.
Pattern engine. The pattern engine is the core of the system used to manage the binding process based on the pattern specification. The pattern engine transforms a partially bound pattern into a **task list**. This is done by generating a task for each unbound role that can be bound in the current situation, taking the dependencies and cardinalities of roles into account. Tasks can be performed in two modes. A role can be bound to an existing element, or a new element is first generated according to a role specification and then the element is bound to the role. The pattern engine updates the task list after a task has been performed, usually creating new tasks. When updating the task list, the pattern engine also checks that the constraints of the roles are satisfied, and generates corrective tasks if this is not the case.

Semantics. In JavaFrames terminology, sets of role types are referred to as semantics. This is the part of the system that connects the supported role types to the pattern engine and where new role types are introduced. Originally, JavaFrames has been developed as a pattern-driven specialization environment for Java-based frameworks, implementing the pattern concept for Java role types [HHK+01b]. In order to support other artifact types, we have added new role types. For modeling purposes, a set of UML-specific role types, representing UML class diagrams, has been developed [P4]. A third set of role types currently exists for supporting XML semantics. Besides, the pattern tool has been extended with simple role types for supporting arbitrary text files. In addition, there is a number of other simple role types representing informal tasks and user input values. Roles of such types are not bound to explicit concrete elements. The former is used to remind users of certain development issues whereas the latter is used to get string values from users.

Utility Functions. JavaFrames comes with a number of tools for specifying and applying patterns. The pattern view is used to construct and modify patterns while the architecture view is used to organize the existing patterns based on their relationships. The task view is utilized for displaying new and recorded tasks. The binding process is performed using task wizards. The synchronization tool is used to check the concrete model against the pattern specification, i.e. monitoring for possible broken bindings or violated constraints. Finally, the visualization and tracing tools are used to show and analyze the effect of patterns in the concrete models.
The JavaFrames pattern mechanism is independent of the tools where the individual artifact types are managed. In order to access the concrete models, several default editors have been integrated in the MADE environment. It is possible, however, to switch to new CASE tools as long as these tools provide well-defined APIs for accessing the content of the artifacts. The following default editors are used.

- **Rational Rose**: For UML support, Rational Rose has been selected. The reason is that the tool provides a proper API to its contents and that it is used by our industrial partners. In order to make it easier to change to another UML tool, without rewriting the model processing code, a tool-independent UML processing platform xUMLi (executable UML interface) has been used.

- **Eclipse Java editor**: Eclipse offers an integrated editor for managing Java source code. The editor comes with a variety of useful features such as code formatting, built-in refactoring, development history, documentation assistance, and context-sensitive development. In order to be able to access the elements of Java models independent of the used editor, JavaFrames keeps an own representation of Java source code (i.e. abstract syntax tree model). For accessing the content of the Eclipse Java editor, the environment defines a mapping between the MADE Java metamodel and the Eclipse Java metamodel.

- **MADE XML tool**: For managing XML artifacts, an experimental Eclipse-based XML editor has been built. The MADE tool uses DOM [DOM05] to represent the content of XML documents and XPath [XPa05] to access the XML model elements. This makes it possible to switch to a new XML editor.

- **Eclipse text file editor**: In addition to managing Java source code, Eclipse built-in text editor is used in the MADE environment for accessing text files. Similarly, the MADE tool uses a tool-independent metamodel for accessing the file system.
3.2 USER INTERFACE

Figure 3.3 depicts an overview of the MADE tool environment. The bottom part shows the JavaFrames component, which is implemented as an Eclipse perspective consisting of two main views: the architecture view (left part) and the pattern view (right part). The architecture view represents the part of the environment where aspectual patterns are specified, organized, and applied. Patterns are represented by circular graphical icons. In this view, the two nodes 'Instances' and 'Patterns' represent architectural nodes that are mainly used to organize the patterns into hierarchical structures defining the context where these patterns are defined and applied.

To give an example of pattern instantiation, pattern 'Security' is used. This pattern is defined under the node 'Patterns'. An instance of this pattern, named 'FileAccess', is shown under the node 'Instances'. The pattern instance is for developing a security policy for file system access. In the terminology of JavaFrames, pattern 'FileAccess' extends pattern 'Security'. When a pattern is instantiated, the MADE environment transforms the pattern into a task list.

The pattern view displays the current status of pattern role bindings. Considering the specification of pattern 'Security', the left pane shows, for example, the binding of role 'SecurityManager' to the model element 'SecurityManager' and role 'ConcreteSecurityManager' to the model elements 'WeekEndsSecurityManager' and 'WorkingDaysSecurityManager' (Role names are not shown in this view, role information is displayed when a given binding is selected). In this way, the environment records the information how patterns are applied to concrete models. The remaining possible role bindings are displayed in the right part of the pattern view as tasks.

Depending on the current context, tasks can be optional or mandatory. Optional tasks are marked with a white (hollow) dot while mandatory tasks are marked with a red (filled) dot. For example, the pattern view in the figure displays an optional task for binding role 'ConcreteSecurityManager' to a UML class. The task is optional since the cardinality of the role is [1..n] and the role has already been bound twice.

The upper right part of the pattern view presents the task title whereas the bottom right part gives a detailed task description. In the MADE environment, tasks can be performed in two different modes. The developer might bind the pattern role to an existing model element. In this case, a binding between a pattern role and an existing model element is established but the model does not change. The second mode is to generate a new model element and bind the role to this generated element. The generated element is added to the proper editor depending on the type of the element, implied by the role type. If the model element is a UML class, for example, the element is shown in the UML editor.

Pattern 'Security' has roles representing UML model elements. Thus, the model elements bound to the pattern role are managed using a UML editor. In MADE environment, Rational Rose is used as the UML editor. This is shown in the upper part of Figure 3.3. For instance, the UML class 'WeekEndsSecurityManager' displayed in the Rational Rose view is bound to role 'ConcreteSecurityManager' in pattern 'Security'. The communication between JavaFrames and the used editors does not restrict the models from being freely edited through their dedicated tools. For example, it is possible to add new UML elements to the UML model (using Rational Rose interface) or modify the existing elements. If a model is modified, the worst that can happen is that some bindings in an existing pattern become in-
A UML model being constructed according to a pattern

Rational Rose

Editors for Java, XML and Text files

A view to an instance of a pattern (bindings)

An Eclipse-based GUI

Java source code

Pattern description

The used pattern library

Instructions for performing a binding task

A task list to carry out the bindings at this point

**Figure 3.3** The MADE environment user interface
valid (for example when deleting ‘WorkingDaysSecurityManager’ from the UML model) or certain constraints defined by the pattern are violated (for example when the inheritance relationship between ‘WeekEndsSecurityManager’ and ‘SecurityManager’ is removed). In this case, the MADE tool warns the developer about the inconsistencies and proposes corrective actions. The warnings are displayed to the user as new tasks. It is then up to the developer to either correct the situation or ignore the warning.

The next chapter, which presents the case study, will illustrate the above tool properties using concrete scenarios. In addition to UML, pattern roles can be used to represent other notations such as Java or XML. In this case, the concrete model elements bound to the roles should be managed in their proper editors. The usage of other editors will be illustrated in the case study. Another important issue, which has not been discussed yet, is how aspectual patterns are specified in the MADE environment. This will be addressed in the next section.

3.3 PRESENTING ASPECTUAL PATTERNS IN MADE

For illustration purposes, this section uses pattern ‘Security’ presented earlier. Figure 3.4 depicts a simplified version of the pattern. Since the focus is on the way patterns are specified, the instantiation part in the original pattern role diagram has been omitted. Furthermore, three roles have been prebound. Prebound roles, which are marked with darker color, represent existing elements in the system model to which the pattern is to be applied. A textual representation of the pattern specification is illustrated in Table 3.1. The table shows a number of new role properties in addition to the information shown in Figure 3.4. The purpose of these newly added properties is to guide developers through the pattern instantiation process. For brevity, we have omitted some less essential properties in the table.

The definition of role ‘ConcreteSecurityManager’ includes a ‘defaultElementName’, which is used as a default name when generating the UML class to be bound to the role. The ‘defaultElementName’ property is related to the current MADE approach to model generation, which is discussed in Section 3.4. The value of this property is expressed as a template referring to the name of the concrete element bound to role ‘SecurityManager’. The property ‘taskTitle’ is used by the MADE environment for generating a title for the corresponding task. The property ‘description’ is used to present a detailed description for the task. Concrete usage of the latter two properties has previously been illustrated in Figure
Table 3.1 Textual representation of pattern 'Security'

<table>
<thead>
<tr>
<th>Roles</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecurityManager: UML Class, prebound</td>
<td>description: Class encapsulating security managers.</td>
</tr>
<tr>
<td>checkPermission: UML Operation, prebound</td>
<td>description: Operation representing a security policy.</td>
</tr>
<tr>
<td>ConcreteSecurityManager: UML Class (+)</td>
<td>description: Concrete security manager implementation.</td>
</tr>
<tr>
<td></td>
<td>defaultElementName: My-#:/SecurityManager.i.shortName&gt;</td>
</tr>
<tr>
<td></td>
<td>taskTitle: Provide an UML class for the role 'ConcreteSecurityManager'.</td>
</tr>
<tr>
<td>inheritance: Constraint dependency</td>
<td>value: /SecurityManager.i</td>
</tr>
<tr>
<td>checkPermission: UML Operation (1)</td>
<td>description: Operation encapsulating a concrete security policy.</td>
</tr>
<tr>
<td>parameter: Constraint dependency</td>
<td>value: /ConcretePermission.i.shortName&gt;</td>
</tr>
<tr>
<td>overriding: Constraint dependency</td>
<td>value: /SecurityManager/checkPermission.i</td>
</tr>
<tr>
<td>Permission: UML Class, prebound</td>
<td>description: Class encapsulating the kinds of permissions.</td>
</tr>
<tr>
<td>ConcretePermission: UML Class (+)</td>
<td>description: Concrete permission kind.</td>
</tr>
<tr>
<td></td>
<td>defaultElementName: My-#:/Permission.i.shortName&gt;</td>
</tr>
</tbody>
</table>

3.3. The property 'defaultElementName' is typically used in the 'generate element wizard' illustrated in Section 3.5.

The 'inheritance' constraint is used to enforce the generalization/specialization relationship between concrete elements bound to role 'ConcreteSecurityManager' and the concrete element bound to role 'SecurityManager'. In this case, the specification says that any UML class bound to role 'ConcreteSecurityManager' should inherit from the UML class bound to role 'SecurityManager'. The cardinality of roles is given between parentheses after role names. The dependency associated with role 'ConcreteSecurityManager' has value '/SecurityManager' meaning that there is a dependency from role 'ConcreteSecurityManager' to role 'SecurityManager', this dependency is implied by constraint 'inheritance'.

In the MADE environment, patterns are created using a dedicated editor. Figure 3.5 shows a representation of a part of the pattern 'Security' in the MADE pattern editor. The left view displays the pattern roles and their associated constraints. In this case, the properties of role 'ConcreteSecurityManager' are displayed. The right upper view shows the (outgoing) dependencies of the selected role whereas the lower part exposes the properties of the role expressed as text templates. As mentioned earlier, these properties are typically used by the MADE tool to guide the binding process.

3.4 TOOLS FEATURES

For exploiting aspectual patterns, the MADE environment provides a number of mechanisms and features that allow users to cope with the major sources of difficulty of software develop-
The various tool characteristics will be illustrated in the next section and demonstrated in the case study (Chapter 4). In this section, we enumerate the major tool characteristics, relate them to the general tool requirements presented in the previous chapter (section 2.3.3), and give the subsections where they are illustrated in the case study. In addition, the discussion gives the corresponding references to the included publications where these tool characteristics have been exploited. Even though we discuss the tool features in the context of the MADE environment, many of these features are already available in JavaFrames. The main tool characteristics are as follows.

**Stepwise artifact development** (requirements 1 and 4). The MADE tool transforms a pattern specification into a task list guiding developers step by step through artifact development. In this context, a task stands for a simple action that adds a model element or enforces a property on system models. In the MADE environment, the task-based approach can be made flexible by providing users with alternative tasks to choose from. Depending on the choice, artifacts can evolve differently. In practice, this means the development of alternative models. Stepwise development is a fundamental concept of the tool that will be used throughout all the phases of the case study presented in the next chapter. This feature has been used to produce a stepwise J2EE application development environment [P3], a stepwise model development environment in UML for Symbian applications [P4], and a stepwise maintenance tool [P5].

**Model generation** (requirement 1). In the MADE environment, the outcome of performing a task can be a new element added to the model to which the pattern is applied. For every pattern role, we can define a default element that is specified by the role. A default element exhibits properties conforming to the constraints set for the corresponding role. As the models themselves can be at different levels of abstraction, they are expressed using different notations. Currently, the tool provides support for generating models expressed in UML, Java, and XML. Model generation will be illustrated in the different scenarios of the case study. In addition, this core feature has been used in the included publications to generate Java code [P3], XML data [P7], and UML models [P4].

**Automatic detection and repair of broken model conventions** (requirement 2). If a model is manually edited, the environment provides immediate validation of the model against the
pattern specification. In the case of a constraint violation, a repair task is created to inform the user about the violation. In most cases, the tool provides the option to automatically repair the violation, for example restoring a generalization/specialization relationship between two classes. A second problem that might be caused by manual model changes is the accidental deletion of model elements. The MADE tool is capable of detecting broken bindings between pattern roles and the deleted model elements. Similarly, this is a core feature of the tool that is typically used in all tool usage scenarios. This feature has been used to ensure that models across multiple abstraction levels are kept aligned [P8].

**Monitoring for model changes** (requirement 2). The MADE environment can be used to inform the developer about any kind of change occurring in a model fragment. This is particularly important if the change can break certain assumptions in the model. The MADE environment provides a special role constraint type called ‘Watcher’, which can be used to monitor for changes to the concrete element bound to a role. Currently, it is possible to monitor Java elements. For example, this feature can be used to manage the fragile base class maintenance problem [P5]. A similar situation where this feature is utilized is presented in Section 4.5.1.

**Concern representation** (requirement 3). The MADE environment can be used to represent different kinds of concerns. Concerns, which might crosscut different abstraction levels, are modeled as aspectual patterns. The MADE tool exploits the recorded role bindings to link together different representations of a concern at different abstraction levels. This is important, for example, to trace features in system models [P7]. Binding information can also be used to highlight the effect of a concern in system models, which might be used to promote model comprehension [P6], for instance. Aspectual patterns have been used to represent different kinds of concerns such as features [P7], maintenance concerns [P5], comprehension concerns [P6], design rules [P2] or extension points [P2, P4]. Every experiment in the case study represents a different kind of concern. To give an example of concern operations, tracing is illustrated in Section 4.4.1 and highlighting is used in Section 4.6.2.

**Pattern composition** (requirement 3). System concerns can be treated as compositions of smaller sub-concerns. The MADE environment provides tool mechanisms for composing aspectual patterns together in order to model the larger concerns. Pattern composition has been addressed in [P8]. Regarding the case study presented in the next chapter, this feature is used in Section 4.3.1.

**Adapting to developer’s context** (requirement 4). The MADE environment is able to record the performed bindings, which are carried out as tasks, and exploit the recorded information when documenting the next tasks. The tool, for example, uses the names of concrete elements in the textual descriptions of related tasks. Similarly to features 1 and 2, adaptation is a core quality of the MADE environment. For example, the development environments presented in [P3] and [P4] use this tool feature extensively.

**Openness to new design conventions** (requirement 5). The MADE environment is not bound to a fixed set of patterns. As new concerns are identified, new patterns can be easily introduced to the existing architecture. In addition, as the application domain evolves, existing
pattern specification can be updated and outdated patterns can be removed without affecting the actual system. It is also possible to change the specification of patterns if the roles are not yet bound. For example, this feature has been used in [P3] to extend the specification presented in [P2]. In this case, the extension stands for augmenting the EJB pattern system [P2] with new patterns for specifying the presentation layer of J2EE applications [P3].

In addition to the above features, the design of the MADE platform has been built taking into consideration two important extensibility issues: the possibility to support new artifact notations (requirement 7) and the ability to incorporate new artifact tools (requirement 6). The former issue has been discussed in [P5]. While the MADE tool does not currently provide support for this kind of extensibility, it is possible to use the MADE tool itself to generate the required code. This can be specified as aspectual patterns. A simple experiment in this regard has been carried out in [Hau05]. Considering the second extensibility issue, the MADE environment uses tool-independent metamodels for accessing the model elements. This allows easier integration of new artifact tools.

3.5 USING ASPECTUAL PATTERNS IN MADE

An example deployment scenario for pattern 'Security' is presented in Figure 3.6. The figure should be read from top to bottom and from left to right. The original design model specifies three kinds of access permissions, 'DBPermission' (for database access), 'WebResourcesPermission' (for access to Web resources), and 'FilePermission' (for file access). In this case, the user decides to use the pattern to define concrete security strategies for file access permissions. First, pattern 'Security' is instantiated (dialog A). As a result, a task list is displayed. The user then decides to provide a concrete element for role 'ConcretePermission' (dialog B), and chooses to locate an existing model element (dialog C). As a result, a binding is performed between role 'ConcretePermission' and the selected UML class 'FilePermission'. As a next step, the user adds a new concrete security manager class for 'FilePermission' (Dialog D). Consequently, a second binding between role 'ConcreteSecurityStrategy' and element 'WeekEndsSecurityManager' is added. The newly generated element is added to the UML design model. In a similar fashion, we assume that the user adds another concrete security manager class 'WorkingDaysSecurityManager'.

The visualization features of the MADE environment are illustrated in dialogs E and F. In dialog E, it is shown how to retrieve a concrete element taking part in a given binding. In the figure, the model element 'SecurityManager' is pointed out. This feature is important for tracing specific elements in a complex model. In dialog F, the user highlights the effect of applying pattern 'Security'. As a result, five UML classes involved in the pattern are highlighted in the Rose model; the highlighted model elements are displayed in blue (darker) color.

As an example of model monitoring and checking, let us assume that the generalization relationship between 'SecurityManager' and 'WeekEndsSecurityManager' (which has been specified in the pattern as a role constraint) has been accidentally deleted. This is marked in the figure by a dashed cross on the generalization relationship. This action results in a violated constraint message, displayed to the user in the task view (dialog G). In this case, the violated constraint can be automatically repaired. Similarly, let us suppose that a model element introduced by the pattern has been deleted from the model. As soon as the element is
Figure 3.6 Using patterns in MADE
deleted, the environment detects an invalid binding and relates it to its corresponding pattern (dialog H). It is the task of the user to restore the binding.

3.6 DOCUMENTING MADE PATTERNS

As we have seen earlier, a typical usage of the MADE tool involves two main steps: building the patterns and then using the patterns to present a particular concern. Therefore, one can identify two kinds of pattern stakeholders: those users whose role is to specify aspectual patterns (pattern creators) and those who deploy patterns (pattern users). In order to ease the communication between the two kinds of users, an informal pattern description language can be used. This is, in fact, used for all kinds of patterns in software engineering. For example, well-defined pattern description templates have been suggested for design patterns [GHJV95, BMR+96].

As aspectual patterns are system-specific, the pattern description format used in this work will be different from other used formats. In the context of aspectual patterns, the following items are stressed.

- **Name**: The name of the aspectual pattern.
- **Concern type**: The concern type addressed by the pattern.
- **Intent**: The general purpose of the pattern.
- **Problem**: The problem, forces, and context where the aspectual pattern can be applied.
- **Solution**: A detailed specification of the aspectual pattern. Role diagrams (simplified versions such as the one shown in Figure 3.4) can be used to present the structure of the pattern.
- **Usage scenario**: An example usage scenario illustrating the deployment of the pattern in the pattern tool (i.e. the MADE environment). For example, UML sequence diagrams can be used to specify the deployment steps.

For example, the following description represents the specification of pattern 'Security'.

- **Name**: Security.
- **Concern type**: Variability.
- **Intent**: Modeling security in the system design model.
- **Problem**: You want to define new security strategies concerning access permissions for the various system resources as specified in the system design model. It is possible to define more than one security strategy for a single system resource. Model elements representing a security strategy should be organized according to a well-defined architectural rules.
• **Solution**: A role diagram for pattern 'Security' has been presented in Figure 3.4.

  Pattern roles:

  - **Permission**: A prebound UML class role denoting system permission kinds.
  - **ConcretePermission**: A UML class role representing concrete permission kinds. It is possible to define more than one concrete permission class.
  - **SecurityManager**: A prebound UML class role for abstract security manager.
  - **checkPermission (in 'SecurityManager')**: A prebound UML operation role representing a security policy.
  - **ConcretePermission**: A UML class role representing concrete security managers regarding a concrete permission kind. It is possible to define more than one concrete security manager class per permission kind.
  - **checkPermission (in 'ConcreteSecurityManager')**: A UML operation role for implementing a concrete security policy.

  Role constraints:

  - **inheritance**: Any UML class bound to 'ConcretePermission' should inherit from the UML class bound to 'Permission'. Similarly, any UML class bound to 'ConcreteSecurityManager' should inherit from the UML class bound to 'SecurityManager'.
  - **overriding**: Any UML operation bound to 'checkPermission' in 'ConcreteSecurityManager' should override the UML operation bound to 'checkPermission' in 'SecurityManager'.
  - **parameter**: Any UML operation bound to 'checkPermission' in 'ConcreteSecurityManager' should take an instance of 'ConcretePermission' as a parameter.

• **Usage Scenario**

In order to include a usage scenario of pattern 'Security' in the general description of the pattern, the information presented in Figure 3.6 should be abstracted in a tool-independent form. However, something could still be assumed about the tool user interface. Figure 3.7 depicts an abstract specification of pattern deployment. The notation, which is based on UML sequence diagrams, will be extensively used in the case study presented in the next chapter.

The user and the elements of the user interface are displayed in the upper part of Figure 3.7. The element 'Main View' stands for the pattern view and the architecture view of the environment. The next three elements to the left represent the dialogs for instantiating a pattern, generating a new model element, and locating an existing model element. The last element denotes the UML model to which the pattern instance is applied. The communication between the various elements is represented using arrows. An arrow starts from the element triggering the request and ends at the element handling the request. For example, the first arrow stands for instantiating the pattern. This request is triggered by the user and is handled in the pattern instantiation dialog. When a request is executed, the effect of that request to the system is shown...
in the figure as a graphical note box. For instance, immediately after binding element 'FilePermission' to role 'ConcretePermission', a new binding is added to the binding view and two tasks are added to the task list. In this case, the mandatory task is for providing a concrete security manager class for the selected permission kind and the optional task is for specifying a second concrete permission class. Mandatory tasks are indicated using the letter 'M' at the end of the task title, whereas optional tasks are denoted using the letter 'O'.
Chapter 4

Case Study - Nokia GUI Platform

The purpose of this chapter is to demonstrate the multi-dimensional nature of software systems, to evaluate the methodology introduced in Chapter 2, and to illustrate the main features of the MADE tool environment. The first section gives an overview of the case study. The next section presents a brief introduction to the experiment, around which the chapter is formulated. The experiment is categorized into different groups. Each category is discussed in a separate section. Finally, the last section reports the experiences related to the case study and the presented methodology.

4.1 INTRODUCTION

Network management represents one of the core businesses of Nokia. For managing networks and network elements, the company produces a family of NMS (Network Management System) and EM (Element Manager) applications. The GUI parts of the applications are developed based on a common platform. The main purpose of the platform is to help developers to build Java-based GUIs and to make sure that NMS and EM applications share the same features. The GUI platform has been developed as an object-oriented Java framework. New management applications are constructed by specializing parts of the framework and using the common services offered by the platform.

Figure 4.1 depicts a logical decomposition of the GUI platform consisting of several independent logical blocks. The figure shows those blocks that are relevant to the experiment presented in this dissertation. The upper layer (Management Applications) stands for the NMS and EM applications. They are not part of the platform but they are built on top of it. The middle layer shows a number of platform subsystems. In this dissertation, we discuss three major subsystems.

- **Platform Application Management**: The purpose of this subsystem is to keep track of running applications and to manage the starting of new applications. A key feature is to allow applications to be loaded from different code bases. For example, new applications may be loaded from different file systems.
Platform Service Interface: This component defines the abstract interfaces to the services offered by the GUI platform. Example services include logging, authentication, and online help facilities. Applications can get references to the used service implementations through a Service Registry.

Platform Application Libraries: This subsystem provides common features used for building new network management applications. Example features include concurrency control and an MVC (Model View Controller)-based framework. The MVC architectural style [KSP88] is used to provide a clear separation between the graphical user interface and the other parts of the application.

The bottom layer shows a block representing Platform Service Implementations. The GUI platform comes with a number of default service implementations. If needed, application developers can provide their own service implementations based on the service interfaces.

4.2 EXPERIMENT

The GUI platform presented in the previous section is an example of a complex software system exhibiting a variety of dimensions (as discussed in Chapter 2). Table 4.1 depicts eight situations encountered during the development of new applications based on the GUI platform. Each development scenario can be associated with a typical software engineering activity. The experiments are grouped into four categories.

Specialization. The first two experiments concern model development through the specialization of common platform components. Specialization is an example of variability management, where reuse is typically white-box. The first scenario is for developing an application design model by specializing an MVC-based framework. The second is for constructing a new service implementation based on a well-defined service interface.
**Table 4.1 Experiments**

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Artifact dimension</th>
<th>Concern dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Artifact type</td>
<td>Tool</td>
</tr>
<tr>
<td>Feature variation</td>
<td>Platform feature model, application detailed design</td>
<td>Rational Rose</td>
</tr>
<tr>
<td></td>
<td>Application source code</td>
<td>Eclipse Java editor</td>
</tr>
<tr>
<td></td>
<td>Managing the use of platform services</td>
<td>Platform feature model, application detailed design, application deployment descriptor</td>
</tr>
<tr>
<td></td>
<td>Varying the settings of a service</td>
<td>Platform feature model, application source code</td>
</tr>
<tr>
<td>Maintenance</td>
<td>Platform source code</td>
<td>Eclipse Java editor</td>
</tr>
<tr>
<td></td>
<td>Maintaining platform service implementations</td>
<td>Platform feature model, application source code</td>
</tr>
<tr>
<td>Comprehension</td>
<td>Application deployment descriptor</td>
<td>Eclipse Text editor</td>
</tr>
<tr>
<td></td>
<td>Learning platform libraries</td>
<td>Platform detailed design</td>
</tr>
</tbody>
</table>

**Feature variation.** The second and third experiments address the problem of feature-driven model development. A feature model is defined to represent the variation points at the requirements level. Application developers need to bind these variation points to a specific variant. The former experiment is for including selected platform services to the application under development. The latter case is for associating application GUI components with online help service, using the right settings.

**Maintenance.** The next two experiments represent maintenance activities. Here, maintenance is considered as an activity that is mainly dealt with at the design and implementation phases. In contrast, variability management is typically addressed in the requirements and architecture levels. The first scenario is for modifying the authentication interface of the platform. The other scenario is for maintaining the platform feature model by introducing a new service implementation.

**Comprehension.** The last two scenarios tackle the issue of model comprehension, where models can be expressed using different notations. Here, comprehension is based on customizing original complex models according to the actual needs of the learners. The purpose of the first case is comprehending application deployment policies whereas the other case is for comprehending application management facilities offered by the platform.
Each of the eight development scenarios is associated with specific dimension values. For example, consider the dimension values involved in the first experiment. Regarding the artifact dimension, the artifact types, tools, and notations involved are detailed design, Rational Rose, and UML respectively. As for the concern dimension, the following are involved: architectural patterns/specialization interfaces (as concern categories), application designers/developers (as stakeholders), and application development (as development needs).

In chapter 2, we have studied the various relationships between the two main dimensions structuring a software system: the artifact dimension and the concern dimension. Table 4.1 illustrates the complexity of such relationships. For example, the same artifact type can be involved in different development needs. The platform feature model, for example, is involved in both application development and platform maintenance. The same tool can be used for different purposes. Rational Rose, for instance, is used for managing application detailed design and for learning system functions. Furthermore, the same notation can be used for different development phases. To give an example, UML is used for application development, comprehension, and maintenance. As for the concern dimension, the same concern type can be addressed in multiple artifact types. Features, for instance, have their representation in feature models, detailed designs, source code, and deployment information. Similarly, the same concern type can be relevant to various stakeholders. In the case study, maintaining a service interface is important for both platform and application maintainers. Finally, the same development need can involve different concern types. As shown in the table, comprehension involves both system functions and product configurations.

The MADE tool can be used to manage the development scenarios presented above. The subsequent sections present the four applications of heterogeneous aspectual patterns: feature variation patterns, maintenance patterns, specialization patterns, and comprehension patterns. Table 4.2 shows the purpose of each pattern category, the different stakeholders that are involved in creating the patterns and those involved in applying them. Feature variation patterns, for example, are used to manage feature variability in software systems. In this case, heterogeneous aspectual patterns are used to represent the different variation points and to make sure that when a specific feature variant is selected, that decision is propagated to all models of the application. These kinds of patterns are usually created by product-line developers specifying the different variation points in the system architecture. Product developers then apply these patterns to design and implement new products.
4.3 SPECIALIZATION

This section discusses the use of specialization patterns [P2]. A specialization pattern is an aspectual pattern used to document the specialization interface of a system, possibly across different levels of abstraction.

4.3.1 Building New Platform Applications

The GUI parts of EM and NMS applications are developed by specializing an MVC-based framework. The principle of the framework is that every view object is managed by exactly one controller object. Every controller, except the main controller, is managed by a parent controller. An application has exactly one main controller associated with one main view. In addition, the application could define other kinds of controllers for handling different kinds of views. In addition to the main view, there are four kinds of views: subviews, dialog views, panel views, and internal frames views. Similarly, there are four kinds of controllers, in addition to the main one: subcontrollers, dialog controllers, panel controllers, and internal frame controllers.

In order to manage the above situation, we define three separate concerns: creation, controller, and view. The creation concern covers the creation of application main classes, the controller concern is used to define application controllers, and the purpose of the view concern is to define application view objects. These three concerns need to be combined together in order to achieve the overall expected behavior of the application. We define three specialization patterns for representing the concerns. The purpose of the patterns is to capture the specialization steps required for generating a detailed design for the application. The design model is specified using UML class diagrams.

Pattern descriptions

Figure 4.2 depicts the three specialization patterns 'Creation', 'View', and 'Controller', represented as three cloud shapes. In addition to the patterns, the figure shows how the patterns should be combined, i.e. overlapping relationships. An overlapping relationship between two roles is graphically represented using a dashed double-headed arrow linking these two roles and are textually expressed in terms of composition rules [P8]. In this case, the rules imply that during the binding process, roles 'ApplView' of 'View' and role 'Managed-View' of 'Controller' should be bound to the same UML class representing a view object. Otherwise, the two patterns would combine in a wrong way leading to an undesirable representation of the concerns [HHP+05]. The two class roles are said to be overlapping. The purpose is to ensure that for every view object created using the 'View' pattern there is a corresponding controller object provided in pattern 'Controller'.

In the remainder of this section, we separately present the structure of each pattern, we then show a unified usage scenario for all the three patterns. For brevity, we do not show all the roles and we omit the discussion of role constraints.
A pattern for the creation concern:

- **Name**: Creation.
- **Concern type**: Specialization interfaces.
- **Intent**: Developing application main classes.
- **Problem**: You want to generate application main classes so that the application is associated with the framework system. In other words, the application should be created according to the well-defined mechanism defined by the framework. This is the first step in creating the application.
- **Solution**: Figure 4.2 depicts a role diagram for pattern ‘Creation’.

Pattern roles:

- **Application**: A prebound UML class role denoting the framework class ‘Application’, which the application main class should extend.
- **ManagedApplicationFactory**: A prebound UML class role representing the framework factory interface for creating applications.
- **ApplMain**: A UML class role representing the application main class.
- **ApplMAFactory**: A UML class role for the concrete factory creating the application.
- **createApplication**: A UML operation role for the method creating the application.
creates: A UML dependency role standing for the instantiation relationship between the factory class and the application main class.

A pattern for the view concern:

- **Name**: View.
- **Concern type**: Specialization interfaces, architectural pattern.
- **Intent**: Developing application view components.
- **Problem**: You want to define views for the application. The application should have exactly one main view and zero or more views of other kinds. A view could define any number of GUI components.
- **Solution**: Figure 4.2 depicts a role diagram for pattern 'View'.

Pattern roles:

- **MainView**: A prebound UML class role representing the framework class 'MainView', which is a base class for the application main view.
- **ApplMainView**: A UML class role for the application main view.
- **Component (in 'ApplMainView')**: A UML attribute role for GUI components in the main view. A view may define any number of components.
- **OtherViewType**: A UML class role representing the other view kinds, which the application can define. The application can define any number of views of these kinds.
- **ApplView**: A UML class role for the other application views.
- **Component (in 'ApplView')**: A UML attribute role for GUI components in the application other views. A view may define any number of components.

A pattern for the controller concern:

- **Name**: Controller.
- **Concern type**: Specialization interfaces, architectural pattern.
- **Intent**: Developing application controller components.
- **Problem**: You want to define controllers for the application. Every view object in the application should be associated with a controller. The application should define exactly one main controller and any number of controllers of other kinds depending on the number of application views. Each controller, except the main one, should have a parent controller.
- **Solution**: Figure 4.2 depicts a role diagram for pattern 'Controller'.

Pattern roles:

- **MainController**: A prebound UML class role representing the framework class 'MainController', which is a base class for the application main controller.
Case Study - Nokia GUI Platform

- **ApplMainCont**: A UML class role for the application main controller.
- **OtherContType**: A UML class role representing the other controller kinds, which the application can have. The application can define any number of controllers of these kinds.
- **ApplCont**: A UML class role for the application other controllers.
- **parentCont**: A UML class role for the parent controller of any application controller, except the main one.
- **ManagedView**: A UML class role for the view to be managed.

**Usage Scenario:**

Figure 4.3 shows an example scenario of patterns 'Creation', 'View', and 'Controller'. The scenario is started by instantiating pattern 'Creation'. The first task is to generate a new application class ('MyApplMain'). For brevity, we skip the remaining tasks generated by the pattern instance. The next major step is to instantiate pattern 'View' for defining application views. According to the scenario, the main application view is provided. It is an optional task to provide other kinds of views. The third major step is the instantiation of pattern 'Controller'. In the context of this pattern instance, the main controller has been provided. Even though an application could provide other kinds of controllers, we do not have optional tasks for these. The reason is that no views, except the main one, have been generated in the instantiation of pattern 'View' and that there is a composition rule between role 'ApplView' in pattern 'View' and role 'ManagedView' in pattern 'Controller'. Even though the composition rule could be considered as bidirectional, the pattern tool interprets the rule in this example as follows. First, a concrete element for role 'ApplView' is provided, then the element is automatically bound to role 'ManagedView'.

### 4.3.2 Providing a New Service Implementation

Nokia GUI platform provides default service implementations. Application developers, however, can build and use their own implementations if needed. Service implementations need to conform to well-defined service interfaces. For example, let us suppose that a new logging service (trace, system, and error logs) implementation is needed. There are separate interfaces for each logging service. Even though it is in practice possible that one class implements more than one service interface, it is advised that a service implementation class addresses one service only. For instance, it is preferred to have separate implementation classes for trace, system, and error logging services. Aspectual patterns can be used to manage the development of new logging service implementations.

**Pattern description**

A specialization pattern named 'LoggingServiceImpl' is used to specify the development steps required for a new logging service implementation.

- **Name**: LoggingServiceImpl.
- **Concern type**: Specialization interfaces.
- **Intent**: Specialization interfaces - Developing new implementation for logging service.
Figure 4.3 Usage scenario for patterns 'Creation', 'View', and 'Controller'

- **Problem:** You want to generate a new implementation for platform logging service. The logging service includes trace, error, and system logs. It is advised that each service type is implemented in a separate implementation class. Method implementation should conform to well-defined rules. For example, all methods and constructors should throw an exception of type 'IOException'.

- **Solution:** Figure 4.4 depicts a role diagram for pattern 'LoggingServiceImpl'. The internal representation for error and system logging have been omitted.

Pattern roles:

- **TraceLogImpl:** A Java class role denoting the implementation class for trace log service. The cardinality of the role is [0..1] since the developer might choose not to provide such service type (this is the case for the next two roles as well).

- **ErrorLogImpl:** A Java class role denoting the implementation class for system log service.

- **SystemLogImpl:** A Java class role denoting the implementation class for error log service.
– writeMessage: A Java method role for writing a specified message to the trace log.

– writeSeqMessages: A Java method role for writing a specified sequence of messages to the trace log.

– TraceLogConstructor: A Java constructor role for the constructor of the implementation class. The developer might define more than one constructor.

Role constraints:

– IOException: An exception constraint stating that all methods and constructors should throw an exception of this type.

– realization: This constraint, which can be specified as an inheritance constraint, ensures that the implementation class realize a proper interface.

• Usage Scenario: An example usage of pattern 'LoggingServiceImpl' is shown in Figure 4.5. The case is to provide a new implementation for the trace log service. First the pattern is instantiated and tasks are displayed to the user. Then, example concrete elements, i.e. a Java class, a constructor, and a method, are generated and added to the model. For brevity, we do not show the execution of the remaining tasks.

4.4 FEATURE VARIABILITY MANAGEMENT

This section focuses on the use of feature variation patterns [P7]. A feature variation pattern is an aspactual pattern used to manage feature variability in a system across different software artifacts.

4.4.1 Managing the Use of Platform Services

Nokia GUI platform provides a number of services useful for GUI applications. There are services for system logging, online help, user authentication, product internationalization, clipboard usage, CORBA-based communication, and licensing. Depending on the environment used, each service may have different implementations. Applications, built on top of
the GUI platform, get a reference to the proper service implementation from a registry file. It is essential that product developers register the right service implementation.

In this experiment, we consider platform services as features. Figure 4.6 depicts a feature model representing the main services provided by the GUI platform. All services are optional. It is up to the product developer to decide which services to use. In the next discussion, we show how this situation can be managed by an example heterogeneous aspectual pattern.

Pattern description
A feature variation pattern named 'Services' is used to manage the variability issues related to the use of features.

- **Name**: Services.
- **Concern type**: Feature variability.
- **Intent**: Use of platform services.
- **Problem**: You want to manage the use of platform services in the GUI application. Typically, services are represented across different application artifact types including detailed design, source code, and deployment information. As a service might have multiple implementations, it is necessary to register the right service implemen-
There cannot be more than one implementation of one interface registered in the service registry file.

- **Solution:** Figure 4.7 depicts a role diagram for pattern 'Services'.

  Pattern roles:
  
  - Service: A prebound UML class role denoting the feature to be managed.
  - ConcreteService: A UML class role representing the selected service to use (selected feature variant). More than one service can be selected.
  - Application: A prebound UML class role representing the GUI application class.
  - UMLServiceComponent: A UML class role for the class representing the selected service at the design level.
  - Uses: A UML association role standing for the association between the application class and the selected service.
  - JavaServiceComponent: A Java class role for the Java class representing the desired service implementation.
Usage Scenario: Figure 4.8 shows an example scenario of pattern ‘Services’. The pattern instance is used to manage the representation of the selected service (TraceLog) across the different application artifacts. The service is selected from the feature model represented in Figure 4.6. The consequences of the pattern is to represent the TraceLog service in the detailed design model (UML class ‘TraceLogService’), to check and select a desired service implementation (Java class ‘TraceLogImpl’), and to represent the service implementation in the registry file (XML fragment). The last step in the scenario is to trace the element corresponding to the service in the design model. As a result, class ‘TraceLogService’ gets selected in the UML model. For brevity, the binding of role ‘Uses’ is not shown.

4.4.2 Varying the Settings of a Service

The GUI platform provides a help service that can be used to associate application GUI components with online help. Help request can vary from simple services like retrieving help strings to more advanced services such as index, table of contents, or search. Figure
4.9 depicts a feature model specifying such variability issue. First, the variant ’User_Event’ indicates that the call resulted from a user request whereas ’System_Event’ indicates that the call originated from the application. GUI components can support either event types but not both at the same time. Second, developers must specify what type of help service is requested. There are five variants for such service type. ’Contents’, for example indicates that the target of the request is a table of contents whereas ’Search’ indicates that the target of the request is a search page. Similarly, the choice of the service type is exclusive. We show how an example heterogeneous asp ectual pattern can be used to manage the variability issue concerning the use of online help service.

**Pattern description**

A feature variation pattern named ’OnlineHelp’ is used to manage the problem of associating application GUI components with online help service.

- **Name**: OnlineHelp.
- **Concern type**: Feature variability.
- **Intent**: Settings for online help service.
- **Problem**: You want to associate help service for selected application GUI components. For this, proper service and event types have to be chosen. The selected service and event types have to be properly registered in the service initialization Java method. There can be more than one GUI component defined in the application class.
- **Solution**: The role structure of pattern ’OnlineHelp’ is shown in Figure 4.10.

Pattern roles:
- **EventType**: A prebound UML class role denoting the event type feature.
- **ServiceType**: A prebound UML class role denoting the service type feature.
- **ConcreteEventType**: A UML class role representing the event type variant to be used.
- **ConcreteServiceType**: A UML class role representing the service type variant to be used.
- **Application**: A prebound UML class role representing the GUI application class.
– initializeHelpService: A prebound Java method providing the initialization code for help service.

– GUIComponent: A Java attribute role standing for the GUI component to associate the help service with. It is possible to select more than one component.

– Implementation: A Java code fragment role for registering the proper help service to the selected GUI components.

• Usage Scenario: An example usage of pattern 'OnlineHelp' is presented in Figure 4.11. The scenario is to associate a GUI component name 'newSession' with online help. This is done by generating a Java code fragment for registering the help service with variants 'User_Event' as an event type and 'Topic' as a service type.

4.5 MAINTENANCE

This section illustrates the purpose of maintenance patterns [P5]. A maintenance pattern is an aspectual pattern used to document anticipated tasks for maintaining a given system.

4.5.1 Modifying a Service Interface

As we have seen earlier, the GUI platform describes well-defined service interfaces. As the platform evolves, it might be required, in some situations, to modify the original service interfaces. This can be the case, for instance, when releasing a new version of the service. As an example, let us consider the authentication service. The authentication service can be enhanced by supporting new authentication schemes such as adding support for single sign on, which would allow the user to get authenticated once when accessing multiple products. Such an enhancement requires adding new methods to the interface so that all service implementations will take care of this enhancement. However, in order to avoid version compatibility problems, the interface modification should be carried
out in a rather controlled way. The aspectual pattern presented next handles such a situation.

**Pattern description**

A maintenance pattern named ‘AuthenticationInterface’ is used to document the maintenance tasks for the service interface.

- **Name**: AuthenticationInterface.
- **Concern type**: Maintenance interfaces.
- **Intent**: Maintaining the authentication service interface.
- **Problem**: You want to modify the interface of the authentication service for supporting new authentication schemes. The modifications should not affect the existing methods. The newly added authentication methods should be named ‘authenticate’ and should incorporate at least one parameter; the parameterless authentication method is used for a simple authentication scheme.
- **Solution**: An example maintenance pattern solving the above problem is illustrated in Figure 4.12.
Pattern roles:

- **Interface**: A Java class role that should be bound to the interface to be modified.
- **existingMethod**: A Java method role representing the existing interface methods. The original interface can define multiple methods.
- **newMethod**: A Java method role denoting newly added authentication methods. It is possible to add more than one method.

Role constraints:

- **watcher**: A watcher constraint for monitoring changes in the original interface methods.
- **methodName**: A name constraint requiring that the name of new interface methods be ’authenticate’.
- **parameter**: A parameter constraint requiring that newly added interface methods have at least one parameter.

**Usage Scenario**: An example usage scenario of pattern ’AuthenticationInterface’ is illustrated in Figure 4.13. The scenario starts by selecting the service interface to modify (’AuthenticationService’). The second task is to specify the original interface methods. For example, method ’getUserName’ is selected. In order to illustrate the usage of constraint ’watcher’, the user accidently changes the signature of method ’getUserName’. The tool reacts to this change by detecting a watcher constraint violation. In this case, it is not possible to automatically repair the violated constraint. The final task is to provide a newly added interface method according to the specification.

### 4.5.2 Maintaining Platform Service Implementations

In Section 4.3.2, we have seen that product developers can provide their own service implementation if the default implementations provided by the platform do not exhibit the desired functionality. Even though a new service implementation is typically built for a specific product, it may be suitable for other products as well. In fact, this is a common maintenance situation in the field of software product-lines as it is generally thought that product-specific software may provide useful hints about likely useful extensions to the functionality supported by the reusable parts of the software product line [Bos00]. A clear advantage of
Figure 4.13 Usage scenario for pattern 'AuthenticationInterface'

managing this maintenance problem is to prevent multiple same implementations of platform services, thus reducing development costs. In the next discussion, we show how this situation can be handled using aspectual patterns.

Pattern description
A maintenance pattern named 'ServiceImpl' is used to document the maintenance tasks for introducing a new service implementation to the list of implementations originally offered by the platform. The pattern specification assumes that a new service implementation is given in terms of a Java class and that the original list of platform service implementations is given in terms of a feature model such as the one given in Figure 4.6. In the feature model, it is assumed that features represent the platform services and possible implementations for a service are described as variants for the corresponding feature. Finally, we also assume that the task for including the actual service implementation in the platform binaries is given as an informal task.

- **Name**: ServiceImpl.
- **Concern type**: Maintenance interfaces.
- **Intent**: Maintaining available service implementations.
- **Problem**: You want to include a new service implementation, built for a specific application, in the platform. It is possible to include more than one service implementation. It is also assumed but not recommended that one service implementation class may address more than one service.
Solution: An example maintenance pattern addressing the problem is illustrated in Figure 4.14.

Pattern roles:

- ServiceImpl: A UML class role for a new service implementation. The user may select more than one service implementation.

- Services: A prebound UML class role for platform services.

- ConcreteService: A UML class role for the concrete service realized by the selected implementation. Multiple services can share the same implementation class.

- NewImpl: A UML class role for representing the new implementation in the platform feature model.

- Updating: An informal task role for reminding the user to include the new implementation in the platform binaries.

Usage Scenario: An example usage scenario of pattern 'ServiceImpl' is illustrated in Figure 4.15. Once the pattern is instantiated, the user is prompted for a service implementation. A new implementation for the trace log service ('MyTraceLogImpl') is selected. As a second task, the user specifies 'TraceLog' as the corresponding platform service. The third task is to include the new implementation in the platform service model. Finally, the user is reminded to include the new implementation in the actual platform binaries.

4.6 COMPREHENSION

This section discusses the application of comprehension patterns [P6]. A comprehension pattern is an asceptual pattern used to document the comprehension tasks related to learning a system model through model customization.
4.6.1 Learning Product Configuration Facilities

When developing a new GUI application, the usage of the platform is expressed using a deployment descriptor file specified in XML. The file provides a start up mechanism for the platform in the context of the application. For example, the deployment information contains configuration options related to the use of platform services. Therefore, the deployment descriptor file should be carefully designed since any errors could result in a malfunction of the application.

Table 4.3 represents a number of example configuration elements in a deployment descriptor file. The use of the configuration elements can be mandatory or optional; an element can occur once or multiple times. Aspectual patterns can be used to help developers in the process of comprehending the structure of deployment descriptor files. This is done through a stepwise development of an example file according to the context of the learner.

Pattern description

A comprehension pattern named 'DeploymentDescriptor' is used to document the comprehension tasks related to learning the platform deployment features.

- **Name**: DeploymentDescriptor.
- **Concern type**: Comprehension, product configuration.
- **Intent**: Learning product configuration facilities.
- **Problem**: You want to learn the structure of application deployment descriptor files. Depending on the context, deployment information may include different configuration elements. When building a deployment file, it is essential to conform to a well-defined structure indicated by a DTD (data type description) and to comprehend.
### Table 4.3 Allowed tags in platform deployment descriptor

<table>
<thead>
<tr>
<th>Element</th>
<th>Parent</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>platform</td>
<td>–</td>
<td>The root element in the deployment descriptor</td>
<td>Mandatory tag</td>
</tr>
<tr>
<td>platform-settings</td>
<td>platform</td>
<td>The common settings of the platform</td>
<td>Mandatory tag</td>
</tr>
<tr>
<td>application</td>
<td>platform</td>
<td>The application definitions</td>
<td>One to many applications</td>
</tr>
<tr>
<td>common-services</td>
<td>platform</td>
<td>The services settings</td>
<td>Mandatory tag</td>
</tr>
<tr>
<td>appmanager</td>
<td>platform-settings</td>
<td>The ApplicationManager class name</td>
<td>Optional tag</td>
</tr>
<tr>
<td>publish</td>
<td>application</td>
<td>The class names that should be published from this applicationbase</td>
<td>Optional tag (zero to many)</td>
</tr>
<tr>
<td>name</td>
<td>application</td>
<td>The name of the application</td>
<td>Mandatory tag</td>
</tr>
<tr>
<td>description</td>
<td>application</td>
<td>The description of the application</td>
<td>Optional tag</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

![Figure 4.16](image)

**Figure 4.16** Role diagram for pattern 'DeploymentDescriptor'

...hend the exact semantics of the elements. There can be only one deployment file per application.

- **Solution**: An example structure of pattern 'DeploymentDescriptor' is shown in Figure 4.16. For brevity, the structure depicts only a small set the allowed configuration tags.

Pattern roles:

- DDFile: An XML file role representing the deployment descriptor file.
- platform: An XML tag role for deployment element ’platform’. This is a mandatory tag.
- application: An XML tag role for deployment element ’application’. This tag can occur at least one time.
- **platform-settings**: An XML tag role for deployment element ‘platform-settings’. This is a mandatory tag.

- **common-services**: An XML tag role for deployment element ‘common-services’. This is a mandatory tag.

- **appmanager**: An XML tag role for deployment element ‘appmanager’. This is an optional tag.

- **publish**: An XML tag role for deployment element ‘publish’. This tag can occur one or many times.

- **name**: An XML tag role for deployment element ‘name’. This is a mandatory tag.

- **description**: An XML tag role for deployment element ‘description’. This is an optional tag.

**Usage Scenario**: An example comprehension scenario using pattern ‘Deployment-Descriptor’ is presented in Figure 4.17. The scenario is started by instantiating the pattern. The role bindings that are left to be carried out are considered as learning tasks. The first learning task is to provide the deployment descriptor file. Once the file is created, the tool displays tasks for generating the allowed tags. The task capabilities of the tool (i.e. ordering, task descriptions, etc) are used to highlight the purpose of each tag and its relationships to the other tags. In this example scenario, all elements are provided using the generative feature of the tool. When the pattern is fully applied, a customized deployment descriptor file is generated. Figure 4.18 shows the contents of the generated file following the steps executed in the example scenario.

### 4.6.2 Learning Platform Libraries

Once a GUI application has been developed, it is loaded and started using the platform. This kind of service is offered by a separate subsystem known as platform application management. The subsystem comes with numerous features. Depending on the context of the application, certain features are used and others are omitted. Figure 4.19 depicts a brief illustration of the application management features. For example, in addition to the core elements, there are features for publishing and caching functionality.

Considering the core features, there are application-specific elements (‘AnyApplication-FactoryImpl’), possible library extensions (‘AnyCodeBaseImpl’), and optional elements (a code base implementation needs to implement ’ResourceFinder’ only if it is desired to support searching for resources). Obviously, the learner of the system can be overwhelmed if all the library information is given her at once. Instead, the learner should be shown only those parts of the system description that are relevant for her purposes. It is a fact that individuals can only learn a limited amount of information at one time [Mil56]. It would be also beneficial to use the learner’s own context such as her own library extensions or her own concrete names for the application-specific elements. In this way, the effort required for the learning process could be greatly reduced. In the next discussion, we show how this situation can be managed using aspcausal patterns.
Pattern description

Comprehension patterns can be used to assist the learner through the learning process. Assuming that the description of the system is given in terms of a UML class diagram, such as the case of this experiment, the patterns can be used to build, step-by-step, a customized model of the system reflecting the learner’s own context. In this experiment, a separate comprehension pattern is used to document the comprehension tasks required for learning.
a specific system feature. For example there is a pattern for learning core elements, and another for learning caching features. For brevity, we discuss only one comprehension pattern and we present a very small part of the pattern structure.

- **Name**: ApplicationManagement.
- **Concern type**: Comprehension, system functions.
- **Intent**: Learning application management library.
- **Problem**: You want to comprehend the core elements of the application management system. It should be possible to provide application-specific concrete names, provide own extensions, if any, and to decide which optional features to use.
- **Solution**: Figure 4.20 depicts a role diagram for pattern 'ApplicationManagement'.

  **Pattern roles:**
  - ApplFactImpl: A UML class role for the application-specific element denoting the application factory.
– CodeBase: A UML class role for element 'CodeBase'.
– FileCodeBase: A UML class role for element 'FileCodeBase'. The use of this element is optional.
– FileCBRealization: A UML realization role for the realization relationship between elements 'FileCodeBase' and 'CodeBase'.
– CodeBaseImpl: A UML class role for the learner's own implementations of 'CodeBase'. The user can define any number of implementations.
– ImplCBRealization: A UML realization role for the realization relationship between a possibly added code base implementation and 'CodeBase'. The relationship is required.
– ResourceFinder: A UML class role for element 'ResourceFinder'.
– ImplRFRealization: A UML realization role for the realization relationship between a possibly added code base implementation and 'ResourceFinder'. The relationship is optional.

- **Usage Scenario**: An example usage scenario of pattern 'ApplicationManagement' is demonstrated in Figure 4.21. The scenario is started by instantiating the pattern. As a first task, the learner provides a UML class for the application factory, specifying a concrete name ('MyApplFactImpl'). The next two steps provide the two required elements 'CodeBase' and 'ResourceFinder'. Next, the learner chooses to provide a code base implementation ('MyCodeBaseImpl') of her own instead of the predefined file-based implementation ('FileCodeBase'), the new implementation does not use the resource searching functionality. Finally, the scenario is ended by highlighting the effect of the pattern in the model. In this case, the user wants to view the core elements (and the user's own extensions) in the library model, assuming that the model consists of other elements as well. The customized model, which is generated according to this scenario, is illustrated in Figure 4.22. It is assumed that elements 'CodeBaseNode' and 'ManagedApplicationFactory' have also been provided.

4.7 EXPERIENCES AND EVALUATION

We evaluate the case study from three main perspectives. First, we relate the experiment to the problem of essential difficulties. Then, we report our experiences with the identification and the creation of aspectual patterns. Finally, we evaluate the use of the patterns and the MADE environment applied to the case study.

4.7.1 Dealing with Essential Difficulty

The case study demonstrated the essential sources of difficulty of software development: conformity, complexity, changeability, and invisibility. The specialization tasks, encountered during the development of new applications, reflected the conformity issue. Product and service implementations had to conform to certain architectural rules and well-defined interfaces implied by the platform. Similarly, feature variability revealed the problem of
Figure 4.21 Usage scenario for pattern 'ApplicationManagement'

Figure 4.22 Customized model for platform application management
complexity. We needed to make explicit the problem features addressed by the product-line and to express the solution features across multiple artifact types. It was essential to keep all artifacts aligned and consistent. Furthermore, the maintenance scenarios, presented in the case study, exhibited the notion of changeability. Changing the authentication service interface, for example, was a result of a new requirement. Changeability should be carefully managed in order to prevent broken conventions and incomplete maintenance actions. Finally, the issue of invisibility was highlighted in the two comprehension situations. It was shown how system models could be too complex to comprehend. Usually models, such as the ones discussed in the case study, present more information than needed and their static structure impedes the comprehension process.

The case study showed that difficulty of software development can in fact be associated with the notion of software concerns. Each source of difficulty reflected certain concern types. Various types of concerns have been identified, including specialization interfaces (framework hotspots), architectural concerns (MVC), features (platform services), and development process concerns such as maintenance (service interfaces) and comprehension (platform libraries). We have shown how the same concern type can reflect different development needs and can be addressed by multiple stakeholders. In the case study, for example, it was shown how specialization tasks can be involved in both application development and platform maintenance, and is addressed by both designers and programmers.

Similarly, the case study demonstrated how sources of difficulty can be attributed to the fact that a software system involves multiple artifacts. The issue of complexity, as reflected in feature variability for instance, involved artifacts specified in UML, Java, and XML. Another illustration was given in the case of comprehension, where we (as system learners) had to deal with different artifact types such as detailed design and deployment description, and different notations such as XML and UML. The notations were handled using different tools such as Eclipse editors and Rational Rose. Therefore, the case study validated our approach to analyzing the structure of software systems.

4.7.2 Identifying and Building the Patterns

The development scenarios identified in the scope of the case study reflected different variability issues in the platform and the product development process. For identifying the variability points, we have spent considerable effort on studying the platform itself and interviewing several stakeholders participating in the development and use of the platform. This allowed us to identify the various dimensions presented earlier. The next major step was to structure the elements addressed in each of the experiments in a way that makes it suitable for identifying the patterns. For example, we had to dig up the feature models from the platform documentation for constructing the feature variation patterns. The documentation was not structured according to our methodology. Another example is that we had to structure the specialization interface of the MVC-based framework for constructing the specialization patterns.

Once the patterns have been identified and specified as role diagrams, the next challenge was to represent them in the pattern tool. We have used the MADE environment to build the patterns. Currently, MADE patterns are constructed manually using the pattern editor provided by the tool, which was in some cases a laborious process. Ideally, it should have been possible in some situations to build the patterns, at least partially, in an automatic
or semi-automatic fashion. For example, feature variation patterns could have been partially constructed out of feature models. Similarly, specialization patterns could have been partially deduced from well-defined interfaces (service interfaces, for example) or reverse engineered from concrete specializations.

In this work, aspectual patterns provided support for UML, Java, and XML semantics. In the case study, this was considered as a limitation since several platform artifacts were specified in a different notation. For example, platform services were described in terms of Word documents. A more convenient option would have been to link the relevant parts of these documents to the patterns, rather than transforming them to feature models expressed in UML. Nevertheless, this option can still be realized as new role types covering elements in Word documents could be added to the patterns. For the MADE implementation, this means adding support for communication with Microsoft Word editor.

4.7.3 Using the Patterns

Throughout the case study, we could conveniently use our pattern concept to manage various development scenarios encountered during application development. Nevertheless, the set of patterns we have discussed represents only a part of a bigger annotation. The purpose of the patterns presented in this dissertation was to demonstrate the applicability of our pattern concept and tool environment (also to our industrial partners). Accordingly, the patterns have mainly been tested by the project members. Once the annotation is completed, the environment will have to be tested by the industrial partners.

The experiments illustrated the various properties of aspectual patterns being notation and artifact neutral (patterns were used to describe various kinds of concerns such as features and specialization interfaces involving different artifact types and notations), system-specific (all the patterns presented in the case study are specific to the GUI platform, yet they offered a convenient level of reuse as the development activities they address represent recurring problems), and heterogeneous (a feature variation pattern, for example, had roles supporting UML, Java, and XML).

In general, any increase in the size of the pattern library, in terms of number of patterns, might raise applicability problems as users might be confused with which patterns to deploy and which instantiation path to take. These kinds of problems are solved by providing extensive documentation attached to the pattern catalogue. The case study demonstrated that the approach scales up. As new patterns are identified and added to the library, the pattern system has been restructured accordingly. However, this does not mean that all the patterns were organized into the same repository. Our approach was to organize different categories into different pattern assemblies. The disadvantage of this approach is that patterns in different categories cannot inter-communicate. Relationships between patterns were expressed using compositions rules. Currently, composition rules are specified manually, which might become a challenging task as the number of rules grows high. This problem can be partially resolved by keeping the patterns small in size and loosely coupled.

Another issue related to scalability is the suitability of the stepwise development approach, as the number of pattern roles gets larger. The number of tasks generated by a pattern instance is approximately equivalent to the number of roles in the pattern. Our experiences showed that stepwise development is a useful technique in the sense that we had better control over the development process and we could immediately see the effect of a
development step in the model. However, in some situations, the approach turned out to be too fine-grained as increased number of development tasks required too much user interaction. This problem, however, is mainly related to how the MADE environment executes the task list but not to the approach itself. To manage such a situation, the MADE tool currently provides facilities to perform a set of tasks at once and to automatically bind roles (based on the composition rules, for example). Another enhancement would be to give users the option to completely instantiate patterns in a batch-like mode. In this case, the concrete elements will be generated using the default properties of pattern roles.

In Chapter 2, we have defined four basic operations on aspectual patterns: synthesis, checking, tracing, and visualization. The usage scenarios presented in the case study confirms the need for such operations and show their applicability in the MADE environment. Synthesis (generating and locating elements) has been illustrated in all experiments. One weakness of the generative approach of the MADE environment, in the case of Java elements, was that the tool provided limited support for the individual Java statements in method bodies. The tool uses the concept of a Java code fragment (there is an equivalent Java code fragment role) to represent the internal elements of method bodies. However, the tool cannot remember the placement of the code fragment once it is generated.

In the case study, checking has been used to watch for illegal modification of source code elements. A useful conformance checking mechanism, which the tool does not currently provide, is the checking of existing models against the pattern specification. This can be done by automatically binding pattern roles to existing models elements, perhaps by using heuristics such as element properties and model structural relationships. The tool then could report all the missing elements and/or the broken architectural conventions. Highlighting, in turn, has been used in the comprehension scenario to identify certain kinds of elements in the overall model. Currently, highlighting UML elements only works at the class level, it is not possible to highlight individual operations or attributes. Tracing has been applied in feature variation patterns to view the design element corresponding to a certain feature (platform service). In the MADE environment, it is not possible to trace the concrete elements back to the roles they are bound to, which might be useful during the development of complex models.

Nevertheless, we think that the MADE environment provided reasonable support for the tool requirements identified in Chapter 2. Using the same tool, we could conveniently generate development environments suitable for different software engineering activities such as application development, maintenance, and comprehension. The tool has been used as a unified interface to several artifact editors such as Rational Rose and Eclipse Java editor. During the pattern instantiation process, the MADE environment provided a smooth switch from one tool to another, and support for the communication between the different artifact types. The tool limitations we have encountered during the case study could be considered as a basis for future tool development.
Chapter 5

Related Work

In this chapter, we discuss research work and tool environments related to structuring of software systems and the management of heterogeneous software artifacts. We then compare our methodology to other related development approaches, i.e. model-driven development, software product-lines, and generative programming.

5.1 CONCERN-ORIENTED STRUCTURING OF SOFTWARE SYSTEMS

Concerns have been a primary motivation for organizing and decomposing software systems into smaller, more manageable and comprehensible parts. Recent research on aspect-oriented software development (AOSD) [FECA04] suggests that the main structural decomposition of a system should be complemented by software entities cross-cutting the primary structure of the system. Typically, the introduced entities correspond to specific concerns identified in the problem domain. AOSD provides means for the systematic identification, separation, representation, and composition of crosscutting concerns [RMA03]. According to Kiczales [Kic05], two crosscutting structures imply that neither can fit neatly into the other.

Aspect-oriented programming (AOP) is a programming paradigm incorporating the ideas of AOSD. AOP organizes the crosscutting concerns into separate modules called aspects. Aspects are merged with base programs or models of programs. The process of merging is called weaving. There are two ways in which aspects are weaved: static or dynamic. Static weaving modifies a structural base model by inserting new model elements. Dynamic weaving adds new behavior to applications at runtime. In this dissertation, binding pattern roles to concrete elements can be considered as a static weaving mechanism.

Support for aspects have been researched and developed within different levels of abstraction. At the source code level, AspectJ [Asp05] 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 crosscutting concerns. At the design level, there have been many proposals on how to model aspects in UML, for instance [Cla02, KK03]. 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. Compared to traditional aspect-oriented development environments, the weaving of aspectual patterns is considered as an interactive, incremental process where the join points are located under the guidance of a pattern tool, rather than in a fully automated fashion.

Nevertheless, it has been recognized that the goals of separation of concerns have not been achieved in practice [TOSH04]. The reason is that there can be different kinds of concerns depending on many factors such as the nature of the development activity, the stage of the software lifecycle, and the kind of involved developers. Thus, concerns should be separated and defined according to the needs of the actual situation. In order to overcome these problems, the concept of separation of concerns has been extended to cover multiple dimensions under the title multi-dimensional separation of concerns (MDSOC). Similarly, this dissertation argued that the concept of a concern could be extended to include other concern types such as the variation of a feature, the specialization interface of a system or even development-process concerns such as maintenance tasks and comprehension needs. Our term of 'concern types' is referred to in MDSOC as dimensions of concerns.

An approach to provide multi-dimensional separation of concerns has been presented in the Hyperspace model [OT00], other approaches based on the ideas of MDSOC include Parallax [SS05] and Cosmos [SR02]. Hyperspace, which is the closest to our methodology, is a conceptual model for advanced separation of concerns that must be instantiated for some language or notation before applicable. In this approach, a system is described using hyperslices, which can be seen to correspond to aspectual patterns. Hyperslices can be composed recursively to hypermodules, which contain composition rules for the component slices. A concrete support for hyperspaces in Java has been implemented [Hyp05]. In our methodology, we also have to instantiate the pattern formalism for a specific metamodel by constructing role types for the element kinds in that metamodel. The advantage of our approach, however, is that new role types can be constructed using the pattern concept itself [Hau05] [P5].

Another instantiation of the Hyperspace approach has been presented in Theme [BC04, Cla02]. Theme is based on ideas of subject-oriented design [Cla99] combined recently with the support for requirements capturing and management. Subject-oriented design is a methodology for finding and managing the design subjects (concerns) of a system. A theme is any feature, concern, or requirement of interest that must be handled in the system; the concept of theme corresponds to our concept of aspectual pattern. Themes are composed with base models (which are themselves expressed as themes) using so-called composition patterns. Compared to aspectual patterns, the rules of binding a theme to a base model are not made explicit. For example, the number of concrete elements that a theme element should be bound to is not explicitly specified. The MADE environment achieves many of the goals set for Theme, namely binding themes to base models and composing together individual themes [P8].

Aspectual patterns, presented in this dissertation, correspond to other decomposition mechanisms such as subjects [Cla99], contracts [Hol92], and viewpoints [NKF94]. Subjects are class hierarchies representing a particular viewpoint of a domain model. Contracts operate on object-oriented artifacts and are used to represent objects and their interactions separating them from other interactions involving the same objects. Viewpoints, in turn, are used to represent developers’ views at the requirements level. Different viewpoints can be described using different notations. Compared to these concepts, aspectual patterns are
not bound to a specific artifact type and can be used to represent any type of structural configuration including text document, for example.

The Concern Manipulation Environment (CME) [CME05] share similar ideas and goals as those of the MADE environment. The idea of CME is to offer end users an open suite of tools for use in creating, manipulating, and evolving aspect-oriented software, across the full software lifecycle. The environment helps in interoperating and integrating different AOSD development tools and paradigms, and comes with an initial set of components. For instance, the Concern Manager (ConMan) tool models software systems in terms of arbitrary concerns and their interrelationships. A strong similarity between MADE and ConMan is the support for a wide variety of concern structures and software decompositions across heterogeneous artifact types.

5.2 MANAGING HETEROGENEOUS SOFTWARE ARTIFACTS

In practice, the ideas of multi-dimensional separation of concerns can be regarded as extending the representation of concerns to cover heterogeneous artifact types. Similarly to our approach, a single concern might have various representations at different model levels, usually expressed using different notations. The traditional approach to manage heterogeneous software artifacts is to provide various mappings from one artifact type to another, often supported by automated transformations [CE00]. In the forward engineering direction, high-level UML models can be transformed into lower level ones (e.g., [MDA05]), UML design models can be transformed into source code or into XML descriptions, etc. In the reverse engineering direction [CC90], information extracted from source code can be transformed to high-level representations that are often graph-based. Occasionally the high-level representation is expressed using the same notation used during system design, for example in UML-based reverse engineering approaches [RSSX04]. Transformations have traditionally been applied as one-shot activities used to automatically or semi-automatically generate new artifact representations based on other input representations.

From the viewpoint of developers, such transformational activities are considered insufficient for managing the underlying multi-dimensional structure of artifacts as they efface the relationships exploited in the transformation. In addition, they can lead to serious problems when maintaining the automatically produced artifacts. In contrast, aspectual patterns can be thought of as a mechanism for incremental transformation, where a transformation step involves generating the possible default elements for unbound roles based on their specification. Transformation information is furthermore stored as role bindings.

Another area where the multi-dimensional nature of software systems has been traditionally taken into account is configuration management [Bur99]. Typically, not only source code but also other artifact types, corresponding to other software development phases, are stored into configuration management systems (CMS). The key functionality of CMS is to identify software-related artifacts and manage their evolution. The main target of these systems is to provide common source of information for development teams, at the same time enabling isolated development environments for each team, mitigating the problems caused by concurrent development.

Automated version management can be considered as a key enabler for CMS’s, the managed configurations are snapshots of a version tree at some point in time. Change man-
agement can be based on version numbers indicating that something has changed, or on textual comparisons. A limitation of CMS’s is that they are not aware of the structure or semantics of the artifacts. For instance, a CMS cannot decide, in general, if changes in the feature models imply changes in the design model or not. In the pattern-based approach, the structure of the artifacts is known by the tool, so that fine-grained relationships can be maintained between different artifacts.

The ability to track relationships between software artifacts has been a central aim in requirements engineering [LW00]. Rational RequisitePro [Req05b] is an example requirement management tool. In RequisitePro, use cases are written as Word documents, which are stored into a relational database. Use case diagrams in Rational XDE [XDE05] can be associated with use case documents; sequence diagrams implementing the use cases can be linked to the use case diagrams, and class diagrams can be linked to the sequence diagrams [Rat05]. This kind of traceability is based mainly on explicitly created links relying on a disciplined software process. In the pattern-based approach there are no explicit links between the artifacts themselves, but instead we specify a particular concern as a pattern and bind the roles of the pattern to certain elements of the artifacts. The role bindings result from performing a task list that is generated by a pattern capturing a particular concern. In this regard, our approach could be regarded as a task-oriented tracing tool aiming to cover the dependencies implied by a well-understood concern.

For developers who play multiple roles in the development process, working with heterogeneous software artifacts is considered as a challenging task since they have to interact with multiple tools for managing the various artifacts. Generally, using one integrated development environment, as the MADE tool, instead of independent tools can minimize the overhead of learning and switching between tools, and enhances the inter-tool communication. A considerable amount of research work on tool integration has been conducted. For example, in [BCM+94] the authors define strategies and principles for integrating CASE tools. The aim is to help organizations select tools that would work together and to guide them how to effectively use integration technologies.

An approach to maintain the consistency of software artifacts has been discussed in the work of Reiss [Rei02]. The approach is tool, language, and notation independent offering developers the flexibility to choose the right environments and representations. It consists of using a constraint-based integration mechanism to manage the consistency of software artifacts during evolution. Our approach addresses many of the problems identified but not solved by the work of Reiss. Firstly, we have developed tools for visualizing and fixing the inconsistencies between the artifacts. Secondly, the MADE environment not only detects and fixes the inconsistencies between multiple software artifacts, but also can be used to generate these artifacts.

The most closely matching tool related to the MADE approach is Rational XDE [XDE05]. Similarly to our approach, XDE includes a pattern engine that can be used for model development. However, patterns are applied differently in the MADE environment than they are in XDE. In our approach, it is possible to apply a pattern in small increments whereas an XDE pattern is only applied in full. The reason is that the MADE tool treats each role binding as a separate task. After an XDE pattern is applied, the integrity of the pattern against the model is not automatically supervised, one has to revalidate the pattern bindings each time the model changes. In contrast to the MADE environment, XDE does not detect all violations (for example a deleted generalization relationship between two classes) when the
model is manually edited. Another significant difference is that aspectual patterns can have roles representing non-software entities like text files and user input values or even roles representing informal entities like reminders.

5.3 OTHER RELATED DEVELOPMENT APPROACHES

The following areas are related to certain parts of the dissertation. The field of software product-lines represents the scope of the case study. Model-driven development (MDD) is related to our approach in the sense that MDD promotes a methodology where different parts and aspects of a software system are specified using models across multiple abstraction levels. Generative programming addresses the problem of synthesis, which is a central concept in our methodology and tool environment.

5.3.1 Software Product-Lines

In this dissertation, aspectual patterns have been applied to manage variability in a software product-line. A product-line is a family of products sharing the same assets, variability management has been considered as a key issue in the field of software product-lines [Bos00, CN01, JRVdL00]. A product-line architecture explicitly captures the commonality and variability in the family of systems that constitutes the product line [Gom04], promoting the variation between different products, i.e., the use of variants and variation points [JGJ97].

We have provided two-way support for managing variability: using specialization patterns and feature variation patterns. Specialization patterns are used to document the specialization interface of object-oriented frameworks [FSJ99], which is a popular way to implement product-line architectures. Feature variation patterns are used to manage variability expressed as feature models [CE00]. Other approaches to variability management include the use of architecture description languages (ADLs) [BCK97, VdH00] and XML-based program specification [Cle01]. Software factories [GSCK04], in turn, represent another approach to automated development of product-lines. The approach is based on the use of highly tuned domain-specific languages (DSLs) [VDKV00] and XML as source artifacts, to capture variability among products. The approach then relies on model transformation, code generation, and other forms of automation for building individual products.

A wide range of tool support has been provided for software product-lines [GEA05, LNVV04, Req05a]. The main features of the GEARS tool [GEA05] include creating and maintaining product-line infrastructures, assembling of products, facilitated configuration management, and support for interoperability. In [LNVV04], the authors present an approach to achieve traceability in software product families. A concrete tool support for traceability of requirements to architecture, components, and source code is presented. The tool, which is an extension to the Together ControlCenter environment [Tog05], supports traceability for feature models down to source code and back. The RequiLine tool [Req05a], in turn, comes with a number of important capabilities such as modeling of features and requirements, product configuration, product consistency checking, and querying and viewing of features. Compared to these tools, the MADE environment falls short in its support to several of the features desired for managing the development of software product-lines,
such as the explicit distinction between the product family, product, and component implementation layers. However, the ideas of the MADE tool are applicable to a wider context in the sense that the tool has been used to manage variability issues regarding different phases of the development process such as maintenance and comprehension.

5.3.2 Model-Driven Development

Model-driven development (MDD) [MCF03] promotes an approach where models of the same system are usually derived from each other leading to better alignment between the models. The models are typically expressed using heterogeneous notations, thus managed using different artifact editors. The issue of separation of concerns has as well been discussed in the context of model-driven development. According to Kulkarni et al. [KR03], separation of concerns should be dealt with at both the model and code levels.

Model-driven architecture (MDA) [MDA05] is a recent initiative by OMG for supporting MDD principles. MDA defines three views of a system: a Computation Independent Model (CIM), which is a representation of a system from a business viewpoint, a Platform Independent Model (PIM), which is a representation of a system ignoring platform specific details, and Platform Specific Model (PSM), which is a model of a system that covers both platform independent information and details about a specific platform. In this dissertation, we have used feature variation patterns to document CIMs (features models) and PSMs (other model kinds).

In response to the needs of MDA, the QVT-Partners has released a revised proposal for the OMG’s QVT RFP (Request For Proposal) [QVT05]. QVT stands for Queries / Views / Transformations and represents a standardized transformation language to allow UML models to be transformed into usable software. The MADE approach solves some of the issues mentioned in the RFP. The aspectual pattern concept can be used as a technical infrastructure for managing the various transformations between models [SKS04].

The MADE environment can be considered as an MDD tool [P8], as it provides mechanisms for transforming input models into models of other kinds. A number of these tools along with a detailed comparative evaluation are presented in [Bel05]. Some of these tools have not been originally built for MDA but were later tuned to support its principles. Similarly, our pattern concept and pattern engine have initially been implemented for specializing Java-based frameworks but was later extended to cover other purposes and notations. Most tools (if not all) do not implement all the features of MDD (or MDA). Instead, each tool, similarly to the MADE environment, considers a restricted set of features.

5.3.3 Generative Programming

Aspectual patterns are exploited in the MADE environment for the synthesis of software models based on well-defined specifications (captured in the patterns). Synthesis represents a core concept of generative programming (also referred to as generative software development) [CE00], where the goal is to develop programs that synthesize other programs. In the context of software product-lines, generative programming can be thought of modeling and implementing system families where a given system can be automatically generated based on a specification. The specification can be written in one or more textual or graphi-
cal domain-specific languages [CHE04]. In generative programming, the focus has mainly been on automatic code generation.

The principles of generative programming have been realized in practice to build useful environments for automatic code generation. For example, Harrington [Her03] presents various types of code generators ranging from simple code processors to more complex generators for maintaining entire application tiers. Example areas include database access, user interface, and test cases. Cleaveland [Cle01], in turn, presents a step-by-step guide to creating "wizard-like" program generators with Java and XML. The author argues that such approach helps improving software development productivity. Some code generators have been built for specific environments, such as the generators defined for Microsoft .Net [Dol04]. Most generators, however, are based on the use of wizards, exhibiting several limitations related to comprehensibility, maintainability, and poor support for separation of concerns [P4]. These issues have been among the primary requirements considered during the development of the MADE environment.

Feature models play a key role in generative software development [CHE04]. Batory et al. take an approach to feature-oriented programming where models are treated as a series of layered refinements [BSR04]. Individual features are composed together in a step-wise refinement fashion to form complex models. Models can be programs or other non-code representations. In order to support their concepts, the authors have developed a number of tools for feature composition, called AHEAD toolset. The toolset provides similar functions to those of MADE. The MADE environment, however, solves two problems not otherwise addressed in [BSR04]: tracing features across different artifacts and checking the validity of models.
Chapter 6

Introduction to the Included Publications

This chapter presents a summary to the eight included publications in the dissertation. Then, the author’s contribution to the individual publications is explained.

6.1 SUMMARY OF THE PUBLICATIONS

The concept of an aspectual pattern has been introduced in [P1] to provide an approach to aspect-oriented pattern-driven model development. Aspectual patterns combines a role-based generic pattern concept with aspect orientation. The paper also argues that the realization of an aspect (concern) 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. In order to validate the approach, the use of aspectual patterns has been demonstrated with the case of the JUnit testing framework [Jun05]. The MADE environment support for aspectual patterns in UML has been presented.

The generic pattern mechanism, presented in [P1], has been used in [P2] to build an architecture-centric wizard that guides the user through application development. This is achieved by exploiting the pattern concept to document the specialization interface of a Java-based framework. The case study has been the Enterprise JavaBeans (EJB), which is Java’s component architecture for server-side distributed enterprise applications. As a first step, we have identified and collected proven EJB-specific design solutions as a set of design patterns. Next, we have used the EJB design patterns to build a tightly interconnected pattern system. We have shown that the pattern system can significantly ease the development process of EJB-based applications and improve the quality of the produced software. In [P2], an earlier version of the JavaFrames environment known as Fred has been used. The platform has been used to generate an EJB programming environment, when given the specifications of the EJB design patterns as input.
The results presented in [P2] have been extended in [P3] to cover other parts of the J2EE framework, illustrating the extensibility and adaptability qualities of the approach. In addition to the business logic and data access layers, which were discussed in [P2], additional support for the presentation tier has been added. The additions did not affect the EJB-specific parts. The pattern-based annotation of the presentation tier, as well as the other two tiers, has been built around proven design solutions, known as J2EE design patterns [ACM01]. It is shown how a general architectural tool (a standalone version of JavaFrames) can be used to model such a pattern system and to generate an architecture-centric environment for developing J2EE applications. The methodology has been demonstrated and validated with the help of an example web-based application. In the case of [P1] and [P2], the term specialization pattern has been used to indicate the purpose of the pattern mechanism, i.e. documenting the specialization interface of frameworks.

Another exploitation of specialization patterns has been presented in [P4]. The goal was to generate a pattern-based modeling environment for Symbian applications. For this reason, patterns have been presented in the context of UML. A major motivation for the work was that in case specialization patterns get bigger, it becomes harder to adapt the patterns into a particular design, due to increasing number of bindings between pattern roles and the elements of the design. To tackle such a problem, a concern-based approach (i.e. concern architecture views) has been utilized. The methodology was to group specialization patterns into concerns yet allowing user-controlled instantiation of individual patterns. Tool support for the methodology has been implemented in the MADE environment. An example usage scenario has been presented throughout the paper.

A second application of aspectual patterns has been to record information about maintenance tasks [P5], where the pattern mechanism is referred to as maintenance patterns. Such pattern kinds can be regarded as organized collections of software elements relevant for a particular maintenance concern. It is shown how maintenance patterns can be exploited in the JavaFrames environment to keep track and guide the system developer step by step through a maintenance procedure. For illustration purposes, the paper presents three maintenance situations: extending an existing system, copy-paste programming, and the fragile base class problem. The paper, further, discusses the way the three situations can be managed using the methodology. Finally, it is shown how the approach can be applied to the JavaFrames system itself to construct new role types, this has been considered as an example of adaptive maintenance.

Paper [P6] studies the problem of learning complex software libraries modeled in UML. The paper argues that the learning process can be supported with a tool environment that allows the customization of the UML model according to the context of the learner, step-wise and dynamically chosen learning tasks, and focusing on a particular learning concern at a time. It is shown how such an environment (i.e. using the MADE tool) can be achieved based on the concept of aspectual patterns (referred in the paper as comprehension patterns). The approach is demonstrated with a part of the Symbian platform architecture and evaluated in a case study where a pattern-driven learning environment is constructed for JPEG interchange file format specifications [T.892].

The final application of aspectual patterns has been to manage feature-driven variability in a software development process [P7]. It is argued that feature variation concerns can be presented as pattern-like entities - called feature variation patterns - cross-cutting heterogeneous artifacts. Feature variation patterns, typically, cover a wide range of artifact types
from a feature model to implementation. The idea is demonstrated, using the MADE tool environment, with an example taken from J2EE. The paper further studies the practical applicability of the approach in an industrial product-line, which is a Nokia GUI platform for building network management systems (the case study presented in Chapter 4).

In paper [P8], it is argued that a software system can be viewed as a combination of separate concerns covering various artifact types and cross-cutting the primary structure within each artifact type. This paper presents a concern-based approach to model-driven development. The aspectual pattern concept is used to represent concerns at different model levels. It is shown how the MADE tool can be used as a task-based model-driven development environment, providing facilities for model generation, checking, and tracing. The paper discusses pattern-based support for variability management, framework specialization, maintenance, and system comprehension in the context of model-driven development.

6.2 AUTHOR’S CONTRIBUTIONS

The research work of this dissertation was carried out at the Institute of Software Systems, Tampere University of Technology. The author has been a member of an active research group involved in studying architecture-centric software development and building tools for relevant tasks such as variability management, model development, and model conformance checking. The members of the research group have been in close collaboration under the guidance of the thesis supervisor Prof. Kai Koskimies and Professors Tommi Mikkonen, Tarja Systä, and Ilkka Haikala. However, the author’s contribution to all of the publications has been essential in that he developed the idea of aspectual pattern and its applications, participated in the development of the tool environment, performed the case studies, and co-prepared the manuscripts.

In [P1], the author has introduced the concept of aspectual patterns for representing concerns as role-based patterns, has suggested a new notation for describing patterns, and conducted the JUnit case study. In [P2] the author has reviewed a number of well-known Enterprise Java Beans patterns. In addition to the existing patterns, the author has suggested the use of several new patterns that can be used to solve specific interests in the EJB framework. The patterns are then put together forming a pattern system. The author has shown how each EJB pattern can be modeled as a JavaFrames pattern. In this context, JavaFrames patterns are considered as framework specialization patterns. In [P3], the author has extended the pattern system defined in [P2] with new patterns in order to build a pattern-based J2EE application development environment.

In [P4], the author has contributed to the integration of the JavaFrames environment in a toolset providing tool support for concern-based framework specialization in UML. In addition to framework specialization patterns, the author has discussed in [P5] how a JavaFrames pattern can be used to model maintenance patterns. Maintenance patterns are used to document maintenance tasks. The author has shown how maintenance patterns can be built by considering maintainability as a concern. The author has also prepared the illustrative examples and the case study.

Similarly, the author has suggested in [P6] the use of comprehension patterns for encapsulating the tasks required for learning a complex system. For building the pattern system, the author applied the methodology explained in [P4]. Furthermore, the author has planned and
co-organized the case study. In [P7], the author’s contributions include the use of heterogeneous patterns for managing feature-driven variability. The author prepared the illustrative example and contributed to the case study. Paper [P8] was exclusively contributed by the author.

Concerning the MADE tool environment, the author has contributed with the following major design and implementation tasks.

- Pattern support for general file and UML semantics.
- Support for explicit modeling and visualization of concerns.
- Design and implementation of pattern composition.
- XML-based specification of patterns.
- Enhancement of task wizards and task automation.

The first three tasks have been essential to the ideas presented in this dissertation, i.e. the support for heterogeneous artifact types and the explicit representation of concerns. The last two tasks have been implemented in order to improve the usability and the user-friendliness of the tool.
Chapter 7

Conclusions

First, we summarize the main results of the dissertation. Then, we discuss the limitations of the approach and recommend several research topics to be investigated in the future.

7.1 SUMMARY

This dissertation described a methodology for addressing essential difficulty in software development, i.e. complexity, conformity, changeability, and invisibility. We argued that multi-dimensional structure of software systems is a major cause of difficulty. Therefore, the difficulty issue could be managed by addressing the various dimensions of software systems, both at the methodological level and at the level of tool support. Recognizing this, the thesis is motivated by the lack of well-established formalisms and tool support for managing the multi-dimensionality issue, despite the latest advancements in software research and technology.

First, the dissertation presented an approach to structure a software system along two major dimensions: the concern dimension and the artifact dimension. These two structuring elements can themselves be structured into further sub-dimensions. Software concerns can be understood in terms of certain concern types, stakeholders, and development needs whereas artifacts can be associated with given artifact types, notations, and tools. To manage these dimensions, the dissertation suggested a pattern-based tool concept referred to as aspectual patterns. Aspectual patterns are generic artifact neutral mechanisms for capturing system concerns represented across heterogeneous artifact types.

The dissertation then discussed a general architecture for tool support exploiting aspectual patterns. It is argued that such tools conforming to the architecture should provide support for four kinds of tool functionalities: synthesis, checking, visualization, and tracing. The four functionalities represent a basis for addressing each of the difficulty sources. The discussed tool architecture exhibits several desirable characteristics including openness, extensibility, and artifact tool independence. Several other characteristics have been stressed to enhance the usefulness of the tool, including stepwise artifact management and support for adaptive development.
A concrete tool implementation, known as MADE (Modeling and Architecting Development Environment) was developed to demonstrate the pattern-based approach and the general tool architecture. The implementation consisted of extending an existing pattern-based specialization environment for Java frameworks called JavaFrames. In addition to the existing support for Java artifacts, the JavaFrames environment has been extended with support for UML, XML, and general text files. Such extensions allowed the management of other artifact types such as design model documents and deployment descriptor files. To support the editing of the used notations, we integrated Rational Rose and an Eclipse-based text editor with the pattern tool. Similarly, we have generalized the concept of specialization pattern introduced in JavaFrames into aspectual pattern to cover other concern types such as maintenance, comprehension, and feature variability.

For the validation of the tool implementation and the methodology, we applied the approach to manage various development scenarios in an industrial-level software platform, which is a Nokia product-line for building the GUI parts of network management applications. The discussed scenarios were mapped to various development phases and associated with different sources of difficulty. Consequently, every problem exhibited its own properties regarding the concern and artifact dimension. Depending on the addressed dimensions, a certain category of aspectual patterns have been used as a solution. Four categories of aspectual patterns have been discussed: specialization patterns, feature variability patterns, maintenance patterns, and comprehension patterns. The case study, in addition to the case studies reported in the included publications, has demonstrated the applicability of the approach and the usefulness of the tool environment. Nevertheless, in the next section we identify several limitations with the methodology and results.

7.2 LIMITATIONS

We summarize the main limitations of this work in the following discussion.

The issue of difficulty. In this dissertation, we have considered complexity, conformity, changeability, and invisibility as four major sources of essential difficulty in software development [Bro87]. We have discussed each difficulty kind from a rather narrow perspective. For example, conformity has been studied in the context of conforming to the specialization rules required for product development. This represents just one example instance of conformity, naturally other forms of conformity problems do exist. This observation is valid for the other sources of difficulty as well. Furthermore, the software community has suggested additional sources of essential difficulty such as the problem of interoperability [FS05], which we have not considered.

Structure of software systems. We have studied software systems based on two major structuring dimensions being the artifact and the concern dimensions. One could argue that there are other alternative dimensions, which could be considered. For instance it is possible to organize software entities into problem-domain concepts and solution-domain concepts, this can be referred to as the domain dimension.
**The pattern formalism.** Aspectual patterns as discussed in this dissertation represent system-specific configurations of software elements addressing a specific concern. Though we used informal descriptions, for example in the case study, to document the problem and the context in which the individual patterns can be applied, our pattern concept remains rather solution-oriented in the sense that the problem being solved is not documented enough, especially as far as the relationship to other patterns is concerned. Towards this direction, we have used concern architecture views to structure pattern relationships and relate them to the problem domain [HHP+05]. We still need to formalize the bridging between the concept of an aspectual pattern and that of a concern in concern architectures.

**The case study.** The annotation discussed in the case study represents a step in the process of identifying and managing variability issues in the presented product-line. The results we achieved have mainly been evaluated by the project members. For a more accurate evaluation, we still need feedback from our industrial partners, once the case studies have been completed. In some situations, we had to define usage scenarios based on our own understanding of the case study, rather than working on concrete problems. The main reason is that the case studies have been defined while the methodology is still being developed, thus it was rather early for our industrial partners to build a complete picture of the approach.

**The MADE tool environment.** Our experiences, which are based on the case study and the experiments reported in the included publications, raised a number of questions regarding the MADE tool environment mainly related to scalability, extensibility, and user-friendliness. Scalability became an issue as the pattern library grew very large. Flexibility was mainly related to the lack of support for other notations and artifact tools. Finally, user-friendliness limitations were mostly related to the effort needed for constructing the patterns, the amount of user interaction needed to instantiate the patterns, and the kind of interaction with the artifact tools.

### 7.3 FUTURE WORK

There are many directions in which the research results in this dissertation can be extended and improved. In the following, we present the main future research topics.

**Support for other development scenarios.** We have used aspectual patterns and the MADE environment to address software engineering activities such as feature variability management, framework specialization, maintenance, and comprehension. We think that the approach can be used to address other areas. Using the MADE environment, we could, for example, assist developers in generating test code and test cases based on well-defined testing strategies. Another topic we are currently researching is the use of our methodology to achieve a quality attribute-driven design environment.

**Pattern categories.** Considering the four categories of aspectual patterns, we have applied specialization patterns mainly to white box specialization, but we have little experience with black box specialization. Similarly, maintenance patterns have been used to manage adaptive maintenance, but not other forms of maintenance (i.e. corrective, perfective, and
CONCLUSIONS

preventive). Comprehension patterns have been considered in learning static structure of system models. Comprehensibility, however, can be enhanced by studying the behavioral aspects of systems.

Support for other notations. Addressing other development scenarios and needs requires the support for other artifact types and notations. For example, in order to be able to comprehend the behavioral aspects of software systems, we are planning to add support for UML sequence diagrams. In this direction, we are working at transforming the tool into a framework for constructing new role types.

Managing heterogeneous artifacts. The MADE environment can be used to support heterogeneous artifacts, which are processed in their own tools. Usually, the artifacts are related with each other, this is for example the case of feature variation patterns. Currently, MADE environment offers only limited support for expressing the inter-artifact relationships. A future direction is to enhance the alignment and traceability between the various artifact types.

Relations between patterns. In our methodology, the relationships between patterns are represented using compositions rules. Typically, the composed patterns are of the same category (e.g. specialization patterns). We have not studied the relationships between different categories. For example, how comprehension patterns would relate to maintenance or specialization patterns, especially that aspectual patterns are used to study and document the variability issues of the same system, as the case of the product-line presented in this work.

Need for metrics. A convenient way of demonstrating the applicability and the usefulness of the MADE environment is to compare the achieved results against those obtained using other approaches. This, however, requires the definition of clear metrics that could be used to systematically assess the quality of the tool environment. The metrics could, for example, include the use of statistical data regarding code generation ratios, development time statistics or the cost required for maintaining the pattern-based development environment (modifications made to the pattern library).

Industry application. The industrial partners involved in this research work are expected to evaluate the methodology and the tool environment. In case the techniques prove to be useful, the MADE environment will be used in the partner companies. In order to reach this goal, we need to improve certain aspects of the environment, mainly related to user-friendliness and performance.
References


REFERENCES


REFERENCES


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


REFERENCES


REFERENCES


Publications
Publication 1


Copyright © 2004 IEE.
Reprinted, with permission, from the proceedings of WoDiSEE 2004.
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
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.

<table>
<thead>
<tr>
<th>Constraint Role type</th>
<th>Stereotype</th>
<th>Abstract</th>
<th>Visibility</th>
<th>Inheritance</th>
<th>Multiplicity</th>
<th>Aggregation</th>
<th>Return Type</th>
<th>Parameter</th>
<th>Overriding</th>
<th>Type</th>
<th>Navigability</th>
<th>Participant</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML Package</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Class</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Operation</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Attribute</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Association</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Association End</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Realization</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML Dependency</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1. Pattern roles and constraints

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-
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.

4 Applying Aspectual Patterns

4.1 Developing JUnit Design Model

As 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
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 CollectingParameter 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.
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 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

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
Publication 2


Copyright © 2002 IEEE.
Reprinted, with permission, from the proceedings of COMPSAC 2002.
Generating a Pattern-Based Application Development Environment for Enterprise JavaBeans

Imed Hammouda and Kai Koskimies
Institute of Software Systems, Tampere University of Technology
P.O. Box 553, FIN- 33101 Tampere, Finland
{imed, kk}@cs.tut.fi

Abstract
Enterprise JavaBeans (EJB) is Java’s component architecture for server-side distributed enterprise applications. The architecture of EJB applications is based on well-established solutions common to most distributed business systems. To utilize the architecture in an optimal way, proven EJB specific design solutions have been identified and collected as a set of design patterns. The use of these EJB design patterns as a tightly interconnected pattern system can significantly ease the development process of EJB based applications and improve the quality of the produced software. We will study in this paper how a general architectural tool (Fred) can be used to generate an EJB programming environment, when given the specifications of the EJB design patterns as input. This environment can be viewed as an architecture-centric wizard that guides the user through the development of the application, following the EJB design patterns.

1. Introduction

Enterprise JavaBeans (EJB), part of the Java 2 Enterprise Edition Platform, is an architecture for setting up program components that run in the server parts of a client-server system. This architecture allows developers to quickly create and maintain scalable enterprise applications at lower cost and higher usability. Developers need to concentrate on writing the business logic rather than dealing with general distributed systems problems such as persistency, transaction management, security etc. EJB architectural standard is currently supported by many products from different vendors, implementing a software platform for distributed business applications. An important advantage of EJB is that applications become independent of the environment and can be easily transported from one environment to another, assuming that both support EJB.

Enterprise JavaBeans development is still an evolving art. During the last few years, EJB has gained a lot of popularity in industry, but there is still relatively little experience in writing high-quality EJB applications among the majority of EJB developers. Hence it is extremely important that proven, good design solutions are systematically collected and made available to EJB developers. These solutions are known as EJB design patterns. Recently, a collection of EJB patterns has been published in a text book [2] and widely distributed in the industry, following the traditions of [1].

While EJB patterns are being recognized and published, several commercial and non-commercial software products with EJB tool support have been released and adopted by the industry. These systems offer a development environment for EJB applications, providing assistance in generating code, testing the applications and deploying them within the environment. Some environments even allow the user to select certain EJB patterns and generate code according to those patterns [7]. The idea of having programming environments dedicated to a certain category of applications is of course not limited to EJB. Such environments are called Application Development Environments (ADE).

However, a general problem with the conventional ADEs is that the tools hide the architectural aspects and the design decisions of the automatically produced code. Hence, the programmer has little hope to be able to understand the generated source code, and to modify it when necessary. Maintenance becomes problematic if the generation steps cannot be repeated, for instance after adding hand-written code. This is a common problem of tools that generate large portions of source code without user intervention.

Another significant limitation of the ADEs is that the tools assume a particular architecture and code generation patterns. The user can affect the produced code by changing some parameters, but the overall structure of the code is predetermined. This implies that it is hard to exploit the tools for systems that do not comply with the architectural assumptions of the ADE. It may also be hard to integrate the generated code with legacy code. In the case of an EJB-based ADE, the environment should understand and support EJB patterns, but the problem is that the set of EJB patterns evolves and grows all the time, and there is often need for domain-specific or even company-specific patterns. Hence no predefined set of
patterns can be sufficient for all purposes and for a long time.

These observations suggest that an EJB ADE should adapt itself to arbitrary architectural or design patterns, given by the user of the ADE. The pattern specifications should serve as an engine that runs a generic ADE, guiding the programmer through the architecture and making sure that the given patterns are followed. We also propose that an EJB ADE should be open in the sense that it does not perform mysterious generation actions behind the scenes, but rather offers an intelligent architecture-sensitive editor. If the ADE generates code, it should do it only in a limited context where the programmer can easily understand the meaning of the code. For example, the code generated for an individual attribute or method of an existing class can be usually understood by the programmer fairly easily, whereas generating a whole set of classes leaves the programmer hopelessly without track of the process.

There have been several attempts to formalize the concept of a (design) pattern and to develop tool support based on such formalizations (for a recent work and summary of existing approaches, see e.g. [6], [4]). These approaches are often motivated by the need to define the extension interface of application frameworks, so that the application-specific code can be generated according to the requirements of the framework. Such an interface specification can be defined as a set of patterns. In this context, a pattern is viewed as a set of roles and constraints. Each role can be bound to a program element (say, class), and the constraints define requirements that must be satisfied by the elements bound to certain roles. A framework extension point is defined by a pattern in which some of the roles are bound to program elements in the framework and some roles are left unbound. The latter roles will be bound to application-specific program elements, following the constraints.

In this paper we demonstrate that it is possible to solve the above mentioned problems of conventional ADEs for EJB by applying a general pattern-based approach for architectural modeling and framework tool support. We will utilize a prototype tool called Fred ("Framework Editor", [3], [4]) providing pattern-based assistance for framework specialization. We show that the specialization pattern concept of Fred fits the EJB patterns very well. A covering set of EJB patterns has been specified as Fred patterns, and an ADE for EJB has been produced by Fred based on those patterns. The resulting ADE has been used to develop a small example EJB application to demonstrate the capabilities of the pattern-based environment.

We argue that the Fred-based EJB environment in particular solves the basic problems with ADEs discussed above: hidden code generation cycle and limited extensibility. The Fred-based environment allows the programmer to select an EJB pattern, and apply it in the system through a sequence of small tasks. A pattern specification can be augmented with instructions to carry out each task. The programmer understands the purpose of the pattern and the roles of its parts, seeing its construction step by step. The environment is not bound to a fixed set of architectural patterns, but new patterns can be easily introduced to the system at any time. The pattern descriptions can be generated automatically as XML files, to be further processed for documentation purposes.

In addition, the Fred-based approach offers extra benefits that are not found in conventional EJB environments:

1. **EJB sensitive source code editor.** The integrated Java editor keeps track of the application of EJB patterns and checks the constraints of the patterns during editing on the fly: after each editing action the constraints are re-checked for the changed parts of the source, and possible conflicts are notified as new mandatory tasks for the user. When the user corrects the source text, these tasks automatically disappear.

2. **Application-oriented instructions for EJB patterns.** The task lists and the associated instructions are not static but generated dynamically taking into account the decisions and application-specific names given so far. Hence the instructions are not on the abstract level of general EJB patterns, but on the concrete level of a particular application. This makes the instructions much easier to understand and follow.

All these benefits are direct consequences of using Fred. To produce this kind of EJB ADE with Fred, one only needs to specify the required EJB patterns using Fred's pattern tool, and give these specifications as input for Fred. The EJB patterns specialize the generic Fred environment into an EJB environment.

The remaining of the paper is organized as follows. In the next section we discuss the main features of the EJB patterns used in this work, and how they are organized as a small pattern system. An overview of the Fred environment is presented in Section 3. In Section 4, we show how a representative EJB pattern is specified in Fred. In Section 5 we use a case study to demonstrate our approach. Section 6 compares the work to other existing solutions. Finally, in Section 7 conclusions are drawn and possible future work is highlighted.

2. An EJB Pattern System

In the past few years, the EJB community has come up with a multitude of ideas on how to optimize the architecture and deployment of distributed business applications. Since then, the term “EJB Design Pattern” has become a hot topic in the on-going discussions [2]. Some of the patterns are built on other patterns described in well-known literature such as [1] whereas others are unique to the EJB technology. Some EJB patterns deal with certain category of applications while others can be
considered in the design of most distributed enterprise systems.

In this section, we will concentrate on patterns that might improve the architecture of the business logic tier, paying less attention to those of the presentation and data tier. In addition, we will discuss few integration tier patterns. Rather than going into details, we will just give a brief overview of each pattern. More details about the patterns can be found in [2]. Nevertheless, we will choose to closely study the Session Façade pattern to show how it is translated into a Fred specialization pattern in Section 4. Finally, we discuss the relationships of these patterns and present them as an integrated pattern system that can be used by an EJB developer to cover central parts of the design.

Session Façade
The communication between the presentation layer and business layer in distributed business applications often leads to tight coupling between clients and the business tier. The interaction could get so complex that maintaining the system becomes difficult. The solution to this problem is to provide a simpler interface that reduces the number of business objects exposed to the client over the network and encapsulates the complexity of this interaction. At run-time, the client calls a method on a Session Façade, which in turn calls several methods on individual business objects. Figure 1 shows the UML class diagram representing the Session Façade pattern.

![Diagram of the Session Facade pattern](image)

**Figure 1. Structure of the Session Facade pattern**

The primary benefit of Session Façade is to provide a centralized control over the business tier and ease of understanding and the maintainability of the system. In addition, the façade represents an access control layer to manage the relationships between user requests and business methods, and a transactional control layer where a transaction starts by calling a number of methods on the individual entities and commits by returning to the client. This pattern is based on the Façade pattern in [1].

Value Object
In J2EE applications, the client needs to exchange data with the business tier. For instance, the business components, implemented by session beans and entity beans, often need to return data to the client by invoking multiple get methods. Every method invocation is a remote call and is associated with network overhead. So the increase of these methods can significantly degrade application performance. The solution to this problem is to use a Value Object to encapsulate the business data transferred between the client and the business components. Instead of invoking multiple getters and setters for every field, a single method call is used to send and retrieve the needed data.

Business Delegate
By using the Session Façade pattern, we did not rule out all the design problems involving the interaction between the client and the business layer. We do have a centralized access to the business logic but still the session bean itself is exposed to the client. Enterprise beans are reusable components and should be easily deployable in different environments. Changes in the business services API should not affect in principle the implementation of the beans. To achieve loose coupling between clients at the presentation tier and the services implemented in the enterprise beans, Business Delegate Pattern is used. This hides the complexities of the services and acts as a simpler uniform interface to the business methods.

Service Locator
In J2EE applications, clients need to locate and interact with the business components consisting of session and entity beans. The lookup and the creation of a bean is a resource intensive operation. In order to reduce the overhead associated with establishing the communication between clients and enterprise beans (clients can be other enterprise beans), the Service Locator Pattern is used. This pattern abstracts the complexity of the lookups and act as a uniform lookup to all the clients.

Data Access Object
Enterprise Java Beans are reusable components and should be deployable in different environments with lesser effort. Implementing a so-called bean-managed persistence entity bean means that the programmer should provide all the persistent code (JDBC code). However, the API to different databases is not always identical so the bean programmer should consider different persistence code for different data sources. Depending on the data source, one specific implementation is used. To make the enterprise components transparent to the actual persistent store, the Data Access Object pattern should be used.

In addition to the standard EJB design patterns above, we have defined several custom patterns that have been translated to Fred specialization patterns.

Primary Key Pattern
Entity beans need to be occasionally stored in the database and loaded to memory to guarantee data
A number of Session Façades are used to encapsulate entity beans. In order to lookup and use bean instances and home objects, the Session Façade gets the service from the Service Locator pattern. Other clients as well, including session and entity beans, locate other beans using the Service Locator. Individual beans could be tested using the Tester pattern.

3. Fred Environment

A basic concept for defining the architectural units in Fred is a specialization pattern. In Fred this concept is typically used for an abstract structural description of an extension point of a framework. Basically, 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 (or bound to) structural elements of a program, such as classes or methods. The same role can be played by a varying number of program elements. This is indicated by the multiplicity of the role; it defines the minimum and maximum number of bindings that may be created for the role. Combinations are from one to one (denoted (1,1)), from zero to one (0,1), from one to infinity (1,n), and from zero to infinity (0,n). A single program element can participate in multiple patterns.

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

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

The central part of the user interface of the Fred environment shows the current bindings of the roles for a selected pattern, structured according to the containment relationship of the roles. Since this relationship corresponds to the containment relationship of the program elements playing the roles, the given view looks very much like a conventional structural tree-view of a program. In this view, a red spot marks undone mandatory tasks, optional tasks are marked with a white spot. The actual to-do tasks are shown with respect to this view: for each bound role selected from the view, a separate task pane shows the tasks for binding the child roles, according to the containment relationship of the roles. The user interface of Fred is shown in Figure 3.

Figure 3. Fred interface

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

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

4. Presenting EJB Design Patterns in Fred

Each EJB pattern discussed earlier needs to be represented as a Fred specialization pattern. A typical Fred pattern is composed of several roles; each role corresponds to one program element (class, method, field...). Few other role types are used to represent certain relationships such as inheritance or alternative use between two roles. Roles may have properties like dependencies on other roles, multiplicity, constraints, and templates.

We will discuss here the Fred specification of the Session Façade Pattern in more detail. Figure 4 presents the roles and properties of the Session Façade Pattern.

Figure 4. Structure of Session Façade pattern

Figure 4 shows the roles, their multiplicities (in the upper right corner of role names), and the constraints between the roles. Containment constraints are expressed simply by nesting, and other constraints indicating dependencies between roles are denoted by arrows. For readability we have omitted some roles and other details. Roles Remote, Bean, and Home should be bound to the remote interface, bean class and home interface of the Session Façade, respectively. Roles EntityHome, EntityRemote, and EntityPrimaryKey should be bound to home interface, remote interface, and the primary key class (instance of the Primary Key Pattern) of the encapsulated entity bean, respectively. The field bound to entity role holds a remote reference of the encapsulated entity bean. There can be any number of actual fields bound to entity since one Session Façade can encapsulate several entity beans. We can also see that there is a dependency between resetEntities method role and...
connectToEntities method role. The reason is that resetEntities reconnects the Session Facade to its entity beans by calling connectToEntities.

In Section 2 we showed how the patterns constitute an integrated pattern system. This is reflected in pattern specifications as well. For example, the method bound to Session Façade’s getHome role uses the ServiceLocator class role, which belongs to the Service Locator Pattern.

In order to express the pattern structure as a Fred specialization pattern we need to use Fred’s pattern editor tool. The pattern editor is opened in Figure 5 for the annotated Session Façade Pattern. The Home role is selected for detailed editing, showing the role editor in the right-hand side pane.

![Figure 5. Fred pattern editor](image)

Instead of going through the pattern specification as it is given using the pattern editor, we will give below a textual specification of the pattern, with some clarifying comments. This representation follows roughly the structure of the XML description generated automatically by the tool, but we have transformed it to a more readable form. This description gives an idea of the kind of information that has to be given as input for Fred. For simplicity, Table 1 shows the textual representation of a small part of the specialization pattern illustrated in Figure 5. Bound roles are common to every pattern instance and should appear in the framework with the specified name. Unbound roles are specific to every pattern instance; this allows the pattern to be used in different contexts. For instance, the unbound role Home, home interface of the session bean, extends the bound role EJBHome, a built-in Java interface that every EJB home interface should extend. Every role has a set of properties; the defaultModifiers property of the getEntity role for example, has the value “private” to say that the method should be private. The definitions of properties may refer to other roles; such references are of the form <#r>, where r is the identification of a role. This is used for producing adaptable textual specialization instructions. For example, the description of getHome method role contains a reference to EntityRemote class role. In constraints, references to other roles imply relationships that must be satisfied by the program elements playing the roles. For example, the type of the field playing the role entity should be the class playing the role EntityRemote. The multiplicity symbol “+” that comes with EntityHome means that there can be more than one program element playing the role EntityHome.

<table>
<thead>
<tr>
<th>SessionFacade</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bound roles</td>
<td></td>
</tr>
<tr>
<td>EJBHome : class</td>
<td>description Java’s built-in EJBHome interface</td>
</tr>
<tr>
<td>SessionBean : class</td>
<td>description Java’s built-in SessionBean interface</td>
</tr>
<tr>
<td>EJBObject : class</td>
<td>description Java’s built-in EJBObject interface</td>
</tr>
<tr>
<td>Unbound roles</td>
<td>Properties</td>
</tr>
<tr>
<td>Remote : class</td>
<td>operation : method</td>
</tr>
<tr>
<td>type</td>
<td>java(&quot;javax.ejb.RemoteException&quot;)</td>
</tr>
<tr>
<td>EntityRemote: class</td>
<td>description Remote interface of the encapsulated bean</td>
</tr>
<tr>
<td>EntityHome + : class</td>
<td>description Home interface of the encapsulated bean</td>
</tr>
<tr>
<td>Bean : class</td>
<td>getHome: method</td>
</tr>
<tr>
<td>type</td>
<td>EntityRemote</td>
</tr>
<tr>
<td>entity : field</td>
<td>defaultInitializer null</td>
</tr>
<tr>
<td>description</td>
<td>A field that holds the remote reference for the &lt;EntityRemote&gt; entity bean.</td>
</tr>
<tr>
<td>taskTitle</td>
<td>Provide</td>
</tr>
<tr>
<td>defaultImplementation</td>
<td>&lt;#EntityRemote&gt; home = &lt;#Bean.getHome&gt;(); return (&lt;#EntityRemote&gt;) home.findByPrimaryKey(pk);</td>
</tr>
<tr>
<td>defaultModifiers</td>
<td>private</td>
</tr>
<tr>
<td>defualtName</td>
<td>get&lt;%Bean.entity&gt;Entiti</td>
</tr>
<tr>
<td>description</td>
<td>Private method to get &lt;EntityRemote&gt; entity.</td>
</tr>
</tbody>
</table>

Table 1. Textual representation of part of the Session Façade pattern

The EJB design patterns discussed in Section 2 are typically deployed in certain combinations. For example, an EJB application often requires several Session Façades. Each façade encapsulates a set of entity beans and for each façade there is a corresponding Business Delegate. To handle this kind of composition of the EJB design patterns, we have applied the composite pattern approach used in [5]. Composite patterns are combinations of other patterns. Such highly adaptable (at deployment time) components are important in situations where a similar functionality is needed in several places in an application, with slight variations. Currently our environment
distinguishes between the following substructures that serve as either simple or composite patterns:

- Session Facade that encapsulates a set of entity beans.
- Business Delegate that forwards client requests to Session Facades.
- Bean managed entity bean that has a primary key class and a Value Object, and uses Data Access Objects to implement the JDBC code.
- Bean managed entity bean implemented as a session bean and a Data Access Object.
- Container managed entity bean that has a primary key class and a Value Object.
- Session bean (stateful and stateless).
- Tester client.

A typical scenario is to start creating a set of strongly coupled entity beans that can take any form discussed above, encapsulate them with a façade, and then map a Business Delegate to that Session Facade. If needed, the user could generate a number of session beans by specializing the right component in different ways. In order to encapsulate a different set of entity beans, the composite pattern responsible for the “delegate-façade-entity beans” substructure should be considered once again. The user, with the help of the environment, can integrate custom business logic code with the design patterns. She can define any number of business methods for entity and session beans. The environment ensures that every method defined in the remote and home interface is implemented in the bean implementation class. If the user chooses to implement methods that access data from an Enterprise Information System, the required tasks for using the Common Client Interface (CCI) are generated.

5. A Case Study

We applied the environment produced by Fred to implement the business logic of a simple to-do list application. The application accesses a list of users (administrators and normal users) and associated tasks stored in a relational database. Operations on the lists include adding, updating, deleting, and searching database entries. We created a bean-managed persistence entity bean to encapsulate the table of users and a container-managed persistence entity bean to represent the table of all tasks. A session bean is then used as a façade to both entity beans. The application is implemented according to the architecture proposed by the design patterns of the environment. Since the specifications of the design patterns are associated with various default implementations, a substantial amount of code will be automatically generated during the development process. However, this code is produced in small, understandable pieces to fulfill certain limited tasks.

Figure 6 shows a sample view of the environment at the specialization phase. The Architecture View to the left gives an idea about the overall application architecture. The Session Facade composite pattern Manager for example encapsulates two entity beans. Users composite pattern represents a bean-managed persistent entity bean whereas Tasks composite pattern encapsulates a container-managed entity bean. Users has a business logic part and a business-data integration part. In the left pane of the Task View we can see the list of the tasks that have been carried out by the user. In the other half, the environment suggests several other tasks, finder methods in this case. Besides, an optional task to define other bean persistent fields is shown.

Figure 6. Sample EJB application development session

Moreover, adaptive documentation helps in assisting the user with explanations on what tasks should be provided and which missing program elements should be added. This is illustrated in the right bottom pane of the task view in Figure 6: the spécializer is asked to provide the findByPrimaryKey method. The documentation explains the user the purpose of the method, its parameter list and return value. It then asks the user to provide the SQL code of the method in the implementation class of the bean’s Data Access Object. Note the use of application-specific names (like Users). The integrated Java editor can be used to add custom business logic in addition to viewing and maintaining the code that has been generated. Recall that Fred keeps track of any broken dependencies or inconsistencies between the code and the given architecture.

The environment acts as a tool for compile time checking of an EJB application. For instance, the system checks if for every method declared in the remote interface of a bean, there is a corresponding method definition in the bean implementation class. Hence in a sense the environment acts itself as a Business Interface Pattern, a common practice used to avoid inconsistencies between remote interfaces and bean implementation. This
-pattern assumes an interface that the remote interface extends and the bean class implements. This design pattern is very specific to EJB.

The Value Object specialization pattern can be considered as a factory for Value Objects. Complex applications need to use several custom Value Objects for the same entity bean. For simplicity, the environment we propose uses default the domain Value Object, a Value Object that has all the persistent fields of the bean.

If the user decides to switch from container-managed persistence to bean-managed persistence for example, the right actions and modifications are generated and displayed to the user in the form of tasks. In this respect the environment supports also an evolutionary mode of work. Another important feature is flexibility; for example, the user is free to choose whether to put the JDBC code in a separate Data Access Object or in the bean implementation class itself. As the business tier gets larger, the proposed architecture makes it easier to understand, control and maintain the system.

Deployment descriptors are text files describing various properties of the beans in XML format, required for the deployment of an EJB application. Generating deployment descriptors can be supported by Fred; the user selects the pattern instance that represents the bean and applies an XSLT transformation using an XSL file that comes together with the environment. Fred parses the pattern instance, extracts the data needed for the bean deployment such as the name of the bean’s remote interface, and puts it between the opening and closing tags of the corresponding fields in an XML file.

The environment has been tested against several other similar, simple applications. The experiences showed that the environment can improve the quality of the software and reduce the development effort. The improved quality is a consequence of making desirable EJB solutions available for the application developer and guiding her to use these solutions. For example, using Data Access Objects makes the persistence code of entity beans independent from the used data source, increasing the flexibility of the system design. Using Value Objects reduces the network overhead when accessing persistent fields, optimizing the performance of the application. Much of the work behind coding Enterprise Java Beans was reduced to mouse clicking. Although it is hard to define specific metrics to compare the development process with more conventional ones, it seems obvious that this approach improves the quality of the software and reduces the development effort considerably. The benefits apply both for experienced EJB programmers and novices: for the former, the environment releases the developer from the burden of writing a lot of straightforward but complicated code and remembering all the details of the EJB conventions; for the latter, the environment helps the developer to learn the EJB patterns and successfully produce an application even with limited understanding of EJB. In the next section we discuss the advantages and limitations of the Fred EJB environment in comparison with some other tools.

6. Related Work

Several commercial and non-commercial tools providing support for the EJB technology is being used by the industry. Some tools require users to provide a specific representation of the enterprise beans such as XML files. An example of such products is the realMethods tool [7]. The tool is capable of generating Java code, SQL code and deployment descriptors on the basis of the input XML file that represents the object model. The user can select a set of design patterns, which are used to build the application infrastructure. The advantage of this technique is that once you have your bean specifications ready, generating the equivalent Java code becomes a simple and fast operation.

Other widely used application development environments offers a more standard way of support for EJB technology. An example of such environments is JBuilder [8]. The tool is for example capable of generating code for enterprise beans in a visual mode. It can also construct entity beans out of data models and create relationships between entity beans using drag-and-drop. Changes can be made in both the generated beans code and in the visual tool, keeping the two versions in synchronization. A big advantage of such a tool is the built-in application server where the generated beans could be deployed and tested.

However, the two environments fall short in many features available in the Fred EJB development environment.

- Almost all tools supporting the EJB technology focus on code generation but pay less attention to the design and architecture of the application. Design patterns (as well as other best practices) are used in isolation. Fred environment pays special attention to the architectural aspect of reuse. Fred EJB framework is a collection of collaborating patterns and several other programming rules that could be easily extended and updated.
- Tools that require prior representation of the business logic and data such as the realMethods tool comes at the cost of spending time and effort in preparing correctly the input specifications. Any error in the specifications could lead to serious problems. The maintainability of the system can become as hard as implementing the whole application from scratch. Fred environment gently guides the user in a step-by-step basis to do the job.
- The JBuilder environment does not provide much help to users on what tasks should be done next. The documentation that comes with the environment is rather static and same for all applications. Fred provides adaptive help, documentation are specific to every framework specialization instance.

---

IEEE Computer Society
Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC’02) 0730-3157/02 $17.00 © 2002 IEEE
• Using Fred, the user could build a better understanding of the problem domain. The pattern editor and the architecture view of the framework make it possible to obtain a quick and deep view of the overall structure of the application.

• Fred could automatically detect and locate violations in the design rules as presented by the framework. The environment enforces the cardinalities, the naming and the dependencies of the EJB pattern elements and complain if the contract between the architecture and the developer is broken.

There are few other experimental tools that resemble the pattern-based approach of Fred (see e.g. [6]). In principle, such tools could be used to produce support for EJB application development environment based on a selected set of EJB patterns. However, Fred is unique in its support for fine-grained task-driven development and adaptive on-line guidance. Hence the resulting environment would lose much of the interactive flavor of a Fred-based environment.

7. Conclusions

We demonstrated that it is possible to automatically generate a pattern-based EJB application development environment that provides stronger support than existing EJB environments and removes some of their typical problems. To produce such an environment, a coherent collection of EJB patterns was developed and specified using the notation of the Fred tool. On the basis of such specifications, Fred was effectively turned into an EJB environment. Early experiences with the produced environment confirm our expectations of the benefits of the approach. When compared to conventional EJB environments, the Fred-based ADE allows the generation of large amounts of code without loosing the programmer's control of the code, and the environment can be easily extended with new patterns when needed. However, further evaluations are still necessary to validate our approach.

We are currently integrating Fred with a commercial Java ADE. In this way the rich (but somewhat ad hoc) functionality of a modern ADE can be combined with the more systematic pattern-based software development paradigm of Fred. An even better solution would be to implement the characteristics of the Fred-based ADE directly within a traditional ADE.

There are also some limitations and challenges that were recognized during this work. Some programmers may feel that the task-based wizard limits the freedom of the programmer. In principle this is not true, because the programmer can always resort to the integrated Java editor and write arbitrary code; however, the benefits of the system would be then lost. Hence the programmer has full freedom to decide which parts of the application are supported by the Fred patterns. The environment could allow more freedom by offering different alternatives on how to implement the design patterns, instead of sticking to one implementation strategy. Another limitation is the strict step-by-step working mode enforced by the tool; this appears to be too tedious in cases where user input is actually not needed. For example, several EJB callback methods come with default empty implementations and could be generated without user involvement. Finally, Fred provides no support for specifying how the method bodies should be written when the default implementation is not sufficient.

So far we have used a fairly limited collection of EJB patterns, and the environment should be extended with other J2EE patterns. The Fred environment could then be used to generate an environment for developing complete J2EE compliant applications. Another important limitation is the lack of support for the new EJB 2.0 specification. Currently the environment does not support message beans, local interfaces and the notion of a query language. Finally, a serious challenge would be to generate custom deployment descriptors for different application servers.

References


Publication 3


Copyright © 2002 NJC. 
Reprinted, with permission, from NJC, Volume 9, Number 3, Fall 2002.
A PATTERN-BASED J2EE APPLICATION DEVELOPMENT ENVIRONMENT

IMED HAMMOUDA
KAI KOSKIMIES
Institute of Software Systems, Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
{imed|kk}@cs.tut.fi

Abstract. J2EE is Java’s platform for building distributed enterprise applications. The platform takes advantage of a wide range of new and evolving technologies and has been enriched by proven design solutions. These solutions are formulated and documented in what is known as J2EE design patterns. Rather than applying the patterns in isolation, a complete design system can be composed of the patterns. This pattern-based system can significantly ease the development process of J2EE applications and improve the quality of the produced software. In this paper, we will show how a general architectural tool (Fred) can be used to model such a pattern system and to generate an architecture-centric environment for developing J2EE applications. The environment is a task-based wizard that guides the user through the development of the application by enforcing certain design rules.

CR Classification: D.2.10, D.2.11, D.2.13

Key words: J2EE, framework, design patterns, software architecture

1. Introduction

J2EE is a component-based and platform-independent architecture for building enterprise applications. This architecture offers a multitiered distributed application model. Each tier is usually implemented by a different group of developers and communicates with the other tiers via a standardized interface. The advantages of n-tiered architecture are promoting software reusability, easier system maintenance, and more effective use of data and networks. Most web-based enterprise applications are split into three logical tiers, Presentation Tier, Business Logic Tier, and Enterprise Information System Tier. J2EE provides standard solutions to each of these tiers.

Using J2EE, the presentation tier, which aims at presenting the business information to the user, is implemented using Servlets, JSPs, and HTML or WML pages. The business tier, where core business mechanisms are implemented, is usually encapsulated in EJBs (Enterprise Java Beans). The enterprise information system tier, which represents different kinds of legacy systems, database servers, etc is usually accessed through the JDBC API and other standard interfaces provided by the J2EE Connector Architecture.

Received September 27, 2002; revised November 11, 2002; accepted November 13, 2002.
J2EE application development is still an evolving art. In order to share good design solutions, proven design and programming techniques are available and shared by J2EE developers. These solutions are known as J2EE design patterns. However, because the technology is moving so quickly, designers and developers are struggling to understand and apply the architecture. Recognizing the need, several commercial and non-commercial software products with J2EE tool support have been released and adopted by the industry. These systems offer a development environment for J2EE applications, providing assistance in generating code, testing the applications, and deploying them within the environment. Such environments are called Application Development Environments (ADE).

Even though these ADEs provide substantial help for system designers and application developers, they open a new set of problems: the produced code hides the architectural aspects and the design decisions that have been taken, developers need to examine the generated code in order to maintain it. Another problem is that these development environments can be seen as closed systems, making it hard to adapt them to new design decisions, which contradicts with the evolving nature of J2EE patterns. In this work, we show how a general architectural tool called Fred (“Framework Editor”, [5], [6]) can be used to generate an architecture-centric task-based environment for developing J2EE applications. The discussed pattern-based environment overcomes the mentioned problems of conventional ADE’s.

The remaining of the paper is organized as follows. In the next section, we discuss the main features of the J2EE patterns used in this work, and how they are organized as a small pattern system. An overview of the Fred J2EE environment and a small case study is presented in Section 3. In Section 4 we compare the work to other existing solutions. Finally, in Section 5 conclusions are drawn and possible future work is highlighted. The EJB part of the work has been previously discussed in [7].

2. A J2EE pattern system

In this section, we will just give a brief overview of the patterns we modeled using Fred to generate a J2EE ADE. Our study will focus on patterns that might improve the architecture of J2EE applications. Most of these patterns are presented in more details in [1]. We show how to present these cooperating patterns as an integrated pattern system that can be used by J2EE developers to cover central parts of the design.

2.1 Business-tier patterns

Session Facade
The communication between the presentation layer and business layer in distributed business applications often leads to tight coupling between clients and the business tier. The interaction could get so complex that maintaining the system becomes difficult. The solution to this problem is to provide a simper interface that reduces the number of business objects exposed to the client over the network and encapsulates the complexity of this interaction.
Value Object
In J2EE applications, the client needs to exchange data with the business tier. For instance, the business components, implemented by session beans and entity beans, often need to return data to the client by invoking multiple get methods. Every method invocation is a remote call and is associated with network overhead. So the increase of these methods can significantly degrade application performance. The solution to this problem is to use a Value Object to encapsulate the business data transferred between the client and the business components. Instead of invoking multiple getters and setters for every field, a single method call is used to send and retrieve the needed data.

Business Delegate
By using the Session Facade pattern, we did not rule out all the design problems involving the interaction between the client and the business layer. We do have a centralized access to the business logic but still the session bean itself is exposed to the client. Enterprise beans are reusable components and should be easily deployable in different environments. Changes in the business services API should not affect in principle the implementation of the beans. To achieve loose coupling between clients at the presentation tier and the services implemented in the enterprise beans, Business Delegate pattern is used. This hides the complexities of the services and acts as a simpler uniform interface to the business methods.

Service Locator
In J2EE applications, clients need to locate and interact with the business components consisting of session and entity beans. The lookup and the creation of a bean is a resource intensive operation. In order to reduce the overhead associated with establishing the communication between clients and enterprise beans (clients can be other enterprise beans), the Service Locator pattern is used. This pattern abstracts the complexity of the lookups and act as a uniform lookup to all the clients.

2.2 Integration-tier patterns
Data Access Object
Enterprise Java Beans are reusable components and should be deployable in different environments with lesser effort. Implementing a so-called bean-managed persistence entity bean means that the programmer should provide all the persistent code (JDBC code). However, the API to different databases is not always identical so the bean programmer should consider different persistence code for different data sources. Depending on the data source, one specific implementation is used. To make the enterprise components transparent to the actual persistent store, the Data Access Object pattern should be used.

2.3 Presentation-tier patterns
Front Controller
A View represents information that is retrieved from the data model and displayed to the client. View navigation is a key issue in web-based enterprise applications. Because views usually share common logic, a centralized access point for view
navigation can be introduced in order to remove code duplication and improve view manageability. This pattern controls and coordinates the processing code across multiple requests. It centralizes the decision with respect how to retrieve and process the requests. A common strategy to implement this pattern is to use command pattern [4].

**Intercepting Filter**
Requests and responses need sometimes to be processed before being passed to handlers and other clients. An example of request processing is form validation, user authentication, or data compression. The solution is to create pluggable filters to process common logic without requiring changes to core request processing which improves code reusability and decouples request handlers. Servlet specification version 2.3 comes with a standard Filter approach that can be used to apply this kind of processing.

**Dispatcher View**
The system needs to control the flow of execution and the navigation between views. In particular, the system needs to know to which view to dispatch next based on the request. It is vital to separate the logic on deciding which view comes next from the view components themselves. The pattern does not perform heavy processing on the request but can be seen as a simple forwarding facility.

**View Helper**
In J2EE applications, a view is used for content presentation, which may require the processing of dynamic business data. Business and system logic should not be placed within views. Encapsulating business logic in a helper instead of a view makes the application more modular and promotes code reuse. View Helper pattern enforces the use of only presentation format inside the view and not the processing required to get the presentation content.

In addition to the standard J2EE design patterns above, we have defined several custom patterns that have been translated to Fred specialization patterns.

**Primary Key pattern**
Entity beans need to be occasionally stored in the database and loaded to memory to guarantee data consistency. Also some finder operations need to uniquely identify the entity bean in the underlying storage. Such operations need to have a primary key object that would allow the environment to uniquely specify the target bean. In EJB technology, the primary key class is optional. However, in order to enhance the readability and the maintainability of the system, our environment requires the use of a primary key class for every generated entity bean.

**Session (Entity) Bean pattern**
EJB is a component architecture that offers a common standard for distributed applications. Session beans (as well as entity beans) share the same architectural skeleton. This makes cross-vendor, cross-platform components easy to integrate together. In our environment, we have defined a programming pattern for session beans and another for entity beans. The persistence of an entity bean can be either container-managed or bean-managed. Session beans can be either stateless or stateful.
Tester pattern
This is a simple pattern that acts as a test client for the generated enterprise beans. The client tests all possible operations on beans such as create, finder and custom business methods. The behaviour is observed through console output.

2.4 Pattern system
In order to generate a working development environment, we need to put these patterns together in an integrated scenario to form a more comprehensive solution to J2EE application development. In [11], the author sees that a promising way of working with design patterns is to group interrelated patterns, those which cover a specific problem domain, together to form pattern languages. Pattern languages can then be used to generate language specific frameworks. In [3], the authors introduce a process for framework construction based on pattern languages. We used a similar, yet more practical, approach to combine the patterns discussed in the previous section to define a pattern system which will then be modelled with Fred into an architecture-centric development environment for web-based applications that conform to the J2EE standards.

Fig. 1 shows a pattern system for J2EE applications. The proposed system is a modified version of the pattern system used in [1]. The modifications have been considered in order to make the system easier to implement and use. For instance, each entity bean is associated with a primary key class generated by the Primary Key pattern. This practise is required in our environment whereas it is an optional feature in EJB technology. Knowing the primary key class allows the environment to generate more code, for example the type of the parameter passed to the findByPrimaryKey method can be deduced by the system. The same information can also be used when generating deployment descriptors for the entity beans. The architecture in Fig. 1 defines an entity bean, a primary key, a value object, and a possible data access object (in case of bean-managed persistence) as one block. This structure is usually used recurrently in the same application, thus it should be defined as a separate substructure. Imposing this kind of organization enhances the architecture of the generated application, which in turn improves the performance of the application and promotes its maintainability and portability as we have seen in the previous section. In addition, a Tester pattern has been added to the pattern system in order to individually check the generated beans.

The presentation tier is implemented around the MVC pattern [4]. Controller, and business code is separated from the dynamic content and the view components. Client requests can be pre-processed and post-processed using intercepting filters. Front controllers further process the request by executing the right command, which in turn executes the associated task on the business-logic tier. The user is dispatched to the next view using dispatcher components. View helpers are used to present business data in separate view components.

At the business tier, the environment generates skeleton code for both entity and session beans. Every generated entity bean has one primary key class and at least one Value Object. If the entity bean has bean-managed persistence then a Data Access Object is used in order to encapsulate JDBC code. A number of Session
Facades are used to encapsulate entity beans. In order to lookup and use bean instances and home objects, the Session Facade gets the service from the Service Locator pattern. Other clients as well, including session and entity beans, locate other beans using the Service Locator. Individual beans could be tested using the Tester pattern. A Business Delegate is used as a communication point between the presentation and business tier acting as a proxy for the Session Facade.

During the development of a typical web-based enterprise application, a set of use cases is generally identified from the application requirements. A variation of the pattern combination shown in Fig. 1 can be used to implement each use case. In this way, the whole application is developed around the same design rules enhancing the readability and the maintainability of the system. Besides, applying such a pattern combination for each use case promotes better separation of concerns. Each programmer can concentrate on her specific development role defining clear interfaces how to interact with the other components. The clear separation between presentation and business layers makes page designers for example independent from other programmers who are implementing the business logic. However, other J2EE applications have very specific business domains and so they require specific design solutions. In some cases, neither the discussed pattern system nor a variation of it can be used to design the application. Nevertheless, we believe that
a software pattern system is by definition an open system that is dynamic, adaptive and evolving. The system can be extended by new design rules and ideas that could make it suitable for wider range of application problem domains.

3. Fred-based J2EE environment

In this section, we will give an overview of the Fred J2EE environment. We will first describe the main features of Fred tool and then illustrate the concepts with a small case study.

3.1 Fred environment

An object-oriented framework is a reusable design expressed as a set of classes implementing the basic architecture for a family of software systems. A pattern-based framework is a framework that takes as input a set of collaborating patterns which model the aspects of the framework that are likely to differ from one application to another. The process of generating applications out of a framework is known as the specialization process. Stepwise specialization, in particular, offers an attractive approach for incremental application development. The programmer executes a list of simple tasks; each task adds a small portion of source code to the application. Fred is a prototype tool for enforcing pattern-based software development in Java. The software architect specifies the architectural decisions of a system as a set of patterns. In Fred, the pattern concept is called a specialization pattern. This is a concrete implementation of the general pattern concept. A specialization pattern is composed of several roles; each role corresponds to one program element (class, method, field). Roles may have properties like dependencies on other roles, cardinality, constraints, and templates. The task of the software developer is to bind the roles of the patterns to actual program elements following the dependencies of the roles. In this way, the tool can guarantee that the given source code conforms to the architecture. More information about Fred tool can be found in [5] and [6]. Specialization patterns represent the core of the tool. The framework developer creates the patterns for the framework and puts them together in a unified scenario. The application developer uses the framework to generate a specific instance. Each pattern we discussed in the previous section is modelled as a separate Fred specialization pattern.

Fig. 2 shows an annotation example of the Session Facade pattern. The pattern consists of several roles, these are shown in the left view of Fig. 2. The right view of Fig. 2 shows the properties of the class role. It is a relatively easy task to model the specialization pattern in Fred once they are specified in terms of roles and their properties. For example, it took a couple of hours to model the Session Facade pattern shown in Fig. 2. Documenting the pattern (role and task description) takes up to half of the annotation time. We have applied the composite pattern approach used in [8] to create macro patterns. Composite patterns are combinations of other patterns. For example a BMP entity bean is used together with a Primary Key, a Value Object, and a Data Access Object. Once the patterns have been constructed, Fred is used in a simple visual mode to put these patterns
in a specific combination forming the pattern framework. At this level, roles that represent the commonalities between all the applications are bound to concrete program elements whereas roles that represent the variabilities are left for the application developer. New patterns can be created by developers and deployed in the existing architecture. The time spent to represent the whole framework in Fred is proportional to the number of used patterns. In our case, it took a dozen of hours to generate the whole J2EE environment. The complete framework is shown in the architecture view of Fig. 3. A detailed description of the environment is presented in the next section. The environment generated adds a number of qualities that we find desirable and essential for J2EE application development. These include:

- **Task-based stepwise framework specialization.** The developer can select which pattern to use and can eventually switch to other patterns during the specialization process. The programmer executes a list of simple tasks. This gives her better understanding of the pattern.

- **Extensible architecture.** New patterns can be easily introduced to the system at any time. The pattern descriptions can be generated automatically as XML files, to be further processed for documentation purposes.

- **Source code editor with interactive constraint checking.** The environment can check the source code against the pattern constraints. The user is informed about possible conflicts every time a new task is provided.
Adaptive documentation. During the specialization process, the instructions given to the user take into account previous decisions and concrete names used for the program elements. This makes it easier to follow the specialization process.

3.2 Example

Fig. 3 shows a sample view of the environment during the development phase of a sample J2EE application. The developer is implementing a web-based to-do list where a list of users and their associated tasks can be accessed, manipulated, and stored in a relational database. The application is implemented according to the architecture proposed by the design patterns of the environment. The Architecture View to the left gives an idea about the overall application architecture in terms of patterns. The Session Facade composite pattern Facade, for example, encapsulates two entity beans. User composite pattern represents a bean-managed persistent entity bean whereas Task composite pattern encapsulates a container-managed entity bean. User has a business logic part and a business-data integration part. In the left pane of the Task View we can see the list of the tasks that have been carried out by the user. In the other half, the environment asks the user to provide findAllByPrimaryKey method. An optional task is to define a new persistent field. The tasks can be carried out by indicating the concrete program elements that play the roles. The environment generates default code for the concrete element based
on the role properties. The system checks the bound element against the constraints of the pattern each time the user performs an action. Fred supports adaptive documentation that helps in assisting the user with explanations on what tasks should be provided and which missing program elements should be added. Also, remedial tasks are generated when the constraints of the architecture are violated. This is illustrated in the right bottom pane of the task view. A substantial amount of code, shown in the Java Editor, is automatically produced as instructed by the pattern specification. The pattern editor is synchronized with the task view and the architecture view. The user applies a pattern by clicking on the corresponding pattern instance in the architecture view. As a result, the tasks associated with that pattern instance are displayed in the task view. When a task is performed, the integrated Java editor shows the corresponding program element and new tasks might be generated. The user is free to choose which patterns to apply and which to leave out. Also, the user can decide which tasks to generate and which to ignore.

The environment has been tested against several other similar, simple applications. The experiences showed that the environment could improve the quality of the software and reduce the development effort. The improved quality is a consequence of making desirable J2EE design solutions available for the application developer and guiding her to use these solutions. Although it is hard to define specific metrics to compare the development process with more conventional ones, it can be seen that this approach reduces the development effort considerably. Methods, for example, are usually generated with default implementations. A big part of the programming tasks is reduced to mouse clicks, the user is often asked to provide a program element with the name proposed by the environment. The benefits apply both for experienced J2EE programmers and novices. In the case of our example application, up to 60% of the total lines of code was automatically generated by the environment. Most of the generated source code represents the program elements required for the J2EE component environment, the overall code structure of the used patterns, and the way these patterns are interacting. Manually, it could take up to 20 hours to build the discussed application code from scratch. Using Fred J2EE environment the same code has been generated within a couple of hours. Approximately, one third of the development time is spent with using the environment for automatic code generation, whereas the rest was the programmer’s part to provide custom business code and implement the user interface. However, these figures should not hide the improvement in the design quality of the generated application that cannot be quantified.

4. Related work

There are several commercial and non-commercial environments that enable rapid J2EE application development. RealMethods [10], for example, is a tool that provides design pattern-based implementation on all J2EE tiers (presentation, business, and integration). The tool requires the user to import in the system a standard XML file that represents the object model. The XML file is parsed and represented as a tree before generating the required code by a single mouse click.
The advantage of this technique is that once you have your bean specifications ready, generating the equivalent Java code becomes a simple and fast operation.

Other widely used application development environments offer a more standard way of support for J2EE technology. An example of such environments is JBuilder [2]. The tool, for example, is capable of generating code for enterprise beans in a visual mode. It can also construct entity beans out of data models. A big advantage of such a tool is the built-in application server where the generated beans could be deployed and tested.

The third ADE we have evaluated is TogetherJ [12]. The tool supports the use of patterns when developing J2EE applications. Developers can also create their own patterns. The use of patterns in TogetherJ, however, is limited compared to our environment. For example, the tool does not detect the design violations during the specialization phase, developers can freely change the structure of the patterns. Another important feature of this environment is the fact that pattern elements are displayed in a visual mode and are well synchronized with the corresponding source code.

However, The three environments fall short in many features available in the Fred J2EE development environment.

- Some tools, JBuilder for example, pay more attention to code generation but give less care to design and architecture solutions. The Fred environment pays special attention to the architectural aspect of reuse. The environment constitutes of a collection of collaborating patterns and several other programming rules that could be easily extended and updated.

- Tools similar to realMethods require the preparation of standard representation of the object model. This is a sensitive and time consuming operation. Any errors in the specifications can affect the produced code. Every time the specification changes, new code should be generated. Fred offers a smooth evolution of the application that goes in parallel with the specification.

- Adaptive documentation, supported by Fred, is an essential factor to architecture-oriented software development. In JBuilder environment, documentation is rather static. The environment does not record the history of the developer’s tasks and does not inform her what to do next. This limitation is also true for TogetherJ, the tool does not guide the developer through the application development.

- The architecture view of Fred environment makes it easy for the developer to understand the problem domain. The task view reflects the status of the application.

- Using Fred environment, it is easy to see how the different tiers of the J2EE application communicate with each other. The user does not need to refer to the generated code as in the case of realMethods.

- Fred could automatically detect and locate violations in the design rules as presented by the framework. The environment enforces the cardinals, the naming and the dependencies of the J2EE pattern elements and complains if the contract between the architecture and the developer is broken. TogetherJ does not support this feature.
5. Conclusions

In this paper, we have presented the main features of the J2EE Enterprise Java Beans component architecture. We have briefly studied some proven design solutions to J2EE application known as “J2EE design patterns”. These solutions could significantly improve the design and architecture of distributed enterprise applications. Using a general architectural tool called Fred, we have developed a pattern-based framework for J2EE applications. The generated environment can improve the quality of J2EE software and can minimize development time and effort. We compared the framework to some other products that offer tool support for the technology. Fred environment looked very promising and contributed with many useful new features to conventional application development environments for J2EE systems.

Nevertheless, we are aware of several limitations that we recognized in our approach. First, rather than considering one implementation strategy, the environment could present the user with different solutions, at different levels of abstractions. Second, the environment comes with minimal control on what developers add to the generated code. For instance, it would be very beneficial to check if the user has implemented the methods as intended by the framework developer. In addition, because Fred cannot generate non-Java code, view components at the presentation tier have to be written in Servlets. View helpers implemented with Servlets are hard to understand and maintain, it is more convenient in implement them using JSP’s for example. Moreover, the environment could be enriched by new J2EE patterns and should provide support for EJB 2.0 specification.

However, by adding new patterns to the system, the framework could become more and more complex, which may lead to a longer learning curve. Also, the application development phase may become more complicated and error-prone. As the framework gets larger, using a standard generic architecture may require a bigger coding effort and so there will be no control over the overhead associated with the application development. It is essential to keep the size of the framework as small as possible. In this way, the framework remains simple and understandable, and therefore can be quickly adopted by novice developers. Also smaller frameworks tend to be more reusable and can be applied to wider problem domains than complex frameworks. Instead of continually adding new patterns to the existing framework, it is more convenient to create separate framelets [9]. Framelets are small frameworks consisting of a small number of classes that represent a clear substructure of a component. Framelets could either be used by the framework developer to build more complex frameworks or by the framework specialist to generate application code. A future plan of this work is to develop a collection of framelets that can be used either discretely or in combination to cover a wider range of problem domains.

Developing frameworks can be regarded as a delicate operation. Any bugs in the framework can very likely propagate to the individual generated applications. In this case, it would be hard to locate the source of program errors. The malfunction could either be a result of application-specific code as typed by the application developer, or simply due to a problem inherited from the specialized framework.
In order to generate applications with the correct behaviour, the framework should be designed and implemented error-free. Minimising framework defects can be achieved either by testing the framework itself or by testing the individual generated applications. During the testing process, possible bugs are located and the framework (or the application) is then fixed. Framework construction, therefore, can be seen as an iterative and incremental process. Moreover, testing framework specializations is a possible way of ensuring that the user has implemented the framework methods, for example, as intended by the framework developer. An important challenge for Fred environment is to provide testing support for both frameworks and framework specializations.

Acknowledgements

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

References

Publication 4


Copyright © 2004 IEEE.
Reprinted, with permission, from the proceedings of ASE 2004.
Adaptable Concern-based Framework Specialization in UML

Imed Hammouda, Johannes Koekinnen, Mika Pussinen, Mika Katara and Tommi Mikkonen
Institute of Software Systems
Tampere University of Technology
P.O. BOX 553, FIN-33101 Tampere, Finland
{firstname.lastname}@tut.fi

Abstract

Architectural-level reuse of software can be achieved in the form of application frameworks. Then, the architecture of a system can be copied from a framework, and the developer is liberated to application development. In this scheme, patterns utilized for specializing the framework play a critical role. Unfortunately, the bigger the specialization pattern, the harder it is to adapt the pattern to a particular design due to increasing number of bindings between pattern roles and the elements of the design. In this paper, we introduce a tool supported methodology based on UML in which specialization patterns are grouped to match different concerns, i.e. conceptual matters of interest, they treat. Also, user-controlled instantiation of individual patterns is allowed to promote learning the architectural conventions. We argue that this approach overcomes some limitations, especially the lack of adaptability, of wizards that are commonly used for similar purposes.

1. Introduction

One of the recent trends in software development, driven by practical needs, is architecture-level reuse. A pragmatic way to reuse architectures is simply to copy-and-paste a previous implementation and to modify its code. However, it has been considered a harmful practice as the developer may not fully understand the reused (sub)system, nor be able to adequately modify the correct parts of it. This can lead to severe consequences such as lack of traceability and maintainability [2].

More advanced approaches include the use of wizards that automatically generate code, which follows certain guidelines based on user input. While wizards enable fast code generation, it is not a very descriptive way to produce designs, as the internals of the wizard are not revealed to the application developer. Similarly, maintenance of the wizard when changes are needed to its contents may be problematic. The same applies to user-tailored customization.

The above problems have resulted in the development of application frameworks that can be easily adapted to fit the purposes of different applications via the process of specialization. This adaptation takes the architecture specified by a framework as the starting point, and adds application dependent issues, for example by allowing developers to override some methods with application-specific ones.

However, contrary to using wizards, the framework approach reveals unnecessary implementation details of the framework and its specialization interface, leading to imposing additional complexity to the developer. Furthermore, separation of concerns is usually poor as the concerns, i.e. conceptual matters of interest, of different stakeholders are addressed by several different architectural units each treating various concerns. The situation is worsened by the lack of tools, which would assist the developer to carry out the specializations at the level of visual modeling languages, such as UML.

This paper introduces a methodology where framework specialization is carried out in a way where individual design decisions are made visible to the user in UML. In fact, these design decisions can even be revisited, if the underlying framework is modified or some customization is needed. The methodology is based on combining three approaches from the existing body of research in software architectures:

- Application frameworks, defining common software architecture for a family of applications, are a proven software development approach. A framework consists of predefined classes, some of which may be abstract, and their relationships that facilitate application development for some specific domain or platform.
• Specialization patterns [7] are used for annotating the specialization interfaces of frameworks. Instantiating a pattern results in a more specialized version of the framework. After instantiating all the patterns, the framework is fully specialized.

• Concern architecture views [10] are used to structure sets of patterns and their dependencies to match different concerns in UML. Then, in order to provide a treatment for a concern, one only has to specialize the framework according to the patterns associated with that particular concern. This enables aspect-oriented architecture where each specialization step produces a more detailed UML model treating the corresponding concerns and nothing else.

The contributions of this paper are in conceptually integrating the above techniques into a seamless application development approach. In addition, to support the approach, we introduce a prototype toolset called MADE. The toolset automates the framework specialization process and supervision of architectural constraints, among other things.

The rest of the paper is structured as follows. Section 2 introduces the motives behind our work, together with the way we have integrated the available techniques into a toolset. Section 3 demonstrates the use of the approach using Symbian application framework as a case study. Section 4 presents the MADE toolset in detail. Finally, Section 5 describes related approaches and draws conclusions.

2. Motivation and theory

Applications derived from the same framework share a similar structure and common architectural rules. Typically, the same framework classes are specialized and a slightly differing set of methods are overridden from those classes, making framework specialization a recurring problem. Specializing a large framework manually is both error prone and inefficient. One has to make sure that all the necessary classes are inherited, methods are overridden correctly, and the implementation conforms to architectural rules and conventions.

This is also true for Symbian OS [16], an operating system targeted for mobile devices, which we use as a running example. What makes the specialization of this framework harder than many other frameworks is its reliance on architectural rules expressed as coding conventions. Therefore, even if the developer has specialized correct classes and has overridden proper methods, the result may be an invalid Symbian application. For instance, the last letter of a method name tells if the method may or may not throw an exception.

Methods throwing an exception must either catch the exception or indicate in its name that the exception is propagated. Because of the multitude of essential conventions and unavailability of tool support, the complexity of the Symbian platform often results in copy-paste programming utilizing source code from previous projects. A better approach, however, is to use tools automating the specialization process.

In general, the current best practice for automated framework specialization is the use of wizards. The framework developer is responsible for creating an installer-like program that guides application developers through the necessary steps in the specialization process. Although this is clearly a step forward, wizards are merely a more civilized form of copy-paste programming. Both approaches raise serious concerns regarding comprehensibility and maintainability. For copy-paste, problems are well-known [2].

For wizards, consider the following. First, the specialization process does not support learning. Wizards usually lack documentation describing the kind of problems that are being solved and how these problems relate to other parts of the application. In many cases, additional documentation and example applications are needed. Furthermore, wizards typically generate applications in one shot. Framework specialization, however, might involve separate interests in the application. Therefore, a better approach is to organize the specialization interface into groups of related specialization steps. Generating an application one building block at a time contributes to understanding both internals of the framework and the application. Second, wizards have mostly been used to generate application code. However, various relationships are easier to comprehend from visual models. Thus generating models of application, in addition to its code, might be beneficial. Finally, as the framework evolves, wizards need to be maintained. Consider an example where the framework developer decides to augment a platform with an optional feature based on the feedback from application developers. This means that a new version of the wizard, including that feature, is created. There are now two versions of the framework corresponding to the two versions of the wizard, and it might be difficult to persuade all the users to upgrade unless they really need the newly added feature. Therefore, a framework specialization environment should allow "plug-and-play" of new framework features.

2.1. Basic idea

We propose the use of concern architectures, called also aspect architectures [10], to capture framework
knowledge into separate groups (concerns) of related specialization tasks. These specialization tasks are individual design decisions a designer must take to specialize the framework. Specialization patterns [7] are then used to implement these concerns by decomposing the large task groups into a set of simpler tasks. At the tool level, a generic pattern-oriented toolset is used to transform each specialization pattern specification into a task list that guides the developer through the specialization process step by step. Each step can be attached with documentation about the step and its role in the framework.

The specialization process generates a UML model documenting the generated application. The developer can enhance the generated UML diagram either manually or by deploying yet other patterns. Based on the pattern specification, the toolset monitors the application model detecting and repairing possible architectural violations. Finally, the information stored into the diagram and the deployed patterns can be used to create a code skeleton for the application. The toolset can be augmented with new patterns providing new solutions whenever needed, thus adapting the toolset to new requirements. Moreover, upgrades to existing patterns are also allowed.

Concern architectures, specialization patterns, and the toolset are discussed in the following sections.

2.2. Concern architectures

Figure 1 depicts a simplified UML class diagram of a typical Symbian application. The diagram shows only the classes forming the application and framework classes that are directly inherited. Symbian applications are built around the Model-View-Controller architectural style. This is reflected in Figure 1 so that classes CMyEngine, EngineDlEntryPoint, and CMyDocument form the Model part, class CMyView corresponds to the View part, and class CMyAppUi acts as Controller. Furthermore, Symbian imposes a creation chain between the main classes restricting how the corresponding objects are created when the application is started. This can be seen as a separate interest cutting across all the classes except EngineDlEntryPoint.

In the case of Symbian, application architecture clearly separates between different parts. Therefore, the specialization interface can be structured, at least partly, into separate concerns in accordance to architectural parts. In other situations, however, the concerns are not directly reflected by the architecture. For example, in the Symbian case, the creation concern crosses multiple architectural entities making it harder to be identified.

Concern architecture views can be used to factor out the related parts in a framework specialization interface and document them into separate concerns. When generating application specific design models, such a concern covers all the tasks for generating the model elements relevant to a certain part or aspect in the derived application. The idea is to make explicit the parts of the application design that correspond to different concerns. Additionally, concern architecture views document the order in which different parts of the application are generated. For example, the Model part of the application should be provided before the Controller part since CMyAppUi (in Controller) uses operations provided by CMyEngine (in Model).

Based on the discussion above, Figure 2 depicts the following important concerns in the Symbian architecture, which should be lifted as first class entities at the
• MVC concern, corresponding to the Model-View-Controller model.
• Model, View and Controller as sub-concerns of the MVC concern. In addition, Model has an internal structure of its own, which must be obeyed.
• Application concern, corresponding to the actual application.
• Creation concern, discussed above, forms an interesting pattern of its own. Without it, the framework would be sound in the object-oriented sense, but invalid in the domain of the Symbian platform.

In addition to the individual concerns, some of which are included in the figure, the Symbian application concern itself can be considered to include all the concerns of the framework. This is intuitive because if any of the concerns would not be applied, the application would not conform to the Symbian architecture.

As already discussed, we will represent a concern as a set of specialization patterns, where each specialization pattern takes care of a more limited set of specialization tasks. Specialization patterns are discussed in the following section.

2.3. Specialization patterns

A pattern is an arrangement of software elements for solving a particular problem. To be able to define a pattern independently of any particular system, it is defined in terms of element roles rather than concrete elements. Each role corresponds to one concrete element (class, method, or field). Roles may have properties like dependencies on other roles, cardinality, constraints, and templates. To illustrate these properties, let us consider two pattern roles A and B that stand for class roles. Role A is said to be dependent upon role B if the binding (associating the role to concrete classes) of role A depends on the binding of role B. The cardinality of role A specifies the number of concrete elements that may play the role. An example constraint on pattern role A could be an "inheritance constraint" meaning that the class which plays pattern role A should extend the class which plays role B. Finally, the "instance name" template, for example, represents the default name of the concrete element bound to the role.

In this work, we use a concrete pattern concept called a specialization pattern, whose purpose is to document the specialization interface of object-oriented frameworks. When instantiating a specialization pattern, the pattern roles are bound to concrete elements.

A UML class role, for instance, is bound to a class in the UML model of the application under development. In our work, role binding is considered as a specialization task. This enables the developer to better learn how the framework is specialized and control the development of the application. When all the application-specific parts of all the specialization patterns are bound, the framework is fully specialized.

Specialization patterns are used to represent the different concerns in the framework specialization interface. In the case of Symbian architecture, Figure 2 depicts which specialization patterns implement which concerns. The Model concern for instance is implemented by two patterns: Engine and Document. The dependency arrows show the order in which the patterns should be applied. For instance, pattern Engine should be applied before pattern Document since the implementation of Document stores a reference to the concrete engine.

Because the pattern specialization steps can be expressed as a simple task list, a tool can be used to maintain the task list implementing the pattern role properties and checking possible constraint violations. Next, we will introduce our prototype implementation.

2.4. Toolset introduction

In order to illustrate our approach, we have built a generic concern-based pattern-oriented development environment known as MADE. In this section, we discuss the main properties of the environment. Detailed requirements and architecture of the toolset will be discussed in Section 4.

Figure 3 depicts the main view of MADE as seen by
a framework specializer. UML models are visualized and modified using Rational Rose [13]. Previously described pattern and concern concepts have been implemented as Eclipse [5] plugins referred to as JavaFrames. The main window of JavaFrames is divided into Architecture, Pattern, Documentation, and Code Generation View. The Architecture View is used to catalog patterns and further organize them into deployable concerns, as already illustrated in Figure 2. The Pattern View is used to both create and maintain pattern specifications as well as to show bound UML elements during the binding process. The upper right-hand corner of the Pattern View shows the tasks available for the node selected in the left pane of the Pattern View. Task-specific documentation is shown for each task in the Documentation View. Tasks can be dependent on other ones, as implied by the dependencies between the pattern roles. When a task is performed, new tasks depending on the performed task, become available. Tasks can be either mandatory or optional. A pattern is fully specialized when all mandatory tasks are carried out. When specialization process is complete, the Code Generation View can be used to create code skeleton based on the generated UML model and on the information stored into the patterns specifications.

3. Case study on Symbian OS

Nokia Series 60 [14] is a software platform built on Symbian OS introducing a product family specific layer of framework classes on top of native Symbian ones. In addition, Series 60 comes with a set of applications, and the platform is licensed as such also to other mobile phone manufacturers. Software development kit (SDK) for Series 60 [15] is freely available for developers, and therefore used as a sample development kit in this paper. In the following, we use this platform as an example to demonstrate our approach.

3.1. Tool support

Figure 4 depicts wizard pages of the application wizard of the Series 60 development kit, and an application generated with it. The code skeleton generated by the wizard does not provide a very good starting point for a novice developer. One cannot expect such a developer to be familiar with the number of classes created by just running five wizard steps. The black-box nature of the wizard does not support learning and neither does it provide good extension possibilities. There are also more advanced wizard implementations that can be enhanced with project templates (e.g. [12]) but the granularity of the extension is more restricted and less general than in the proposed approach. Specializing wizards with domain specific knowledge results in either a separate wizard or a separate project template.

The deeper insight of the platform can be achieved via examples and documentation. Current best practices for architecting and developing Symbian applications are described in [16, 4], and they give explanations to the patterns, idioms, and conventions designed for the platform. Sample applications of software development kits, in particular, are important because many programmers like to learn rather by example than by reading a pile of documents.

UML-based tool support brings best practices to the software development kits guiding and even enforcing developers to proper architectural decisions and conventions. Currently used code examples can be enhanced with a set of patterns and semi-instantiated UML designs targeted for a specific problem. A developer could then start either from scratch by using the patterns, or use a combination of semi-instantiated models and patterns, with visual UML representation easing the comprehension of the solution as a whole. The semi-instantiated models could be modified for the purposes of target applications, and instantiated in full to create a model detailed enough for code generation. As development moves towards implementation at code level, the developer could pick appropriate helper patterns used to accomplish tedious programming tasks that are required by the underlying platform but take focus away from the development of the actual application logic. If the creation process of an application is made more transparent and documented, learning becomes a part of the process.
In the following, an example is given in which a fully functional application skeleton is created from scratch using MADE instead of SDK wizards.

3.2. Example

Symbian OS comprises a large framework offering classes for diverse purposes. Symbian Application Framework is its important subset, which forms a framework of its own. Figure 5 shows a class diagram consisting of the core classes upon which a new Series 60 application is specialized. In the beginning of the specialization process only these classes are stored and visualized in the UML modeling tool. Four application classes, forming the user interface of an application, are inherited from those framework classes in a predefined way. One of the application classes is embedded in a dynamically linked library being responsible for user interface independent application logic.

Figure 6 shows the concerns and patterns used to guide the specialization process. Catalogue node shows a flat list of patterns, which are further categorized under SymbianApplication node according the concerns depicted in Figure 2. The structure of SymbianApplication concern is not visible in the figure, but it has a structure identical to its instance, MyApplication, seen under Deployment node. README pattern selected in the Pattern View documents concerns, thus acting as a task-based user manual.

Figure 7 goes through a scenario in which a developer has started to deploy Engine pattern and created a package called engine, seen as package node in the left-hand side pane of the Pattern View. A circle in the lower right-hand corner of the node informs that there are tasks to be performed right under the node (circles have red color giving better distinction than can be seen in gray-scale figure). As the developer selects the node, tasks appear in the right-hand side pane of Pattern View. First, the developer has performed a task that asks user to either create or provide the concrete UML class that plays role Engine (step 1). Because the UML model does not yet have any application classes, the developer decides to generate a new class that plays the required role (steps 2 and 3). The specialization process proceeds in this fashion. The user performs individual tasks, and the environment prompts her for the necessary information. There can be also tasks using fixed naming scheme, thus requiring no input from the developer. Such tasks can be performed automatically in the recursive manner, which allows an advanced developer to go through familiar phases faster than only task by task (step 4). When the developer has performed all tasks of Engine pattern (step 5), the depicted model fragment has been added to the UML model.

In Figure 8 the whole set of patterns has been deployed. The obtained class diagram is almost detailed enough to be used as a base for the code generation. Still, additional information can be provided in the form of code templates stored into the pattern specifications. In the final phase, role names in the code templates are replaced by the names of the concrete UML elements, thus complementing the information provided by the UML model. Code generation phase is guided by a code generation wizard similar to the wizards our approach is compared to. The code generation wizard (Figure 9) creates project directory structure with related source code and the project files shown in the console window in Figure 10. The project files
The code generation phase

Figure 9. Code generation phase

The initial project is created. The developer is not alone with the starting point, but can apply new problem specific patterns in order to enhance the design and the Symbian project with new classes and related code. As an example of such support, there is category HelperPatterns in the architecture view that contains a simple helper pattern called HeapObject. Each dynamically allocated Symbian class must inherit from framework class CBase and be equipped with a set of convenience methods that make sure that the class follows the memory allocation principles. These convenience methods are replicated for each such class. By using the pattern, the developer can introduce a new heap-based class into some package, enhance the generated UML class with new application specific methods, and create a code skeleton for the class. Thus, the developer avoids the awkward and error prone copy-pasting or writing these methods from scratch.

4. MADE toolset

To accommodate the specialization of frameworks utilizing the discussed methodology, we have listed the following non-functional requirements for a tool:

- Flexible task-based guidance for application developers.
- Conformance checking, guiding the user to repair broken architectural conventions.
- Separation between different matters of interests in the application.
- Multiple levels of abstractions, from abstract models to code.

Figures 7, 8, and 9 illustrate the usage scenario, final UML model, and code generation phase, respectively.
• Interoperability with existing CASE tools and different IDE (Integrated Development Environment) tools.

• Openness to new design conventions in the specialization interface.

The above requirements have led to a prototype toolset MADE (Modelling and Architecting Development Environment). MADE is the result of the integration of a number of different tools. The main tools deployed are Eclipse-based JavaFrames and Rational Rose as already described in section 2.4. Communication between JavaFrames and Rational Rose is achieved through a UML model processing platform, xUMLi [1]. Figure 11 shows the architecture of the toolset integrating these tools. The individual tools forming the architecture are introduced in the following subsections.

4.1. JavaFrames

JavaFrames [7] (previously called Fred) was originally developed as a tool for specializing frameworks implemented in Java. The pattern engine, which is an integral part of the JavaFrames system, has been designed to support different domains. In JavaFrames terminology, we refer to these domains as semantics. In the same domain, there can be different semantics types. For example, in the UML domain, there are different semantics types for UML classes, UML operations, etc. Currently, the tool supports Java and UML semantics. However, it can be extended to support other semantics such as C++ or XML semantics. Figure 12 shows the main components of the system:

- Pattern engine: This is the core of the system which updates the task list and manages the binding process by following the inter-pattern relationships within the architecture and the inter-role relationships within individual patterns.

- Semantics: This part of the system connects the domain supported by the tool to the pattern engine. When supporting a new domain, appropriate pattern roles need to be implemented in order to model all possible semantics types in that domain. In addition, role properties and possible constraints on the roles need to be provided.

- Utility Functions: JavaFrames comes with a number of tools that are used to develop and deploy the patterns. For example, the architecture view shown in Figure 3 manages the different concerns in the framework specialization interface and shows the constituent pattern relationships; the
pattern view constructs the individual patterns; the task list shows the task to be performed; the task wizards are used to bind the roles to concrete elements; and the synchronization tool is used to check the constructed model against the pattern specification.

In addition to being used as a development environment, Eclipse is used to implement the user interface for developing and deploying the patterns. The main properties of JavaFrames environment are:

- Simple task lists: JavaFrames environment models a specialization interface in the form of role-based patterns. When applying these patterns, the tool generates a number of tasks to be carried out in order to meet the specifications. By task, we mean a simple action that adds an element or enforces a property on the model. Tasks are kept simple enough so that their immediate result is easy to track and can be changed if needed. A sample task is to provide an operation for a class in a UML model. In order to express users' own context, task descriptions might refer to the concrete names of the elements specified earlier.

- Reactive model checking: Every time the domain is changed, for example a new element has been added to a UML diagram, the environment rechecks the constraints, and the user is asked to repair the violations, if any.

- Separation of concerns: The architecture view of the tool lists the different concerns in the specialization interface. When the concerns are applied, it is possible to highlight the generated application elements corresponding to a specific concern.

- Generating UML models and source code: When applying the patterns, the environment generates abstract application models expressed in UML diagrams. In addition, the environment can be used to generate code which corresponds to these models using the information stored in the patterns. Hence, JavaFrames can generate different application views offering multiple levels of abstractions.

- Openness for new patterns: The environment is not bound to a fixed set of architectural patterns, but new patterns can be easily introduced to the existing architecture at any time. The system can be extended by new design rules and ideas that could make it suitable for wider range of application problem domains.

4.2. UML editor

For UML support, we decided to use an existing CASE tool. In order to be able to introduce our own extensions, we therefore needed a documented application programming interface (API) for communicating with the tool. Moreover, we wanted to ensure that the tool would be in wide use in the industry so that all our results could be benefited from in practice.

At the point of selecting the tool, Rational Rose was considered as the best candidate. It offers an API to its contents, and is widely deployed in companies that we believed would be interested in our approach. Moreover, the possibility to create UML models single-handedly with Rose was also considered important, as then the toolset can be used as an extension of an existing development process.

4.3. xUMLi

Although many available UML-based CASE tools (like Rose) provide proprietary APIs for extending the tool, it was considered advantageous to implement a tool-independent API for processing UML models which provides high-level support for implementing model processing actions. By using CASE tool independent API, rewriting the model processing code is not necessary when the used CASE tool changes. Such a tool-independent UML processing platform xUMLi (executable UML interface) has been introduced in [1].

UMLi (xUMLi API) has been implemented using the Microsoft COM component model [3]. Therefore, a bridge between COM and Java virtual machine is required. In the toolset we use an open source tool called Javlin [9]. To integrate xUMLi with the toolset, a Java class layer for UML support has been written to hide implementation details of UMLi and Javlin. Every UML metaclass supported by xUMLi is translated into Java class providing methods to set, add and get values of the properties of the metaclass. The class layer also provides methods for the navigation within the model using OCL (Object Constraint Language [11]) as well as importing and exporting the model with importers and exporters provided by xUMLi.

Because xUMLi was originally designed for scripting and batch processing, user activity (such as changing the model in the editor) has to be caught some other way. We have written a Rose add-in to catch and forward the events that Rose sends while the model is being changed. The messages from the add-in are interpreted in the Rose plugin of JavaFrames, and a notification is sent to registered plug-in listeners.
5. Discussion

In this paper we have introduced a methodology for adaptable concern-based framework specialization in UML, together with prototype tool implementing the approach. The aim has been at supporting application development using frameworks by providing tools that operate on a higher level of abstraction and automation than conventional ones. The approach is at its best when adaptability or learning are important aspects of the development, as then revealing the details of the design to the developer introduces most benefits. The introduced toolset has been developed by reusing existing technologies and integrating proprietary and open source tools from external projects as well as our own.

As demonstrated by the Symbian OS example, the approach allows to capture meaningful concepts of the software architecture using concern architectures and specialization patterns. The application developer is liberated from the intricacies inherent in the target architecture. Moreover, the patterns can be used as building blocks in a concern architecture to separate concerns inside the framework. This allows the specialization process to advance in well-defined steps that are also meaningful to the application developer.

Concerning related work, one of the early attempts to define tool support requirements for developing and applying design patterns is presented in [6]. The authors discuss fundamental issues and requirements a pattern-based tool must address. Their tool, called OMT-Tool, assists a developer in generating program elements, integrating pattern occurrences with the rest of the program, checking the invariants governing the patterns, and repairing the violations. An important requirement that this approach lacks compared to ours is the task-based guidance and the adaptive documentation presented to the application developer. In addition, the architecture of our toolset supports the interoperability with other CASE tools and IDEs.

The most closely matching commercial tool is Rational XDE [8]. Like in our approach, XDE uses its own dedicated pattern editor to specify patterns, but the instantiation process differs. In MADE, it is possible to apply a pattern in small increments whereas an XDE pattern must be applied in full. In addition, our tool supervises the integrity of the pattern automatically. In case of violations, new tasks will appear which guide in fixing them. In XDE, a separate validation step is needed. Furthermore, there is no support for explicit modeling of concerns cutting across several patterns. However, an Eclipse-based UML modeling tool would provide more interactive use with JavaFrames than Rose currently does. Thus, XDE might be a good alternative if Rose needs to be replaced in MADE.

Our long-term goal is to develop an industrial-strength application development tool. Naturally, larger case studies need to be conducted in an industrial setting to validate the appropriateness of the approach. The future work includes also gathering statistical data comparing our methodology and toolset to the wizards and copy-paste approach currently in practice.

References

Publication 5


Copyright © 2004 IEEE.
Reprinted, with permission, from the proceedings of CSMR 2004.
Documenting Maintenance Tasks Using Maintenance Patterns

Imed Hammouda
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
imed@cs.tut.fi

Maarit Harsu
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
harsu@cs.tut.fi

Abstract

A common problem in software maintenance is the lack of documentation required for carrying out the maintenance tasks. Both expected and unexpected maintenance tasks use and produce important information, which is not systematically recorded during the evolution of a system. As a result, maintenance becomes unnecessarily hard. In this paper, we suggest the use of maintenance patterns to record information about maintenance tasks. Maintenance patterns are organized collections of software elements relevant for a particular maintenance concern. Such a pattern can be exploited for advanced tool support where the tool keeps track and guides the system developer step by step through a maintenance procedure.

1 Introduction

A truly successful software system is one that is continually modified and enhanced [22]: prolonging the lifespan of a software system has always been one of the main purposes of software maintenance. However, a common problem in software maintenance is the lack of documentation needed for the maintenance tasks. In fact, it is well known that maintenance should be considered already in the design phase [21]. Unfortunately, this suggestion is usually not followed, partly due to the limitations of the current technologies and methodologies. Even though a designer anticipates certain maintenance tasks during design, there are no systematic means to record this information in such a form that it could be used to provide tool-assisted guidance for the maintainers. In the case of an unanticipated maintenance activity, the information concerning the activity is typically lost completely, or documented scarcely at best.

However, this information would be valuable in the future maintenance, because it reveals a logical connection between different parts of the system that is relevant from the maintenance viewpoint. Such information is valuable if similar maintenance tasks are repeated, or if other maintenance tasks concern parts that have been involved in previous tasks. In the latter case, information about previous maintenance tasks can be exploited to guarantee that the new maintenance task does not corrupt the effect of the previous action. Again, this information can be used to provide tool support for repeating similar maintenance tasks, for reversing maintenance tasks, and for sustaining compliance with earlier maintenance tasks.

In this paper, we introduce a way to document software maintenance via so-called maintenance patterns. In this context, a pattern means a specification of a set of software entities that are logically related from the viewpoint of some concern. More precisely, a pattern consists of roles that can be bound to program entities (like classes, methods and attributes) and a set of constraints that these bindings must satisfy. A pattern can exist without any bindings, or it can be partially or fully bound. A maintenance pattern is a pattern that captures the set of program elements involved in a logical maintenance task. The roles are used to store the information about the participants of a maintenance task whereas the constraints state the relationship that must be followed by the program elements bound to different roles. In addition to the constraints specifying the relationships of the program elements, a maintenance pattern can be associated with other information concerning the maintenance task, like informal instructions for the maintainer or default code templates for some program elements.

Maintenance tasks can be divided into smaller subtasks that require modifications to individual small program elements scattered around the system to be maintained. This gives maintenance patterns a cross-cutting nature. In this sense a pattern is conceptually close to an aspect [2], but since we will have a fairly specific technical realization of this concept, we want to avoid using this term. We will refer to the cross-cutting concern represented by a maintenance pattern as maintainability concern.

Software maintenance may include different tasks: the
system can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [1]. In fact, it is acknowledged that the maintenance phase in software development is the most difficult and thus causes major costs [5]. We think that these deficiencies can be partially solved by associating the system with maintenance patterns. In the paper, we show the benefits of applying such patterns in different maintenance situations such as system adaptation. We also show how errors resulting from maintenance tasks could be reduced when using our approach.

An implication of our approach of documenting maintenance tasks as patterns is that we can exploit generic pattern-based tool support for maintenance. In particular, we use a tool called JavaFrames (previously Fred) to implement the idea of maintenance patterns. JavaFrames [12] is an Eclipse [3] plug-in which allows the specification and use of patterns in a Java programming environment. Given a pattern specification, the tool provides task-driven guidance for a developer to carry out the binding of the roles as required by the pattern. Although the tool was originally developed to support the specialization of a framework, we argue that the generic pattern concept of JavaFrames can be used as a basis of maintenance patterns as well. The benefit of exploiting JavaFrames is that various kinds of useful tool support become readily available:

- A task list can be generated automatically on the basis of a maintenance pattern, to guide the maintainer in carrying out the maintenance task foreseen by a designer (and expressed as a maintenance pattern).
- If a performed maintenance task is recorded as a maintenance pattern, the tool keeps track of the different program elements involved in the task and enforces the relationships between those elements, as specified by the constraints in the pattern.
- If some program element involved in a maintenance action can be deduced on the basis of the other elements involved in that task, the former element can be automatically produced.
- Various kinds of textual information can be conveniently associated with single maintenance actions. Such information can be used to guide the maintainer to carry out the maintenance action, or to store explanations of the maintenance action in a form that can be easily retrieved during the evolution of the system.

The rest of the paper proceeds as follows. In Section 2 we discuss maintenance patterns and their use in more detail. We give three concrete examples of well known maintenance problems and show how these problems can be solved using maintenance patterns. In Section 3 we discuss the JavaFrames tool which serves as our basis of the realization of maintenance patterns and show how to use the tool in applying maintenance patterns. Section 4 discusses maintainability concern architectures in general, while Section 5 considers maintainability concerns of JavaFrames itself. In Section 6 we present related work and finally conclude in Section 7.

2 Using maintenance patterns

In this section we first discuss the conceptual basis of maintenance patterns. Then we introduce three examples that apply maintenance patterns.

2.1 Maintenance patterns

A maintenance pattern consists of roles, associated with constraints and properties. A role has a type which determines the kind of program elements that can be bound to a role. For example, a class role can be bound to a concrete Java class. We assume that there are appropriate role types covering the basic Java concepts. In addition, a note role can be bound to an acknowledgment provided by the maintainer. A note role is used to associate informal instructions at certain points in the maintenance pattern.

Constraints are structural conditions that must be satisfied by the program element bound to a role. For example, a constraint of class role $P$ may require that the class bound to $P$ must inherit the class bound to another role $Q$, or that the method bound to role $R$ must be located within the class bound to role $S$. The properties include various additional specifications that are useful in practice, like a short (informal) description of the binding task or a default implementation for the program element to be bound (used for automatically generating the program element). A property can also define the multiplicity of a role, that is, the lower and upper limits for the number of the instances of the role. For example, if a method role has multiplicity $0..1$, the method is optional in the pattern, because the lower limit is 0.

We assume tool support which can generate a task list for a partially bound maintenance pattern. The task list displays a task for each role that can be instantiated and bound in the present situation, taking into account the dependencies (implied for example by the constraints) and the multiplicities of the roles. We also assume that the tool is integrated with a program editor and that the constraints are checked after every editing action. We will not explain in detail how the tool does all this; an account of the mechanisms is given in [12]. Every time a binding action is performed, the tool generates an updated task list (because new tasks will probably become doable). Thus, each binding action represents one maintenance action within the maintenance task managed by the maintenance pattern, and the task list generated...
by the tool provides a step-by-step instruction on how to carry out the maintenance task. Usually a binding action means that the maintainer should provide a new program element at a certain place, modify an existing program element, or simply check manually that the code satisfies certain requirements. The latter can be accomplished using note roles.

With this rough idea of a maintenance pattern in mind, in the next subsections we give three example situations where a maintenance pattern is applied. We will give a more detailed description of the maintenance patterns in terms of JavaFrames in Section 3.

2.2 Extending an existing system

As an example of adaptive maintenance, consider a typical situation in an object-oriented system where the system should be modified by providing a new implementation for a certain concept represented by a base class or an interface. This kind of maintenance task is often anticipated, and indeed maintenance may have been the driving concern in the design solution leading to the use of a base class or an interface.

The above situation is depicted in Figure 1. Originally, we have three classes: a base class Concept, a second class ConcreteImplA extending Concept, and another arbitrary class Client that instantiates ConcreteImplA. At some point, the maintainer of the system finds herself in a situation where she should provide a new specialization of Concept that replaces ConcreteImplA with ConcreteImplB. If the system is well-designed, references to ConcreteImplA (like instantiations) are isolated to few places, but these places may nevertheless be hard to locate. The designer should therefore document the places as a maintenance pattern (called, say, "New implementation of Concept"). Let us assume here that the only place where ConcreteImplA is referenced is the instantiate method of Client.

![Figure 1. Extending an existing system.](image)

Figure 1 uses a modified version of UML notation for patterns because our pattern concept differs slightly from that of UML. The patterns are denoted by ellipses. The tiny circles on the boundary of the patterns denote the roles that are associated with a maintenance pattern. For instance, the role bound to the superclass Concept suggests that there is a single code point (class Concept) played by the pattern role. The multiplicity of a role can be more than 1. The role referring to both ConcreteImplA and ConcreteImplB is an example of a role that refers to more than one code point playing the same role.

When applying the maintenance pattern in the situation described above, all required modifications are presented as simple maintenance tasks. The pattern has roles referencing the class Concept and the code points where its implementation class is instantiated. Another role is used to refer to the implementation class itself. When the user decides to change the existing implementation class by a new one, the pattern enforces the inheritance relationship between Concept and the new implementation class. In addition, the pattern warns the user to revisit the code points where the implementation class is used. For example, the pattern might ask the user to change the reference to ConcreteImplA in the method instantiate of class Client to a reference to ConcreteImplB instead.

2.3 Copy-paste programming

Newly-implemented programs are often based on earlier constructed ones. In this case, the programmer typically copies the older program and modifies it to suit the new purpose. In other situations, a new program may be constructed out of combining together several pieces of code extracted from a number of other programs. In both cases, the practice is to first copy and paste different pieces of code and then try to refactor the assembled elements. This coding technique is commonly used when maintaining existing systems as well, especially when the purpose of the maintainer is to adapt the system to a new environment or to improve some of the system qualities.

Copy-paste programming may add uncontrolled errors to the system. This is illustrated in the case where the original piece of code, i.e. the code which has been copied, has been modified in order to correct some errors. Most probably, all code replica need to be modified, too. However, it is not easy to know where all the copied pieces lie. Finding out these code pieces can be a painful and costly activity.

Copy-paste programming is identified as an AntiPattern called cut-and-paste programming [6]. It is considered as a bad programming style, and the best choice is to avoid it. However, copy-paste style of program development is sometimes unavoidable, because refactoring the code to eliminate all code duplications would lead to complicated and unnatural decomposition of the system. Thus, copy-paste occurrences are a kind of micro aspects of the system: they represent an implementation level cross-cutting
concern that should be made explicit in the system. From a maintenance viewpoint, the above practice can be seen as lacking control over the maintenance activity. There is no clear documentation on what and where modifications should be carried out. Maintenance patterns can be used to document and keep track of copied program elements.

![Copy-paste pattern for methodA](image1)

**Figure 2. Pattern for copy-paste programming.**

Figure 2 shows a maintenance pattern that manages the copy-paste style of programming. The pattern adds more control to the maintenance activity in the sense that the copied code (marked with dark color), is replicated and inserted in the right code points (marked with grey color) as specified by the pattern. The pattern has a role referring to the copied code and a second role referring to the code points where it has been pasted. When the copy-paste situation is identified, the programmer records this as copy-paste maintenance pattern, binding the roles to the concrete program elements. If the user modifies the original code, the pattern warns her to consider the replica by revealing their location. In some situations, a modified version of the copied piece is needed. In this case, too, the maintenance pattern contains the tracking information about the code pieces that are involved in the maintenance task.

Copy-paste programming is quite similar to implicit coding rules [16]. Implicit coding rules cannot be typically found in software documentations, instead, they are in the heads of experienced maintainers. An example of such a rule is that a certain global variable should be initialized before calling a certain routine. Violations of these rules are called bug code patterns. In the similar way, to handle copy-paste programming without violating the code the maintainer should know the occurrences of the copied code. Usually, such knowledge is not presented explicitly in software documentation.

2.4 Fragile base class problem

Let us assume a simple class inheritance situation, a derived class inherits from a base class. The fragile base class problem is the situation where modifications in the base class may cause the derived class to malfunction. The base class is said to be fragile because changes in it break its role as a base class. A thorough study of the problem is presented in [17]. Figure 3 shows an example of the fragile base class problem. A class library provides a class Container for storing objects. The class has two methods add inserting a new object in the container and addSet invoking the add method to add a set of objects to the container. Suppose that an application developer decides to extend Container. She derives a class CountedContainer introducing an instance variable counter and redefining the add operation to increment counter every time a new object is added to the container. At a later time, the library maintainer decides to optimize the class Container. She provides a new implementation for addSet without invoking add thinking that the new system is compatible with the previous one. Unfortunately, when the new version of Container is used as a base class for CountedContainer, the system returns the wrong number of objects since counter is not updated anymore.

![Pattern for changing the implementation of addSet](image2)

**Figure 3. Fragile base class problem.**

In this kind of situation, a maintenance pattern can be used to monitor changes in the superclass. The pattern has a role referring to the base class method and a second role referring to the subclass method being used by the superclass. Initially, the roles are bound to the concrete methods. If the maintainer modifies a referred method in the superclass, the pattern asks her to verify the related method in the derived class. The warning may take the form of an informal task description. Thus, possible problems related to the fragile base class case can be eliminated.

3 Maintenance patterns in JavaFrames

In this section, we first discuss the main features of JavaFrames tool. We then show how to present the maintenance patterns discussed in Section 2 and how to apply them using the environment.

3.1 JavaFrames environment

JavaFrames is initially a pattern-oriented task-based prototype tool for framework specialization. The tool has been used as a Java development environment and is currently
being adapted to support UML semantics [13]. However, the tool can be easily extended to support any other type of semantics.

The tool comes with several useful features. Three of these main qualities are:

**Simple task lists**: JavaFrames environment models a set of specification in the form of role-based patterns. When applying these patterns, the tool generates a number of tasks to be carried out in order to meet the specifications. By task, we mean a simple action that adds an element or enforces a property on the model. Tasks are kept simple enough so that their immediate result is easy to track and can be changed if needed. An example task is to provide a Java method.

**Adaptive documentation**: JavaFrames environment is able to record the history of the user’s tasks and informs her what to do next using the recorded information. The tool, for example, uses the concrete names of the Java classes specified earlier by the user when documenting next related tasks. The concrete names can be used both in the informal task descriptions and the generated default code.

**Integrated source editor**: The integrated Java editor keeps track of the application of the patterns and checks the constraints during editing on the fly: constraints are re-checked every time the source is changed and the user is asked to repair the violations.

The tool implementation of the general pattern concept in addition to the qualities mentioned above makes it possible to adapt the environment to support software maintenance activities. For this purpose, we treat JavaFrames patterns as maintenance patterns. A maintenance pattern then consists of several roles, each role corresponds to a code point we are interested in during the maintenance activity. Role properties define how the maintenance task should be performed and the inter-role dependencies show when these maintenance tasks should be carried out. Violated constraints are also considered as maintenance tasks.

In JavaFrames, patterns are usually put in a hierarchical structure defining a unified context. During a maintenance session, the pattern roles are either bound to new program elements, for example by creating a new Java class for a class role, or bound to existing program elements. The action of associating pattern roles with concrete program elements represents a maintenance task the user should perform. When a task is performed, new maintenance tasks might be generated.

### 3.2 Presenting maintenance patterns in JavaFrames

In this subsection, we discuss the JavaFrames specification of the Extending Pattern shown in Figure 1. This maintenance pattern consists of four roles: two class roles, a method role, and a code snippet role. The class roles represent the base class `Concept` and the class that inherits from it. The method role specifies the method `instantiate` in class `Client`. Finally, the code snippet role stands for the implementation of the method `instantiate`.

![Figure 4. JavaFrames maintenance pattern.](image)

![Figure 5. Textual representation of the pattern.](image)

Figure 4 shows the representation of the pattern in JavaFrames pattern editor. The left view displays the pattern roles and their constraints. The right upper view shows the dependencies of the selected role whereas the lower part exposes the properties of the role expressed as text templates. In order to clarify the structure of the pattern, we give in Figure 5 a more readable textual specification. Every role has a set of properties; the `taskTitle` property of the `ConcreteImpl` role for example, tells the maintainer what task should be performed. The definition of this property refers to `Concept` role in the form of `<#:/Concept.i.shortName>`, this tag refers to the concrete name of the superclass. The `inheritance` constraint
has value `<#:/Concept.i>` saying that any instance of the `ConcreteImpl` role (class `ConcreteImplA` and class `ConcreteImplB`) should inherit from instances of `Concept` role. The multiplicity symbol “+” that comes with `ConcreteImpl` means that there can be more than one program element playing the role `ConcreteImpl`. When no multiplicity is given, the role should be bound to exactly one program element, this is the case of the `Concept` role. Finally, the code snippet role `Instantiation` has default implementation that can be used by the maintainer resulting in a concrete action.

**Figure 6. Sample maintenance tasks.**

Figure 6 shows sample maintenance tasks to be performed when applying the maintenance pattern discussed above. In task 1, the user provides a new subclass `ConcreteImplB` inheriting from class `Concept`. The next task is to remind the user that the newly generated subclass has to be instantiated instead of the existing subclass. The environment generates default implementation and suggests the right class to be instantiated.

The other two maintenance problems mentioned in Section 2 can be handled with JavaFrames, assuming that the environment can detect changes in the source code. In order to achieve that, a new “unchanged source” constraint that is invalidated by any change in a bound element is needed. Support for such a constraint can be added to JavaFrames.

The pattern that manages copy-paste programming has two roles; a first role with a reference to the code to be copied and a second role with a reference to the program elements where the copied code should be pasted. There is a dependency from the latter role to the former role and there is an “unchanged source” constraint on the first role. When applying the pattern, the first task is to specify the original code (methodA in the example) whereas the remaining tasks are to provide the code points where to find the copied code (methodB, methodC, and methodD in the example). At a later time, when the maintainer decides to make changes in the original code, the “unchanged source” constraint is violated and the pattern warns the user to revisit the replica.

As for the fragile base class problem, the pattern has two method roles in addition to two class roles associated with the base class and the subclass. The first method role refers to the method of the base class (addSet in the example) whereas the second method role points to the method of the subclass (add in the example). There is a dependency from the first role to the second role since addSet invokes add. In addition, there is an “unchanged source” constraint on the first role. If the library maintainer performs changes in the base class method, the “unchanged source” constraint is violated and a task is generated warning her of possible problem in the subclass. The environment can suggest the user to add a copy of the base class method with the original implementation to the subclass.

From the discussion above, we see how JavaFrames environment can manage maintenance activities via simple task lists. The tasks are easy to follow and their actions can be visually controlled. The patterns are used to refer to the code points involved in the maintenance activity. In the task descriptions, concrete names such as `ConcreteImplB` in task 2 of Figure 6, are used. Moreover, default implementation required for the maintenance actions can be suggested. This offers the maintainer a stepwise adaptive maintenance environment. However, the number of maintenance tasks should be carefully planned since a huge number of tasks can annoy the maintainer and slow down the maintenance phase.

A related problem is how to manage the maintenance activity of a large system via small-grained maintenance patterns instead of using one huge macro pattern. In the next section, we propose a possible solution to such problems.

## 4 Maintainability concern architectures

In large real systems maintenance patterns can be overlapping and co-operate with each other as illustrated in Figure 7. Several maintenance patterns are connected to each other via the accompanying classes which require maintenance actions. Each time a maintenance action (described with a maintenance pattern) occurs, the connected maintenance patterns should be checked to verify whether these patterns produce any modification requirements. Thus, maintenance patterns support change impact analysis to find the points of the system where the maintenance task extends.

In more detail, Figure 7 shows the code points that
should be considered when maintaining a specific software system. Code points that are related to each other are grouped together under the same unit. This is marked in the figure by a circular shape. For example, the required modifications in ClassA and ClassB are tied together. The role of the maintenance pattern is to contain related code points. For instance, pattern P2 groups the code points in ClassA and ClassB. In some cases, a pattern needs information that is not concerned by the pattern. For example, P1 uses the information of the existing system ClassX in order to document the modification tasks required for ClassC, ClassD, and ClassE. A similar situation appears when dealing with P2: the pattern uses the information of ClassC which should already be generated by P1.

In this way, we can see how certain patterns may depend on other ones defining a partial order of applying the patterns. This can also be understood by the fact that in a typical maintenance situation, some maintenance tasks may depend on other ones. The result of applying the maintenance patterns on an existing system can take two forms. In some cases, the pattern adds a new element to the system. For example, pattern P1 adds the new class ClassC to the system. In other situations, the pattern may just modify an existing system element. For example, pattern P2 modifies class ClassA by probably adding a new operation to the class or by adding a new code snippet to an existing operation.

Building the system consisting of several maintenance patterns (as in Figure 7) can be regarded as a three phase process. First, modification points need to be identified. Second, related points should be tied up under the same pattern. Third, the relationships among the patterns of the system should be recognized to provide the documentation for the whole maintenance process.

In the building process, we can rely on earlier work about concern architectures [13, 14]. For example, in [13], it is shown how concern architectures might ease documenting framework specialization using patterns. When considering maintainability as a concern [15], we can talk about maintainability concern architectures. Typically, each concern tackles a specific area of interest in the maintenance activity. Concerns can, in turn, be composed of smaller units of interest. In this work, we treat each of these small units as a maintenance pattern.

Figure 8 shows the maintainability concern architecture of a specific software system. The architecture consists of three concerns X, Y, and Z. Individual concerns may further be decomposed into several smaller sub-concerns. Concern X, for example, is decomposed into two sub-concerns X1 and X2. Concern Z consists of two smaller units of interest. These two units are shown in the figure as two patterns. Concerns Y and Z are overlapping meaning that changes in concern Y, for example, might also affect concern Z. This means that the maintainer needs to pay more attention when maintaining such overlapping concerns. The architecture may also define inter-concern relationships. This is translated into dependencies when constructing the patterns. For example, in the right part of Figure 8, we see the pattern system that should be constructed; the dependencies between the pattern are shown using dashed arrows.

Nevertheless, a typical maintainability concern architecture does not cover all possible maintenance tasks of a system. In fact, maintenance patterns are mostly suitable for adaptive maintenance but are less used for managing other maintenance tasks such as correcting faults or improving system attributes. As a result, a number of code points may remain uncontrolled by maintenance patterns. However, even in this situation, maintenance patterns may prove to be useful. This is obvious when the execution of a maintenance action, that is not controlled by a pattern, breaks other pattern controlled decisions. For example, the maintainer may attempt to improve a certain attribute of a system by splitting an existing class into two new ones. If the original class is referred by a maintenance pattern, the binding between the corresponding pattern role and that class becomes unresolved. As a result, a warning for unresolved element is reported by the pattern.
5 JavaFrames maintainability concern architecture

In this section, we consider applying our approach on JavaFrames system itself. This can be considered as an example of adaptive maintenance.

The pattern engine, an integral part of the JavaFrames system, has been designed to support different domains. In JavaFrames terminology, we refer to these domains as semantics. In the same domain, there can be different semantics types. For example, in the Java domain, there are different semantics types for Java classes, Java methods, etc. Currently, the tool supports Java and UML semantics. However, it can be extended to support other semantics such as C++ or XML semantics.

Figure 9 shows the maintainability concern architecture of the JavaFrames system. Our interest in the system is from the angle of adapting it to new semantics. The figure defines three concerns:

- Semantics Creation Concern: encapsulates the aspects of creating a new semantics type. It consists of three overlapping sub-concerns that have one pattern in common. The pattern specifies how to create a new semantics type. The three sub-concerns, in turn, are:
  - Synchronization Concern: synchronizes the pattern model with the real world of concrete elements. It is composed of a listener pattern that specifies how to listen to changes in the real world that may affect the current role bindings. Let us suppose, for example, that a Java class role is bound to Java class A. If class A happens to be renamed or deleted, the associated binding should be broken.
  - Validity Checking Concern: deals with defining possible constraints on the newly created semantics type. It consists of a pattern that generates code for constraint implementation. An example constraint is enforcing the right access permissions of files.
  - Structure Generation Concern: abstracts the properties of the newly created semantics type. It defines a pattern that guides the user how to implement new templates for new type. An example template is the directory information of files.
- Pattern Functionality Concern: defines rules how to extend the existing pattern functionality so that newly generated semantics types are supported. The enclosed pattern, for example, specifies the steps required for creating the right pattern role when reading XML representation of the corresponding type.
- User Interface Concern: encapsulates the steps required for generating wizards for providing concrete elements that play the role of the new type. The 'Wizard Sheet' pattern specifies steps how to define new wizard sheets whereas the 'Sheet Implementation' pattern defines tasks to provide concrete implementations of the wizard sheets.

Figure 10 shows an example maintenance session for adding a new semantics type to the set of existing types. The left view of the figure shows the hierarchical structure of the deployed patterns. The patterns have been identified in the maintainability concern architecture presented in Figure 9. The middle bottom view shows the currently performed bindings. The right view presents the maintenance task to be performed. The upper part of the view indicates the title of the task whereas the bottom part displays a detailed task description. The upper view shows the integrated editor which displays the outcome of performing the maintenance task.

6 Related work

In this section, we investigate related approaches that try to solve maintenance problems at both the methodology and the tool support levels. We start by discussing related methodologies, and then compare our environment with several related maintenance tools. After that, we consider maintainability from the point of view of aspects. Finally, we compare maintenance patterns to related approaches.
6.1 Work related to maintenance patterns

There is a lot of work done concerning different kinds of patterns. Technically, maintenance patterns resemble design patterns [10], but they are usually system-specific. Although our maintenance patterns may emerge in the design phase, their purpose is to support software maintenance. There are other patterns occurring especially in the maintenance phase. AntiPatterns describe bad programming styles that can be found during maintenance and that should be replaced with better styles [6]. In addition, there is ongoing work concerning identification of design patterns afterwards during software re-engineering or maintenance [7].

Our maintenance patterns are related to re-engineering patterns [8]. Re-engineering patterns provide guidelines to some reverse engineering and re-engineering tasks. They describe common situations that may occur during re-engineering and give advice how to manage these problematic situations. Compared to maintenance patterns, re-engineering patterns are meant to be in post-delivery situations of software, they cannot be applied in the design phase like maintenance patterns. In addition, re-engineering patterns are usually common to all systems whereas maintenance patterns are more system-specific.

Concerning maintenance, there are also lifecycle maintenance patterns or evolution patterns [4]. However, lifecycle maintenance patterns are not system-specific, because they describe typical evolutionary changes that are common to several software systems. Furthermore, lifecycle maintenance patterns try to find common general rules how systems evolve, while re-engineering patterns describe problematic situations that might occur during system evolution and present possible concrete solutions.

6.2 Work related to maintenance tools

There are several maintenance tools for different purposes. Some tools are meant for searching interesting code pieces (for example [19]). There are also tools that enable program representation as hypertext (for example [9, 18]). In these tools, for example, a function name (in function call) provides a link to the corresponding function implementation.

As shown earlier, we have used JavaFrames for documentation purposes needed in software maintenance. However, maintenance tools supporting documentation are typically such that they produce new documentation (e.g. different graphs) according to source code. Their purpose is to clarify the dependencies between different code elements. Together with these graphs, the program becomes easier to understand. The JavaFrames approach is different, because documentation is needed in order to link together code pieces that belong to the same maintenance task. Thus, concerning the provided links, JavaFrames (or actually the maintenance patterns) has similarities to such tools that include the hypertext feature.

There are tools that support change management by locating features in the program code [24]. These features can be domain-specific concepts or functional properties of the system. To make modifications to the features, their location should be identified in the code. Some of these tools, such as presented in [11], apply a static approach. Tracing of related code pieces is based on graphs telling the dependencies between program elements, for example, in procedure calls and data flow.

Searching for feature location can also be implemented dynamically [23, 25, 26]. Searching for the code points where a certain feature is implemented is based on running the program to be changed. The user may run a set of tests that are divided into invoking tests and excluding tests [23]. The execution traces are analyzed to find the code components that were executed in the invoking tests but not in the excluding tests. The set of tests must be selected very carefully, because poorly selected tests will lead to inaccurate selection of the components.

A program feature may correspond to a maintenance task, if a maintenance task is a modification to an existing feature. To complete the task, the involved code pieces should be known. In the aforementioned tools, this identification is performed afterwards and repeated when necessary. In JavaFrames (and especially via maintenance patterns), the documentation, preferably made during software design, include the information that the other tools try to find. Thus, the task of the other tools can be considered as some kind of maintenance pattern mining.
6.3 Work related to aspects and concerns

Aspect-oriented software development (AOSD) supports several concerns that cut across the most evident decomposition of the system [2]. A typical system consists of several kinds of concerns such as business logic, performance, data persistence, logging and debugging [15]. In addition, it is possible to detect development-process concerns, for example, comprehensibility, maintainability, traceability, and evolution ease. In this paper, we are especially interested in maintainability concern.

Maintainability can be considered as a concern. However, at a finer-granular level, we can consider individual maintenance tasks as crosscutting concerns in the system to be maintained. We use maintenance patterns to recognize and manage such concerns. Concern architectures are considered in [13, 14]. In the terminology of [13], maintainability can be managed by identifying maintenance patterns and their relationships. The pattern relationships in [13] are used to define a partial order for applying the patterns. In addition, the relationships identify the parts where modifications to program elements referred by a pattern can affect other elements bound by the overlapping patterns.

Maintenance patterns are closely related to concern graphs [20]. In concern graphs, related program features (or code pieces) form a concern that is represented as a graph. Code pieces forming a concern belong to the same maintenance task. In the same conceptual way, maintenance patterns collect together the related pieces of source code. However, concern graphs differ from maintenance patterns in that concern graphs are extracted from the existing source code repeatedly, and they are not permanent like the information in maintenance patterns. Moreover, concern graphs do not support maintenance documentation.

6.4 Comparison of maintenance patterns to related approaches

The tools presented in Figure 11 are related to each other, because they all have the common purpose, i.e. change management. Our approach is referred in the figure as maintenance patterns (MP). Most of the methods support identification of feature locations. Maintenance patterns are mostly related to concern graphs, because both methods support concerns. We can consider concerns as a mechanism that brings and links together associated pieces such as patterns in our approach. In this standpoint, hypertext links can serve the same purpose, and thus, the HyperSoft tool [18] has also been checked off in that row.

The other considerations discussed in previous subsections are also collected in Figure 11. Maintenance patterns can be characterized such that although their purpose (change management) is similar to other tools, they support the purpose in a different way. For example, they provide permanent information about related code points, whereas other tools extract information repeatedly. Moreover, maintenance patterns support maintenance consideration already in the design phase.

7 Conclusions

In this paper, we have introduced maintenance patterns to support software documentation especially for maintenance purposes. Maintenance patterns aid in linking together those program parts that need modification due to the same larger maintenance task. A program piece may belong to several maintenance tasks creating a link between the corresponding maintenance patterns. Documenting these links support change impact analysis.

We show how maintenance patterns can be applied using JavaFrames even though JavaFrames is originally not built as a maintenance tool, but is meant for application development purposes. For example, the task list property supports maintenance tasks directly, because maintenance tasks can typically be divided into smaller tasks that can be performed in a certain order.

So far, we have only limited experience in applying the approach presented in this paper to real maintenance situations. A real case study is the next action of future work, and for this purpose we have tool support available. In this way, we can find out how appropriate and practical the discussed approach is and how it scales up to real world examples.

As another future work, maintenance patterns can be used to provide information about maintenance history. We can collect useful information about those code points that need recurring modifications and/or those code points that belong to several maintenance patterns. It can be assumed that typically there are code points that meet frequent change actions and others that are more stable. Via maintenance patterns, we could find these different points. In the future, there could even be a property which can be used to visualize maintenance patterns and the evolution of the system. Another future research direction is to investigate how
the use of maintenance patterns could ease system testing. We can assume that using maintenance patterns it is possible to select the test cases more precisely. Particularly in regression testing, we need not test such parts of the system that are not involved in the implemented change.

Acknowledgments

This work is funded by the National Technology Agency of Finland (Tekes), Nokia Mobile Phones (NPM), Nokia Networks (NET), Plustech, Almare, Profit, and the Academy of Finland. The authors would also like to thank Kai Koskimies, Tommi Mikkonen, and Mika Katara for their comments.

References

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
Publication 6


Copyright © 2004 IEEE.
Reprinted, with permission, from the proceedings of IWPC 2004.
Tool-Supported Customization of UML Class Diagrams for Learning Complex System Models

Imed Hammouda  Olcay Guldogan  Kai Koskimies  Tarja Systa
Institute of Software Systems, Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
Nokia Corporation
Tieteenkatu 1, 33720 Tampere, Finland
{imed, kk, tsysta}@cs.tut.fi, olcay.guldogan@nokia.com

Abstract

To employ an existing software library, its structure should be first learned and the required elements should be identified. This can be challenging if the library is large and only a specific part of it should be comprehended. In this paper, we study the problem of learning complex software libraries modeled in UML. It is argued that the learning process can be supported with a tool environment that allows the customization of the UML model according to the context of the learner, stepwise and dynamically chosen learning tasks, and focusing on a particular learning concern at a time. We show how such an environment can be achieved based on the concept of a pattern, using existing tool support. We demonstrate the idea with a part of Symbian platform architecture. The approach is evaluated in a case study where a pattern-driven learning environment is constructed for JPEG interchange file format specifications.

1 Introduction

Learning complex systems is a challenging task. The learning curve often depends on what experience the learner has and how familiar she is with the system domain. Beginners (for example students), in particular, may face serious problems in understanding the details of complex systems like various software platforms, standard API’s or formats.

The understandability of a complex system can be improved by graphical representations. A good candidate for the graphical description of software-related systems is the Unified Modeling Language (UML)[20], which is a widely used standard design notation and thus assumingly familiar for most software engineers. Such a graphical description effaces unessential technical details like the intricacies of a particular textual syntax or implementations of methods, and shows the abstract relationships between the different parts of the system. In this paper we assume that the system description is given as a UML class diagram.

The problem is, however, that even the purely logical, essential information can be overwhelming for a particular person who is interested in applying (and understanding) the system only in her own, limited context. If we could somehow show this person only that part of the system description (e.g. a UML model) which is relevant for her purposes, the target of the learning process would be greatly reduced.

On the other hand, it is well-known that the comprehensibility of a system is mainly determined by the possibility of studying the system one part at a time [15]. Such a part can be a component, a subsystem, or even an aspect [1]; the important thing is that the learning task can be divided into pieces which make sense from the learner’s viewpoint. This makes it possible for the learner to focus on one issue at a time, and to restrict herself to those issues that are relevant for her.

The set of potential learning issues can be anticipated beforehand, but the individual learner eventually decides which issues to focus on. However, since the selection of the relevant topics itself requires certain understanding of the system, we cannot assume that the learner is able to make deterministic choices concerning the learning issues in the beginning of the process. Instead, the learning process should be interactive, allowing the learner to select the learning paths dynamically as she understands the system better.

Separate learning issues typically contain sequentiality; certain pieces need to be learned before the other ones can be fully understood. For instance, to learn how to use a
certain feature provided by a software library, the engineer needs to understand how its individual parts relate to each other and in which order the individual implementation steps should be taken. Software documentation seldom provides that kind of support.

Thus, our research problem is the following: what kind of technique would support a learning process that (i) is based on the UML model of the target system, (ii) allows the customization of the UML model according to the needs of the learner, (iii) allows the learner to make dynamic choices concerning the subjects to be learned, and (iv) supports sequentiality in specific learning concerns.

We argue that these requirements can be satisfied technically using so-called patterns. In this context, a pattern is a specification of a configuration of modeling elements. Thus, a pattern is a generic concept that can be used to present the structure of virtually any collection of logically related (software or modeling) elements. We argue that a learning issue can be conveniently specified as a pattern on the UML model of the system. In the sequel we will use the term comprehension pattern for such patterns.

Separation of concerns (SoC) can be used to factor out different aspects (concerns) of a model [5]. We propose the use of SoC to decompose and organize the tasks required for learning a complex system into separate comprehensibility concerns. Such a concern covers all the model elements relevant for learning a certain part or aspect of the system. We will represent a comprehensibility concern as a set of comprehension patterns, where each comprehension pattern takes care of a more limited learning issue pertaining to the concept. We will show in this paper how an existing tool providing generic pattern support in UML environment [12] can be used to implement the idea of comprehension patterns and comprehensibility concerns in a way that satisfies our requirements (i) - (iv). Additional benefits for comprehension patterns offered by this tool include:

- A task-driven learning environment: The tool generates a task list on the basis of comprehension patterns, to be carried out by the learner. This kind of stepwise learning-by-doing paradigm makes the learning process more natural.
- Immediate feedback: When the learner makes a mistake, the tool immediately generates a new task for correcting the mistake. Usually, the learner is provided with tool-supported repair functionality.
- Informal explanations: Comprehension patterns can be augmented with informal explanations concerning the roles of the model elements in a pattern. Such explanations can be shown by the tool when the learner focuses her attention to a particular element.

In order to demonstrate our approach we use the MADE tool [12] to model a collection of comprehension patterns for the JPEG interchange file format specifications [18].

The remaining of the paper is organized as follows. In Section 2 we discuss comprehension patterns in more detail. Section 3 gives an introductory example of the application of our approach in the case of Symbian OS architecture. In Section 4 we present the tool platform we are using, and in Section 5, we briefly explain the JPEG interchange file format specifications and show how comprehension patterns and comprehensibility concerns can be identified and used in this domain. Related work is discussed in Section 6. Finally, we conclude with some remarks concerning future work in Section 7.

2 Comprehension Patterns

A pattern is an arrangement of software elements for solving a particular problem. Depending on the nature of the problem, we may speak of analysis patterns [9], architectural patterns [3], design patterns [3], coding patterns [3] etc. 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.

In this work, we introduce a concrete pattern concept called a comprehension pattern whose purpose is to facilitate the learning of existing systems. Comprehension patterns are used to represent learning tasks in a form that makes it suitable for tool support. In the case of a comprehension pattern, binding a role to a model element is considered as a learning task. In this way it becomes clear to the learner which model elements play certain roles in the system. Assuming that each binding task is associated with informal explanations of the role to be bound (as is the case in our tool environment), the learner understands the pur-
pose of the related model element. Each binding task can be attached with a description detailing its objectives and the way it should be carried out.

We assume that each role in a pattern can be associated with the specification of a default model element, so that this model element can be generated and bound to the role automatically when the binding is requested, if the learner wishes. This is a key property in the tool, because it allows the generation of a customized version of the system model during the learning process, according to the learning paths the user has taken. Basically, the default model elements constitute roughly the complete system model. We will study the principles of pattern construction in the next section.

When modeling a comprehension pattern, we distinguish between two kinds of roles. Roles, which represent model elements existing in the original model (called original model roles) and roles that stand for possible model extension (called model extension roles). An original model role may have cardinality ‘1’ or ‘0..1’. A ‘1’ cardinality means that the model element represented by the role is required and must be present in the customized model. A ‘0..1’ cardinality signify that the corresponding model element is optional. Thus, if the role is not relevant from the viewpoint of the learner, she may ignore the role, which will have the effect that all the other roles which depend on this role through their constraints will not appear in the task list. For example, assume that a particular association end in the UML model has multiplicity ‘0..1’. If the learner knows that in her context there will actually be no instances of the class involved, she will ignore the corresponding role, with the effect that the class will never appear in the generated customized system model. More importantly, other classes depending only on uninteresting classes will not appear in the model, either. Thus large portions of the model that are not interesting for the learner can be omitted in the produced customized model. A model extension role, on the other hand, may have cardinality ‘1’, ‘0..1’, ‘1..*’, or ‘0..*’. Compared to original model roles, there can be more than one model element played by a model extension role. The reason is that the learner can provide different kinds of extensions for the same extension point.

In software engineering, separation of concerns (SoC) refers to the ability to identify those parts of software 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. In fact, SoC has been applied to different kinds of concerns ranging from business logic, performance, and security to other development-process concerns, such as comprehensibility, maintainability, and traceability. In this work, we are particularly interested in the comprehensibility concern. Enhancing system comprehensibility can facilitate the learning process in complex domains since these domains are usually composed of numerous parts and complex interactions between these parts.

3 Example: Learning the Symbian Architecture with Comprehension Patterns

Symbian [19] is an open standard operating system for data-enabled mobile devices such as PDAs and smartphones. The standard architecture comes with a large number of components providing support for a wide range of applications and functionality. Figure 1 shows a UML class diagram describing an overview of the Symbian communications and networking facilities, which is a core part of the Symbian platform architecture. Every class represents a component within the architecture. In overall, the architecture covers four different application types: message applications, web applications, connectivity applications, and comms applications. More detailed information about each application type can be found in [19].

![Figure 1. Original model](image)

From the application type viewpoint, the architecture in Figure 1 can be decomposed into four main parts: each part represents the components that are relevant to each of the application types we have previously identified. Furthermore, it seems natural to define a separate comprehensibility concern for learning each of these four model parts. Therefore, we identify four comprehensibility concerns: messaging, connectivity, web, and comms (communications). The messaging comprehensibility concern, for example, covers all the model elements relevant for learning the messaging aspect of the platform.

The Symbian messaging comprehensibility concern can be further encapsulated by two comprehension patterns: The first pattern takes care of learning the elements that are shared by other application types such as the ETEL tele-
phony server, whereas the second pattern represents the elements specific to messaging only, such as "Message type user interfaces". The two patterns define roles for these model elements and constraints on their relationships. For example, there is a model extension role for extending the original model element "Message app". This role has cardinality '1..' since the user can specify any number of own messaging applications types. The pattern defines another role for the "Message type user interfaces". In addition, the pattern defines a constraint on the uses relationship between the two roles meaning that the element bound to the model extension role uses the element bound to role "Message type user interfaces".

1. The concerns that are not relevant to messaging applications are dropped out. In our example, the elements that are only relevant to connectivity, web, and comms are left out.

2. Within the messaging concern, the elements that are not relevant to the specific application configuration are also dropped. The "Internet Protocols" component, for example, is dropped since it is relevant to Email messaging but not to SMS messaging.

3. The optional model elements that are relevant to our specific application configuration are enforced. Even though the use of the "C32 Serial Comms Server" (used for serial communications) is optional, the user thinks that every device should support this communication type. We, therefore, enforce its use in the customized model by changing the multiplicity of the corresponding association end from '0..1' to '1'.

4. Using concrete names for the model elements that are application specific can enhance the understandability of the platform model. In the customized model, the messaging application component "Message app" is replaced by a new component named "My message app".

5. The customized model can extend the original model by defining new elements that correspond to possible extensions in the platform. According to the Symbian specification, clients can provide custom extensions to the "ETEL Telephony Server" component. This is illustrated in our example customized model by the new element "My Telephony Server".

4 Tool Platform: MADE

MADE [12] is an experimental platform for pattern-driven UML modeling. The platform itself is the result of the integration of a number of different existing tools. JavaFrames [10] and Rational Rose [16] represent the key components of the integrated environment. JavaFrames is a pattern-oriented task-based development environment built on top of the Eclipse [7] platform. The pattern concept implemented by JavaFrames is an instance of the generic pattern concept discussed previously. Rational Rose is a UML...
modeling tool widely used by the industry. The communication between JavaFrames and Rational Rose is achieved through a UML model processing platform, xUMLi [2], providing a tool-independent API for accessing the UML models.

The MADE environment introduces patterns in the UML domain. MADE allows the specification of patterns and the binding of the roles of a pattern to UML model elements. The types of the roles are thus element types (metaclasses) of UML; for example, there are class roles, attribute roles, operation roles etc. The roles can be associated with various kinds of constraints on UML model elements. The constraints may refer to the elements bound to other roles, thus creating implicit dependencies between the roles. As an example, a naming constraint of a role may require that the name of the element bound to another role must appear in the name of the element bound to this role, according to a particular convention given as a regular expression.

The main contribution of the pattern tool [10] is that it maintains a dynamic task list for a partially bound pattern, indicating the binding tasks that are possible to carry out at a particular point of time. Recall that the action of binding a role to an (automatically generated) UML model element is here considered as a learning task. The tool keeps track of the mutual dependencies of the roles, reevaluates the constraints after each binding action, and updates the task list. Typically, the execution of a task generates new mandatory or optional tasks. The tool smoothly integrates patterns with UML models, so that the designer is free to edit the UML models as desired: if some of the constraints of a pattern bound to the model is broken as a result of an editing action, the tool generates immediately a new task to repair the broken constraint (and in some cases even an automated repairing option). The dynamic task list, integrated with the UML editor, allows the learner to proceed in a stepwise manner, understanding and producing the customized model one step at a time, and making editing experiments with the produced model.

The MADE pattern concept allows the specification of a partial order for the roles, implying that the binding of some roles must be carried out before certain other roles can be bound. This is used in our work for defining the order for individual learning tasks. The partial order is obtained simply by defining explicit dependency relationships between certain roles.

Another technically fairly simple, but in our context important contribution of the pattern tool is that one can associate informal instructions with each role, displayed when the binding task of the role is selected from the task list. In our work, we exploit this feature for informing the user about the meaning of the role (or the element bound to the role) from the viewpoint of the learning issue covered by the pattern. The tool also allows the highlighting of the model elements bound to the roles of a particular pattern, allowing the learner to easily see the essential parts of the learning issue. Finally, an essential property of the tool environment from the viewpoint of comprehension patterns is the possibility to associate a default UML model element for each role, to be automatically generated and bound to the role when the user performs a binding action (that is, a learning task).

5 Case Study: JPEG File Formats

In this section, we apply our approach to develop a learning environment for a software library representing the JPEG file format specifications.

5.1 Introduction to JPEG File Formats

Image and Video file format related software implementations have been widely used in practice. These systems are mostly complicated due to the underlying technical standards and specifications. Customizability and extendibility are desired characteristics for such systems, since they cover a broad range of considerations on the related field. In order to learn these systems, one has to learn their uses, considerations, and design models of the technical specifications. One of the well-known file formats for compressed images is JPEG (Joint Photographic Experts Group) [18]. JPEG is an image compression standard issued by ITU-T and JPEG committee of ISO in 1992.

![Figure 3. The JPEG file formats](image-url)

JPEG compression processes cover various modes and methods for lossy and lossless compression in order to support a wide application scope. However, this wide coverage makes JPEG a complicated standard. Various file format specifications followed the JPEG standard to provide interoperability and simplicity of digital representation in practice.
JFIF (JPEG File Interchange Format) [11] is a relatively simple and widely used file format. EXIF (Exchangeable Image File Format) [13] and DCF (Design rule for Camera File system) [17] are mostly used by digital cameras for providing image-specific ancillary data. For simplicity and clarity of the examples in this paper, we refer to the EXIF file format containing only JPEG compressed data. Figure 3 illustrates the relationship between JPEG and related file formats.

5.2 Comprehensibility Concerns for JPEG File Formats

Based on the relationships presented in Figure 3, we have derived a comprehensibility concern architecture for JPEG file formats shown in Figure 4. Comprehensibility concerns are represented by circular shapes. Patterns are denoted by rectangular shapes whereas the arrows depict the inter-pattern dependencies. The dependencies show the order of the larger learning tasks. The following four concerns are illustrated:

- **JPEG Format Concern** represents JPEG digital image representation in the interchange format as defined in the technical specification. It is composed of six patterns: 'JPEG Core', 'JPEG Baseline', 'JPEG Extended', 'JPEG Lossless', 'JPEG Hierarchical', and 'JPEG Image Extension'. 'JPEG Core' represents the core structure common for all JPEG images. 'JPEG Baseline' pattern deals with the simplest JPEG file format, which is supported by every JPEG related application. 'JPEG Extended' pattern is responsible for representing additional lossy compression processes in the file format. 'JPEG Lossless' and 'JPEG Hierarchical' patterns are not significant for the example in this case. 'JPEG Image Extension' pattern defines extensions and/or restrictions for enhancing or adapting the file format for particular applications, such as a newer version of EXIF format.

- **JFIF Format Concern** abstracts the JFIF file format, which is based on JPEG file format. JFIF file format mainly defines basic restrictions and extensions on the JPEG file format. A JFIF image may include a thumbnail - a small copy of the original image - as well as the main image data. Thus, this concern is composed of two new patterns in addition to four JPEG patterns. 'JFIF Image' pattern handles the representation of the pre-defined restrictions and extensions on the underlying JPEG pattern. 'JFIF thumbnail' pattern is similar to the 'JFIF Image' pattern, however it doesn’t define any extensions.

- **EXIF Format Concern** encapsulates the EXIF file format representation and related EXIF data structures. EXIF is a more complicated file format specification than JFIF. Similar to the JFIF concern, this concern contains two patterns for thumbnail ('EXIF Thumbnail') and main image data ('EXIF Image') in addition to three JPEG patterns. 'EXIF Image' pattern is mainly responsible for the primary image and associated ancillary data representation in the specified structure. 'EXIF Thumbnail' pattern handles the thumbnail image representation with the associated ancillary data. 'EXIF Image' pattern may use the 'EXIF Thumbnail' pattern, where the image contains thumbnail image.

- **DCF Format Concern** contains three new patterns, which basically extend the EXIF-specific patterns, in addition to the EXIF patterns. 'DCF Basic Image' pattern is the significant pattern, which handles DCF main image and related restrictions and extensions. In contrast to JFIF and EXIF, DCF defines two types of thumbnail images. 'DCF Basic thumbnail' is similar to the aforementioned thumbnail images, and is contained inside the DCF image file format. 'DCF Thumbnail' is a stand-alone file format, which is independent from the main image.

5.3 EXIF Image Comprehension Pattern

Figure 5 shows a textual representation of the 'EXIF Image' pattern modeled in JavaFrames. For brevity, the figure describes a small part of the pattern structure. The left section displays a number of roles along with their constraints.
and cardinalities whereas the right section entails other role properties. Role and constraint names are shown in bold font, role and constraint types are displayed in regular font, and cardinalities are specified in parentheses. The EXIFApplicationSegment role, for instance, represents a class role that stands for the application segment storing EXIF specific information. It has cardinality '1' since an EXIF image must have an EXIFApplicationSegment. The inheritance constraint associated with this role indicates that an EXIFApplicationSegment is an ApplicationSegment.

In addition, every EXIFApplicationSegment should have a constant identifier allowing the identification of an EXIF application segment. This is encapsulated by the UML attribute role EXIFIdentifier. The UML Association role ('contains') represents the composition relationship between the elements EXIFApplicationSegment and EXIFImage since an EXIF image stores an EXIF application segment. The relationship is further enforced by constraining its participants. This is shown in Figure 5 by the two constraints participantA and participantB. Furthermore, the figure shows a UML class role 'GpsIfd' with cardinality '0..1'. Such cardinality means that the element is optional. In our example, an EXIF application segment might or might not contain an IFD (Image File Directory) for recording GPS information.

As we have mentioned earlier, JavaFrames transforms the pattern specification into a task list. Figure 6 shows two learning tasks. Every task is associated with a task title and a task description as specified in Figure 5. A red dot (filled) marks a mandatory tasks whereas a white dot denotes an optional task. The upper task stands for providing an association for the UML association role specified in Figure 5. The other task represents the 'GpsIfd' role in Figure 5. This task is optional since the cardinality of the role it represents is '0..1'. In addition to displaying the tasks to be performed, the environment shows the learner how many mandatory and optional tasks are left to complete the actual customization session. This is shown in the title section of the figure.

5.4 An Example Customization Session

Figure 7 shows an example customizing session for learning the JPEG interchange file format specifications discussed in this paper. The patterns which have been identified in section 5.2 are shown in the bottom left view of the Figure. In the same view, the environment allows the user to choose between different JPEG concerns. Each concern corresponds to a specific set of features. The user, for example, might choose to learn the features of the EXIF image file format with or without the thumbnail information.

The learning tasks and their detailed description are shown in the bottom right view of the figure. Each task corresponds to understanding the role of a specific element in the standard. Figure 7 shows one mandatory task and two optional tasks. When a task is carried out (by double clicking on the task title), the corresponding model element is added to the system model and new tasks might be generated. The tool keeps track of all task information. This is important when the user wants to retrieve the description of the elements that have been constructed earlier. The recorded tasks are shown in the bottom middle view of the environment. Finally, the upper part of Figure 7 shows the customized model.

In a typical scenario, the learner chooses the part of the model she wants to customize/learn by selecting the corresponding pattern from the 'Concerns/Patterns' view. The learner then reads the available task descriptions and decides which learning step to take (in case more than one task is displayed). In the case of Figure 7, she might choose to provide the required model element 'IfdEXIF' by double clicking on the upper most task title. As a result, a new
UML class named 'IfdEXIF' is added to the Rose view. The learner might proceed by generating more model elements. If the learner decides to quit the learning session, she has to save the generated model. She can later proceed with the session since all session information is automatically saved and recovered by the environment.

Using the tool, it is possible to highlight the elements that correspond to a specific pattern or concern. Figure 7, for example, shows in blue (darker) color the elements corresponding to the EXIF image information as specified earlier by the 'EXIF Image' pattern in the EXIF concern. Using this feature, the logical structure of the customized model can be retrieved at any phase during the customization session. This might enhance model comprehension.

5.5 Experiences

Figure 8 shows a comparison between the original model of the JPEG library structure versus several customized models with respect to the model elements that have been considered in the case study. The customized models represent different concerns in the library. Depending on the selected concern, the customization resulted in a reduced number of model elements. For example, this number is reduced by 50% when the user focuses on the JFIF related features.

In order to evaluate the JPEG learning environment presented in this work, we have requested the feedback of two experts in the field of image processing (but not familiar with JPEG file formats structure): The first is a member of an image processing group at Tampere University of Technology and the second works for a telecom company. The questions that both learners had to answer fall into three main categories: the positive qualities, the negative aspects and the missing features of the environment. The main positive experiences were:

- Using our approach, it was easier to understand and memorize the elements and the various relationships in the model. The "learning by doing" approach, in particular, seemed useful and contributed to model comprehensibility.
- From an industrial point of view, our methodology can be useful for companies that are interested in the same software components from different technical perspectives.

The main critics of the methodology were:

- Learning tasks should be carefully explained and the descriptions should be detailed.
- The order of the learning tasks should be carefully planned since there can be different paths when constructing models, for example the top down versus the bottom up approaches.
- Clustering the model elements that belong to the same concern into one abstract element can be a further enhancement of the approach. It should be then possible to expand this element to view the inner details.

Identifying comprehensibility concerns, and thus comprehension patterns, for learning a software system depends heavily on its architecture. If the architecture clearly separates different concerns, the comprehension patterns may at least partly follow those concerns. In this case the comprehension patterns may be easier to find, but on the other hand there is less need for comprehension support. Comprehension patterns are especially valuable for systems with cross-cutting concerns that are not directly reflected by the architecture. Once identified, it is a relatively easy task to model the patterns in MADE. For this, the roles and the properties

<table>
<thead>
<tr>
<th>UML Model</th>
<th>Classes</th>
<th>Associations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Model</td>
<td>70</td>
<td>50</td>
</tr>
<tr>
<td>JPEG Baseline</td>
<td>27</td>
<td>19</td>
</tr>
<tr>
<td>EXIF without thumbnail</td>
<td>49</td>
<td>37</td>
</tr>
<tr>
<td>EXIF with thumbnail</td>
<td>55</td>
<td>43</td>
</tr>
<tr>
<td>Whole JFIF</td>
<td>32</td>
<td>24</td>
</tr>
</tbody>
</table>

Figure 8. The original model versus several customized models
6 Discussion and Related Work

6.1 Program Comprehension

Software comprehension can be seen as a task of building mental models, i.e., internal working representations of the subject software, of the underlying software at various levels of abstraction. In addition to static entities (such as text structures or plans), the mental models also contain dynamic entities: a dynamic entity might be a sequence of actions while following a plan to reach a particular goal [21]. Activities and approaches supporting software comprehension include software visualization, documentation generation, and reverse engineering. The goal of reverse engineering is to construct abstractions and design information from the subject system [4]. Tools supporting software visualization and reverse engineering often visualize the subject system using a graphical notation but rarely use the same notation as used for software design, namely, UML in the case of object-oriented systems. In addition, the importance of the dynamic entities of cognitive models is often overlooked in software comprehension tools. In the approach presented in this paper, the learner follows a step-by-step process that helps her in learning a specific learning issue.

Von Mayrhauer and Vans [21] point out a common inadequacy in several cognition models supporting program comprehension they compared: "Most models assume that the objective is to understand all the code rather than a particular purpose. While these general models can foster a complete understanding of a piece of code, they may not always apply to specialized tasks that more efficiently employ strategies geared toward partial understanding". In our approach, partial understanding is supported by dividing the model to be learned into learning concerns and further into comprehension patterns.

Erdem et al. propose a method and tool called MediaDoc to generate tailored software explanations [8]. Task and user models are first constructed by studying a series of user questions while performing a certain task. These models are then used in various ways to generate explanations tailored to user’s task and expertise. The user can pose queries either by directly entering a query or by interacting with MediaDoc. MediaDoc either generates an explanation that directly answers the query, or reformulates the query as a new subtask. One of the motivating observations when developing MediaDoc was the importance of dialogues with experts when learning a system, even if a good documentation is available. One reason for this is that the dialogue between the learner and the expert facilitates the refinement of the questions [23]. In our approach, gathering the set of potential learning concerns is assumed. This could be done, e.g., by studying a series of user questions as done in MediaDoc. The clarifying learning issues, e.g., issues that need to be first understood, and the sequentiality of the learning process is captured using comprehension patterns. The expertise of the learner is not taken into account. Instead, the system models are customized based on the learning task only. Also, we do not allow the user to pose free-form textual questions, but rather aim at structuring the learning concerns and tasks in a manageable and well-defined way and mapping the directly with them system models under examination.

6.2 Learning Complex Structures

The approach presented in this paper represents an active learning environment for learning complex domain models. Active learning has been defined as the process of continuous and active construction and reconstruction of experiences [6]. The comprehension patterns in our methodology encapsulate the learning experience in the form of a set of learning tasks. The comprehension patterns themselves can be updated when more knowledge about the domain is obtained. Learning is then carried out by constructing the personalized system model as instructed by the patterns.

Constructivist learning systems have been discussed in [22]. The authors define the constructivist view of learning as a process in which learners play active roles in constructing the set of conceptual structures that constitute their own knowledge base. In our approach, the learner constructs her own conceptual structure of the system by creating personalized models of the complex domain. The learner, in addition, takes an active role in defining the learning environment. This is exhibited by the fact that the environment adapts the task descriptions to the concrete names given by the learner in earlier tasks. Moreover, depending on the chosen alternative, the environment behaves differently when the learner is presented with a number of alternatives.

From the viewpoint of structuring learning material, a similar approach to our work is presented in [14]. The authors propose an architecture for a goal-oriented way of teaching. They illustrate their concepts by presenting a web-based adaptive learning environment. The methodology described in [14] structures the domain knowledge of courses into three layers: the educational material, the concepts, and the knowledge goals. Similarly, we have structured the domain knowledge of complex system models into three layers: the model itself, the concerns abstracting the model, and the patterns encapsulating the learning tasks. Using the terminology of [14], our models represent the material, our concerns represent the concepts structuring this
material, and our comprehension patterns stand for the concrete learning goals.

7 Conclusions

In this work, we have described an approach for facilitating learning software libraries through the customization of the model entailing their structures. Our approach is to encapsulate the model into separate concerns. Every concern corresponds to a set of related features (matters of interest) in the software library. We further used comprehension patterns to encapsulate the smaller learning tasks within each of these concerns. We have applied our methodology to construct an interactive environment for learning a software library representing the JPEG file format specifications. The structural model of the JPEG library is composed of numerous components, interactions, and restrictions making it harder to learn and use. In order to model the JPEG library comprehension patterns, we have used a generic pattern-based development environment known as MADE.

Our early experiences suggest that our methodology contributes to the comprehensibility of system models. Stepwise learning by doing, separation of unrelated model features, grouping of related model elements, and customizing original models constituted the key benefits of our approach. Nevertheless, in order for our environment to be useful, the order of the learning tasks should be carefully planned and the descriptions of the tasks should be described in detail.

A possible direction for future work is to further evaluate the effectiveness of the presented methodology and tool support. In this regard, we might consider using the discussed case study to teach engineering students the fundamentals of the JPEG file format specifications. This assignment can be given in the scope of a Digital Image Processing course. We also consider applying our methodology to a more complex industrial case study.

Acknowledgments. This work is funded by the National Technology Agency of Finland (Tekes) and the Academy of Finland. The authors would like to thank Esin Guldogan and Mejdi Trimeche for their feedback on the case study.

References

Publication 7


Copyright © 2005 Springer.
Reprinted, with kind permission of Springer Science and Business Media, from the proceedings of FASE 2005.
Managing Variability Using Heterogeneous Feature Variation Patterns

Imed Hammouda, Juha Hautamäki, Mika Pussinen, and Kai Koskimies

Institute of Software Systems, Tampere University of Technology,
P.O. BOX 553, FIN-33101 Tampere, Finland
{firstname.lastname, juha.o.hautamaki}@tut.fi

Abstract. Feature-driven variability is viewed as an instance of multidimensional separation of concerns. We argue that feature variation concerns can be presented as pattern-like entities - called feature variation patterns - cross-cutting heterogeneous artifacts. We show that a feature variation pattern, covering a wide range of artifact types from a feature model to implementation, can be used to manage feature-driven variability in a software development process. A prototype tool environment has been developed to demonstrate the idea, supporting the specification and use of heterogeneous feature variation patterns. We illustrate the idea with a small example taken from J2EE, and further study the practical applicability of the approach in an industrial product-line.

1 Introduction

The software engineering community is becoming increasingly aware of the nature of software systems as multi-dimensionally structured collections of artifacts: no single structuring principle can cover all the possible concerns of the stakeholders of a software system. This observation has far-reaching implications on how we construct, understand and manage software systems. Multi-dimensional approaches to software engineering have been the target of active research for a long time [1][2][3].

In the context of software product-lines [4][5], one of the central concerns is variability management. The aim of variability management is to change, customize or configure a software system for use in a particular context [6]. In feature-driven variability management, variations of software products are expressed in terms of feature models (e.g. [7]). Selections of certain variants in a feature model are reflected in the design and implementation of the resulting product. Thus, a feature and its variation points constitute a slice of the entire system, cross-cutting various system artifacts ranging from feature models to implementation. Although variability management has been recognized as one of the key issues of software product-lines, its tool support lacks systematic approaches: most tools used in the industry are specific to a particular domain or product-line platform. Typically, automated support for variability management
is based on product specifications given in, say, XML, used to generate the actual product by a proprietary tool.

A particular challenge for more systematic tool support for variability management is the fact that variability concerns span different kinds of artifacts and different phases of the development process. Even if we forget informal documents, the artifacts involved in variability concerns may include formal requirement specifications, design models, Java source files, XML files, scripts, make files, etc. The languages these artifacts are expressed in vary from graphical notations like UML \[8\] to various textual languages. Within UML, so-called profiles can be further used to create specialized modeling languages as extensions of UML for various purposes. Thus, we need a tool concept for variability concerns which is easily adapted to any reasonable artifact format.

In our previous work \[9,10,11,12\] we have studied how a generic pattern concept can be used as a basis of tool support for various cross-cutting concerns like framework’s specialization interfaces, maintenance concerns, and comprehension concerns within artifacts of a particular kind (e.g. UML design models or Java source code files). Here the term pattern refers to a specification of a collection of related software entities capturing a concern in a software system; a pattern consists of roles which are bound to the concrete entities.

In this paper we generalize the pattern concept to allow multiple artifact types within the same pattern, thus satisfying the needs of feature variation patterns. We argue that the pattern concept is particularly amenable to present such heterogeneous patterns, since the basic pattern mechanisms are independent of the representation format of the artifacts, as long as there is a way to bind certain elements appearing in the artifacts to the roles of the pattern. This is in contrast to traditional aspects \[13\] which are presented using language-dependent mechanisms. The main contributions of this paper are the following:

- An approach to provide tool support for representing concerns within heterogeneous artifact types covering different phases of the development process
- The concept of a feature variation pattern as a model for tool-supported feature-driven variability management
- A prototype tool environment allowing the specification and use of feature variation patterns, together with early experiments

We proceed as follows. In the next section we briefly sketch the main idea. In Section 3, we explain the basic structuring device our approach is based on, the pattern concept. In Section 4, we give an overview of our current prototype environment supporting heterogeneous patterns. In Section 5, we illustrate our approach with an example concerning feature variation management in the J2EE environment. In Section 6, we present a small case study where we have applied the idea of feature variation patterns to a product-line provided by our industrial partner. Related areas in software engineering are discussed in Section 7. Finally, we summarize our work in Section 8.

\[1\] Our pattern concept has little to do with, say, design patterns: a pattern is a low-level mechanism that can be used to represent a design pattern or some other concern.
2 Basic Idea

We propose that feature variability can be managed using an artifact-neutral structuring device, a feature variation pattern. We will explain the pattern concept in more detail in the next section. Basically, a pattern consists of roles which are bound to actual system elements located in various artifacts; the pattern defines the required relationships between the elements bound to its roles. A feature variation pattern collects together elements relevant for realizing the anticipated variability of a feature across multiple artifacts. Ranging from requirements descriptions to actual implementation, these artifacts may be created in different phases of the software development process, and manipulated by different tools.

A single tool is used to manage the patterns, communicating with artifact-specific tools through their APIs. The existence of a feature variation pattern makes it possible for the tool to guide the product developer in exploiting the variability provided by a product-line platform, to assist in the generation of product-specific parts of the system, to make sure that the product has been developed according to the assumptions of the platform, to trace design-level variability support back to requirements, and to extract a system slice representing a single feature variation concern. This paper focuses on the two first issues. The general idea of the tool concept is illustrated in Figure 1.

Our approach is conservative in the sense that we do not make assumptions about languages or design methods: the only thing we assume is that the tools used to process the relevant system artifacts offer an API which allows the pattern tool to access the elements of those artifacts and to catch certain events (e.g. when an artifact has been modified). This assumption holds for many modern tools; in this work we have used Rational Rose for UML models and Eclipse for Java and XML. The artifacts can be freely edited through their dedicated tools: if an artifact is modified, the worst that can happen is that some bindings in an existing pattern become invalid or certain constraints defined by the pattern are violated. In this case, the pattern tool warns the developer about the inconsistencies and proposes corrective actions. It is then up to the developer to either correct the situation or ignore the warning. A prototype tool environment is presented in more detail in Section 4.
3 Patterns

We view a feature variation pattern as an organized collection of software elements capturing any concern in a software system related to the variation of a particular feature. The target system could be a software product-line, or a single product with anticipated needs for feature variation.

Figure 2 depicts a conceptual model in UML for (feature variation) patterns. A pattern is a collection of hierarchically organized roles. A pattern is instantiated in a particular context by binding the roles to certain elements of concrete artifacts. Each role can be associated with a set of constraints expressing conditions that must be satisfied by the element bound to a role.

Artifacts contain models which can be expressed in different notations following well-defined metamodels. Here we regard any formal presentation describing some system properties as a model, including, say, UML models and source code. A metamodel is assumed to define a containment relationship between the model elements. In any binding, the containment relationships of the bound elements must respect the hierarchy of the roles.

The metamodels of the notations used in a model define properties for the model elements that can be checked by constraints. Constraints may refer to the elements bound to other roles, implying dependencies between the roles. For example, in a pattern which covers UML models, a constraint of an association role may require that the association bound to this role must appear between the classes bound to certain class roles, thus implying a dependency from the association role to the two class roles.

A role is associated with a type, which determines the kind of model elements that can be bound to the role. A role type typically corresponds to a metaclass in the metamodel of a given notation. As indicated in the lower part of Figure 2, a pattern can be associated with multiple sets of role types (for example UML, Java, etc). Each set groups together related role types. For example, there is a set

Fig. 2. Conceptual model for feature variation patterns
of role types for representing UML model elements. In this paper, we use patterns with role types covering a subset of UML (for representing feature models and design models), Java (for representing the actual implementation), and general text (for representing deployment descriptors). Each set of role types can be associated with specialized constraints applicable only for the roles in that set. For example, a Visibility constraint checks the visibility option of classes and their members in Java.

A multiplicity is defined for each role. The multiplicity of a role gives the lower and upper limits for the number of elements playing the role in an instantiation of the pattern. For example, if a class role has multiplicity \([0..1]\), the class is optional in the pattern, because the lower limit is 0.

4 A Prototype Environment - MADE

Our experimental environment supporting feature variability management is the result of the integration of several existing tools. Eclipse \([14]\) is used as a platform for JavaFrames \([9]\) that implements the previously described generic pattern concept for Java role types. Eclipse acts also as a Java IDE in our work. The pattern engine of JavaFrames has been lately exploited in the MADE tool for creating a pattern-driven UML modeling environment \([12]\). This has been done by adding UML specific role types and integrating the resulting UML pattern tool with Rational Rose \([15]\). We have further extended the pattern tool with simple text file role types, allowing text files or their tagged elements to be bound to the roles as well. Thus, we can bind the roles to, say, XML files or items.

The MADE tool transforms a partially bound pattern into a task list. This is done by generating a task for each unbound role that can be bound in the current situation, taking into account the dependencies and multiplicities of roles. By performing a task, the designer effectively binds a role to an element. In order to use patterns as a generative mechanism (as in this paper), a default element can be defined for every role. If a role with a default element specification is to be bound during the pattern instantiation process, the binding can be carried out by first generating the default element according to the specification, and then binding the role to this element. The pattern engine updates the task list after a task has been performed, usually creating new tasks. When updating the task list, the pattern engine also checks that the constraints of the roles are satisfied, and generates corrective tasks if this is not the case.

A main principle in the design of our environment has been avoiding any kind of compulsive working mode of the designer. The existence of patterns does not prevent normal editing of, say, UML models or Java code; the purpose of the patterns is to offer additional support rather than a strait-jacket. If some constraint of a pattern bound to an element (or the binding itself) is broken as a result of an editing action, the tool generates immediately a new task to repair the broken constraint or rebind the role. In many cases the tool can even offer an automated repairing option.
5 Illustrative Example: Managing Persistence in J2EE

We illustrate the idea of feature variation patterns with a simple J2EE application. A typical J2EE application makes use of a persistent data storage which can be realized by different database products. We assume that the developers of the application want to make it easy to select the most suitable database solution for different customers. Thus, the possible variations of the database solution are specified in the feature model, and the design is given in such a way that all the desired variations can be easily achieved.

Assuming bean-managed persistence, the bean should be able to decide for optimization reasons which data source implementation to use. After the right implementation has been established, the bean can rely on a standard interface (DAO, Data Access Object) implemented by all the different data sources. There are two common solutions to select a data source implementation. The first is to hardcode the name of the implementation class in a specific method in the bean. When the data source changes, the implementation of that method should be changed so that it would return the proper implementation class. Another solution is to store the name of the implementation class in the deployment descriptor of the bean, as an environment variable. The bean decides at run-time which data source to use by looking up the value of this environment variable.

In both techniques, either the Java code or the deployment descriptor should change according to the data source selected. However, even if the developer decides to hardcode the implementation class in the bean code, storing the information of the used data source in the deployment descriptor might serve other purposes such as application documentation. In addition, the design model of the application should change in the sense that a class corresponding to the selected data source is added to the model filling the variation point.

5.1 Representing Database Selection as a Feature Variation Pattern

The above situation can be described using a feature variation pattern. The pattern has roles representing concrete elements at four abstraction levels: feature model, design model, Java source code, and deployment descriptor (XML). The pattern is given using a dedicated editor in MADE; however, we illustrate the pattern in Figure 3 with a dependency graph. The four different artifact types are represented by circular shapes. Roles are denoted by rounded rectangles, with role type marks (<<role type>>). Prebound roles are shaded. Role dependencies are drawn as broken arrows, while containment relationships are presented with diamond edges.

In the feature model part, there is a role named ‘ConcreteDatasource’. This role represents the data source variant to be used. In the design model part, the UML class role named ‘BeanDAOImplementation’ stands for the model element indicating the proper DAO implementation. This role should be bound to a UML class in the application design model. Role ‘Implementation’ is used to reflect the data source decision at the code level. This role should be bound to a Java
There is a dependency from the four roles 'Datasource', 'Implementation', 'BeanDAOJavaImpl', and 'BeanDAOImplementation' to 'ConcreteDatasource' since the concrete elements bound to these four roles depend on the chosen data source implementation. The pattern tool generates the tasks following the partial order defined by the dependency relationships of the roles.

The actual pattern specification defines a set of tool-related properties for each role such as the task prompt, an informal description of the task (shown to the user together with the task prompt), and the possible default element generated prior to binding. The table in Figure 3 illustrates how these properties of the roles are specified in MADE. The table presents the properties of role 'BeanDAOImplementation'. Note that the specifications refer to the names of the elements bound to other roles using the <# : ... > notation, an expression of a simple scripting language used in our tool.

Due to such references, the values of these textual properties of the role are adapted to the current binding situation; for example, the task prompt and task description always use the application-specific names. In this case the default element is a template for the UML class of the proper DAO implementation. The template gives the name of the DAO implementation class and refers to the concrete name of the DAO class. In addition to role properties, the table in Figure 3 includes an example inheritance constraint which is used to check the
generalization/specialization relationship between the UML class bound to role 'BeanDAOImplementation' and the UML class bound to role 'BeanDAO'.

5.2 Using the Pattern

Figure 4 shows a scenario for applying the pattern. The MADE tool transforms an unbound role into a task, shown as a textual prompt. The execution of the task results in binding the role. The figure includes four tasks and their outcome.

The scenario starts from the left upper corner. First, a task prompt asks the user to select the data source to be used. The user is shown the list of available data sources: MySQL, Oracle, and PostgreSQL (1). The user decides to use the Oracle database. Next, a new task for providing a UML class named 'BeanDAO_Oracle_Impl' is shown (2). The UML class stands for the implementation class of the DAO interface. Note that the environment adapts the task description to the context of the user: the selected database name 'Oracle' is used in the default name of the UML class (3). The next task is to register the DAO Java implementation class to be used by the bean (4). A Java code fragment is then generated (5). The code creates a new instance of class 'BeanDAO_Oracle_Impl' and assigns it to the bean field holding the DAO object. Finally, the last task is to store the name of the DAO implementation class in the deployment descriptor of the bean (6). For this purpose, a new environment entry 'DAO_CLASS_NAME' is generated. The value of the entry is 'BeanDAO_Oracle_Impl' (7).

Figure 5 shows an overall view of our environment after the tasks described above have been carried out. In the upper half of the screen, Rose displays the feature model (on the left) and the design model (on the right) as UML class diagrams. In the lower half of the screen, the Eclipse Java environment displays
two textual editors (above) and the integrated pattern tool (below). The latter is further divided into three panes: the pattern view showing the roles in a containment tree (left), the task view (upper) and the instruction view (lower). The textual editors are for Java code and for XML.

When a pattern is selected in the pattern view, the task view displays the tasks generated by that pattern. In Figure 5, however, no doable tasks are shown since the pattern is fully bound. Instead, the view reveals a description of a task (data source selection) that has already been carried out. The outcome of every performed task can be retrieved in this way at any time. The bindings, visualized with the arrows in Figure 5, show how a pattern acts as a connecting artifact between different model levels.

6 Case Study - Nokia GUI Platform

6.1 Target System

Nokia produces a family of NMS (Network Management System) and EM (Element Manager) applications, which are software systems used to manage networks and network elements. For this purpose, the company has developed a Java GUI (Graphical User Interface) platform to support the implementation of the GUI parts for the variants of this product family. The platform is used as an object-oriented Java framework, in which the application is created by deriving new subclasses and by using the components and services of the framework.
The platform provides a number of services useful for GUI applications. There are services for system logging, online help, user authentication, product internationalization, clipboard usage, CORBA-based communication, and licensing. Depending on the environment used, each service may have different implementations. Applications, built on top of the GUI platform, get a reference to the proper service implementation from a registry file. It is essential that product developers register the right service implementation. Due to the confidential nature of the platform, detailed descriptions about its architecture are omitted.

6.2 Experiment

The goal of the case study was threefold: identifying variability in the GUI platform, expressing the identified variation points in terms of feature models, and using our pattern concept and prototype tool to achieve an environment where variability is managed across multiple artifacts.

Our first step was to analyze the platform documentation and interview several of the platform users. As a result, we have identified a number of variability issues regarding how platform services are being used. The next step was to construct feature models realizing the variability issues we have identified. Figure 6 depicts, in lighter color, a feature model representing the main services provided by the GUI platform. All services are optional. It is up to the product developer to decide which services to use. The 'I18n' service stands for the internationalization service.

As a third step, we have developed a system of feature variation patterns for expressing which platform services are used and which service implementations are considered. Because a service is regarded as a separate concern in the platform, a pattern (or a set of related patterns) is used to represent that service. The patterns cover four abstraction levels: the feature model representing the services, the design model of the product, the Java implementation of the application, and the product service registry files.

When instantiating the patterns, developers decide which services the application should incorporate and which service implementation to use. If a service is selected, MADE uses the corresponding pattern to generate tasks for registering the service in the application service registry. Furthermore, the pattern ensures that the right service implementation is added to the design model of the application, that the Java implementation exists, and that the right (and only one per service) service implementation is registered to the property files.

Each of the services shown in Figure 6 in lighter color can be further represented by its own feature models. The figure, for example, depicts, in darker color, a feature model specifying how application GUI components can use the on-line help service. First, the variant 'User_Event' indicates that the call resulted from a user request whereas 'System_Event' indicates that the call originated from the application. GUI components can support either event types but not both at the same time. Second, developers must specify what type of help service is requested. There are five variants for such service type. 'Contents', for example indicates that the target of the request is a table of contents whereas 'Search'
Managing Variability Using Heterogeneous Feature Variation Patterns

Fig. 6. Feature models for platform services and online help service

Fig. 7. Role diagram for the OnlineHelp pattern

indicates that the target of the request is a search page. Similarly, the choice of the service type is exclusive.

A feature variation pattern is used for representing the feature model for online help. The pattern is used to generate tasks for selecting the GUI components which incorporate a help service and the help event and service types associated with those components. In addition, it ensures that only one variant is selected and that it is correctly represented in the Java implementation. Furthermore, the pattern is used to associate, in the design model, the help service with the selected GUI components.

The role structure of the pattern is shown in Figure 7. The two roles named 'ConcreteEventType' and 'ConcreteServiceType' represent the event and service type variants to be used. In the design model, role 'GUIComponent' should be bound to a UML class representing the GUI components associated with the help service. The '+' symbol in front of the role name stands for the multiplicity value meaning, in this case, that there can be any number of GUI components (UML classes) bound to the role. At the implementation level, the pattern uses the variants in the feature model to generate a Java code fragment for registering the proper help service to the selected GUI components. This is illustrated by role 'Implementation'. The code fragment is inserted in an initialization method of the application.
6.3 Experiences
As explained earlier the goals of the case study have been to study the expressive power of our formalism and the applicability of our approach in an industrial setting. Our first challenge has been to dig up the feature models from the platform documentation, where the documents were not structured according to our methodology. Another problem, during this phase of the experiment, was the fact that the platform is used by different groups in the company having different interests towards the product line. Thus, we had to interview each of these parties. Furthermore, the platform comes with different versions making it harder to identify the variation points.

The variability aspects of the platform were mostly specified in Word documents. An attractive option would have been to link the relevant parts of these documents to the feature variation patterns, rather than (or in addition to) feature models. However, this would require different structuring of the Word documents and new role types covering elements in these documents. In the case study, we have specified all platform services as optional features, even though services such as internationalization or online help are in practice required. At this stage, we could conveniently present a number of variation points in the platform with a set of heterogeneous patterns. However, we still need to construct more patterns in order to cover variability in other platform components.

7 Related Approaches

Feature Variability Management
One of the key issues in software product lines is variability management. A product line architecture makes it easier to manage the product family as it promotes the variation between different products, i.e., the use of variants and variation points [16]. The software community has taken different approaches to variability management. Our methodology is based on feature models [7]. Other methodologies include architecture description languages (ADLs) [17] and different XML-based program specification [18].

Framed aspects [19] are another approach for representing features in software product lines. The purpose of the method is to support evolution in product lines rather than the development of products. First, aspects are used to encapsulate tangled features. Then, frames are used to provide parameterization and reconfiguration support for the features. Compared to our approach, framed aspects are applied at the implementation level only.

Model-Driven Engineering
MDD (Model-Driven Development) [20] promotes an approach where models of the same system are usually derived from each other leading to better alignment between the models. MDA (Model-Driven Architecture) [21] is a recent initiative by OMG for supporting MDD principles. MDA defines three views of a system: a Computation Independent Model (CIM), which is a representation of a system from a business viewpoint, a Platform Independent Model (PIM), which is a rep-
representation of a system ignoring platform specific details, and Platform Specific Model (PSM), which is a model of a system that covers both platform independent information and details about a specific platform. Compared to MDA, our feature models correspond to CIM whereas other model kinds can be viewed as PSM. In this paper we do not discuss support for PIM. An MDA-oriented approach, using our pattern concept, is presented in [22].

Batory et al. take an approach to feature oriented programming where models are treated as a series of layered refinements [23]. Features are composed together in a step-wise refinement fashion to form complex models. Models can be programs or other non-code representations. To support their concepts, the authors have developed a number of tools for feature composition, called AHEAD toolset. The toolset provides similar functions to those of MADE. MADE environment solves two problems not otherwise addressed in [23]: tracing features across different artifacts and checking the validity of models.

**Separation of Concerns Across Artifacts**

The idea of representing concerns within and across different artifacts has been addressed in the work on multi-dimensional separation of concerns [2]. The authors present a model for encapsulating concerns using so-called hyperslices. These are entities independent of any artifact formalism. Our heterogeneous pattern concept can be considered as a concrete realization of the hyperslice concept. Other concepts such as subjects [24], aspects [1], and viewpoints [25] also resemble our patterns. Subjects are class hierarchies representing a particular view point of a domain model. Thus subjects deal only with object-oriented artifacts. Aspects, on the other hand, have been mostly used to represent concerns at the programming level. Viewpoints, in turn, are used to represent developers’ views at the requirements level. Different viewpoints can be described using different notations. Compared to these concepts, our pattern approach is not bound to a specific artifact type but can be used to capture concerns cross-cutting different phases of the development process.

**Tool Support for Traceability**

The ability to track relationships between artifacts has been a central aim in requirements engineering [26]. Rational RequisitePro [27] is a market-leading requirement management tool. In RequisitePro, use cases are written as Word documents which are stored into a relational database. Use case diagrams can be associated with use case documents; sequence diagrams implementing the use cases can be linked to the use case diagrams, and class diagrams can be linked to the sequence diagrams. A related approach has been proposed to achieve traceability in software product families [28], linking requirements, architecture, components, and source code. In both works, traceability is based mainly on explicitly created links. In our approach there are no explicit links between the artifacts themselves, but instead we specify a particular concern as a pattern and bind the roles of the pattern to certain elements of the artifacts.
8 Concluding Remarks

We have developed a prototype environment supporting the representation of concerns cross-cutting not only components but also various artifacts produced in different phases of a software development process. Our first experiences with the environment are encouraging: we could conveniently present variation points in a product-line with a heterogeneous pattern covering multiple artifact types. Using existing tool technology for pattern-driven software development, we could achieve an environment where the pattern guides variation management from feature model to actual implementation.

However, feature variation management is just one example of the potential benefits of our environment. The main point is that the pattern stores the information of the existence of a concern among the artifacts. This information can be exploited in many ways. For example, it is often useful simply to generate a single view where all the fragments related to a particular concern can be browsed, a kind of concern visualizer. This kind of support is readily available in our tool. Heterogeneous patterns can be used for tracing as well: the designer can follow the dependencies between the roles of a pattern and find out why, for example, a particular class has been introduced in the design model.

In order to support new representation formats of artifacts, we are working at transforming the tool into a framework for constructing new role types. Other future directions include enhancing the alignment and traceability between the various artifact types, and proceeding with the case studies provided by our industrial partners.

Acknowledgments

This work is supported financially by the Academy of Finland (project 51528) and by the National Technology Agency of Finland (projects 40183/03 and 40226/04), and is partly (first author) funded by Nokia Foundation.

References

Publication 8


Copyright © 2005 Springer.
Reprinted, with kind permission of Springer Science and Business Media, from Model-driven Software Development - Volume II of Research and Practice in Software Engineering.
A Tool Infrastructure for Model-Driven Development Using Aspectual Patterns

Imed Hammouda

Institute of Software Systems, Tampere University of Technology,
P.O. BOX 553, FIN-33101 Tampere, Finland
imed.hammouda@tut.fi

Summary. A software system can be viewed as a combination of separate concerns covering various artifact types and cross-cutting the primary structure within each artifact type. This chapter presents a concern-based approach to model-driven development. An aspectual pattern concept is used to represent concerns at different model levels. We back our approach with a concrete tool implementation called MADE (Modeling and Architecting Development Environment). The tool has been used in a number of development scenarios providing support for variability management, framework specialization, maintenance, and system comprehension. The tool introduces a task-based model-driven development environment and provides facilities for model generation, checking, and tracing.

1 Introduction

In software engineering, a model of a system is a description or specification of that system and its environment for some certain purpose [317]. A single system can be represented using multiple models at various abstraction levels ranging from requirement models to source code. Different models can be expressed using different forms. Requirements documents can be written in natural language, design models can be expressed using visual modeling languages such as the Unified Modeling Language (UML), and implementation is usually coded using high-level languages such as Java. Even though these models take different forms, they all represent properties of the same software system, realized in machine instructions. A clear benefit of this diversity is that each form may serve the need of different stakeholders in the software development cycle. Business managers, for example, are mostly comfortable with informal textual information and simple visual diagrams. System designers, however, prefer to analyze systems using more detailed design diagrams whereas programmers express their software solutions in terms of code.

Model-driven development (MDD) [288] is a practice where models not only are used as documentation but also become the backbone of the development process. In MDD, models of the same system are usually derived from each other; a new model can be generated from an existing one. A model and its derivations are usually connected using various kinds of relationships. These relationships can be formal or
informal, complex or simple [180]. One way of building and expressing the inter-
model relationships is the use of mapping functions. Mapping functions represent
expert knowledge to transform between one model and another [288]. As the model
is transformed, the new model may be expressed in another form. For example, a
UML design model can be mapped into Java code. Automation has been recognized
as a key factor when deriving models from others [386]. The idea is that most appli-
cation code, for example, should be automatically generated from design models.

Separation of concerns (SoC [85]), which is a principle to modularize a system
into manageable parts, can be used to structure a model into smaller sub-models
that are easier to handle. Each sub-model represents a specific concern (i.e., matter
of interest) in the system. For example, it is important to separate platform-specific
concerns from application-specific concerns. However, because software systems are
typically specified in terms of models at multiple abstraction levels, individual con-
cerns must be addressed at all levels. Thus, the same concern can be represented at
different levels using different notations and languages.

Generally, the key issues in MDD, such as the ability to generate new models
from existing ones or to express the relationships between a model and its deriva-
tions, are judged according to tool support. In fact, the need for MDD tool support
has been widely acknowledged by software developers, as the success of MDD is
strongly dependent on tools. Nowadays, a variety of tools with various capabilities
of applying MDD principles is available on the market. Another key factor for the
usage of an MDD tool is the support for different software development activities.
Ideally, the same tool should fulfill the need of different stakeholders in the dev-
velopment process, as every stakeholder might use the tool in a different develop-
ment scenario such as model generation, maintenance, or customization.

This work proposes an approach and a tool for MDD by structuring models into
separate concerns at different abstraction levels. The various features and capabili-
ties of the tool allow developers to interact with the environment in different usage
scenarios. The remainder of the chapter is organized as follows. Section 2 discusses
the main characteristics of MDD tools. In Sect. 3, a pattern-oriented infrastructure
for MDD tools is described. Using this infrastructure, Sect. 4 presents a concrete
implementation of an MDD tool. Section 5 illustrates experiences in applying the
proposed methodology and the implemented tool. Related methodologies and tools
are briefly discussed in Sect. 6. Finally, Sect. 7 concludes the chapter.

2 Characteristics of MDD Tools

The main purpose of MDD tools is to facilitate the development of software by rais-
ing the abstraction level at which software solutions are defined. As the rise in the
level of abstraction from assembly languages to high-level programming languages
has improved productivity and increased the quality of software, it is believed that
the same improvements could be achieved by moving the focus of software devel-
opment from programs to models of programs. In order to achieve the goals set for
MDD, these tools should meet the following functional and quality requirements.
The requirements have been abstracted from research on MDA (Model-Driven Architecture) [317] tools such as [35, 369]:

1. Mapping and tracing of models: new models can be derived from existing ones; for example, generating code from visual models or customizing models themselves. Using MDD tools, developers should also be able to trace elements in one model to the elements they are derived from in other models. Consequently, an MDD tool should be able to relate different models expressed in heterogeneous notations and languages. In addition, developers should also be able to modify the derived models by, for example, updating design models or adding code fragments to the generated code.

2. Support for different software engineering activities such as specifying requirements, constructing design models, and code generation as well as other development-process activities such as comprehension and maintenance.

3. Ability to separate and represent different subject matter in models so that relevant matters are considered and non-essential aspects are deferred to later development phases. An MDD tool should also be able to represent the various relationships between these matters of interest.

4. Model mappings and transformations should be performed in a controlled manner. The tool user should be given clear instructions on how to perform the transformations and which model parts are being transformed.

Based on requirements 1–2, we can see that one of the technical foundations of MDD tools is the support for heterogeneous artifacts expressed in different languages and notations. This is essential for managing models at different abstraction levels. A second implication of these two requirements is the support for various development scenarios corresponding to different phases of software development. Requirements 3–4 suggest that every model abstraction level may define its own matter of interest. For example, some platform-specific information is not required for models at higher abstraction levels. Therefore, a desired quality of MDD tools is the ability to express separate concerns of a software system. In the following, these three characteristics are discussed in more detail.

2.1 Support for Multiple Artifact Types

Software development involves different phases ranging from requirements specification to final deployment and maintenance of applications. Each phase is associated with specific software artifacts that are usually expressed using different languages (notations). For example, requirements artifacts can be represented using text documents, design models may be expressed in UML diagrams, and implementation can be written in Java. MDD tools aim at transforming these different artifacts from each other. For this reason, these tools should support heterogeneous artifact types (requirements document, design models, implementation) and different languages used to express these artifacts (UML, Java, XML).

The approach to support multiple artifact types is illustrated in Fig. 1. An MDD tool, which is represented by a cubical shape, handles different software artifacts.
The idea is that such a tool should support various software development processes handling different modeling artifacts given with different notations. Usually, there is a many-to-many relationship between software artifact types and the notations used. A single software artifact can be expressed using different notations. For example, requirement specification documents can be described using both informal textual files and UML use case diagrams. The same language can be used to express different software artifacts. For example, implementation and test code can both be written in Java.

2.2 Support for Various Development Activities

The success of an MDD tool strongly depends on the diversity of the usage scenarios it suggests and on the extent the tool supports the needs of different stakeholders. This is important, especially for developers who play multiple roles in the development process. Generally, using one integrated development environment instead of independent tools minimizes the overhead of learning and switching between tools and enhances the inter-tool communication. MDD tools supporting heterogeneous development activities should incorporate a number of core functionalities including the ability to represent models, to derive new models from existing ones, and to define different kinds of relationships between models.

Generating application code from (visual) models residing at higher abstraction levels has been regarded as the primary expectation of MDD tools. Nevertheless, these tools might help developers deal with the complexity of other development tasks. For example, models may be used for verification and simulation purposes and for increasing the level of automation of testing activities by automatically deriving test cases [35]. Furthermore, MDD tools can be used to facilitate maintenance of existing models. For example, the tool might generate maintenance tasks based on earlier maintenance decisions. Another example scenario is to use the tool to facilitate learning complex system models, for instance through model abstraction (p. 179 of this book) or model customization [188].

![MDD tool support for multiple artifact types](image-url)
2.3 Support for Decomposing Systems into Separate Concerns

The development of a software system can be divided into sub-problems according to the different concerns the application needs to incorporate. Applications are then modeled as hierarchical combinations of concerns. Each concern tackles a specific matter of interest in the application, such as a specific application functionality or quality. It has been shown that separation of concerns improves the alignment between requirements, design, and code of applications and leads to better reuse and evolution of systems [62].

Decomposing systems into separate concerns is particularly beneficial when developing applications of multi-tier architecture such as J2EE systems. The architecture of a J2EE system can be decomposed into three layers: presentation, business logic, and data. Each of these layers represents a separate concern in the system. Different concerns may serve the needs of different stakeholders in the development team. For example, graphic designers are usually interested in user interface matters reflected by the presentation tier only. Even in the case of single-layered systems, separation of concerns proves to be a useful practice. Software development becomes easier if the essential aspects of software such as business logic are separated from other concerns which involve technology-related matters such as security.

![MDD Tool](https://via.placeholder.com/150)

**Fig. 2.** MDD tool support for separation of concerns

Figure 2 shows an arbitrary software system decomposed into a number of concerns. MDD tools should provide a two-way support for such decomposition. First, the tool should be able to represent the individual concerns as separate entities. Second, using the tool, it should be possible to model the various inter-concern relationships. There are two kinds of relationships: overlapping and containment. Figure 2 shows an overlapping relationship between Concern_1 and Concern_2, meaning that those model elements represented by the overlapping region treat both concerns. As an example of a containment relationship, Concern_3 is modeled as the composition of Concern_3.1 and Concern_3.2.

2.4 A Two-Dimensional Development Approach

Figure 3 illustrates a two-dimensional approach for model-driven software development. The figure can be seen as a union of Fig. 1 and Fig. 2: an MDD tool should
represent the different development phases (and the artifact types); an artifact type is structured into separate concerns. Following this approach, an MDD tool can be used to support various development activities as shown in the right part of Fig. 3. The development approach brings a number of benefits including better traceability, comprehensibility, and maintainability, which are essential qualities in model-driven software development.

First, having a concern crosscut multiple software artifacts links the requirements corresponding to that concern to the model elements satisfying the requirements, to the code implementing them, and to all other software artifacts reflecting that concern. This is important in various areas of software engineering. In variability management [82], for example, when a specific variant is selected, that variant should be represented at all model levels. Second, separation of concerns enhances system comprehensibility because a concern localizes the focus of software developers to one specific matter of interest at a time. Furthermore, understanding a certain design model fragment can be enhanced by backward tracing that fragment to the requirement it satisfies. Finally, with a combination of good traceability and good comprehensibility, system maintainability becomes simpler and more obvious. For example, when a requirement is changed, the affected design solution or code is easily tracked so the propagation of change is better controlled.

In a typical situation, however, a concern might not cover all levels of abstraction, as shown in Fig. 3. Some concerns, for example, are first expressed at the requirements level but deferred to later stages in the development process. For instance, user authentication is usually not discussed at the architecture or design levels since at these levels developers prefer to focus on business logic instead. Some other concerns might not be represented in the deployment or documentation levels. An example of such concerns is comprehensibility. Other concerns such as maintainability concerns are not discussed at the requirements level but are often anticipated during the architecture or design phase.

![Fig. 3. A two-dimensional development approach](image-url)
3 Concepts for MDD Tool Infrastructure

We propose the use of a generic role-based pattern concept called aspectual pattern to represent the concern-based decomposition and composition of software systems as discussed in the previous section. First, we review the principle of separation of concerns in software development. Then we discuss the main characteristics of aspectual patterns.

3.1 Separation of Concerns

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. Aspect-oriented software development (AOSD) [124] is a paradigm that addresses crosscutting concerns by providing means for their systematic identification, separation, representation, and composition [357]. According to Kiczales [250], two crosscutting structures imply that neither can fit neatly into the other. Aspect-oriented programming (AOP) is a recent programming paradigm incorporating the ideas of AOSD. AOP organizes the crosscutting concerns into separate modules called aspects.

Aspects are merged with base programs or models of programs. The process of merging is called weaving. There are two ways in which aspects are weaved: static or dynamic. Static weaving modifies a structural base model by inserting new model elements. Dynamic weaving consists of adding at runtime new behavior to applications.

Traditionally, aspects have been used to encapsulate the different concerns cutting across several classes or other units of decomposition within the same level of abstraction, for instance implementation classes. Recent research work (e.g. p. 237 of this book, [419]), however, extends the ideas of aspect orientation to support the representation of concerns within and across software artifacts. Similarly, this work argues that the crosscutting nature of concerns could be extended to cover the different features of software cutting across various levels of abstraction. A single concern would then have a specific representation at different model levels expressed using different artifact types. In the next section, we describe a concrete pattern concept for implementing such concern-based composition and decomposition of systems.

3.2 Aspectual Patterns

A solution for modeling the two-dimensional development approach shown in Fig. 3 is to use the so-called aspectual patterns [191]. An aspectual pattern\(^1\) can be viewed as a configuration that captures an aspect cutting across various software artifacts. It is an organized collection of software elements capturing a concern that is relevant for some stakeholder of a software system.

---

\(^1\) Aspectual patterns have little to do with, say, design patterns: an aspectual pattern is a low-level mechanism that can be used to represent a design pattern or some other concern. Generally, aspectual patterns are system specific.
Aspectual patterns are described in terms of roles and the relationships between them. A role has a type, which determines the kind of software elements that can be bound to the role. A pattern can be associated with multiple sets of role types (e.g., UML, Java, etc.). Each set groups together related role types. For example, there is a set of role types for representing UML class diagram elements. In order for an aspectual pattern to crosscut multiple software artifact types, it should support heterogeneous sets of role types. For simplicity, the term pattern will be used in the remainder of the discussion to refer to aspectual patterns.

Pattern roles are attached with a number of properties. Each role may have a set of constraints. Constraints are structural conditions that must be satisfied by the concrete element bound to a role. A constraint of a UML association role P, for example, may require that the UML association bound to P must appear between the UML classes bound to certain other UML class roles Q and R.

A cardinality value is defined for each role. The cardinality of a role gives the lower and upper limits for the number of the elements bound to the role in the pattern. For example, if a role of type UML operation has cardinality value 0..1, the operation is optional, because the lower limit is 0.

Roles may depend on other roles. For example, there is a dependency from role P to role Q since the binding of P depends on the binding of Q. In this case, a UML class should be bound to role Q before that class is used when binding a UML association to role P.

A default element can be defined for a role. If a role with a default element specification is to be bound during the pattern instantiation process, the binding can be carried out by first generating the default element according to the specification, and then binding the role to this element. The specification of a default element may refer to the (elements bound to the) other roles, implying dependencies between the roles. Often, it is sufficient to calculate the value of the element name before the element gets generated.

In this work, aspects are represented using a role structure that can be instantiated and weave into base models. The weaving corresponds to the binding of the roles: each role stores the information of a joint point. The constraints that are associated with a given role can be used to determine the context where the aspect may appear, and the constraints can be used to check whether the aspect, implemented by the pattern, is correctly weaved. There are a number of advantages in representing aspects as role-based patterns:

- Flexible weaving process: In contrast to traditional weaving, the weaving of aspectual patterns is considered as an interactive, incremental process where the join points are located under the guidance of a pattern tool, rather than in a fully automated fashion. The weaving process is performed as simple tasks in the context of the developer, rather than as a large black-box operation. By task we mean a simple action that adds an element or enforces a property on the base model. In the context of aspectual patterns, a task stands for binding an unbound element or enforcing a role constraint.
• Addressing several key challenges in aspect orientation: First, aspect overlapping can be represented and implemented in a straightforward way using role-based pattern composition techniques where a model element can play different roles in different patterns. Second, aspectual patterns offer a symmetric model where there is no explicit distinction between “base-level” elements and “aspect” elements. Yet, using binding information it is still possible to highlight the effect of aspects in base models. Another desired property of aspectual patterns is the ability to support various phases of the development process as roles can be bound to software and non-software elements covering various kinds of system artifacts. Finally, aspectual patterns can be reused in multiple contexts. Roles are attached with parameterized properties whose values are calculated and adapted to the context where the patterns are applied.

3.3 Pattern Role Diagrams

In order to improve the comprehensiveness of pattern structures, Fig. 4 introduces a notation for visual pattern specification. The figure depicts a role diagram of an aspectual pattern for managing system security. For simplicity, we discuss the pattern structure specified using a single set of role types, i.e., UML element types. In the next section, the representation is extended to multiple sets. Security is recognized as a critical concern, for example, when developing business applications. The aspectual pattern illustrated in the figure shows how security should be modeled in the system under development.

A ConcreteSecurityManager class provides a checkPermission operation which checks the security policy of a custom permission ConcretePermission. The nodes, marked in white, depict the pattern roles. The ConcreteSecurityManager role, for example, stands for any concrete element (in this case a UML class) that implements a custom security policy. The type of the role is specified above 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 checkPermission to role SecurityManager, which is marked with a diamond-headed line, represents the containment relationship between the elements that may play these two roles; and 2) the dependency from role ConcreteSecurityManager to role SecurityManager, which is marked with a light-arrow-headed broken line, stands for a logical relationship. In this case, any element that plays role ConcreteSecurityManager should implement the concrete element that plays role SecurityManager. The cardinality value (“1” for exactly one, “?” for optional, “*” for zero or more, “+” for at least one) of a role comes together with the role name. For instance, the cardinality value of ConcreteSecurityManager is “+” meaning that there should be at least one element bound to this role. If not otherwise indicated, the cardinality of the role is 1.

In order to show how the pattern can be applied, the bottom part of the figure gives a possible concrete binding (weaving). The concrete element FilePermission, represented by a dark-gray node, plays role ConcretePermission. This is marked by the double-arrowed line between FilePermission and ConcretePermission. There are two elements that play role ConcreteSecurityManager. This is a direct implication
of the "+' cardinality symbol associated with this role. 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 of how the bindings should be performed. For instance, the binding of SecurityManager to its concrete element is a prerequisite for the binding between ConcreteSecurityManager and WorkingDaysSecurityManager.

3.4 Using Aspectual Patterns for Multiple Artifact Types

As described earlier, an aspectual pattern captures an aspect cutting across various software artifacts. This situation can be described using a heterogeneous aspectual pattern supporting multiple sets of role types (i.e., UML, Java, etc.). Considering the pattern discussed in Fig. 4, a heterogeneous aspectual pattern for security, for example, can be used to illustrate how security is represented in a software system implementation and documentation in addition to its design model.

Figure 5 shows the security pattern covering three different artifact types. The artifact types are represented by roughly circular shapes. For brevity Fig. 5, which can be regarded as an extension to Fig. 4, does not show role bindings. In addition to the design model, the pattern has roles for implementation and documentation. At the implementation level, there is a role named ConcreteSecurityManager. This
role is of type Java class and is used to represent a class implementing a specific security policy. Furthermore, there is a role at the documentation level for registering the used Java implementation in the system API documentation. This role is named securitySupport and should be bound to a text fragment providing the API documentation fragment. The text fragment should be stored in the API documentation file that should be bound to the APIDocumentation role.

3.5 Composition of Aspectual Patterns

In Sect. 2.3, two kinds of relationships between concerns have been identified: containment and overlapping. Assuming that aspectual patterns are used to specify system concerns, one needs to define proper techniques for representing such kinds of relationships. Let us first discuss containment and overlapping in the context of aspectual patterns.

Containment

This relationship is the simpler of the two. In the context of aspectual patterns, this relationship basically means that it should be possible to group together a number of patterns in order to express a larger concern. Realizing such a concern means applying all the constituent patterns defining that concern.
Overlapping

Using the role-based pattern structure, it is possible to define the overlapping relationship between aspectual patterns in terms of overlapping roles. Assuming that the overlapping region of two patterns is defined using two overlapping roles, applying both patterns means that the two overlapping roles are bound to the same concrete element. The concrete element should reflect the properties of both roles assuming that the two roles do not have conflicting properties.

Composition Operator

We define a composition operator for aspectual patterns that takes care of the above two cases. Using the operator, it is possible to express larger patterns in terms of smaller ones. During the composition process, a role overlapping, if any, is dealt with by merging any two (or more) overlapping roles into one role, meaning that the overlapping roles are bound to the same concrete element. The role overlapping relationships are expressed in terms of composition rules, as illustrated in the next example.

The composition of two patterns results in a pattern. The composition operator we define is a binary operator that takes two arbitrary patterns and returns a possibly larger one. Given two arbitrary patterns X and Y, if roleX and roleY are overlapping roles in patterns X and Y respectively, then the composition of X and Y can be expressed as follows:

$$Z = + (X, Y, \{ (\text{role}X, \text{role}Y) \})$$

Z is said to be the composite pattern of X and Y. The composition formula specifies the two patterns to be composed followed by a set of tuples defining the overlapping roles. In the composite pattern Z, the role representing the overlapping of roleX and roleY, say roleZ, is said to be the unified role of roleX and roleY.

Given the above definition, we can define the following composition properties:

- Two roles roleX and roleY can overlap only if they are of the same role type.
- Two roles roleX and roleY can overlap only if the parent roles of roleX and roleY (the roles where roleX and roleY are contained), if any, are overlapping too.
- If roleX and roleY are two overlapping roles, then the cardinality of the unified role roleZ is reduced to the more restricted cardinality of the two roles.
- If roleZ is the unified role of roleX and roleY, then roleZ has at most the total number of dependencies (both outgoing and incoming) of roleX and roleY. If roleX has n dependencies and roleY has m dependencies, then roleZ has at most n+m dependencies and at least max(n, m) dependencies. This is because some of the dependencies (dependencies having the same target role) of roleX and roleY can be the same.
- Similarly, roleZ is associated with the constraints of roleX and roleY. If roleX has n constraints and roleY has m constraints, then roleZ has at most n+m constraints and at least max(n, m) constraints. The constraints having the same type and value are treated as the same.
• Patterns X and Y are said to be disjoint if they have no overlapping roles. X and Y are said to be fully composed if there is a one-to-one mapping between all roles of X and Y.

**Example**

Figure 6 illustrates a situation where the security policy in a software system is log-based. When designing a software system, security and logging are usually identified as two separate concerns. A common situation, however, is to compose these two concerns in one important concern, called log-based security. This new concern can be considered, for example, in order to check if the software system comes under attack by unauthorized sources at runtime.

![Diagram](image-url)

**Composition Rules:**

{ PL.LoggedClass = PS.ConcreteSecurityManager }

{ PL.LoggedClass.loggedMethod = PS.ConcreteSecurityManager.checkPermission }

**Fig. 6. Composing security and logging**

Figure 6 shows two composed patterns. The pattern for security has the structure explained in Fig. 4. The pattern for logging defines roles for specifying the concrete logger to use and the logged operations to be logged. Role Logger should be bound to a UML class representing the used logger class. Role LoggedClass should be bound to the class to be logged, while loggedMethod represents the individual logged methods of that class.

In the above situation, the log-based security concern can be implemented by applying the two patterns PS (for security) and PL (for logging). During the binding process, roles ConcreteSecurityManager of PS and LoggedClass of PL should be bound to the same concrete element. The purpose is to log the security manager operations being called. The two class roles are said to be overlapping. Furthermore, roles loggedMethod and checkPermission should also be bound to the same concrete operation. The implication of this overlapping relationship is that the method implementing the security policy is recorded in the system logs. An overlapping relationship between two roles is graphically represented using a dashed double-headed arrow.
arrow linking these two roles and is textually expressed in terms of composition rules. The composition rules in Fig. 6 are defined in terms of two role pairs. Each pair represents two overlapping roles.

### 3.6 Identifying and Documenting Aspectual Patterns

In order to be able to identify and structure system concerns (and thus the aspectual patterns), a high-level modularization technique based on concern architecture views is used. Concern architecture views are introduced in [247]. Typically, each concern in a concern architecture view tackles a specific area of interest in the software system. Concerns can, in turn, be composed of smaller units of interest. Each of these small units is treated as an aspectual pattern. In this way, concern architecture views are used to define and structure a system of aspectual patterns for modeling the software system.

![Diagram](image)

**Fig. 7.** Identifying aspectual patterns

The left part of Fig. 7 depicts the concern architecture view of an arbitrary software system. The log-based security concern, discussed in the previous section, is used as one of the example concerns. In addition, the figure shows three other concerns. There is a concern named Resources. This concern stands for adding to the system new resource types such as printer jobs or database connections. There are two other arbitrary concerns, Concern_1.1 and Concern_1.2. The aspectual patterns modeling these concerns are represented using rectangular shapes. In addition to illustrating concerns and patterns, the figure shows, using dashed arrows, possible dependencies between individual patterns. For example, there is a dependency from the pattern for logging (PL) to the pattern for security (PS) since the model elements for security need to be identified before they are logged. Another dependency links pattern PS to the pattern handling resources. The reason is that system developers
want to define a security policy for every added resource type. As explained earlier, the log-based security concern is defined using two overlapping patterns. However, it is possible that a concern is formed using two disjoint smaller concerns. In this case, this concern is modeled using two disjoint aspectual patterns. This is the example of Concern_1.

The right part of Fig. 7 shows the list of patterns identified using the concern architecture view. Assuming a pattern-based MDD tool, it is essential to be able to model both the individual patterns and the various relationships between them. The next section presents a concrete implementation of a pattern-based tool that can be used to specify and apply heterogeneous aspectual patterns.

### 3.7 Aspectual Patterns as Transformations

As discussed earlier, transformation is an essential property for MDD. The main goal of model transformation is to document the relationships between models and to provide rules for deriving a model from another. Models at both ends of the transformation can be either within the same level of abstraction or at different levels. Furthermore, transformation can be applied instantly or incrementally. Instant transformation means that the transformation is carried out in a single batch operation whereas incremental transformation implies that the transformation is performed through a number of transformation steps.

Aspectual patterns can be thought of as a mechanism for incremental transformation, where a transformation step involves generating the possible default elements for unbound roles based on their specifications. The order of the transformation steps is implied by role dependencies. Thus, the relationships between model elements are in fact embedded in the role specification. Because aspectual patterns are heterogeneous, the transformation can be performed from and to any level of abstraction. The idea has been applied to achieve an open MDA environment [393].

### 4 Implementation – MADE

This section discusses the main components and features of the MADE (Modeling and Architecting Development Environment) tool and shows how heterogeneous aspectual patterns are represented and composed in MADE. However, the way patterns are applied is discussed in Sect. 5.

#### 4.1 MADE Toolset

In order to demonstrate the heterogeneous aspectual pattern concept, a prototype tool known as MADE [193] has been developed. The MADE platform itself is the result of integrating three different tools: JavaFrames [183], xUMLi [348], and Rational Rose [368]. JavaFrames is a pattern-oriented development environment built on top of Eclipse [98]. Rational Rose is used as the UML editor. The third component, xUMLi, is a tool-independent platform for processing UML models and is used
for integrating JavaFrames and Rational Rose. In addition to acting as a Java IDE, Eclipse offers a number of facilities useful for software development including a variety of integrated editors (such as for Java and text) and other project management tools. The MADE toolset can be downloaded from http://practise.cs.tut.fi/fred.

Originally, JavaFrames was developed as a pattern-driven specialization environment for Java-based frameworks. For this reason, the pattern role types have initially been Java specific. In order to support the heterogeneous pattern concept, however, support for new role types has been added. For modeling purposes, a set of UML-specific role types has been developed. A third set of role types has been designed for supporting general text file operations. In addition, there are a number of other role types representing informal tasks, reminders, and user input values.

When constructing new role types, the only thing one assumes is that the tools used to process the relevant system artifacts offer an API which allows the MADE tool to access the elements of those artifacts and to catch certain events (e.g., when an artifact has been modified). It is also possible to construct new kinds of constraints and associate them to proper role types. The same constraint kind can be applied to different role types, for instance a naming constraint which requires that the name of an artifact element should conform to a given regular expression.

Figure 8 shows an overall view of MADE. Rational Rose represents the upper part of the environment. As an example of an integrated editor, implementation code is displayed in a Java editor (middle view). The left view represents the part of the environment where patterns are specified and applied. In the next two subsections, pattern specification is discussed in more detail. Patterns are represented by circular graphical icons. In the MADE environment, patterns are instantiated as extensions of other patterns. For example, the Access pattern under the FileSystemAccess node is an instantiation of the Security pattern specified under the Catalogue node. In the MADE terminology, pattern Access extends pattern Security.

When a pattern is selected, MADE transforms the pattern into a task list. This is done by generating a task for each unbound role that can be bound in the current situation, taking into account the dependencies and cardinalities of roles. The task view displays the tasks implied by the pattern. This view is divided into two panes: task titles are shown in the upper pane and detailed task descriptions are presented in the lower one. In the example figure, a task for providing a logger field is displayed. There are two kinds of tasks: mandatory and optional. Mandatory tasks are marked using a red circular dot attached to the task title whereas optional tasks are marked with a white circular dot. The task presented in the figure is mandatory.

Tasks can be performed in two different ways. The developer might bind the pattern role to an existing model element. In this case, a binding between a pattern role and an existing model element is established but the model does not change. In most cases, however, executing a task means generating a new model element. In this case, a binding between a pattern role and the generated element is established and the element is added to the proper editor depending on the type of the element. If the model element is a UML class, for example, the element is shown in the UML editor. The middle view depicts the bindings that have been performed. Models can be freely edited through their dedicated tools: if an artifact is modified, the worst that
can happen is that some bindings in an existing pattern become invalid or certain constraints defined by the pattern are violated. In this case, the pattern tool warns the developer about the inconsistencies and proposes corrective actions. It is then up to the developer to either correct the situation or ignore the warning.

4.2 Presenting Heterogeneous Aspectual Patterns in MADE

Considering the aspectual pattern for security discussed earlier, the actual pattern specification in MADE adds a number of properties to the role information shown in Fig. 5. These properties are mainly used by the tool for guiding the user through the development process. Figure 9 illustrates some of these role properties. For brevity, the figure does not show all the roles and properties. The roles have been grouped into three categories; each category represents a particular artifact type.

At the design model level, the definition of role ConcreteSecurityManager includes a defaultElementName, which is used as a default name when generating the UML class to be bound to the role. The value of this property is expressed as a template referring to the name of the concrete element bound to role SecurityManager,
which in this example represents a base class of all security managers. The property taskTitle is used by the tool for generating a title for the corresponding task. The inheritance constraint is used to enforce the generalization/specialization relationship between concrete elements bound to role ConcreteSecurityManager and the concrete element bound to role SecurityManager. In this case, the specification says that any UML class bound to role ConcreteSecurityManager should inherit from the UML class bound to role SecurityManager. The cardinality of roles is given in parentheses after the role names. At the implementation level, role JavaConcreteSecurityManager refers to role ConcreteSecurityManager for generating the default name of concrete elements. In this case, the specification suggests sharing the same name.

Figure 10 shows a view of the MADE pattern editor specifying the pattern for security. The left pane shows the hierarchical structure of the pattern roles and constraints. In this case, role checkPermission is selected. The right pane is composed
of two parts. The upper view displays the list of roles which the selected role depends on. The lower portrays exposes the checkPermission role properties discussed in Fig. 9.

4.3 Composing Aspectual Patterns

Section 3 discussed two kinds of relationships that may exist between different aspectual patterns: containment and overlapping. A composition operator was then defined in order to represent these relationships. Figure 11 shows how patterns are organized and applied in the MADE tool environment. In particular, the figure shows how individual patterns can be composed to treat larger concerns. Patterns are organized using architecture nodes. There are three types of architecture nodes: catalogue, concern, and application. Individual patterns are created under the Catalogue root node. At this stage, each aspectual pattern is regarded as a separate entity treating a specific concern in a software system and completely unrelated to other patterns. In the example case, there are two patterns named Logging and Security. The two patterns respectively implement the concerns discussed in Sect. 3.

The concerns of a software system are represented using concern nodes and are hierarchically represented under the Concerns root node as shown in Fig. 11. The figure depicts a single concern named LoggedSecurity. As discussed in Sect. 3, this concern is the composition of two component concerns: logging and security. If a concern is implemented by multiple aspectual patterns, that concern may define rules on how the patterns are composed. Composition rules are specified as a property of concern nodes and are currently specified in a text area as shown in Fig. 11. Each composition rule is formulated using a pair of pattern roles. The first rule, for example, says that role LoggedClass in the pattern for logging overlaps with role ConcreteSecurityManager in the pattern for security.

Currently, pattern composition in MADE requires manual enumeration of the composition rules, which might become a challenging task as the number of rules grows high. This problem can be partially resolved by keeping MADE patterns small in size and loosely coupled. Another solution we are investigating is the possibility to build the composition rules on the fly and compose the patterns when applied to base models.

Application development is carried out by considering those concerns relevant to the application needs. Using MADE, the developer selects which concern he or she wants to realize. The environment then takes care of which patterns to instantiate. The Application root node in Fig. 11 shows a concrete realization of the Logged-Security concern. The concrete application is named FileSystemAccess and stands for developing a security policy for file system access. Access and Logs represent respectively the instantiation of patterns Security and Logging. When a pattern is selected, the environment displays the tasks corresponding to that pattern. When tasks are executed, the composition rules are taken into account. If a developer binds a role that has an overlapping counterpart, the overlapping role is automatically bound to the same concrete element.
4.4 Main Features of MADE

MADE comes with a number of qualities desirable for MDD. The main characteristics of the tool are:

1. Stepwise development environment: MADE transforms a pattern specification into a task list guiding developers step by step through model development.

2. Automatic detection and repair of broken model conventions: If a model is manually edited, the environment provides immediate validation of the model against the pattern specifications. In the case of a constraint violation, a repair task is created to inform the user about the violation. In most cases, the tool provides the option to automatically repair the violation, for example, restoring a generalization/specialization relationship between two classes.

3. Adapting to developer's context: MADE is able to record the history of developers’ tasks and use the recorded information when documenting next tasks. The tool, for example, uses the names of concrete elements in the textual descriptions of related tasks.

4. Support for separation of concerns in models: Using MADE, individual concerns in a software system are represented as a set of aspectual patterns.

5. Support for traceability and visualization: Aspectual patterns are used to represent a concern crosscutting various abstraction levels. When a pattern is applied, MADE records role binding information which can be used later to link together different representations of a concern at different abstraction levels or simply to highlight the effect of a concern in a base model.

6. Openness to new design conventions: MADE is not bound to a fixed set of patterns. As new concerns are identified, new patterns can be easily introduced to the existing architecture. In addition, as the application domain evolves, existing pattern specification can be updated and outdated patterns can be removed. It is also possible to change the specification of patterns if the roles are not yet bound.
5 Supporting Different Activities

The MADE tool has been applied to support MDD in a number of ways. In the subsequent sections, four applications of heterogeneous aspectual patterns are presented. Figure 12 depicts four pattern categories relevant to each application: feature variation patterns, maintenance patterns, framework specialization patterns, and comprehension patterns. In addition to the purpose of the pattern, the figure shows the different stakeholders that are involved in creating the patterns and those involved in applying them. In the first category, for example, feature variation patterns have been used to manage feature variability in software systems. In this case, heterogeneous aspectual patterns are used to represent the different variation points and to make sure that when a specific feature variant is selected, that decision is propagated to all models of the application. These kinds of patterns are usually created by product-line developers specifying the different variation points in the system architecture. Product developers then apply these patterns to design and implement new products.

In the remainder of the discussion, each subsection discusses a pattern category, together with an example application. In addition to validating the approach, the aim of the examples is to discuss the details of MADE from different perspectives. In the first two applications, the focus will be on the specification and the deployment of aspectual patterns. The first example shows how the specification of a pattern can be transformed into a task list and how the tasks are carried out in the MADE tool. The second example discusses the issue of viewing and browsing the model fragments that are relevant to a particular concern. In the third and fourth applications, the focus will be on the role of concern architecture views in the identification of aspectual patterns, as discussed in Sect. 3.6. The third case study illustrates the use of concern architecture views to specify the extension points in an example framework specialization interface. The last application shows the use of concern architecture views to describe the customization principles of an example complex model.
5.1 Feature Variability Management

In software product lines, variability means variation in the definition and implementation of a specific feature or additional features [82]. Variability can occur at different levels in the design: product-line level, product level, component level, sub-component level, and code level [412]. Assuming that variation is represented at different model levels, it is essential to reflect the choice of a particular variant in all levels. Typically, every variation point is represented using a separate aspectual pattern. A single variation point may crosscut various model levels. To represent this crosscutting nature, feature variation patterns [190], which have roles covering different artifact types, can be used.

Example

In EJB component architecture [226], for example, the persistence of a BMP (Bean-Managed Persistence) entity bean can be realized by the use of different database products. In order to achieve maximum portability, bean providers choose to support multiple database implementations. Database variation in a BMP entity bean is considered as a variation point in the bean implementation. In order to manage this variation point, bean developers use a solution called Data Access Object (DAO) defining a common interface for all possible data source implementations. When a BMP bean is deployed, only one database implementation is used.

There are two common solutions to select a data source implementation. The first is to hardcode the name of the implementation class in a specific registration method in the bean. When the data source changes, the implementation of that method changes so that it would return the proper implementation class. Another solution is to store the name of the implementation class in the deployment descriptor of the bean, as an environment variable. The bean decides at runtime which data source to use by looking up the value of this environment variable. In both techniques, either the Java code or the deployment descriptor should change according to the data source selected. However, even if the developer decides to hardcode the implementation class in the bean code, storing the information of the used data source in the deployment descriptor might serve other purposes such as application documentation.

Let us assume that the database variation is given as a feature model and that one needs to reflect the database choice in the design model of the bean as well. Both the feature model and the design model are expressed in terms of UML class diagrams. In addition to filling the variation point in the design model, the class corresponding to the selected data source should be represented in the registration method and the deployment description of the bean.

The MADE solution for the above situation is to use a feature variation pattern for managing the database variation. In this context, the concern that the pattern encapsulates is the database implementation feature. The pattern covers four abstraction levels: feature model, design model, implementation, and deployment information. The pattern has roles for representing UML model element (feature model and de-
sign model), Java element (bean implementation), and XML entities (deployment descriptor).

![Diagram of pattern role diagram for database product variation](image)

**Fig. 13.** Pattern role diagram for database product variation

Figure 13 shows a pattern role diagram for database product variation. The diagram uses the notation introduced in Fig. 5. The shaded roles represent pre-bound roles that define the context of applying the pattern. The pre-bound role DeploymentDescriptor, for example, is bound to the XML file that represents the deployment descriptor of the bean. At the level of the feature model, role ConcreteDatabase represents the database product variation point. When the pattern is applied, the role should be bound to a concrete UML class reflecting the selected data source. Several roles in other abstraction levels depend on this role. At the design model level, role BeanDAOImplementation reflects the data source selection in the design model of the bean. At the Java implementation level, role Implementation represents a Java code fragment that registers the proper database implementation class. Finally, at the deployment information level, role Datasource represents an XML tag that stores the name of the selected database implementation into an environment variable.

Figure 14 shows a scenario for applying the pattern shown in Fig. 13. MADE presents a role binding as a task, shown as a textual prompt. The execution of the
task results in binding the role to a concrete element. The first task is to select the data source to be used. The developer is shown the list of available data sources: MySQL, Oracle, and PostgreSQL. The outcome of the task is shown following the task prompt. The user, for example, decides to use the Oracle database. Next, a new task for providing a UML class named BeanDAO_Oracle_Impl is shown. The UML class stands for the implementation class of the DAO interface. Note that the environment adapts the task description to the context of the user: the selected database name Oracle is used in the default name of the UML class. The generated class is added to the design model of the application. The next task is to register the DAO Java implementation class to be used by the bean. As a result, a Java code fragment is generated. The code creates a new instance of class BeanDAO_Oracle_Impl and assigns it to the bean field holding the DAO object. Finally, the last task is to store the name of the DAO implementation class in the deployment descriptor of the bean. For this purpose, a new environment entry DAO_CLASS_NAME is generated. The value of the entry is BeanDAO_Oracle_Impl.

Figure 15 depicts an overall view of MADE after the tasks described above have been carried out. In the upper left part, the feature model is displayed in the UML editor. Using the same editor, the design model of the bean is presented in the upper right view. A proper database implementation class named BeanDAO_Oracle_Impl is used. The Java editor in the middle shows the generated Java implementation of the database registration method whereas the right-hand side shows the environment variable stored in the deployment descriptor file of the bean. The Bindings view groups together all the role bindings at all abstraction levels. Using the tool, it is
possible to trace back the concrete elements bound to the pattern roles. However, it is also useful to know to which roles an artifact element is bound. This feature will be implemented in future versions of the tool.

Experiences

The aim of this section was to illustrate the details of MADE by studying the use of a simple feature variation pattern. An important question is “Can the approach scale up?” To address the question of scalability, we are currently studying the practical applicability of the approach in an industrial product line [190]. The example case study is to use the MADE approach to identify, express, and manage variability in a Nokia GUI platform. The early results have been positive. It was possible to conveniently present a number of variation points in the platform with a set of heterogeneous patterns covering four abstraction levels: the feature model representing
services provided by the platform, the design model of the product, the Java implementation of the application, and the product service registry files.

The biggest challenge in the case study has been identifying the variation points in the GUI platform and expressing them using feature models. The reason is that the platform documentation was not structured according to the needs of the MADE approach. Consequently, considerable effort has been spent on studying the product line itself and interviewing several stakeholders participating in the development and use of the platform.

Concluding Remarks

This example discussed how heterogeneous aspectual patterns can be used to manage feature variability. A feature model is defined to represent the variation points at the requirements level. Application developers need to bind these variation points to a specific variant. An aspectual pattern is used to represent a variation point in the various models of a software system. The purpose of the pattern is both to document the variation points and to generate the required tasks to properly fill in the variation points at different model levels.

5.2 Maintenance

System maintainability is considered as a development-process concern. Maintainability is an important concern that should be considered early in the design phase [413], especially in the case of adaptive maintenance. When considered as a concern, maintainability can be expressed in terms of a related set of maintainability concerns. Each concern corresponds to a large maintenance task that may crosscut different units of decomposition such as classes or modules. A maintenance pattern [189], which is a concrete usage of the aspectual pattern concept, can be used to document the anticipated maintenance tasks. Furthermore, maintenance actions should propagate to all the models of the system under maintenance. A maintenance pattern, therefore, takes care that changes due to maintenance are propagated to all model levels. In addition, the pattern specification can be used to generate maintenance tasks based on maintenance actions carried out at higher abstraction levels. It is important, for instance, that the implementation code of a software system adapts to any maintenance changes occurring in the design model.

Example

Measuring human–computer interaction is very important for improving interactive systems and their user interfaces. In order to conduct this kind of measurement, proper measuring tools have to be implemented. When designing such tools, it is impractical to fix all aspects of the measurement process as new measurements and measuring strategies are discovered after the tool is in use. It is important therefore to anticipate at the design phase those extension points.

As an example, consider some measuring software for making user tests of a web search engine user interface. Using the measuring tool, it is possible to both log and
analyze the results of these tests. The measuring tool first writes user interactions with the search engine into log files, then parses the stored log files, and builds an object structure based on the logged information. In a second phase, a specific component is used to make measurements based on the observed data. There are a variety of possible measurements such as average time needed to complete a task, average number of selections made in one task, or total number of selections made during the test. Many others can be discovered as information is being logged. Furthermore, a measurer component can be defined by combining any of these measurements. Therefore, it is possible to design and use new measurer components as the system runs.

Fig. 16. Pattern role diagram for extending some measurement software

Considering the example, one can define a maintenance pattern for extending measurements and for building new measures. Figure 16 shows a structure for such a pattern. For brevity, some roles have been omitted. The pattern has roles in three model levels. At the design model level, for example, role ConcreteMeasurer stands for a new measurer component and should be bound to a UML class in the design model of the measuring software. The role has two child roles: getNumOfMeasurements represents the operation returning the number of used methods and role calculateMeasurements stands for the operation that performs the actual measurements.

As a new measurer is added to the design model, a corresponding Java implementation class should be defined at the implementation level. This is marked by the dependency from role JConcreteMeasurer at the implementation level to role ConcreteMeasurer at the design model level. Further, let us suppose that every newly
constructed measurer implementation has to pass a number of test cases in order to make sure that the measurer combines the measurements in a correct way. For this, the pattern has roles for generating testing code for every measurer. Role testMeasurer at the tests level represents a Java method for testing the corresponding measure. Similarly to Fig. 13, pre-bound roles are shaded.

Figure 17 shows a possible deployment of the maintenance pattern in MADE. The pattern has been used to provide a new measurement strategy (TimeForTestCompletion) and a new measurer component (TimeMeasurer). Using MADE, it is possible to view the effect of applying the pattern to different model levels. The UML model elements rendered in darker color show the effect of highlighting the maintenance pattern at the design model level. At the Implementation level, the Java source files associated with the pattern are shown in the Java editor. In addition, a method for testing the new measurer is highlighted in the existing TestPlanManager Java class reflecting the effect of the maintenance pattern at the tests level. Pattern highlighting is regarded as a tool for tracing a single concern across multiple software artifacts.
Experiences

As another example of adaptive maintenance, we have applied the approach to the MADE system [189]. The motivation was to show that if the approach works for general anticipated maintenance tasks, it should work for maintaining MADE itself. The pattern engine, which is an integral part of the MADE system, has been designed to support new artifact types and notations. Thus, the case study was to use the tool to construct new role types.

An experiment for creating two new role types has been carried out in [201]. As a result, a total of 11 Java classes (400 lines of code) have been created. Out of these, 315 lines of code have been generated automatically while 85 lines of code have been manually created or edited. Thus, the MADE approach achieved a ratio of 79% of automatic code generation. The other 21% was not automatically generated since it stands for knowledge that is not captured by the pattern specification. In this case, for example, it was not possible to anticipate how to resolve an artifact element bound to a role of the newly constructed type since this depends greatly on the tool where that artifact element is managed.

Concluding Remarks

In this example application, heterogeneous aspectual patterns have been used to document and generate the maintenance tasks required for maintaining a software system. In this case, an example of adaptive maintenance is considered. However, other forms of maintenance can also be supported [189]. Here, maintenance is considered as an activity that is mainly dealt with at the design and implementation phases. In contrast, variability management is addressed in the requirements and architecture levels. Maintenance patterns have roles covering multiple sets of role types in order to reflect the maintenance actions at all possible model levels. For brevity, the discussion does not show how the pattern is applied in MADE. Typically, individual maintenance tasks are similar to the ones described in Fig. 14.

5.3 Framework Specialization

An object-oriented framework is a reusable design expressed as a set of classes implementing the basic architecture for a family of software systems [121]. Framework specialization is the process of adapting a framework to meet the requirements of a specific application. Framework specialization is usually regarded as a complex task. The reason is twofold. First, following the specialization instructions often lack tool support and can be tedious. Second, there is a lack of assisting application developers to carry out the specializations at the level of visual modeling languages, such as UML. As a solution, special-purpose aspectual patterns, called framework specialization patterns [193], can be used to structure and document the specialization interface of frameworks into different specialization concerns. A specialization concern crosscuts different software artifacts of the framework specialization. Patterns are used in order to make the specialization process easier by grouping the specialization tasks into meaningful parts (concerns) and propagating the specialization
J2EE is a component-based and platform-independent architecture for building enterprise applications. J2EE applications are constructed by following the architectural rules imposed by the J2EE framework. On top of these architectural rules, several design conventions have been proposed to bring better reuse, maintainability, and portability to these applications [192]. Figure 18 depicts a concern architecture view structuring the specialization interface of the J2EE platform into a number of concerns reflecting these architectural rules and design conventions.

There are three concerns for the creation of the three bean types: session, message-driven, and entity beans. Each of the first two concerns is represented using a single aspectual pattern whereas the third is composed of several smaller concerns and thus implemented using more than one pattern. Patterns SessionEJB and MessageEJB are used to generate a session bean and a message-driven bean respectively. In order to generate an entity bean, four patterns should be applied. Pattern EntityEJB models the core components of an entity bean. Pattern ValueObject is used to optimize the communication between business components and clients. Pattern DataAccessObject is applied to make the enterprise components transparent to the
data source used. This solution has already been discussed in Sect. 5.1. This pattern is applied for generating entity beans with BMP only. The fourth pattern PrimaryKey is responsible for generating primary key classes for the corresponding entity beans. In order to locate other business components, patterns SessionEJB, MessageEJB, and EntityEJB get the service of a pattern named ServiceLocator whose role is to abstract the complexity of the lookup process.

In order to decouple client access from the core business layer of the application, two patterns are used: pattern SessionFacade is used to reduce the complexity of the interaction between clients and business objects by grouping together the access to multiple beans. In addition, pattern BusinessDelegate is used to achieve loose coupling between clients at the presentation tier and the business services implemented by the enterprise beans.

The presentation layer is defined using three concerns. There is a concern for preprocessing requests. This concern is represented by a pattern named InterceptingFilter. The aim of this pattern is to process requests and responses before being passed to clients. The second concern is for defining a control strategy of how to process requests and view navigation. It is represented by two patterns. Pattern FrontController centralizes the decision how to retrieve and process the requests and pattern DispatcherView separates the logic on deciding which view comes next from the view components themselves. Finally, the third concern deals with the view component of the presentation layer and is modeled using one pattern named ViewHelper. This pattern processes business data for getting presentation content.

Figure 19 shows the patterns and the concerns, which have been identified in Fig. 18, specified in the MADE tool. First, patterns are defined under the Catalogue node, then they are grouped into separate concerns under the Concerns node. In addition, the figure depicts a concrete application implemented according to the architecture proposed by the pattern system. The application is a web-based to-do list where a list of users and their associated tasks can be accessed, manipulated, and stored in a relational database. The Entity_Bean concern is considered to implement an entity bean for representing users. This is depicted in the figure by the node User_Entity_Bean. The Session_Bean concern is used to implement a session bean modeling user tasks. This is illustrated by the Task_Session_Bean. Since the application does not need asynchronous communications, no message-driven bean is developed. This is why the concern Message-driven_Bean is not considered in the application.

Experiences

Using MADE, it was possible to automatically generate up to 60% of the total lines of code for the to-do list application discussed above [192]. Manually, it could take up to 20 hours to construct the code from scratch. Using MADE, the same task took a couple of hours. Approximately, one-third of the development time was spent with using the environment for automatic code generation. The rest was to manually provide custom business code and user interface implementation. The overall design was made according to the widely used J2EE design patterns.
Therefore, the MADE tool could be used to improve the quality of the application and reduce the development effort. Nevertheless, identifying the proper concerns and patterns requires effort and domain expertise. Besides, modeling the patterns in MADE, in terms of roles and their properties, can be a laborious task. For instance, it took a couple of hours to model the EntityEJB pattern and several deployment checks to validate the pattern against its specification.

Concluding Remarks

This section shows how concern architecture views can be used to identify and document the relationships between framework specialization patterns specified as aspectual patterns. Framework specialization is another example of variability management. In this case, however, the decision on the variation points is not bound to a specific feature model. Instead, the specialization decisions are open. For brevity, the structural specification of the patterns is not discussed and the actual deployment of the pattern system has been omitted.

Framework specialization patterns are heterogeneous aspectual patterns. Using MADE, it is possible to generate the design model, the implementation, and the de-
deployment descriptors of the enterprise application. If servlets are used for implementing the presentation layer, the standard Filter approach can be used when applying the InterceptingFilter pattern, which means that the pattern has roles for representing XML files. Furthermore, the patterns at the presentation layer may have roles for supporting different client implementations such as HTML, JSP, or WML.

### 5.4 Comprehension

Customization is an essential activity for comprehending complex model structures since it is easier to study a system one part at a time [343]. Model customization can be used to adapt a model to specific purposes by identifying only those parts of the model that correspond to these purposes. Assuming that software systems are represented using different forms of models, customization decisions should cover all these model levels. For example, the same customization principles should be used to customize a design model and the source code of a software system. In this way, the implementation of the system remains aligned with the design model. Comprehension patterns [188], which are aspectual patterns reflecting system comprehensibility issues, are used to group related model parts of complex systems into separate comprehensibility concerns. Using comprehension patterns, it is possible to show those parts of the system that are relevant for a specific purpose and to make sure that models remain aligned following the customization actions.

#### Example

JPEG (Joint Photographic Experts Group) [417] is one of the well-known file formats for compressed images. The structural model behind this standard is an example of a complex system since it comprises a large number of components, defines complex inter-component dependencies, and is subject to extension. An efficient way of learning this complex system is to customize its model according to the specific needs of the learner, focusing on those parts relevant to the actual context and leaving out the irrelevant parts.

Figure 20 presents the different relationships between the various JPEG-related file formats. The file formats are shown using cubical shapes. There are, however, four main file formats: JPEG, JFIF, EXIF, and DCF. These formats are discussed in more detail in [188]. The dashed arrows describe the dependencies between these formats. The DCF format, for example, is an extension of the EXIF format. Consequently, for learning the DCF format, it is required to learn the EXIF format as well. However, it is not necessary to learn the DCF extensions in order to learn EXIF.

Figure 21 shows a concern architecture view for structuring the comprehensibility of the JPEG-related file formats. There are four concerns; each concern corresponds to learning the corresponding file format and is modeled using a number of comprehension patterns. The DCF format concern, for example, represents the structural components and rules that correspond to DCF file formats. It is composed of the EXIF format concern and three DCF-specific comprehension patterns. These three patterns represent the model elements that are specific to the DCF extensions.
An Extended Jpeg File Format

Baseline Jpeg Image (Lossy)

Extended Jpeg Image (Lossy)

Fig. 20. The JPEG file formats

Fig. 21. Comprehensibility concern architecture view for JPEG-related file formats
only. The EXIF format concern, in turn, is modeled using two EXIF-specific patterns and three JPEG-related patterns. The latter three patterns represent the core model elements required for defining any of the file formats.

Given the above concern architecture view, it is clear that in order to comprehend the DCF part of the JPEG-related file formats, there is no need to study the model elements relevant to JPEG or JFIF formats only. Using MADE, only those patterns corresponding to the DCF format concern have to be applied. The environment provides learning tasks corresponding to the model elements reflecting the structure of DCF.

Figure 22 depicts the MADE specification of the architecture identified in Fig. 21. Individual patterns and concerns are specified in the Catalogue and Concerns nodes respectively. Furthermore, the figure shows three example customization scenarios for learning different formats of the JPEG standard: EXIF, JFIF, and basic JPEG formats. As an example, the environment displays the patterns that should be applied for learning the EXIF format. The complete case study is discussed in more detail in [188].

![Fig. 22. Modeling JPEG patterns/concerns in MADE](image-url)
Experiences

Comprehension patterns are used to represent a structured collection of model elements. Using MADE, developers are able to view only the artifact elements resulting from the application of patterns. The whole original model can be obtained by applying all the patterns. A number of experiments have been conducted to determine the customization capabilities of the MADE tool in the context of the case study. Figure 23 shows a comparison between the original model of the JPEG library structure and several customized models. The customized models represent different concerns in the library. Depending on the selected concern, the customization resulted in a reduced number of model elements. For example, this number is reduced by 50% when the user focuses on the JFIF-related features.

In the case of the JPEG library, it was relatively easy to identify the comprehension patterns based on the concern architecture views. However, in case the architecture does not clearly separate different concerns, it might become harder to identify the patterns. Comprehension patterns, on the other hand, are more useful when the crosscutting concerns are not directly reflected by the architecture.

<table>
<thead>
<tr>
<th>UML Model</th>
<th>Classes</th>
<th>Associations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Model</td>
<td>70</td>
<td>50</td>
</tr>
<tr>
<td>JPEG Baseline</td>
<td>27</td>
<td>19</td>
</tr>
<tr>
<td>EXIF without thumbnail</td>
<td>49</td>
<td>37</td>
</tr>
<tr>
<td>EXIF with thumbnail</td>
<td>55</td>
<td>43</td>
</tr>
<tr>
<td>Whole JFIF</td>
<td>32</td>
<td>24</td>
</tr>
</tbody>
</table>

Fig. 23. The original model versus several customized models

Concluding Remarks

In this section, concern architecture views are used to structure the comprehensibility of complex systems. Heterogeneous aspecual patterns are used to represent the model elements relevant to a specific matter of interest in the system. These patterns are exploited by the MADE tool offering a task-based learning environment. For brevity, however, the detailed specification of the patterns is not shown and the actual learning environment has not been discussed.

6 Related Work

In this section, we discuss work related to the notion of aspecual patterns and the principle of separation of concerns. We then compare our methodology for MDD to other approaches and tools.
6.1 Aspectual Patterns

The term aspectual pattern used in this paper is inspired by the work on aspectual components [274]. The constructs in both approaches are represented in terms of a graph of nodes. In the case of [274], 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 the context of MADE, 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 apply to different artifact types expressed in different notations.

Glandrup and Aksit [149] independently use the term aspectual pattern in a slightly different context. In their work, an aspectual pattern is a language-independent extension for a pattern. The extension is used to describe the crosscutting behavior of a pattern. Similar to the MADE approach, a role-model is used to represent how patterns are superimposed in the implementation and role constraints are used for tracing and verification of patterns in the implementation. MADE can be used to implement the concepts proposed in [149].

6.2 Separation of Concerns

The issue of separation of concerns has already been discussed in the context of MDD. According to Kulkarni and Leddy [263], separation of concerns should be dealt with at both the model and code levels. The authors propose a development architecture that uses parameterized package abstractions to specify and compose concerns at the model level. At the code level, these concerns are managed using aspect-oriented programming. The MADE approach extends the ideas of [263] by supporting separation of concerns at all model levels including documentation, test code, and deployment information if possible. Instead of using different techniques to represent concerns at different levels, the methodology presented in this work uses a solution applicable to all levels. This leads to better traceability between models.

The idea of representing concerns within and across different artifacts has been addressed in the work on multi-dimensional separation of concerns [419]. The authors present a model for encapsulating concerns using so-called hyperslices. These are entities independent of any artifact formalism. Aspectual patterns, presented in this work, can be considered as a concrete realization of the hyperslice concept. Other concepts such as subjects [62], contracts [208], aspects [124], and viewpoints [304] also correspond to aspectual patterns. Subjects are class hierarchies representing a particular viewpoint of a domain model. Contracts operate on object-oriented artifacts and are used to represent objects and their interactions separating them from other interactions involving the same objects. Aspects, on the other hand, have been mostly used to represent concerns at the programming level. Viewpoints, in turn, are used to represent developers’ views at the requirements level. Different viewpoints can be described using different notations. Compared to these concepts, aspectual patterns are not bound to a specific artifact type and can be used to represent any type of structural configuration including text document for example.
6.3 MDD

MDA (Model-Driven Architecture) is a recent initiative by OMG for supporting MDD principles. MDA defines three views of a system: a Computation Independent Model (CIM), which is a representation of a system from a business viewpoint, a Platform Independent Model (PIM), which is a representation of a system ignoring platform (technology) specific details, and Platform Specific Model (PSM), which is a model of a system that covers both platform independent information and details about a specific platform. OMG claims that this hierarchy offers better control for software development and brings up desired qualities such as portability, interoperability, and reusability. In response to the needs of MDA, the QVT-Partners have released a revised proposal for the OMG’s QVT RFP [165]. QVT stands for Queries/Views/Transformations and represents a standardized transformation language to allow UML models to be transformed into usable software. The MADE approach solves some of the issues mentioned in the RFP. The aspectual pattern concept can be used as a technical infrastructure for managing the various transformations between models [393].

Batory et al. treat models as a series of layered refinements [31]. Individual features (reflecting different concerns) are composed together in a stepwise refinement fashion to form complex models. Models can be programs or other non-code representations. In order to support their concepts, the authors have developed a number of tools for feature composition, called the AHEAD toolset. The toolset provides similar functions to those of the MADE tool. MADE, however, solves two problems not otherwise addressed in [31]: tracing concerns in the generated models and checking the validity of models against the architectural rules.

6.4 Tools

As MDA is considered as the most popular approach for MDD, MDD tools are generally referred to in the software community as MDA tools. A number of these tools along with a detailed comparative evaluation are presented in [35]. Some of these tools were not originally built for MDA but were later tuned to support its principles. Similarly, the MADE pattern concept and pattern engine were initially implemented for specializing Java-based frameworks. Most tools (if not all) do not implement all the features of MDD (or MDA). Instead, each tool considers a restricted set of features. MADE takes the same line: the tool supports separation of concerns across models at different abstraction levels.

The Concern Manipulation Environment (CME) [64] shares similar ideas and goals as those of MADE. The idea of CME is to offer end users an open suite of tools for use in creating, manipulating, and evolving aspect-oriented software, across the full software lifecycle. The environment helps in interoperating and integrating different AOSD tools and paradigms and comes with an initial set of components. For instance, the Concern Manager (ConMan) tool models software in terms of arbitrary concerns and their interrelationships. A strong similarity between MADE and
ConMan is the support for a wide variety of concern structures and software decompositions – artifact types.

The most closely matching MDD tool related to the MADE approach is Rational XDE [400]. Similarly to our approach, XDE includes a pattern engine that can be used for model transformation. However, patterns are applied differently in MADE than they are in XDE. In MADE, it is possible to apply a pattern in small increments whereas an XDE pattern is only applied in full. The reason is that MADE treats each role binding as a separate task. After an XDE pattern is applied, the integrity of the pattern against the model is not automatically supervise; one has to revalidate the pattern bindings each time the model changes. Also, XDE does not detect all violations (e.g., a deleted generalization relationship between two classes) when the model is manually edited. In MADE, the conformance to the architecture expressed by patterns is supervised all the time and it is possible to detect all constraint violations in the model against pattern specifications. When a violation is detected, a new task is immediately generated informing the developer about the violation and suggesting an automatic repair. Another significant difference is that aspectual patterns can have roles representing non-software entities like text files and user input values or even roles representing informal entities like reminders. Furthermore, in XDE, there is no support for explicit modeling of concerns.

7 Conclusions

In this work, we have presented a tool infrastructure for model-driven development. The methodology consists of a two-dimensional development approach that is based on separating the different concerns in a software system and representing these concerns at different abstraction levels. In order to implement the proposed tool concept, we have used a structural entity called heterogeneous aspectual pattern. As tool support, we have built a pattern-driven modeling environment known as MADE. In MADE, a pattern is used to represent a concern cutting across multiple levels of abstraction.

We have applied the MADE tool in a number of development scenarios. Depending on the nature of the problem, aspectual patterns provided support for managing feature variability, documenting maintenance tasks, annotating framework specialization, and facilitating system comprehension. We have tested the MADE tool against small illustrative examples and larger industrial-level case studies.

So far, experience indicates that the proposed task-based development environment realizes many of the principles of model-driven development, namely automatic model derivation, controlled artifact generation, and traceability between models. We think that MADE can be further applied to support other software engineering activities such as architecture validation and software testing. Another direction for future work is to generalize the aspectual pattern concept to represent other dimensions in software development. For example, a pattern can be used to encapsulate different views of a given concern. Similarly, one concern may be represented at the
same level of abstraction using different formalisms. Furthermore, one concern may crosscut multiple software systems.

8 Acknowledgements

This work is supported financially by the Academy of Finland (project 51528) and by the National Technology Agency of Finland (projects 40183/03 and 40226/04). The author would also like to thank Nokia Foundation for a grant toward his postgraduate studies and Kai Koskimies, Tommi Mikkonen, Tarja Systä, and the anonymous reviewers for their comments.