

Implementing Architecture Stratification

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades
eines Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Dissertation

von Diplom-Informatiker
Martin Girschick
geboren in Darmstadt

Referent	Prof. Dr. Thomas Kühne
Korreferent	Prof. Dr. Andy Schürr
Tag der Einreichung	26. August 2010
Tag der mündlichen Prüfung	13. Oktober 2010

Darmstadt 2010
D17

Abstract

Although currently software development often separates the design phase from the implementation, the trend towards model-driven approaches is undeniable. As models are the primary development artifact in model-driven development, one has to deal with the problems surrounding huge and therefore unmanageable models. To tackle these issues, systems are often divided into components that are modeled separately. As an alternative, views can be used, which hide certain aspects of the model or present them in a different form. In both cases, several models or at least several views co-exist, which leads to the problem of model synchronization.

One of the main goals of model-driven development is the automatic generation of executable applications. Here, too, model synchronization is problematic, as several information sources may affect the same code fragment. When only parts of the system are generated, the remaining application has to be coded by hand, which complicates the reapplication of the generation process.

In order to achieve a high level of automation in model-driven development, the complete application has to be modeled. In this scenario, the use of several models improves understandability and precision but again leads to model synchronization issues.

In this dissertation I extend and concretize the ideas presented in [AK00] and [AK03]. In [AK00] Atkinson and Kühne introduced the concept of stratified architectures. With it, software systems are described by a number of strata with decreasing levels of abstraction. Each stratum introduces a concern and thereby concretizes the system description. This strict ordering limits dependencies to adjacent strata. In [AK03] the authors complement the concept by using annotations to describe each concern. By introducing “refinement transformations”, which implement abstractly described concerns, annotations enable stepwise refinement for model based system development and—in addition—alleviate the aforementioned model synchronization issue.

In my thesis I discuss similar approaches and compare them to architecture stratification. Additionally, I present a complete implementation of the stratification concept and verify its effectiveness by applying it to a real-world project. The central element

is a combined graphical and textual model transformation language, which optimally fits the needs for stepwise refinement in a model-driven environment. This language enables fine grained and highly parameterizable model transformations. They are organized in concern-oriented transformation rules and described using a detailed metamodel. The rules are presented using a notation, which employs activity diagrams for the control flow and collaboration diagrams for the model transformation. The language also integrates a template-based code generation language and can be extended further by using hand-written code. It is integrated into a CASE tool and provides the ability to trace concerns and their implementations within a stratified architecture.

Zusammenfassung

Obwohl in der Softwareentwicklung derzeit oft noch eine klare Trennung zwischen der Entwurfs- und Implementierungsphase besteht, ist dennoch der Trend zu modellgetriebenen Ansätzen unverkennbar. Modelle sind ein primäres Entwicklungsartefakt modellgetriebener Entwicklung, sie sind jedoch oftmals sehr umfangreich und deshalb schwer zu handhaben. Daher werden Softwaresysteme oft in Komponenten unterteilt und separat modelliert. Alternativ können mehrere Ansichten verwendet werden, die bestimmte Aspekte des Systems verstecken oder in anderer Form darstellen. In beiden Fällen existieren mehrere Modelle oder zumindest mehrere Ansichten, was zu dem Problem der Modellsynchronisation führt.

Eines der Hauptziele modellgetriebener Entwicklung ist die automatische Generierung von ausführbaren Anwendungen. Auch hier existiert das Problem der Modellsynchronisation, da mehrere Informationsquellen mitunter das selbe Codefragment beeinflussen. Falls nur Teile einer Anwendung generiert werden, muss der Rest durch handgeschriebenen Code ergänzt werden. Dies erschwert die erneute Durchführung des Generierungsprozesses.

Um einen hohen Automatisierungsgrad in der modellgetriebenen Entwicklung zu erreichen, muss die gesamte Anwendung modelliert werden. Bei diesem Ansatz verbessert die Verwendung mehrerer Modelle sowohl die Verständlichkeit als auch die Genauigkeit, führt jedoch erneut zum Problem der Modellsynchronisation.

In dieser Dissertation erweitere und konkretisiere ich die Ideen aus [AK00] und [AK03]. Atkinson und Kühne stellen in [AK00] die Architekturstratifikation vor. Damit werden Softwaresysteme durch mehrere Straten mit fallendem Abstraktionsniveau beschrieben. Jedes Stratum fügt einen Aspekt hinzu und konkretisiert damit die Systembeschreibung. Eine solche strikte Ordnung beschränkt Abhängigkeiten auf benachbarte Straten. Um die Aspekte zu beschreiben, ergänzen die Autoren in [AK03] das Konzept um Annotationen. Diese ermöglichen durch die Verwendung von "Verfeinerungstransformationen" die Implementierung abstrakt beschriebener Aspekte und damit eine schrittweise Verfeinerung. Hierdurch mildern sie auch das zuvor erwähnte Problem der Modellsynchronisation.

In meiner Arbeit diskutiere ich ähnliche Ansätze und vergleiche sie mit der Architek-

turstratifikation. Ich beschreibe eine vollständige Implementierung des Stratifikationskonzepts und zeige ihre Effektivität durch die Anwendung auf ein reales Projekt. Im Zentrum steht eine grafische und textuelle Modelltransformationssprache, die optimal auf die Bedürfnisse der schrittweisen Verfeinerung in einem modellgetriebenen Umfeld abgestimmt ist. Diese Sprache ermöglicht feingranulare und hochgradig parametrisierbare Modelltransformationen, die in Transformationsregeln organisiert sind und sich an Aspekten orientieren. Ihre Beschreibung erfolgt durch ein detailliertes Metamodell. Die Notation der Regeln erfolgt durch Aktivitätsdiagramme, zur Beschreibung des Ablaufs, und Kollaborationsdiagramme, für die Modelltransformationen. Die Sprache integriert darüber hinaus eine vorlagenbasierte Codegenerierungssprache und kann durch handgeschriebenen Code erweitert werden. Sie ist in einer CASE-Werkzeug integriert und bietet die Möglichkeit, Aspekte und ihre Implementierung innerhalb einer stratifizierten Architektur nachzuverfolgen.

Preface

After finishing my diploma thesis the desire to do further research was still present. My field of expertise was an ideal match for a newly founded department at the Technische Universität Darmstadt: Metamodeling and its Applications. It presented me with the opportunity to do research and teach at the same time. After starting my work it became clear, that balancing those two wouldn't be an easy task.

The head of the department and thesis supervisor, Prof. Dr. Thomas Kühne, gave me the opportunity to choose my own research subject. I'd like to thank him for that as this is in research institutions not always the case. He also provided continued guidance during my research.

After evaluating different directions I settled on extending a previous research subject of Prof. Kühne. His work on architecture stratification provided an ideal basis for a broad research area. I focussed on model transformation, which is a major research topic within the TU Darmstadt real-time systems group lead by Prof. Dr. Andreas Schürr, who became the second reviewer for my thesis.

I wrote several joined papers with Prof. Kühne and his detailed understanding of the research subject and his experience in writing papers was very helpful. He also reviewed—along with Prof. Schürr—several drafts of my thesis.

In addition to writing papers, well-rehearsed presentations are also very important. The real-time systems department and the software technology group lead by Prof. Dr. Mira Mezini organized regular seminars, in which I was able to present my work and get valuable feedback from the department members.

During my research I supervised several study and diploma theses. All of them helped to improve the contents of my dissertation and I'm very lucky to have found students, who really enjoyed the subject and brought a lot additional input. My thanks go to Felix Klar, Daniel Bausch, Axel Schulz and Anouar Haha.

Finally I would like to thank my wife Joyce Wittur for her support, especially for reading and correcting my thesis. Also my gratitude goes to my parents Emilie and Werner Girschick and to my friends Sven Kloppenburg, Falk Fraikin and Christoph Müller, who always encouraged me to go on.

Declaration

The content of this dissertation is a product of the author's original work except where explicitly stated as otherwise.

Contents

1	Introduction	15
1.1	Model-Driven Software Development	15
1.2	Author's Contribution	16
1.3	Structure of the Dissertation	17
1.4	List of Published Papers	18
2	Trends in Software Development	21
2.1	Concepts	21
2.2	Methodologies and Research Subjects	22
2.2.1	Techniques to Address Separation of Concerns	22
2.2.2	Patterns and Frameworks	24
2.2.3	Domain-Specific Languages and Metamodeling	25
2.2.4	Model-Driven Approaches	26
2.2.5	The Engineering Processes	27
2.2.6	Software Product Lines	28
2.3	Current Model Driven Development Tools	28
2.4	Summary	35
3	Architecture Stratification	37
3.1	The Stratification Concept	37
3.2	Stepwise Refinement	39
3.3	Concern-Oriented Modeling	40

3.4	Concern Descriptions	42
3.5	Automated Refinement and Abstraction	45
3.6	Support for Editing Intermediate Strata	47
3.7	Related Work	51
4	Implementing Stratification	55
4.1	Platform Selection	55
4.2	Modeling with Fujaba	57
4.3	Fujaba Internals	62
4.4	Integrating Stratification into Fujaba	64
4.5	Extending the Modeling Language	66
4.6	Transformation Rules	67
4.7	Rule Library	67
4.8	Stratum Handling	69
4.9	Code Generation	69
4.10	User Interface	69
5	Model Transformation	73
5.1	Concepts	73
5.2	Model Transformation Classification Schemes	74
5.3	Language Requirements	78
5.4	Analysis of Transformation Languages	81
5.5	Story Diagrams and Model Transformations	89
5.6	Integration into SPin	92
5.6.1	Strata and Project Files	92
5.6.2	Metamodel synchronization	93
5.6.3	Transformation Rules	94
5.7	Integrating Code Generation into Story Diagrams	95
5.7.1	Motivation for a Code Generation Facility	95

5.7.2	Criteria for a Code Generation Language	99
5.7.3	Integrating Templates into Story Diagrams	102
5.7.4	Context Transfer between Model and Template	103
5.7.5	Template Stereotypes	104
5.7.6	Template Macro Definitions	106
5.7.7	Measuring Transformation Performance	106
5.8	Preservation of User Edits	107
5.8.1	Target Incremental Transformations	107
5.8.2	Traceability and Model Transformation	108
5.8.3	Stratification Traceability	110
5.8.4	Enabling Incremental Transformation in Stratification	110
5.8.5	Stratification Identifiers	112
5.8.6	Integration into the Transformation Process	114
5.8.7	Related Work	118
5.9	Summary	118
6	Case Study	121
6.1	The Java Pet Store	121
6.2	Other Implementations	122
6.3	The Pet Store Architecture	124
6.3.1	The Client Tier	125
6.3.2	The Web Tier	125
6.3.3	The EJB Tier	126
6.3.4	The Database Tier	127
6.3.5	Packaging, Deployment and Administration	127
6.4	Stratifying the Pet Store	127
6.4.1	Determining System Concerns	128
6.4.2	From Concerns to Stratification	129
6.4.3	Transformation Rules	134

6.5	Evaluation	138
7	Conclusion	141
7.1	Summary	141
7.2	Future Work	143
7.2.1	Stratification	143
7.2.2	Transformation	144
	Bibliography	145

Figures

2.1	Table of model-driven software development tools	30
3.1	A stratified architecture	38
3.2	Interaction refinement [AK00]	41
3.3	Annotated links [AK03]	43
3.4	Annotated links on subsequent strata [AK03]	43
3.5	A collaboration within a class diagram [OMG05b]	44
3.6	Collaboration notation for visitor design pattern	45
3.7	Implemented visitor design pattern	46
3.8	Two-sided strata	50
4.1	Screenshot of Fujaba 5.1	58
4.2	Example of Story Driven Modeling: class diagram	59
4.3	Example of Story Driven Modeling: story diagram of <code>printByCity</code>	60
4.4	Simplified UML metamodel	62
4.5	Abstract Syntax Graph metamodel [Kla05]	63
4.6	Architecture of Fujaba and the plugins implementing stratification	65
4.7	The SPin metamodel extension for annotations	66
4.8	Model for transformation rules metadata	68
4.9	Fujaba's class diagram editor with an annotated class diagram	70
4.10	SPin's annotation editor	70
4.11	SPin's strata navigator and the strata description window	71

5.1	A model fragment represented in concrete and abstract syntax	75
5.2	Comparison of transformation languages	88
5.3	Fujaba's story pattern transition editor	89
5.4	Fujaba's story pattern object editor	90
5.5	Fujaba's story pattern link editor	91
5.6	Model transformation using a story diagram	91
5.7	SPin's project file name structure	92
5.8	Example of the VMSynchronizer	94
5.9	An example transformation rule with one annotation link	95
5.10	Swing Data Dialog: The annotated class diagram	96
5.11	Swing Data Dialog: The result after the transformation	97
5.12	Swing Data Dialog: An excerpt of the transformation rule	98
5.13	A statement activity responsible for code generation	99
5.14	Comparison of templates languages	101
5.15	The template activity equivalent to Figure 5.13	102
5.16	Control and data flow within a template activity [Gir08]	104
5.17	Execution time of statement activity vs. template activity	107
5.18	Table of target incrementality requiring scenarios	108
5.19	Control and data flow during a transformation with controlled merge	111
5.20	Parts of a stratification identifier (SID)	113
5.21	Implementing access methods: The topmost stratum	114
5.22	Implementing access methods: The transformation rule	115
5.23	Implementing access methods: Second stratum and generated method	116
5.24	Implementing access methods: SID of the method <code>getMyBoolean</code> . .	118
6.1	Main components of the Java Pet Store (adapted from [SSJ02]) . . .	122
6.2	Screenshot of Pet Store front-end running in a browser	123
6.3	Four tier architecture of the Pet Store	124
6.4	Screenshot of Pet Store shopping cart [SSJ02]	125

6.5	Web Application Framework control flow (adapted from [SSJ02]) . . .	126
6.6	Pet Store Concerns ordered by dependency	129
6.7	Pet Store: initial stratum	129
6.8	Pet Store: second stratum with refined WebShop	130
6.9	Pet Store: third stratum with refined WebTier	131
6.10	Pet Store: fourth stratum with refined WebShopUserInterface	131
6.11	Pet Store: fifth stratum with refined WebShopUserInterfaceToWAF .	132
6.12	Pet Store: sixth stratum with refined Persistence	132
6.13	Pet Store: seventh stratum with refined IntegrateBusinessLogic . . .	133
6.14	Pet Store: eighth stratum with refined SessionControlling	134
6.15	Pet Store: result of “FrameworkInheritance” refinement	135
6.16	Pet Store: syntactic sugar within transformation rules	136
6.17	Pet Store: template activity “MethodMaker”	136
6.18	Pet Store: template activity “WriteToFile” and “UserMessage”	137
6.19	Pet Store: SessionBean annotation with parameter editor	137
6.20	Pet Store: SessionBean transformation rule	139
6.21	Pet Store: generated <code>perform</code> method with hook spots	140

Listings

4.1	Excerpt of Java code for the class diagram in Figure 4.2	59
4.2	Generated Java code for method <code>printByCity</code>	61
5.1	The Velocity template equivalent to Figure 5.13	100
5.2	Stereotype template: <code>Filewriter.vm</code>	105
5.3	Stereotype parameter definition file: <code>Filewriter.params</code>	105
5.4	Macro definition: <code>addToImports.vm</code>	106
5.5	Excerpt of Java code for the transformation rule from Figure 5.22 . .	117

Chapter 1

Introduction

The introduction starts with a brief overview of ongoing challenges in software engineering and how they are addressed by model-driven software development. Then the author's contributions and the structure of this dissertation are presented.

1.1 Model-Driven Software Development

Brooks wrote in his 1986's essay "No Silver Bullet - Essence and Accidents of Software Engineering" [Bro87] about the ever growing "essential complexity" of software development. It originates from the necessity to solve more complex problems and cannot be removed by using "better" technology. In contrast, "accidental complexity" relates to the used technology and often can be alleviated.

Although the "silver bullet" for software development has yet to come, steady progress in the technological and methodical space helps to deal with the complexity of today's software. Since the invention of high level languages like Fortran [Bac98] and the wider adoption of object-orientation in the 1990s several technological advancements have been made.

According to the CHAOS report, published biyearly since 1994 by the Standish Group, in 2006¹ only one third of the examined software projects were completed in time and budget. The remaining projects missed either time, budget or functionality constraints (46%) or weren't completed at all (19%). Compared to 1994's figures [Sta94], where only 16 percent were successful and 31 percent failed, an improvement can be noted but still leaves enough room for further recovery. The already existing rise can in part be attributed to better project management with standardized processes, closer customer interaction and the application of standard architectures.

¹Numbers taken from <http://www.sdtimes.com/content/article.aspx?ArticleID=30247> (checked August 2009).

As stated by the report, one of the key factors is iterative development. In conjunction with closer customer interaction it becomes clear that a common understanding on the final product between developers and customers has to be found. This eventually led to the creation of standardized modeling languages such as UML².

Modeling languages are primarily used in requirements and design phases. During implementation often design decisions have to be corrected and—in order to keep documentation up-to-date and be able to reuse it in subsequent iterations—the model has to be updated manually. CASE³ tools for both design and implementation are steadily improved and tighter integration offers easier synchronization between the two phases. Still widespread adoption of CASE tools has yet to come and many tasks are currently done manually.

In the introduction of the IEEE Computer special issue on Model-Driven Engineering [Sch06] Douglas C. Schmidt gives an overview on the history of CASE tools, which eventually lead to today's model-driven approaches. The first CASE tools appeared in the 1980s. They used general-purpose graphical programming representations for modeling software systems. According to Schmidt one of the reasons for their failure was the inability to map to the underlying platform due to missing platform abstractions and inferior translation technology. Most of the tools didn't scale to production-scale systems and targeted only proprietary execution environments.

He further argues, that the “Advances in languages and platforms during the past two decades have raised the level of software abstractions available to developers, thereby alleviating one impediment to earlier CASE efforts.”. Due to the risen complexity of today's software systems there's still need for better development technologies. He describes Model-Driven Engineering (MDE) as “A promising approach to address platform complexity”.

According to Schmidt, MDE needs to combine “Domain-specific modeling languages” (DSMLs) and “Transformation engines and generators”. DSMLs are used to formalize a particular domain, they are described using metamodels. Together with the advances of third generation languages the gap between model and code—which was one of the reasons for the CASE tools' failure—can be reduced to a controllable size.

1.2 Author's Contribution

This dissertation is concerned with the model-driven development concept of “architecture stratification”. The three main contributions are:

²Unified Modeling Language, see <http://www.uml.org> (checked August 2009)

³Computer Aided Software Engineering

1. An elaborate description of architecture stratification, building on basic principles outlined in previous work. Stepwise refinement and concern-oriented modeling is related to the idea of interaction refinement and architecture stratification. In addition, presentation techniques for stratified architectures and related work is discussed.
2. An analysis of the stratification approach and related technologies formed the basis of a transformation language for stratification. It is based on a graph transformation language, which combines an imperative control structure with a declarative model transformation.

In order to support both graphical and textual artifacts, a template based code generation mechanism was added. This novel approach offers seamless integration into the graphical transformation process.

Furthermore, an extension for transparent creation of traceability links was devised. Combined with a model synchronization mechanism it enables modeling a software system on several abstraction layers.

The resulting transformation language is ideally suited for the implementation of concerns within a stratified architecture.

3. Under the authors supervision and support, several student projects created a complete implementation of the stratification concept including the mentioned model transformation language. A case-study showed its capableness for real-world application development. Here, various concerns needed to be implemented, ranging from low level design pattern implementations to framework integration all the way up to high level concerns like persistency and user interface creation.

The author concludes, that architecture stratification is a feasible approach to model-driven development addressing several shortcomings of similar approaches. The detailed elaboration on the subject is not limited to architecture stratification but to model-driven development and model transformation in general.

Although the developed implementation is not yet ready for a productive environment it forms a useful basis for future research.

1.3 Structure of the Dissertation

Chapter 1 explains why model based software development approaches are in demand and shows, how architecture stratification fits into this context. The chapter finishes with an overview of the dissertation.

It is followed by Chapter 2, which describes the current state of software development with a focus on model based technologies. This includes both basic methodologies and concrete tools from commercial vendors and research groups. Their shortcomings and strengths are discussed and set in relation to each other.

An in-depth description of architecture stratification follows in Chapter 3. It gives details on the fundamentals of stratification, notational issues, distribution of system concerns to the strata and concern specification by means of parameterizable model annotations.

Chapter 4 describes an integrated development environment supporting architecture stratification. It provides the ability to describe applications on several abstraction layers. Although the approach can be applied to arbitrary description types, the current implementation uses UML class diagrams with attached Java code fragments. This combination of a graphical model with textual behavior description is ideally suited for stratification. As changes on one stratum affect subsequent strata, the tool automatically propagates changes and therefore allows free navigation between strata.

To support automatic propagation and allow automated implementation of abstractly defined concerns, a powerful model transformation language is needed. Chapter 5 introduces a specially tailored language, that fulfills the requirements for artifact transformation and generation needed in the context of stratification. The language is compared to similar approaches from other research groups as well as commercially available tools.

The case study in Chapter 6 uses architecture stratification and the transformation language to rebuilt an existing J2EE web shop application.

Chapter 7 summarizes the dissertation and gives an outlook on future work.

1.4 List of Published Papers

1. Martin Girschick. Integrating Template based Code Generation into Graphical Model Transformation. In Thomas Kühne, Wolfgang Reisig, Friedrich Steimann (Hrsg.), *Modellierung 2008*, Berlin, GI LNI P-127, p. 27-41, March 2008 [Gir08]
2. Martin Girschick, Thomas Kühne, and Felix Klar. Generating Systems from Multiple Levels of Abstraction. In *Proceedings of TEAA 2006*, In LNCS Volume 4473/2007, pp. 127-141, Berlin, Germany, 2007 [GKK06]
3. Thomas Kühne, Martin Girschick, and Felix Klar. Tool Support for Architecture Stratification. In H.C. Mayr, Ruth Breu (eds), *Modellierung 2006*, Innsbruck, GI LNI P-82, p.213-222, March 2006 [KGK06]

4. Felix Klar, Thomas Kühne and Martin Girschick. SPin – A Fujaba Plugin for Architecture Stratification. In Fujaba Days 2005, Paderborn, September 2005 [KKG05]

Chapter 2

Trends in Software Development

The previous chapter showed the demand for new and improved software development technologies. In addition to well established techniques, such as patterns and frameworks, new composition mechanisms to address separation of concerns are explored. The trend towards modeling and model-driven software development lead to the creation of a broad range of tools. Still, all these techniques would be useless without an established process for development.

In this chapter an overview of academic and commercial activities concerning software development is given. Section 2.1 describes the software engineering terms. Section 2.2 focusses on general ideas to improve software development, Section 2.3 then introduces concrete model-driven development tools. It is followed by a discussion, which outlines advantages and deficiencies of these tools.

2.1 Concepts

Firstly some definitions concerning general software development terms are required.

- **Functional requirements** describe *what* a system must do in terms of input, output and behavior. **Non-functional requirements** are additional properties or constraints, which are used to classify the system's operation. This includes aspects such as performance, usability, maintainability and security.
- A software consists of **units** (e.g. classes, methods, build files, test cases, configuration files).
- A **concern** reifies one or more requirements into a more concrete concern description. This **concern description** also contains information on its purpose and on how it affects the software units. Thus a concern serves as an organization principle and connects the requirements to the actual software. A concern

may also refer to one or more **concern realizations**, which implement the concern into the software.

Most of today's new software systems are written in object-oriented languages such as C++ or Java. Although precise sources are missing, this can be deduced from statistical data available from several sources. According to joinvision, an analysis of current job offerings¹ reveals, that six of the eleven most often named languages are object oriented with the topmost two entries Java and C++. Similar tendencies can be seen in automated internet search comparisons from "Language Popularity"² or TIOBE³.

In object-oriented software systems the units are classes, which serve as the predominant organization principle. One of the challenges of software development is to find a suitable mapping from concerns to units. For maintenance reasons, a concern should be addressed by a minimal set of units and a unit shall address only a minimal set of concerns. This idea was first described in 1974 by Edsger W. Dijkstra [Dij82] and termed "Separation of Concerns". As a unit is usually affected by more than one concern, a clear separation is not always possible, resulting in an effect called "tangling". The same effect can be seen, when one concern affects several units. This is called "scattering". Both effects combined result in "Cross-cutting concerns" [KLM⁺97].

2.2 Methodologies and Research Subjects

To tame the increasing complexity, new technologies for software development are required. This involves several areas from improved languages via more flexible frameworks and better tools to standardized and tailored processes. A selection of relevant topics are discussed in this section.

2.2.1 Techniques to Address Separation of Concerns

One of the reasons for the existence of cross-cutting concerns is the "tyranny of the dominant decomposition" as described by Ossher et al. [TOHSMS99]. Current object-oriented languages usually offer only restricted mechanisms for composition and decomposition by providing one dimension for separation. If a concern does

¹<http://www.joinvision.com/jv/x/n/t-TStatOfferHistoryDetail-statistic-pl-loc-en> (checked August 2009)

²<http://www.langpop.com/> (checked August 2009)

³<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html> (checked August 2009)

not fit to that dimension it usually cross-cuts to other units of the software, thus complicating development and maintenance.

Several techniques have been developed to tackle this problem from the language perspective. Ossher et al. describe a concept named “Hyperspaces” [TOHSMS99], which uses “Hyperslices” to capture fragments relevant to one concern. A precursor to this generic concept was “Subject-Oriented Programming” (SOP) [HO93], which was implemented for Java in a tool suite named HyperJ⁴. Here, a Hyperslice is a set of partial classes, which implement a certain concern. This concern, also called “Subject”, thus presents a view on all classes of a system. HyperJ offers several composition mechanisms including extensions, configurations and traceability between all software artifacts (e.g. requirements, design and code).

A similar approach is taken by “Aspect-Oriented Programming” (AOP), first introduced by Kiczales et al. [KLM⁺97]. It adds a second decomposition mechanism to the dominant class paradigm of object-oriented languages. Aspects are used to implement cross-cutting concerns by adding behavior and structure to an existing object-oriented software. This is accomplished by executing additional code (called an “advice”) at certain points within the object-oriented code. These so-called join points are described using a separate “pointcut” language. This composition process, which integrates the aspects into the software, is called “weaving”.

The most popular implementation of AOP is AspectJ⁵. As the name implies it is based on Java and—as the complete functionality is introduced at compile-time—works with standard Java Virtual Machines⁶. AspectJ is used in commercial software projects (e.g. The Spring framework⁷).

“Feature-Oriented Programming” (FOP) [Pre97] proposes a different composition mechanism. Features resemble (incomplete) subclasses containing only the core functionality. Any interaction between features is defined separately, allowing a higher level of reuse. A similar approach to feature composition is taken by Batory et al. [BSR04]. Instead of simple compositions they describe a concept named “feature refinement”, which not only composes features but also adapts elements accordingly. Their implementation—named AHEAD⁸—uses Jak (a superset of Java) to represent programs. Jak embeds languages for refinements, state machines and metaprogramming into Java. A refinement is described by a function, which transforms both Jak artifacts and other (non code) artifacts. As these functions are based on a mathematical foundation, additional reasoning and optimizations can be applied.

⁴<http://www.alphaworks.ibm.com/tech/hyperj> (checked August 2009)

⁵<http://www.eclipse.org/aspectj/> (checked August 2009)

⁶It has to be noted, that AspectJ also allows load-time weaving, which means the aspects can be selectively enabled and disabled at loading time.

⁷<http://www.springframework.org/> (checked August 2009)

⁸Algebraic Hierarchical Equations for Application Design

Mezini and Ostermann [MO04] describe three disadvantages of feature-oriented approaches such as AHEAD. They claim that “FOAs⁹ are purely hierarchical”, which means that an additional feature is always placed on top of an existing one, so free combination of features is not possible. The second disadvantage is “lack of appropriate support for reuse”, because the newly added feature is tangled with the existing program and cannot be deployed elsewhere. As a third problem they note “the lack of support for dynamic configuration”, which is related to the fact, that features cannot be deployed at runtime. The authors propose a different concept: The language CaesarJ [AGMO06], which adds several mechanisms to address these problems. For instance cross-cutting composition to address concerns, bindings for feature composition and dynamic aspect control.

A similar combination of AOP and FOP is described by Apel et al. [ALS06]. They describe “Aspectual Mixin Layers”, which integrate AOP and FOP on an architectural level by using features as higher level structures and aspects to resolve cross-cutting concerns.

Most of the aforementioned approaches have not been widely accepted, yet. They rather present scientific work, which eventually leads to new and more advanced programming languages. A critical discussion of AOP concepts can be found in an essay from Steimann. He argues, “that much of aspect-oriented programming’s success seems to be based on the conception that it improves both modularity and the structure of code, while in fact, it works against the primary purposes of the two...” [Ste06].

2.2.2 Patterns and Frameworks

The following techniques are already widely used within software development projects. For instance the benefits of design patterns have been recognized for quite a while. Those described by Gamma et al. [GHJV95] can be found in virtually any software system. Design Patterns formally document a (reusable) solution for a common design problem in a certain field. They can be applied on a architectural level [BMR⁺96] or on a smaller scale. Useful combinations of patterns are also known as pattern languages (see [BHS07] and the PLoP¹⁰ conference series).

Jan Bosch [Bos98] describes a few problems associated with the implementation of patterns. For instance they cannot be easily detected within a software and reusability is limited, because the pattern implementation is tangled with the remaining application. Also, the implementation of the pattern itself often involves tedious work by implementing the often trivial behavior. The author proposes a “layered object model”, which provides language support for design patterns.

⁹feature-oriented approaches

¹⁰Pattern Languages of Programs, <http://hillside.net/conferences/plop.htm>

Johnson [Joh97] gives an introduction to the use and creation of frameworks and how they differ from large-scale patterns and components. He writes “They are more abstract and flexible than components, but more concrete and easier to reuse than a pure design.”. One problem of frameworks can be seen in the correct usage. Without sufficient documentation or demonstration artifacts it is often complicated to find the needed interactions and extension points between the framework and the application under development. An approach to automate framework instantiations by annotating these extension points appropriately is described by Büchner and Matthes [BM06].

Frameworks also help to “abstract away” technical detail. Johnson [Joh97] describes the similarities between frameworks and application generators. The latter are based on high-level domain specific languages, whereas frameworks employ the language they are developed in to model the abstractions.

2.2.3 Domain-Specific Languages and Metamodeling

In contrast to general purpose languages such as Java or C++, domain-specific languages (DSLs) are designed for a specific application area. This can either be a technical domain or a functional domain. DSLs can either be embedded into a general purpose language (called internal DSL, e.g. LINQ for C#¹¹) or form a separate syntax (external DSL, e.g. HTML, SQL). DSLs enable a concise and precise description of domain aspects.

The process of creating DSLs is described in detail by Czarnecki and Eisenecker [CE00, Chapter 2]. This “Domain Engineering” involves Domain Analysis, Domain Design and Domain Implementation. As an example, the first known Domain Engineering System “Draco”, developed 1980 by James Neighbors [Nei80], is described. It employs transformation to implement domain-specific languages.

Although not limited to them, the term DSL is mostly used for textual languages. In the context of graphical languages, the terms model and metamodel are used instead. One example is the Unified Modeling Language (UML), which is defined by the UML metamodel.

UML was the result of a standardization process for modeling languages in the 1990s. First versions were published by Rational Software¹², which was founded by Grady Booch, Ivar Jacobson und James Rumbaugh. In 1997, the Object Management Group (OMG)¹³ continued further standardization efforts. With the introduction of version 2.0 [OMG05c, OMG05a], UML was based on the meta-metamodel standard

¹¹As of June 2008 the LINQ Project can be found at <http://msdn.microsoft.com/en-us/netframework/aa904594.aspx>.

¹²now part of IBM, <http://www.ibm.com/rational/> (checked August 2009)

¹³<http://www.omg.org> (checked August 2009)

“Meta-Object Facility” (MOF). In order to support domain specific aspects, UML offers a lightweight extension mechanism through UML profiles. Most tools internally use different meta-metamodels to represent models and offer import and export of MOF models using XMI files. XMI (XML Metadata Interchange) [OMG07b] is OMG’s official serialization mechanism (not only) for MOF data. The MOF specification defines two variants named CMOF (for complete MOF) and EMOF (for essential MOF). The Eclipse Modeling Framework (EMF) uses a metamodel named Ecore, which is based on EMOF.

DSLs are often designed with automated code generation in mind, whereas graphical models sometimes serve only informal purposes. Again this can be seen in UML with its different diagram types, of which some cannot be transformed into executable code (e.g. use case diagrams). Fowler and Scott [FS03] describe three modes in which UML can be used: “Sketches” are used to communicate ideas informally, “blueprints” describe a system detailed enough to program it, “programming language” is even more precise and enables automatic code generation.

First used for programming, aspect orientation is now also applied to analysis and design. An overview of current research is given in “Survey of Analysis and Design Approaches” [CRS⁺05] and “A Survey on Aspect-Oriented Modeling Approaches” [SSK⁺07]. Currently, these are primarily focussed towards design without automatic generation of executable systems and thus belong into the aforementioned “blueprints” category. Of more interest in this dissertation is the category “programming language”, where UML models serve as input to automated generation of software. This variant can also be called “model-driven” development.

2.2.4 Model-Driven Approaches

In model-driven software development (MDSD) models become the primary development artifact. They serve as input to model transformations systems, which transform them into a different representation. After one or more transformation steps all necessary artifacts for the executable software system have been produced.

The CASE tools, first appearing in the 1980s, can be seen as ancestors of today’s model-driven efforts. As argued by Douglas C. Schmidt [Sch06] the advances in the past two decades made it possible to realize working model-driven approaches by combining DSLs with model transformation. A slightly different approach is taken by the Object Management Group, which proposes the “Model Driven Architecture” (MDA)¹⁴. They recommend the use of the generic UML as a modeling language in combination with specific UML profiles¹⁵.

The MDA vision aims at addressing the separation of functional and technical as-

¹⁴<http://www.omg.org/mda/> (checked August 2009)

¹⁵Other modeling languages, which are based on MOF, can be used as well.

pects by introducing platform independent and platform specific models (PIM and PSM, respectively). Although often misinterpreted, this is not a limitation to two abstraction levels. Instead each model can play the role of a PIM or PSM depending on whether it serves as source or target of the model transformation.

The wide adoption of the MDA idea by commercial software vendors can be ascribed to its vage definition, which can mainly be found in the MDA Guide 1.0.1 [MJ03]. Thus, many model-based tools which offer UML diagram types and some sort of model transformation or code generation, can call themselves “MDA tools”¹⁶. Section 2.3 gives an overview of diverse model-driven development tools of which some are based on the MDA vision.

Often, available tools only offer code generation facilities, refraining from model-to-model transformations, which transform platform independent to platform specific models. In addition, available transformation languages, both for code generation and model-to-model transformation, are usually proprietary to the used tool. Thus, in 2005 the OMG proposed “Query/View/Transformation” (QVT) [OMG07a] to standardize model transformation languages for MOF and MDA. Implementations relevant to this dissertation are discussed in Section 5.4.

Despite the fact that models are considered primary artifacts within model-driven approaches, it is often not possible to generate the complete application purely from models. A common technique is to create the static structure of the application from a graphical model and to add dynamic behavior by introducing platform specific code into the resulting artifacts. This mixture of generated and hand written code is considered bad style (cf. [VB05]) as it complicates synchronization between these two representations. The automatic synchronization is often called “round-trip engineering” and appears between model and code but also between two or even more models. The problems surrounding round-trip engineering are discussed in detail by Sendall and Küster [SK04].

2.2.5 The Engineering Processes

Every larger software project needs an established process to enable structured development. It usually consists of the description of roles, activities and products within the development cycle. The process is not limited to the product itself but may also include supporting tasks such as project management, requirements, risk analysis and quality control. Early process models, such as the waterfall model (cf. [Roy87]), were purely linear with a list of fixed activities.

Current approaches, such as the “Rational Unified Process” (RUP) [JBR99], offer much more flexibility by allowing specific tailoring to fit the current project and

¹⁶A list of existing MDA tools can be found at <http://www.omg.org/mda/committed-products.htm> (last checked June 2008).

company regulations. RUP is a framework for iterative and incremental development. In iterative development, existing functionality is improved (or fixed), in incremental development new functionality is added. Both are important aspects, which address the shortcomings of linear approaches.

RUP is focussed on work products and documentation, which are especially important in large projects. A different focus is taken by agile approaches such as “eXtreme Programming” (XP) [BA04] or “Scrum” [TN86]. The core principles of these approaches are collected in the “Agile Manifesto”¹⁷. The first—and probably most important—principle is “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.”. This “release early, release often” strategy requires many incremental and iterative release cycles and thus development tools and processes capable of handling it.

2.2.6 Software Product Lines

Parnas [Par76] defines program families “as sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members”. A more recent definition can also be found in [Wit96]: “A product family is the group of products that can be built from a common set of assets.” and further “A product line is a group of products sharing a common, managed set of features that satisfy the specific needs of a selected market.”.

Thus, software product lines are ideal candidates for automated software development using model-driven approaches. One example of this combination is described by Greenfield et al. [GS04]: “Software Factories is a methodology for using domain specific languages and other technologies to automate Software Product Lines.”. Instead of different levels of platform specificity, a Software Factory Schema is organized by concerns and different levels of abstraction. In its simplified version this can be seen as a grid, where a row represents an abstraction level and a column a concern. However, as a concern does not exist on all abstraction levels and concerns affect each other in multiple ways, in reality the schema is rather a graph, with nodes as models and edges as dependencies. These dependencies manifest themselves in model transformations.

2.3 Current Model Driven Development Tools

This section introduces current modeling tools, which are appropriate for model-driven development. Both commercial tools and academic prototypes are presented.

¹⁷<http://www.agilemanifesto.org/principles.html> (checked August 2009)

In order to give a broad overview, a representative selection from the currently available tools has been chosen.

Although the presented tools use different approaches to achieve model-driven development, they also share some similarities. On one hand there are the tools with a fixed set of modeling languages, which are often limited to a selection of commonly used UML diagram types. Some tools enable “lightweight” extensions using the definition of UML profiles or even “heavyweight” extensions, which add new diagram elements to UML. On the other hand there are DSL based tools, which allow the creation of new metamodels. However this added flexibility comes at the price of more complexity during the design of the metamodel and creation of associated editors and code generators.

All tools are capable of generating code from at least a part of the supplied model types. Most of them also allow for the definition of custom model-to-code transformations, often using a template-based language¹⁸. This single step directly from model to code is only viable, when the “abstraction gap” between both is not too big. If the model describes the system on a very high-level of abstraction, the transformation is rather complex or the model even may not contain enough information to create the complete code. If the model is very close to the code, it is merely a different representation of it, not providing the abstraction benefit of model-driven development.

This is one of the reasons, why the MDA approach, described in Section 2.2.4, proposes the use of platform independent and platform specific models. Instead of one model-to-code step, it splits the transformation in several steps, narrowing the gap. Realizations often use one or more model-to-model transformations and a final code generation step.

Support for multiple models and model-to-model transformation is available for most tools. However, this support is often limited to a fixed set of models and transformations between them. This set is chosen at project start from a list of available development scenario. To gain more flexibility, a model-to-model transformation language is required.

As argued in Section 2.2.5, iterative and incremental software development approaches are becoming more important. In order to support this in a model-driven environment, model transformations have to be executed repeatedly and be able to deal appropriately with model changes. This automation support combined with traceability—which enables tracking changes across models and code—is not trivial and thus not fully implemented in most tools.

The table in Figure 2.1 gives an overview of the tools and evaluates them according

¹⁸Czarnecki and Helsén [CH03] see some deficiencies in these languages, as they are usually untyped and thus may produce incorrect code.

Name	Company or University URL and Publication	metamodel support	model transformation language		
			to code	to model	automation
Fujaba Univ. Paderborn/Kassel/Darmstadt http://www.fujaba.de/ [NNZ00]		API	+		
MagicDraw http://www.magicdraw.com/ [Inc08]	No Magic Inc.	profiles		+	
Together http://www.borland.com/us/products/together/ [Fon06]	Borland	EMF	++	++	++
Rational Software Architect http://ibm.com/software/awdtools/swarchitect/ [Cer04]	IBM	profiles/EMF	++	++	+
OptimalJ http://www.compuware.com [Com06]	Compuware	-	++	+	++
ArcStyler http://www.interactive-objects.com/ [Int05]	Interactive Objects	-	++		+
OlivaNova http://www.care-t.com/products/index.asp [Her03a]	CARE-Technologies	-			++
Mia-Studio http://www.mia-software.com/en/ [MS07]	Mia-Software	no editor	++	++	++
MetaEdit+ http://www.metacase.com/ [PK07]	MetaCase	editor	++		
DoME http://www.htc.honeywell.com/dome/ [OSBE01]	Honeywell	editor	++	++	
GME/GReAT http://www.isis.vanderbilt.edu/projects/gme/ [KML ⁺ 04]	Vanderbilt University	editor & API	+	++	++
DSL Tools and GAT/GAX http://msdn.microsoft.com/architecture/aa699360.aspx [Coo07]	Microsoft	editor	++		++
Domain Workbench http://www.intentsoft.com/ [SCC06]	Intentional Software	editor	+	++	++

Figure 2.1: Table of model-driven software development tools

to the mentioned criteria. The column “metamodel support” shows the capability for custom definition or extension of metamodels. Full “editor” support includes the ability to design metamodel and visualization from within the tool. Some tools use “Java” to realize metamodels. “Profiles” enable the simple extension of existing UML diagram types. Two tools support the use of the Eclipse “EMF” Ecore metamodel, which allows the use of other Eclipse plugins for editing.

The next three columns rate the transformation languages and automation support. A single plus sign means basic support, a double sign reflects full support. The following paragraphs give further details on the evaluated tools. Details concerning the transformation languages can be found in Section 5.4.

The open-source CASE tool **Fujaba** [NNZ00] is developed by the universities of Kassel, Paderborn and Darmstadt. It offers basic support for a few UML 1.x diagram types including code generation for Java and C++. A plugin API allows the extension of existing and addition of new diagram types. Although the metamodel can be defined using class diagrams, the visualization has to be programmed manually.

An associated plugin generates code for UML class diagrams, method behavior is modelled by a combination of activity and collaboration diagrams. These so-called story diagrams can also be used for model-to-model transformation (cf. Section 5.5 and [GGZ⁺05]). This requires additional plugins, which may also implement needed automation support for multi-level model-driven development.

The commercial tool **MagicDraw** [Inc08] from No Magic Inc. supports eleven UML 2.x diagram types and several other modeling languages (for some types code generation is available). Simple model transformations—primarily to implement design patterns—are available. It also supports the definition of own transformations using Java code. Due to its good model editor and plugin support it also offers integration with some of the following tools (OptimalJ, ArcStyler, Mia-Studio).

Earlier versions of Borland **Together** [Fon06] used an extendable template-based mechanism for implementing patterns. Model manipulations were possible by using plugins written in Java. Current versions are based on the Eclipse platform and offer support for UML 2.x and custom DSLs. In addition to previous pattern based transformation, QVT/operational [OMG07a] and several template based languages are available. Transformations can be automatized using build scripts. Basic support for tracing is available but incremental transformations are not supported.

Like Together, IBM’s Rational Rose family [Cer04] started with simple design pattern support. Rational Rose XDE adds extensive model transformation capabilities including code and pattern templates. In the current release—**Rational Software Architect (RSA)**, which is based on the Eclipse platform—these capabilities were further extended. In addition to UML 2.x including profiles, support for Ecore metamodels was added. The application of patterns and transformations is stored and can be re-executed later in order to update destination models. Patterns can be associated with templates to perform model-to-code transformations. The model transformation is specified using Java or a simple mapping language. Transformation sets can be packaged to so-called “frameworks”, which can then be used for automated application development.

OptimalJ¹⁹ [Com06] from Compuware is a Java-oriented model-driven development environment specialized to generate J2EE applications. It supports editable source code regions (so-called “free blocks”), which can be used to add hand-written code. These blocks are automatically retained upon re-generation. Generated source code fragments are locked and cannot be edited. OptimalJ provides a guided development process, where first a type of application is selected and then the provided model templates have to be completed by the developer. For example a database definition starts with a domain model²⁰, which is transformed into a database model and can then be optimized further, before the actual database is created. This ap-

¹⁹After an internal reorganization, Compuware decided in May 2008 to discontinue further development and distribution of OptimalJ.

²⁰This is not to be confused with the creation of a domain specific language.

proach basically consists of a chain of transformations. These are differentiated between “technology patterns” (PIM to PSM, model-to-model) and “application patterns” (PSM to code). The latter are implemented using a template language. Model transformation follows a “structure-driven” approach [CH06]: The OptimalJ framework creates the structure of the destination model and the actual objects are created using Java code.

The MDA tool **ArcStyler** [Int05] from Interactive Objects follows the MDA approach where a platform independent model (PIM) is completely parameterized and then transformed to a new platform specific model (PSM). ArcStyler collects transformations in “cartridges”, which are specific to certain destination platforms. Although these cartridges are defined graphically, the actual transformation is implemented in a script language named JPython and primarily used for model-to-code transformations. ArcStyler currently only supports UML 1.x models.

OlivaNova [Her03a] from CARE Technologies offers an extensive set of code generators but does not allow for the definition of custom transformations. The highly guided development process is geared towards .NET, COM+ and J2EE applications and aims at the complete generation of all code without any manual written parts. While suitable for simple scenarios, the employed modeling languages are not sufficient for more complex applications.

More flexibility is available in **Mia-Studio** [MS07] from Mia-Software. It allows for the definition of custom metamodels and transformations. Metamodels have to be implemented manually using Java classes, no custom visualizations can be used. Transformations are defined using a wizard-based process, where first the matching context is described and then the needed transformation actions. The actions are simple assignments but can also call Java code for more complex transformations. Workflow automation is provided by so-called “scenarios”. Similar to OptimalJ, code regions for hand-written code can be defined.

The following tools focus on domain specific languages including the definition of customized textual and graphical editors.

MetaEdit+ [PK07] from MetaCase provides out-of-the-box support for 70 different languages, albeit executable code generation is only available for UML class diagrams. Additional metamodels and code generators can be added, a debugger for the generator is available. An interesting aspect is the tight connection between the metamodel and the code generator. Each type within the metamodel is directly connected to a script within the code generator. On the one hand this simplifies the creation of the code generator but also limits the possibilities for the generated code.

The meta-CASE system Honeywell **DoME** [OSBE01] also support model-to-model transformation using a pattern based transformation language. It can be used to transform between different models but also to create parameterized patterns within

one model. The parameterization includes defined ports, which connect to the remaining model and can be used to implement design patterns. Code generation is accomplished using a separate template language and—similar to MetaEdit+—can also be attached to the types of the metamodel (cf. [SOEB03]).

GME²¹ [KML⁺04], developed by Vanderbilt University, is an open-source platform for the definition and use of domain specific languages. Special emphasis has been placed on the composability of metamodels. This allows for the definition of view-points, which combine data from different models in one view. Also possible is the extension of existing metamodels, for instance to provide annotation mechanisms. By registering hooks and actions with the types of the metamodel, transformations can be executed automatically during modeling. Transformation languages can be added as plugins. **GReAT**²² [AKK⁺06] (also Vanderbilt University) offers a very complex model-to-model transformation engine based on graph-transformation and a graphical defined control flow. A second plugin named C-SAW²³ [GLZ06] (University of Alabama at Birmingham) uses a script language similar to OCL²⁴ to define simple transformations. It has been designed for weaving of aspects on model level.

Microsoft’s vision for model-driven development—Software Factories—has already been mentioned in the previous section. A full implementation does not exist, yet. The **DSL tools** [Coo07] for Visual Studio provide support for metamodel creation including custom editors and code generation capabilities. Code generation is accomplished using a C#-based template language. Automation support is available using the **Guidance Automation Toolkit and Extensions** (GAT/GAX) [RA06]. Currently no model-to-model transformation capabilities exist.

The **Domain Workbench** [CE00, Chapter 11], [SCC06] from Intentional Software unifies the concepts of model and code. All model data is stored in a tree-like data structure with additional “virtual” edges within the tree. It can be presented and edited with different—both textual and graphical—views. Through this a clean separation of domain specific data structure and its visualization can be achieved. The transformation—called “reduction”—is implemented similar to an incremental compiler, where all changes are immediately transformed into a more concrete representation. This process works on multiple levels and DSLs, all stored within on data structure. The most concrete representation is the execution platform and language, which is also represented within the structure. Using this approach it is possible to use legacy code and—given appropriate transformation capabilities—these code fragments can also be transformed into more abstract DSLs. The Domain Workbench is not fully implemented and not available to the public, yet.

²¹Generic Modeling Environment

²²Graph Rewriting And Transformation

²³Constraint-Specification Aspect Weaver

²⁴The declarative Object Constraint Language [OMG06] has been designed for constraints and object queries on UML and MOF models. It is often extended with transformation capabilities.

Discussion

The presented tools offer different approaches to model-driven development with varying degree of flexibility, ease of use and productivity. In the following relevant advantages and drawbacks are discussed.

DSLs are valuable constructs for describing systems on higher levels of abstraction. A simple alternative to them are UML profiles, which provide basic support for the annotation of models using domain specific abstractions. Depending on the context, “heavyweight” extensions or the combination of existing metamodels are a good compromise between both concepts. This approach however can only be realized with a few tools. Fujaba for instance offers a plugin API, which allows the extension of existing and the definition of new metamodels. GME provides a powerful concept for the composition of metamodels into views.

Although most tools support model transformation, the implementations vary in several aspects. MagicDraw, RSA and DoME primarily work within one model, which is modified by the transformation (so-called “in-place” transformation, see Section 5.2). This prohibits incremental development, as more abstract representations are “destroyed” by the transformation. RSA and DoME remember applied transformations, enabling “undo and redo”. OptimalJ, ArcStyler and OlivaNova use a fixed set of models to gather information and to create code or other artifacts. In RSA, Together, Mia-Generation and the DSL Tools more flexibility concerning the use of different models is available, even though the DSL Tools do not support model-to-model transformation.

Transformation between models can be executed in different ways. In GME and the Domain Workbench transformations can be performed automatically, when the model is changed. Besides performance issues this approach is problematic when models are in inconsistent states. In most tools, the transformation is started manually and then builds the destination model from scratch. This approach however discards any manually applied changes to the destination model. OptimalJ and Mia-Studio offer a compromise by defining code areas, which are retained in subsequent transformations. RSA checks the destination model and warns before overwriting any manually changed data. Together, RSA, GME and the Domain Workbench allow incremental transformations, which update the destination model or perform merge operations between the previous and the newly created model.

The employed transformation languages are either used for simple in-place transformations (e.g. for the implementation of design patterns) or as monolithic transformation blocks, which transfer a complete model into one or more other representations. The actual code generation is usually deferred to the last transformation step. Integration of model transformation and code generation are typically accomplished by calling separate code templates from a model-to-model transformation. A full integration, which combines the two, is not available, yet. A minor exception to that is

the Domain Workbench, which integrates a full code representation into the model and thus uses one language for both types, although still a final model-to-code step exists, which produces the textual artifacts.

To sum up, the tools cover very different scenarios ranging from predefined model sets to fully customizable DSL solutions. Even in the latter category, the employed monolithic transformations form a rather rigid structure for application development.

2.4 Summary

This chapter discusses different methods to tame the increasing complexity of today's software systems. On one hand concern-based techniques help to regain both overview and insight into the different aspects of a system. On the other hand model-driven approaches streamline the process from model to executable code by utilizing model transformation.

Both ideas are combined in the following chapter. A novel concept is introduced, which applies stepwise refinement to models. Refinements introduce and implement concerns into a system, this introduction is supported by a model transformation language, which is described in detail in Chapter 5.

Chapter 3

Architecture Stratification

Chapter 2 gave an overview of current software development trends with an emphasis on model-driven approaches. This chapter introduces a methodology named “Architecture Stratification”, which employs concern-oriented modeling and stepwise refinement to achieve flexible model-driven development.

Section 3.1 gives an overview of the basic principles of architecture stratification. Then stepwise refinement is described in Section 3.2 and concern-oriented modeling in Section 3.3. The different mechanisms to describe concerns within models are discussed in Section 3.4. After that Section 3.5 deals with the realization and detection of concerns through automated refinement and abstraction. An important facet of flexible model-driven development is the ability to manually edit intermediate models. Possible solutions are analyzed in Section 3.6. Finally Section 3.7 relates the approach to the techniques presented in the previous chapter.

3.1 The Stratification Concept

In this section, the basic principles of stratification are outlined. According to the Merriam-Webster’s Online Dictionary, a stratum is: “one of a series of layers, levels, or gradations in an ordered system”¹. In “Dimensions of Component-based Development” [AKB99] Atkinson, Kühne and Bunse applied this concept to software architectures by describing a system using a “series of layers”.

They identified four fundamental hierarchies², which dominate the structure of component-oriented software systems. The first three hierarchies (containment, type

¹<http://m-w.com/dictionary/strata> (checked August 2009)

²In this context, a hierarchy is a “set of entities related by some transitive, partially ordered relationship”.

and meta hierarchy) apply to the components themselves. The fourth architecture hierarchy applies to the complete system.

This hierarchy becomes visible, when a software system is viewed on different levels of abstraction. Here, an anomaly is uncovered, which was called by the authors “Interface Vicissitude”: Depending on a chosen abstraction level, the interface of a component changes and often reveals more (technical) detail. This gradual refinement of an architecture can be seen as several abstraction layers with each layer representing the system on a certain abstraction level. Figure 3.1 outlines a stratified architecture. The following paragraph sets the used terms into context:

A **stratified architecture** describes the complete architecture of a software systems on multiple layers of abstraction. The layers are called **strata** and each stratum describes the complete system on a certain abstraction level. The topmost stratum contains the most abstract description of the system, the lowest the most concrete description. The process of switching to lower strata is called **refinement**, when switching to higher strata it is called **abstraction**.

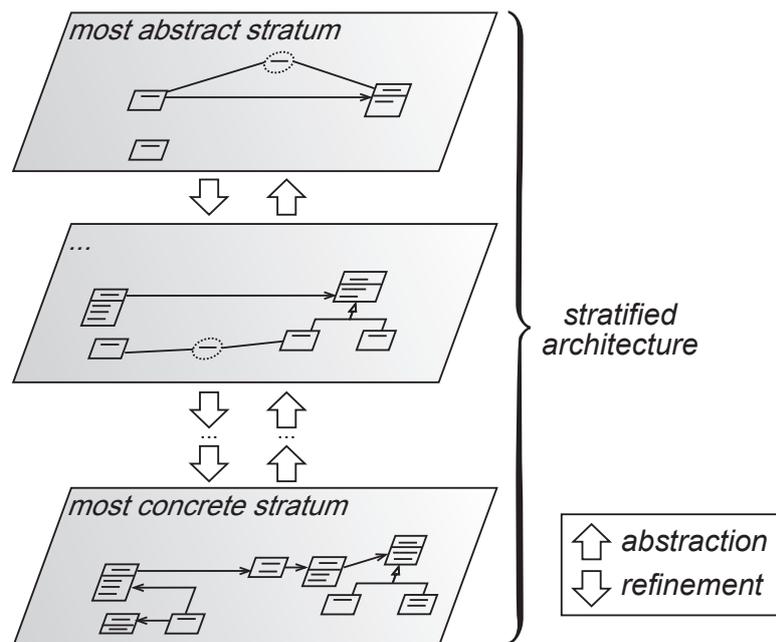


Figure 3.1: A stratified architecture

As argued by Atkinson et al. [AKB99], “strata are not layers in the normal sense”. In a conventional layered architecture, objects only appear on one layer, whereas in stratification, they may appear on several layers, albeit exposing different interfaces. The refinement process is also not to be confused with the stages of a software development process. Each layer within a stratified architecture describes the *complete* system on a certain abstraction level. When information is added to a

stratum, other strata are updated accordingly, keeping all strata synchronized with each other.

3.2 Stepwise Refinement

The process of successive refinement was first described in 1971 by Niklaus Wirth [Wir71]:

The creative activity of programming - to be distinguished from coding - is usually taught by examples serving to exhibit certain techniques. It is here considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures. The process of successive refinement of specifications is illustrated by a short but nontrivial example, from which a number of conclusions are drawn regarding the art and the instruction of programming.

By refining one “specification” at a time, stratification effectively applies the principle of stepwise refinement. His conclusions, although gained from other programming languages, still have relevance today:

1. *Program construction consists of a sequence of refinement steps. [...] Refinement of the description of program and data structures should proceed in parallel.*
2. *The degree of modularity obtained in this way will determine the ease or difficulty with which a program can be adapted to changes or extensions of the purpose or changes in the environment (language, computer) in which it is executed.*
3. *During the process of stepwise refinement, a notation, which is natural to the problem at hand, should be used as long as possible. [...] Each refinement implies a number of design decisions based upon a set of design criteria. [...]*
4. *The detailed elaborations on the development of even a short program form a long story, indicating that careful programming is not a trivial subject. [...]*

The refinement calculus, introduced 1978 in the Ph.D. thesis of Ralph-Johan Back [Bac78] formalized stepwise refinement by specifying behavior using an abstract program and refining it by a series of correctness-preserving transformations. This approach was first applied to algorithms and small programs, later work by the

author also showed its viability to refine object oriented systems. Section 3.7 compares this and other techniques with stratification. A similar approach, the “Harvard Program Development System”, also uses “notational extensions” and refinement transformations for program development (cf. Cheatham et al., [CHT81]).

The Munich project CIP³ [BMPP89], initiated in 1975 by Bauer and Samelson, developed the idea of transformational programming. Here, a formal problem specification is transformed by manageable, controlled, semantics-preserving steps into an executable program.

3.3 Concern-Oriented Modeling

Architecture Stratification makes the intermediate refinement steps visible by presenting them on strata within a stratified architecture. Thus, stratification not only helps to understand a system but also guides the development process itself. This process is not only influenced by the architecture itself but also by the requirements of the developed product. As stratification is concerned with the architecture of the system, requirements cannot directly be used to drive the development process. Thus in Section 2.1, the concept of the “concern” was introduced.

By using concerns as the main organization principle for strata, separation of concerns is implemented on the architectural level. This was suggested by Atkinson and Kühne in “Separation of Concerns through Stratified Architectures” [AK00].

They argue, that current AOP⁴ implementations allow only for code level abstractions. Other means are necessary for architectural separation of concerns. Stratification can address this need by providing a weaving mechanism for the introduction of aspects into systems. The paper shows similarities between the refinement in stratification and the discovery of cross-cutting concerns. Further it is argued that the discovery of such cross-cutting concerns often lead to reconsideration of the high-level architecture. In a stratified architecture, the affected (higher) strata reflect this change, keeping all strata synchronized with each other.

According to [AK00], the disadvantage of code-level weaving as undertaken by most AOP implementations is the interdependence of the aspects themselves. When aspects affect each other it is sometimes unclear, which aspects have to be woven first⁵. This problematic is also discussed by Bodden et al. [BFS06]... The strict hierarchy of a stratified architecture resolves this issue by “weaving” one aspect per stratum.

³Computer-aided, Intuition-guided Programming

⁴Aspect-oriented programming (cf. Section 2.2.1)

⁵More recent works on AOP deal with this issues by defining explicit dependencies between aspects.

The introduction of an aspect into a system leads to a change in behavior of affected components, and—considering the added advices—to a changed structure. This is illustrated in Figure 3.2, where the communication from X and Y is refined by adding two intermediate objects. This structural change shows similarities to the aforementioned “Interface Vicissitude”, because structural changes often also necessitate changed interfaces.

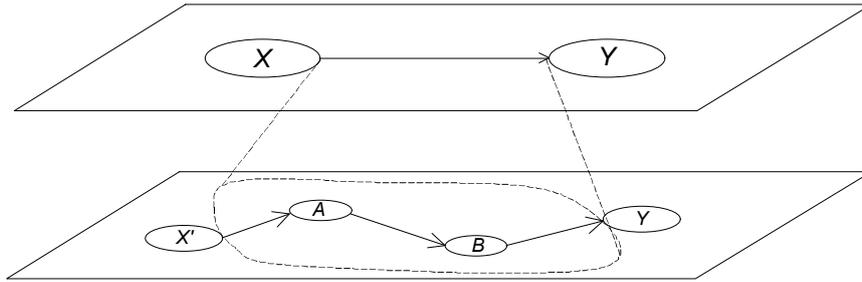


Figure 3.2: Interaction refinement [AK00]

In the context of stratification, this change can be considered an “interaction refinement”, as it primarily affects the communication between units of a system. The concept of interaction refinement is first formally described by Broy [Bro97]. He distinguishes between property refinement (“replace a behavior by one with additional properties”) and interaction refinement (“may change the names and numbers of input and output channels of a system but still relate the behaviors in⁶ a formal way”). According to [AK00], stratification defines the concept of interaction refinement as a refinement pattern, which introduces an aspect into a system. As this refinement takes place on all abstraction levels, it becomes clear that both high and low level refinements—and therefore high and low level aspects—exist.

AOP languages like AspectJ introduce aspects to address cross-cutting concerns. The stratification approach can also be applied to non-crosscutting concerns within a software system. Thus, the more general term “concern realizations” was introduced in Section 2.1. It not only includes domain aspects as mentioned in [KLM⁺97] (e.g. synchronization, failure handling) but also the user-interface of the system, data structures, algorithms etc. It also covers different technical realization techniques like components, design patterns, framework integrations or aspectual language extensions, which are used to address both cross-cutting and non cross-cutting concerns. In comparison to AOP, stratification can thus be seen as a more general concern realization technique. Combinations of aspect languages with stratification are also possible and would add dynamic aspect deployment (e.g. at runtime) to compile time weaving mechanisms, which can already be realized by stratification.

The integration of off-the-shelf frameworks becomes increasingly important in mod-

⁶He probably meant “in”.

ern software projects. Instead of building certain functionalities from scratch, pre-existing software components are integrated, saving effort and therefore time and money. Still, a good deal of time is spent with the integration and parameterization of such frameworks.

In response to this tedious task of framework parameterization Atkinson and Kühne describe the idea of “Aspect-Oriented Development with Stratified Frameworks” [AK03]. By organizing the variation points of a framework according to their concerns and assigning them to separate strata, instantiation of the framework can be simplified and—in part—be automated.

3.4 Concern Descriptions

A suitable mechanism for the parameterization of the framework variation points is needed. As models are the main communication mechanism in stratification, using model annotations is a reasonable choice not only for the description of framework instantiations but also for other concerns of a system. This section describes different variants for model annotations.

In system development, the term “model” often incorrectly implies the use of graphical modeling languages. Although models have a predominant representation system, they can be visualized both graphically or textually. Depending on the meta-model structure, mixed representation are possible as well. Stratification can also be applied to primarily textual models, but the following concentrates on the use of more common graphical modeling languages such as UML.

Conceptually, a stratified architecture is neither limited to a single diagram type nor one diagram per stratum. As with UML, several views of the same architecture can be used on one abstraction level. These views can either use the same or different diagram types.

The following examples and the reference implementation described in Chapter 4 employ a combination of UML class diagrams for the structural description and so-called story diagrams (cf. Section 5.5) to describe behavior. Class diagrams can be used on different abstraction levels, which can also be seen by the fact, that they are already used throughout standard development processes even on early design stages. In addition, more “concrete” class diagrams can easily be mapped to executable code. The missing behavior of created methods is filled by the above mentioned story diagrams.

Bunse and Atkinson discuss the use of UML for different abstraction levels and propose a restricted extension to UML (called “Normal Object Form”) [BA99], which is “semantically close” to object-oriented programming languages. The CASE tool Fujaba, used for the reference implementation of stratification, follows a similar

approach.

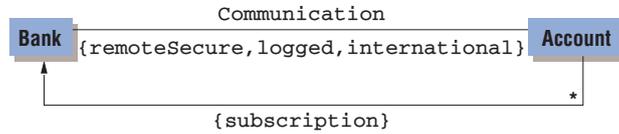


Figure 3.3: Annotated links [AK03]

In the aforementioned paper by Atkinson and Kühne [AK03] a notation similar to UML stereotypes was used to annotate links between objects. Figure 3.3 shows an example from this article. The association between “Bank” and “Account” is annotated with “Communication”, which relates to the interaction between these components. It is further parameterized with three tagged values, which refer to sub-concerns, which appear on deeper strata. The three subsequent strata in Figure 3.4 reveal these sub-concerns through refined interactions. Thus, the model is detailed step-by-step and each step focusses on one specific concern of the system.

The annotations in this example are used to describe framework instantiations. Here, a financial transaction framework is envisioned, which supports—among other features—remote communication and different currency types. Using the annotations, specific features from the framework are selected and parameterized, thus satisfying the concerns and sub-concerns of the stratified architecture.

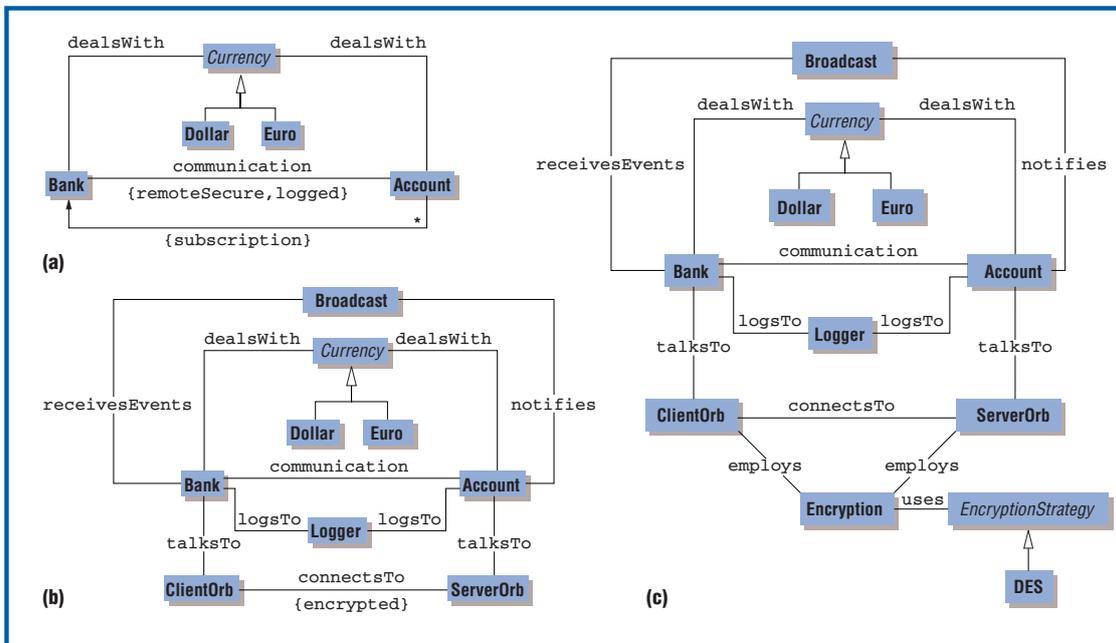


Figure 3.4: Annotated links on subsequent strata [AK03]

Frameworks help to “abstract away” technical detail. Instead of building a full implementation from scratch, they are instantiated and integrated through parameterization. Annotated models contain this information *within* the model compared to separate configuration files, often used in “classical” development with frameworks. This offers several advantages, e.g. guidance for the instantiation process, improved documentation and separation of concerns.

Albeit the notation from Figure 3.3 conforms to UML, it has a few disadvantages. The idea of tagging links within the diagram stems from the notion of interaction refinement, discussed in Section 3.3. This approach, however, is not sufficient for interactions with more than two participants. In addition, stepwise refinement not only involves interaction between components but also the evolution of the components themselves. In comparison to “interaction refinement”, Broy [Bro97] described this effect as “property refinement”. Tagged links are also not capable of assigning named roles to participating components.

An analysis of more complex framework integrations and concerns in general reveals the need for more a sophisticated annotation method. In response Klar et al. [KKG05] presented a notation based on UML collaborations [OMG05b, Section 5.66]. UML collaborations are used to instantiate patterns or templates, which is quite similar to concerns within strata. Figure 3.5 shows a UML collaboration for the design pattern “Observer”⁷. The collaboration name (“Observer”) is shown within a dashed ellipse, the edges are labeled with role names and are connected to participants within the pattern (“Subject” and “Observer”). Additional instantiation parameters are given in an informal note element.

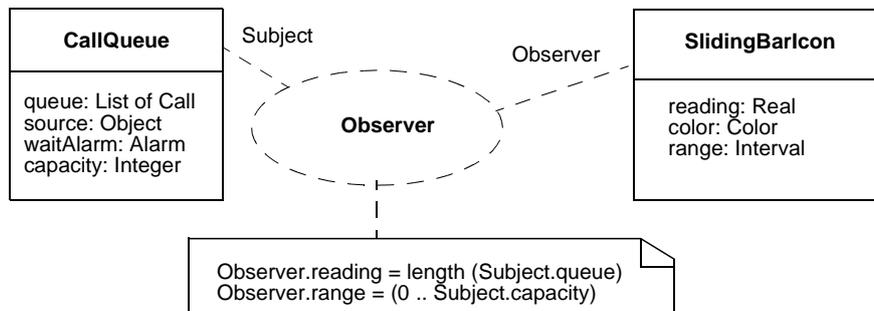


Figure 3.5: A collaboration within a class diagram [OMG05b]

According to the specification, UML collaborations can only be used within class diagrams and only classifiers can be participating elements. Compared to that, the notation presented by Klar et al. [KKG05] is not limited to class diagrams and allows links to arbitrary model elements. The ellipse is called “annotation” and shows the name of the described concern. The named links connect the annotation with the

⁷As described in “Design Patterns: Elements of Reusable Object-Oriented Software” [GHJV95].

model elements, which are affected by the concern. Additional formal parameters, both for the annotation and the links, are specified in a separate editor dialog and are denoted by a boxed “P”-icon.

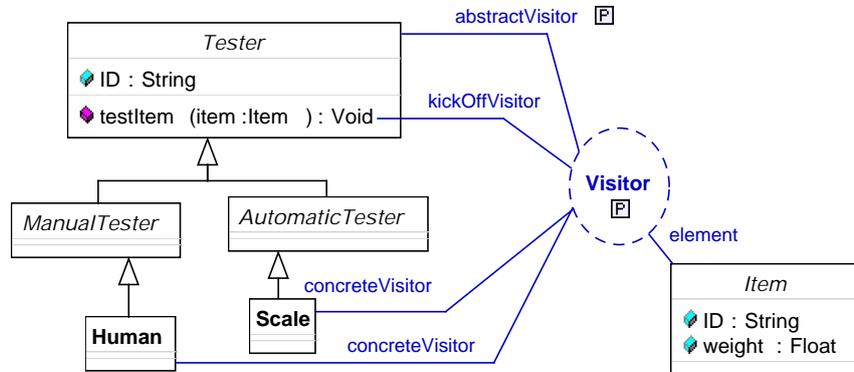


Figure 3.6: Collaboration notation for visitor design pattern

Figure 3.6 shows the collaboration within the design pattern “Visitor” [GHJV95]. The annotation “Visitor” defines—among others—the role of the element to be visited and the concrete visitor classes. The use of this collaboration-like notation avoids the disadvantages of the stereotype notation and offers more flexibility by supporting links to arbitrary model elements.

3.5 Automated Refinement and Abstraction

Having introduced a suitable technique to *describe* concerns, the next logical step is their *realization*. Stratified frameworks, as described in [AK03], already support concern realization by creating necessary binding and parameterization files for framework instantiations. Although sufficient for trivial cases, it is often necessary to create glue code⁸ as well. It ties the application to the framework by adapting data structures and interfaces. Furthermore, frameworks often employ the “template” design pattern [GHJV95], which offers defined parameterization points (so-called “hot spots”). The necessary classes and inheritance relationships are ideal candidates for additional automation support.

An example can be seen in Figure 3.7. Here the design pattern from the previous section has been implemented. As can be seen, an interface with the generalization to the class `Tester` and the methods `visit` and `accept` have been added.

The major difference to the aforementioned “development with stratified frameworks” is the shift from separate artifact generation (e.g. for the instantiation of frameworks) to model-to-model transformation *within* a stratified architecture. This

⁸Glue code adapts application specific data and interfaces to the framework.

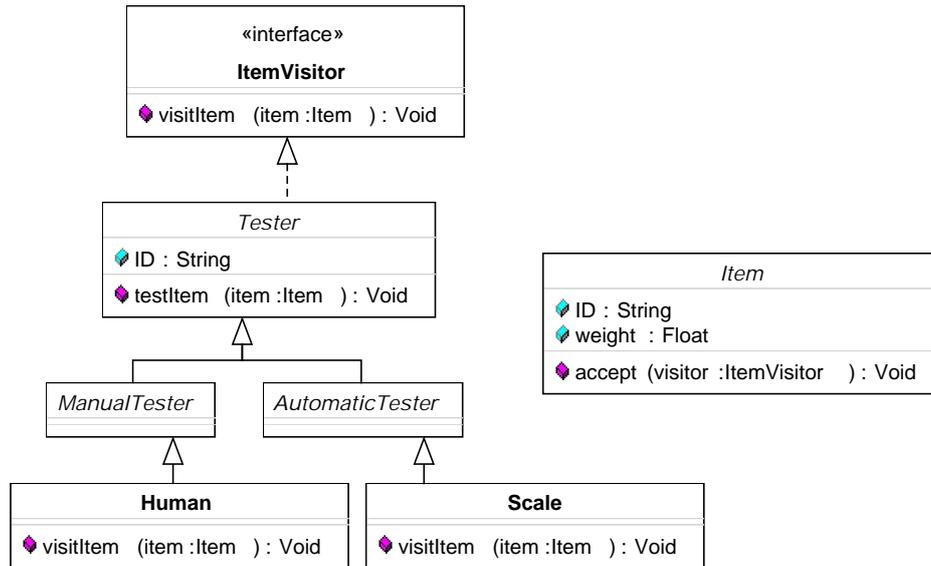


Figure 3.7: Implemented visitor design pattern

enables automated refinement, which transforms the abstract description of a concern on one stratum into its implementation on the stratum below. Annotations serve as input to the model transformation and, therefore, guide the transformation process. Instead of transforming the complete model, annotations thus enable selective transformation by means of concern based transformation rules. As these rules implement concerns, they are also called “refinement rules”.

Annotations and corresponding refinement rules are organized by concern in an extensible library. They implement concerns into software systems and thus enable model-driven, stepwise software development. Frameworks can extend the library by providing annotations, which help to integrate the framework into an application. Similar to design pattern catalogs, the library provides detailed information on annotations. This includes intent, applicability, participants and relations to other annotations.

This model-driven approach also supports iterative and incremental development. Annotations and associated rules “document” the necessary transformations, which allows subsequent iterations to re-execute them automatically, transferring any changes applied to the model to the following strata.

This is one of the key aspects of stratification: The ability to freely navigate between strata. As all strata show the same system—albeit on different abstraction levels—changes on one stratum are propagated automatically to other strata. Keeping the whole stack of strata synchronized is accomplished through model transformation.

In addition to the described propagation towards lower strata, this may also include transformations to higher—less concrete—strata. Two different scenarios have to

be considered here. In the first scenario a concern was implemented manually by the developer and has to be transformed into annotations on a higher strata. Here, “abstraction rules” are used for the transformation process. These transformations recognize concern realizations and transform them to their abstract counterparts.

Defining these transformations is rather difficult, as concerns are not always implemented in the same way—for instance names vary and objects interact differently. Abstraction rules have to accommodate for all these situations. Transformation based pattern recovery is restricted to static analyses, which yields unsatisfactory results (cf. [NSW⁺02]). Current research in the area of reengineering focusses on behavioral detection of patterns using dynamic analyses (e.g. instrumenting application and monitoring at runtime, cf. [HHHL03] and [WO06]), which are too complex in the model transformation context.

In the second scenario, changes to a concern, which was implemented by a refinement, have to be propagated to a higher stratum. Here, “reverse refinement rules”⁹ are applicable. They use traceability information, stored during the refinement transformation, to propagate changes back to higher strata.

In this context, each refinement rule is related to a reverse refinement rule. Both can be combined to form a bidirectional transformation, which then specifies both directions in *one* rule. For instance, Triple Graph Grammars [Sch95] support this approach and use—in addition to the two models between the transformation is applied—a third model, which holds the traceability data by connecting both models.

A third scenario is not strictly related to transformations towards higher strata but nonetheless shall be noted here. If a concern needs to be addressed in a stratified architecture, and no refinement rule exists, two strategies are possible. Either the developer implements the concern manually and defines an abstraction rule, which produces the concern description on a higher stratum, or the developer defines a new refinement rule and adds the concern description to a higher stratum. As the definition of abstraction rules is—as stated above—rather difficult and reusability of refinement rules is usually more important, the latter strategy is preferred.

3.6 Support for Editing Intermediate Strata

Without transformation capabilities, stratification is primarily useful for documentation purposes. With the addition of automated refinement, it now actively supports the development process. Its most simple form can be described as “deterministic stratification”. Here, the most abstract stratum already contains all necessary information to generate the final system and intermediate strata serve as detailed

⁹The term was taken from the book “Software Factories” [GS04, Chapter 15].

visualizations of concerns and sub-concerns. Editing these strata is not possible, as it would contradict the idea of determinism.

The ability to inspect intermediate strata has two main purposes: During development, analyses may reveal errors in the transformation process, which can easily be pinpointed to specific transformations or erratic parameterization on the topmost stratum. After development, they serve as detailed documentation of the implemented system.

In order to support deterministic stratification, highly parameterizable annotations are needed. However, the presented notation with links and parameters is not sufficient. For instance framework integrations are not only based on parameterization but also employ techniques like the template design pattern, where the framework defines the overall control flow and specific behavior is added by means of subclassing framework classes and implementing abstractly defined methods.

This pattern inspired an additional parameterization mechanism for annotations, called “hook spots”. The name is derived from framework “hot spots” (cf. [Pre94]) and the “hook methods” used for example in the template design pattern. The concept, initially proposed in [GKK06], enables transformation rules to mark created elements as hook spots. After the transformation is complete, the developer can modify these newly created elements. This “constrained editing” does not compromise the stratification principle as the modification can be seen as “late parameterizations” of annotations. Compared to parameterization *before* the transformation this offers the advantage of editing within the implemented concern.

When applied to single method bodies, hook spots follow the template pattern. However, they can also be used by code fragments within a method, thus providing clean separation of generated and hand-written code even on a very fine grained level. Both variants can also be employed to generate glue code for frameworks. Applied to general model elements, other usages can be envisioned as well.

With this addition intermediate strata not only serve as documentation but can also actively be edited. Still, the architecture of the complete system is determined by the first stratum and the annotations residing on it. As argued in Section 3.4, the concerns described by these annotations give rise to related sub-concerns on lower levels. This dependency manifests itself in refinement transformations, which create new annotations on lower levels. For instance, a high-level annotation for persistency may add low-level annotations for database connectivity.

Thus, concerns are the main organization scheme in stratification. They are described by annotations and implemented through transformations, which do not only modify the main model, but also add annotations for lower level concerns.

This dependency, where annotations on higher strata create other annotations on lower strata, can also be called a “chain of concerns”. The chain provides a guid-

ance for developing a software system. By offering transformations for different application scenarios, which create annotations for lower level (sub-) concerns, a modular system for application development is created. This guided process shows similarities to architectural patterns and software product lines.

According to Frank Buschmann et al. “Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them.” [BMR⁺96]. By enforcing these “organization schemas” through annotations and providing guidelines by successively creating annotations for sub-concerns, stratification can be used to deploy architectural patterns.

As defined in Section 2.2.6, software product lines share a common, managed set of features. This selection and assembly of features can be guided by annotations and their parameterizations. Dependencies between features manifest themselves in the creation of annotations on lower strata through the above mentioned “chain of concerns”.

The deterministic approach prohibits—with the exception of hook spots—editing intermediate strata. In consequence, all annotations on these strata have to be fully parameterized. Either these parameters already exist on the first stratum or they are determined by the transformation, which creates the annotation. The first variant results in superfluous data, because these parameters belong to a concern, which is described in detail on a lower stratum. The second variant removes flexibility by “choosing” parameters according to predefined rules.

This flexibility can be regained by further enabling controlled editing on intermediate strata. One possible concept is the use of “abstract annotations”. They are created during transformation and serve as placeholders for concerns. When the concern is specified in detail on a lower stratum, the developer chooses a concrete annotation, which “implements” the abstract annotation. The transformation library may contain several implementing annotations for different purposes or platforms. In addition to the parameters defined by the abstract annotation they may require additional input from the developer.

For instance an abstract database connectivity annotation on a higher level is concretized by a vendor specific annotation on a lower stratum. As a consequence, the developer may add parameters and annotation links on intermediate strata. It becomes clear, that often new model elements are required as well. These elements extend the stratum in order to fully describe and implement its concern.

Although the described extensions relax the restrictions of deterministic stratification, they do not violate the stratification approach itself. Added elements are always part of a concern realization and first appear on the stratum, where the concern is detailed. As each stratum describes the complete system, all concerns

are—implicitly or explicitly—present.

Annotations and related transformation rules are not limited to concern realizations. They can also help to solve detail problems, which appear during the specification of a concern. For instance, if a stratum is primarily concerned with the description of a database connection, the used annotation may require a local registry class. The developer adds the class to the model and decides to make it a singleton (cf. [GHJV95]) by adding a “Singleton” annotation to it. Here the annotation does not present a concern itself but merely simplifies development. These additions are always related to the concern described on the current stratum. Thus, each model element can be traced back to its initial creation and the related concern.

Both the hook spot concept and the manual additions to intermediate strata make the stratification approach more flexible. On closer examination, an interesting difference exists between those two. The parameterization of annotation takes place, *before* the concern is implemented. In comparison, the hook spots are filled *after* the implementing transformation has finished. If the transformation defines the transition to the next stratum, the question arises, which of the both strata “describes” the concern? The abstract stratum delivers an abstract definition of it, the more concrete stratum shows the implementation.

As a consequence, each stratum plays two roles, showing the implementation of one concern and describing another. This duality can be eliminated by introducing two-sided strata. The upper side contains the abstract description of a concern, the lower side reveals its implementation. Obviously, the model transformation is executed when switching from the upper to the lower side. When the developer completed all hook spots, he switches to the upper side of the next stratum.

This situation is illustrated in Figure 3.8. The topmost frame shows the implementation side of a stratum (A_i). In the next frame—the description side of the next stratum (B_d)—the developer chooses the annotation to the right for further refinement (depicted by the arrow pointing to the annotation). He also adds an additional class and a link pointing to it. Then the transformation is executed and the concern implemented into the system. This is shown in the following frame, which represents the implementation side of the stratum (B_i). Now the developer is able to complete any hook spots created during the transformation (e.g. method fragments). After he has finished, he switches to the next stratum (C_d) and again chooses the next annotation for further refinement.

3.7 Related Work

After the concepts of architecture stratification have been introduced, this section relates them to the techniques described in Chapter 2. First, the methodologies and

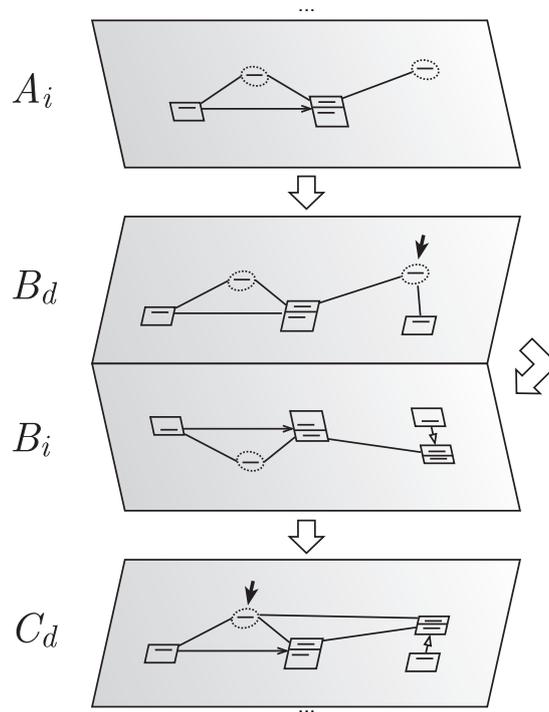


Figure 3.8: Two-sided strata

research subjects are discussed, then model-driven development tools. After that additional research work is presented.

Architecture stratification achieves separation of concerns by distributing them to different abstraction levels and implementing them by means of model transformation. Like SOP¹⁰ and AOP, stratification uses separate entities to describe cross-cutting concerns. Instead of the declarative form of aspects, concern descriptions are contained in model transformation rules. They offer—similar to CaesarJ and AHEAD—a higher level of reuse by applying a parameterizable weaving process. A similar technique to weave aspects on model level and adapt them appropriately is described by Reddy et al. [RGF⁺06].

The weaving of concerns is accomplished at modeling time, allowing the developer to see the effect of the added concern. Other approaches weave concerns at compile time and also offer dynamic deployment of concerns even at runtime. This is a very powerful approach, but the resulting applications are more complex and thus harder to understand. Still, both approaches do not contradict each other and can be combined by adding modeling capabilities for aspects (or other SoC mechanisms) to stratification. This idea is described further in Section 7.2.

¹⁰Subject-Oriented Programming, see Section 2.2.1.

Stratification not only deals with cross-cutting concerns but is also applied to non cross-cutting concerns, which follow the dominant decomposition of the underlying platform (the actual programming language or framework abstractions). Compared to the approach described by Büchner and Matthes [BM06], it does not require the frameworks themselves to be annotated. Instead separate transformation rules are provided, which help the developer to automatically instantiate the framework and integrate it into the application.

Some of the tools, discussed in Section 2.3, support the implementation of design patterns. It is often accomplished using simple annotation mechanisms or a wizard-based process, which adds the necessary elements to an existing model. Stratification in contrast offers an extended annotation mechanism and a more powerful transformation language, capable of adapting existing objects. The separation of the pattern description on one strata and its implementation on the next further improves the use of design patterns.

As argued in Section 3.3, the stratification concept is not limited to UML class diagrams. If a concern can be described more easily with a domain specific language, it can be used in addition to the existing class diagram on the relevant stratum. The implementation of the concern is accomplished using model transformation. A detailed description of this approach is out of scope for this dissertation and subject for future research.

Aspect-oriented modeling approaches primarily address the tracing of requirements to design artifact. The work by Baniassad and Clarke [BC04] is based on the idea of subject-oriented programming [HO93]. It generalizes the concept of an aspect to a “Theme”, which addresses a single feature. The composition of features shares similarities with the implementations of concerns in architecture stratification.

Model-driven development with aspects is often related to model-based weaving. One example is the framework outlined by Simmonds et al. [SRF⁺05], for which no implementation exists, yet. Another example is XWeave, presented by Groher and Völter [GV07]. It uses named based merging for aspects and offers a pointcut language based on openArchitecture [HVEK07].

Stratification also shares many similarities with the MDA idea. Both use several levels of abstraction to model a software system, although most MDA implementations work with predefined transformations between a fixed set of abstraction levels. These “monolithic” transformations are designed to transform a complete model into a less abstract form. Compared to that, stratification is more flexible. New strata can be introduced as needed and transformations implement single concerns by modifying only the relevant parts of the model.

“Classical” (non model-driven) software development does not directly support incremental and iterative development. In subsequent iterations any existing models have to be updated manually and concerns have to be “re-detected” in order to

apply changes. In stratification, the different concerns are stored within models in a persistent manner and can be modified and extended more easily. For instance, the developer may insert additional strata to add new features or edit an existing stratum in order to fix existing bugs. The fine granularity of the transformation rules also enables the developer to apply fixes to the transformation rules themselves. All of the effects are especially important in agile approaches, with their “release early, release often” strategy.

The implementation of the stratification approach (cf. Chapter 4) delivers the necessary support for automated model transformation. The annotation model provides a clean separation of application specific artifacts and the parameterization of elements contributed by the transformation. Combined with the guidance offered by the “chain of concerns”—described in Section 3.6—it forms the basis for software product lines.

Most of the MDSO tools, introduced in Section 2.3, are in principle capable of supporting the stratification approach. However, automation support has to be added and the employed model transformation language may not fit the needs for concern implementation. A detailed evaluation of transformation languages and the requirements for stratification can be found in Chapter 5. Still, most of the tools follow the MDA pattern with a rigid structure of abstraction levels, which may not be adaptable for stratification. In the following characteristic features of the tools, presented in Section 2.3, are discussed¹¹.

OlivaNova offers complete code generation with no support for hand written code or roundtrip engineering. However, the employed modeling techniques only support a limited set of application scenarios. OptimalJ and Mia-Studio use defined code areas for manually written code and prevent the editing of generated code. Other tools (e.g. RSA and Together) provide round trip capabilities, which try to reconstruct the model after the code was changed, keeping model and code synchronized. This often conflicts with the reapplication of transformations.

The modeling language employed by the reference implementation (cf. Chapter 4) uses class diagrams to describe the structure while method behavior is represented by code fragments. Both are strictly separated with no duplication of information, thus no synchronization is required. As code is only stored as part of the behavior model, no round trip capabilities between the structural model and the code are required. The “hook spot” concept, introduced in Section 3.6, shows similarities to OptimalJ by providing regions for manually written code and separating them from generated parts. However, hook spots are more tightly integrated with the transformation rules, which makes them more robust to changes. For instance, when in OptimalJ a class is renamed or some parameters are changed (resulting in different classes to be created), the hand written code might be lost. The transformation rules

¹¹References and web links for the tools can be found in Figure 2.1.

use unique identifiers to handle hook spots, which enable precise recreation during transformation. This concept is not only applied to code fragments but also to model elements such as methods, attributes and classes, which can also be added to models and are retained on re-generation.

Microsoft’s implementation of Software Factories [GS04] does not support hand written code directly. Instead well-known techniques from framework integration such as the template design pattern [GHJV95] or subclassing can be used. Furthermore, C# supports partial classes. These techniques work only on the method level, hook spots can also be used within methods. The Software Factories idea envisions a “grid” of models, where any dependencies between the models are described by model transformations. If the grid is not carefully designed, this results in many conflicts, when different transformations affect the same model. In addition, the developer needs to deal with a multitude of models and at the same time be aware of the interdependencies between them. These problems are circumvented by the strict hierarchy of strata within the architecture stratification approach.

The Domain Workbench from Intentional Software unifies code and model into one data structure. Similar to stratification it represents a software system simultaneously on several abstraction levels. The representation of the data structure is primarily textual. The reduction process used for the transformation (cf. [SCC06]) seems to be rather complicated, albeit powerful.

The following paragraphs discuss current research work, which shows parallels to architecture stratification.

Similar to annotations and framework specific transformation rules, the model transformation framework “Mercator” [WJ04] uses stereotypes to control platform specific parameterization of frameworks. It employs a rather simple expansion process for the stereotypes, which cannot be controlled further.

The work by Almeida et al. [ADP⁺05] specializes in the description of refinements for “interaction between application parts”. They discuss several kinds of interaction refinement on varying abstraction levels. Their analysis reveals different types of operations, needed to describe the refinement. This categorization may help to structure transformations rules and thus—as noted by the authors—contributes to the stratification concept. Currently they apply these operations manually and on an architectural level only.

The refinement calculus, mentioned in Section 3.2, was also applied to stepwise feature introduction within UML class diagrams [Bac02]. The focus is on preserving behavior when introducing new features into a software system. The feature introduction is not automated, instead the refinement calculus uses “abstraction functions” to verify the implementation against a specification.

Czarnecki et al. propose the concept of “staged configuration” for feature model-

ing [CHE04]. This multi-layered modeling approach exhibits some similarities to stratification. The annotations within a stratum can be compared to the features, which are selected in staged configurations. While annotations allow more flexibility, staged configurations are easier to create and apply, because the features are limited to a defined set and less complex than refinement transformations.

Chapter 4

Implementing Stratification

Although architecture stratification can be used with any modeling tool and even by drawing on plain paper, its main advantages of strata synchronization and automated implementation by means of model transformation can only be utilized with special tool support.

An implementation of the stratification concept, based on an open-source CASE tool, was started in 2004 and since then constantly improved. This chapter describes the implementation in detail, leaving out the model transformation language, which follows in Chapter 5.

In Section 4.1, possible choices for a modeling tool as a basis for the stratification approach are discussed. The selected CASE tool—Fujaba—is described in Sections 4.2 and 4.3. It is followed, in Section 4.4, by the integration of stratification. First, the modeling language needs to be extended in order to support annotations (Section 4.5), then the handling of transformation rules and strata is discussed (Section 4.6, 4.7 and 4.8). The separate plugin used for the final code generation step, is shortly described in Section 4.9. Finally, in Section 4.10, an overview of the user-interface is given.

4.1 Platform Selection

As much of the required modeling functionality is already available in existing tools, designing and implementing a modeling tool from scratch was not a viable starting point. Instead a list of requirements was compiled and after careful examination an existing tool was selected. The following characteristics were considered:

Modeling language support Support for commonly used UML diagram types and the ability to add new ones (e.g. for domain specific languages) is mandatory. In order to support the notation described in Section 3.4 not only new diagram types but also the extension of existing types is required. This includes the metamodel data and a customizable visual representation.

Application Programming Interface To manipulate the model, access via a programming interface is necessary. An interface working on the metamodel-level is preferred, as it handles different modeling languages equally. To implement the whole stratification process, it is also necessary to control user interface aspects of the tool (e.g. loading and saving files, showing dialogs etc.).

Model transformation In addition to model manipulation through the API, tools may also offer a built-in model transformation language.

Code generation In order to create executable applications, some kind of code generation facility has to exist, which is capable of creating Java code for the supported diagram types.

Software license Several commercial vendors offer free licenses for educational institutions. These licenses are often time-limited or only work on campus networks, which restricts further development by students or other affiliated research institutions. This limitation does not exist with open-source platforms. In addition, development is simplified by the availability of source-code. Thus open-source tools are preferred.

The selection was based on tools available in 2004. Details can be found in the diploma thesis by Felix Klar [Kla05]. Since then, several new tools entered the market, a reevaluation however revealed no major differences. Under closer examination were the following tools:

Eclipse/EMF (open source)

The Eclipse Modeling Framework¹ provides an API to access models conforming to the Ecore metamodel. Although visual editors can be build easily using the Eclipse “Graphical Editing Framework (GEF)”, EMF itself does not supply any visual editors but only provides the basis for other tools.

¹<http://www.eclipse.org/modeling/emf/> (checked August 2009)

Omondo EclipseUML (commercial, free academic license)

EMF-based UML editor for Eclipse. Announced in 2004, the necessary “OpenAPI” was not released within time, making model access impossible. Moreover, the definition of custom graphical elements is not possible.

Borland Together (commercial²)

In 2004 Together was a stand alone application, featuring a plugin concept and an API for model access. The definition of custom graphical elements was not possible, no model transformation capabilities existed. The current release is based on the Eclipse platform and includes transformation capabilities, namely a QVT/operational³ implementation (cf. Section 5.4).

ArgoUML (open source)

Provides editors for seven main UML diagram types. Development of own plugins is supported, including model access and definition of new diagram types. No model transformation capabilities are available.

Fujaba (open source)

Only three UML diagram types are supported, new types can be added using the very flexible plugin concept. In comparison to ArgoUML, extending existing diagram types is easier to accomplish. Fujaba provides extensive graph transformation capabilities, which can be reused for model transformation.

Although Eclipse is widely used nowadays, the available graphical editors in 2004 did not provide the needed functionality. Fujaba offered the needed editor, a flexible plugin concept and the basic mechanisms for model transformation and was consequently selected.

4.2 Modeling with Fujaba

Development of the CASE tool Fujaba started 1997 at the University of Paderborn. The main focus was to create a modeling tool based on a precisely defined subset

²A price-reduced academic licence is available.

³QVT is a model transformation language standardized by OMG.

of UML, which enables the full generation of executable Java applications. Furthermore, extensive support for roundtrip-engineering is included. Both manifest themselves in the acronym Fujaba, which means “**F**rom **U**ML to **J**ava and **B**ack”.

With the release of version 4, Fujaba added plugin support, which provided the necessary mechanisms to add and extend diagram types and the Fujaba user interface. Support for other programming languages (e.g. C++) was added. The plugin CodeGen2, developed at the University of Kassel further improves code generation by utilizing a template language.

This section describes the current release of Fujaba 5. It forms the basis of the current stratification implementation SPin (for **S**tratification **P**lugin).

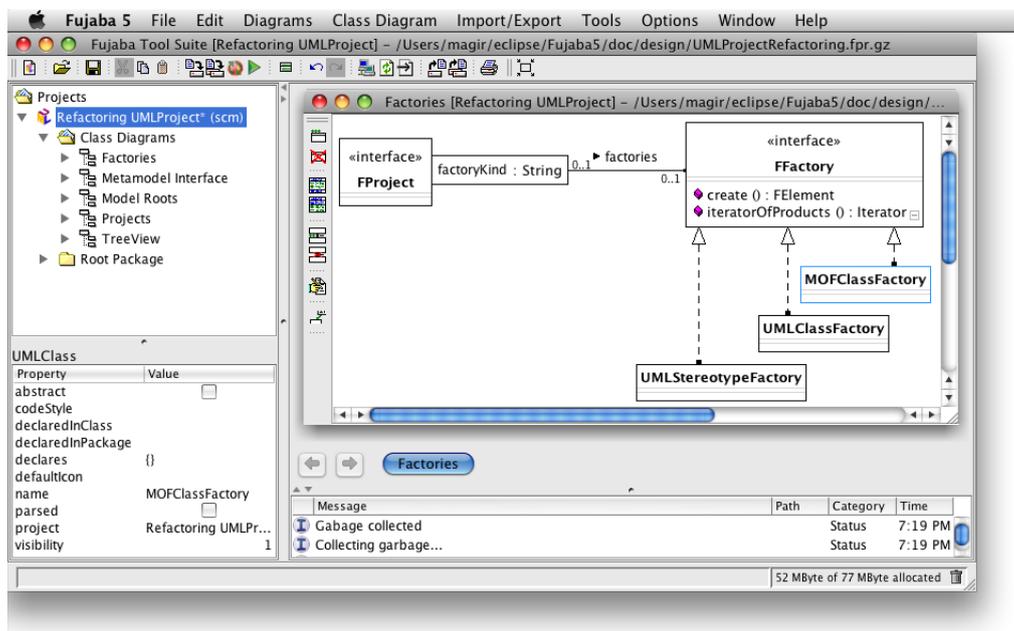


Figure 4.1: Screenshot of Fujaba 5.1

Fujaba requires at least Java SE 5.0, which is available for all major operating systems. Figure 4.1 shows a screenshot of the application. To the left, the list of open projects is shown. Below that, the property inspector for selected elements. The main area is reserved for diagrams. Fujaba supports only a few diagram types, but new types can be added by means of plugins. The lower right area is reserved for status messages.

The main purpose of Fujaba is to create executable Java applications from graphical models and to reverse engineer models from Java code. For the structural description UML class diagrams are available. They are tailored towards code generation for Java (e.g. support for predefined stereotypes for Java and automatic generation for association attributes and accessor methods).

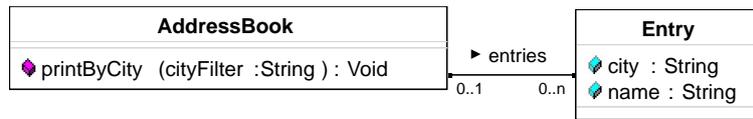


Figure 4.2: Example of Story Driven Modeling: class diagram

An example class diagram is shown in Figure 4.2. It contains a class `AddressBook` and a class `Entry` connected by an association named `entries`. The excerpt of the generated Java code presented in Listing 4.1 shows a part of the generated attributes and methods for the association.

```

1 public class AddressBook
2 {
3     public void printByCity (String cityFilter)
4     {
5         ...
6     }
7
8     /**
9     * <pre>
10    * AddressBook 0..1 entries 0..n Entry
11    * addressBook entry
12    * </pre>
13    */
14    private FHashSet entry;
15
16    public boolean addToEntry (Entry value)
17    {
18        ...
19    }
20
21    public boolean removeFromEntry (Entry value)
22    {
23        ...
24    }
25    ...
26 }
27
  
```

Listing 4.1: Excerpt of Java code for the class diagram in Figure 4.2

The behavior of class methods (e.g. `printByCity` in the example) is specified by means of Story diagrams [FNTZ00], which resemble a combination of UML activity diagrams and collaboration diagrams.

To quote from the UML specification: “An activity diagram is a special case of a state diagram in which all (or at least most) of the states are action or subactivity states and in which all (or at least most) of the transitions are triggered by completion of the actions or subactivities in the source states. The entire activity diagram is attached (through the model) to a classifier, such as a use case, or to a package, or to the implementation of an operation.”⁴ [OMG05b, Section 5.84].

⁴This definition is based on the UML 1.x, in 2.0 the semantics of activity diagrams follow

Fujaba follows this definition and uses Story diagrams to describe method bodies⁵. The UML specification lists only informal activity types, which cannot be used for code generation. Fujaba resolves this problem by providing more formal activity types. In addition to the well-known start and stop activities and decision nodes, Java source code fragments (in Fujaba called “statement activities”) are supported. However, most important are “Story patterns”, which can be considered full diagrams within the activity diagram. Because they describe collaborations between objects, they share similarities with UML collaboration diagrams [OMG05b, Section 5.65]. Their semantics are based on graph grammar theory and thus enable a formal and precise specification.

Similar to collaboration diagrams, story patterns contain object structures (e.g. instantiated classes). To navigate along associations, links between objects are used. The objects can be seen as attributed nodes and the links as edges within a graph. To find actual objects, search operations within the graph are used—a process, which is called “pattern matching”. In order to modify the graph—and thus the object structure—objects can be marked with “create” or “destroy”. Method behavior is specified by executing these operations along the control flow of the enclosing story diagram.

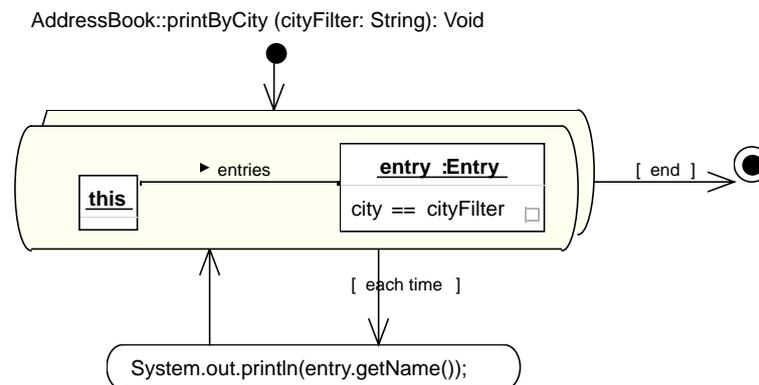


Figure 4.3: Example of Story Driven Modeling: story diagram of `printByCity`

The behavior of the method `printByCity` within the class `AddressBook` is specified by the story diagram shown in Figure 4.3. This diagram contains a start activity at the top and a stop activity to the right. The activity with the double border in the middle of the diagram is a special story pattern, which resembles a “for each” activity. In this case, it searches the `entries` collection⁶ for an entry, where the `city` attribute equals the `cityFilter` variable, which is a parameter of the method. For each found entry, the lower statement activity is executed, which prints the

petri-nets.

⁵called “operations” in the UML specification

⁶Fujaba automatically generates association objects and access methods for all associations.

name of the entry using standard Java code. After all instances have been checked, the stop activity is reached.

The control flow of the diagram also serves as starting point for the code generation. Listing 4.2 presents the code generated for the story diagram by Fujaba. As can be seen, the “for each” activity is mapped to a while loop, which iterates over the collection containing all entries. The statement activity is simply copied into the generated code.

```
1 public void printByCity (String cityFilter)
2 {
3     boolean fujaba__Success = false;
4     Iterator fujaba__IterThisToEntry = null;
5     Entry entry = null;
6
7     try
8     {
9         fujaba__Success = false;
10
11         // iterate to-many link
12         fujaba__Success = false;
13         fujaba__IterThisToEntry = this.iteratorOfEntry ();
14         while ( fujaba__IterThisToEntry.hasNext () )
15         {
16             try
17             {
18                 entry = (Entry) fujaba__IterThisToEntry.next ();
19                 JavaSDM.ensure ( entry != null );
20
21                 // attribute condition
22                 JavaSDM.ensure (JavaSDM.stringCompare (entry.getCity(), cityFilter) == 0);
23
24                 System.out.println(entry.getName());
25
26                 fujaba__Success = true;
27             }
28             catch ( JavaSDMException fujaba__InternalException )
29             {
30                 fujaba__Success = false;
31             }
32         }
33         JavaSDM.ensure (fujaba__Success);
34         fujaba__Success = true;
35     }
36     catch ( JavaSDMException fujaba__InternalException )
37     {
38         fujaba__Success = false;
39     }
40     return ;
41 }
```

Listing 4.2: Generated Java code for method `printByCity`

The syntax of story diagrams is considered to be a complete graph rewrite language. These languages describe subgraph replacements by means of transformation rules. More information on graph based transformation systems and graph grammars can be found in “Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations” [Roz97]. Graph grammars are a generalization

of Chomsky string grammars [Cho57]. Instead of string concatenation, graphs are “glued” together using graph morphisms.

Story diagrams adopt several features of PROGRES (PROgrammed Graph REwriting Systems) [SWZ99], developed at the RWTH Aachen ⁷. These capabilities for graph transformation are not only well-suited for behavior description but can also be employed for model transformation. This is described in detail in Section 5.5. One of the main differences between PROGRES and story diagrams is the removal of backtracking, which enabled rollback of partly executed transformations. The removal also restricted the control flow and enabled the translation to plain Java code [FNTZ00, Section 4]. Fujaba provides only a graphical syntax, whereas PROGRES also offers a textual representation of transformations.

A simplified overview of the employed metamodel is presented in Figure 4.4.

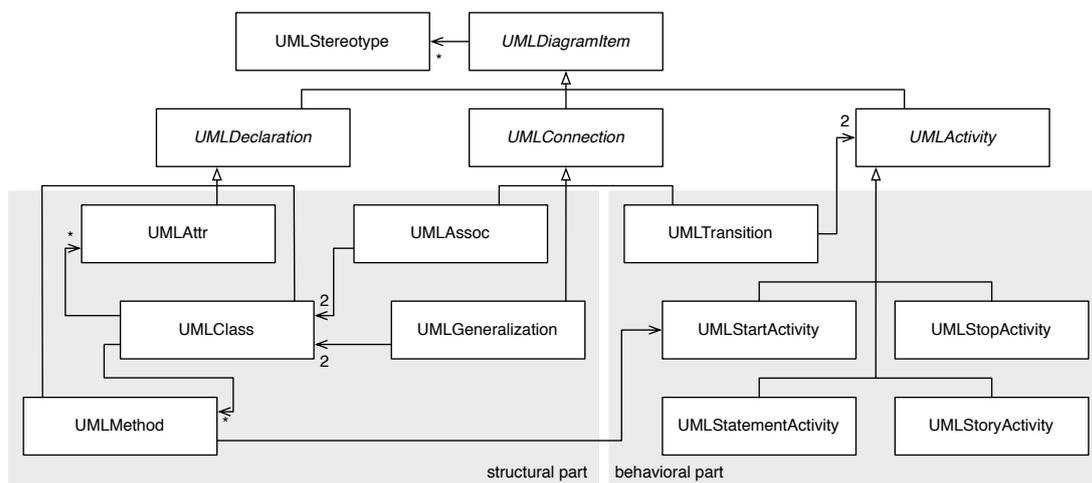


Figure 4.4: Simplified UML metamodel

4.3 Fujaba Internals

In order to understand both the integration of the stratification plugin into Fujaba and the employed transformation language, this section introduces a few necessary internal concepts of the Fujaba tool suite.

Although Fujaba supports UML, its current metamodel is not based on MOF⁸.

⁷http://www-i3.informatik.rwth-aachen.de/tikiwiki/tiki-index.php?page_ref_id=213
(checked August 2009)

⁸MOFLON (<http://www.moflon.org/>) is build on top of Fujaba and offers a MOF metamodel, which will eventually be integrated directly into Fujaba.

Instead a proprietary model is used. The basis forms the ASG⁹ metamodel. A simplified version of the metamodel is shown in Figure 4.5. `ASGElementRef` enables the extension and combination of existing metamodels and is used in Section 4.5 to add model annotations to Fujaba diagrams.

Each model element contains a hash map, which handles references to other elements. For each concrete type, the hash map contains one entry (thus the employed qualifier `getClass().getName()`). This entry (a class implementing `ASGElementRef`) then handles all references of this type.

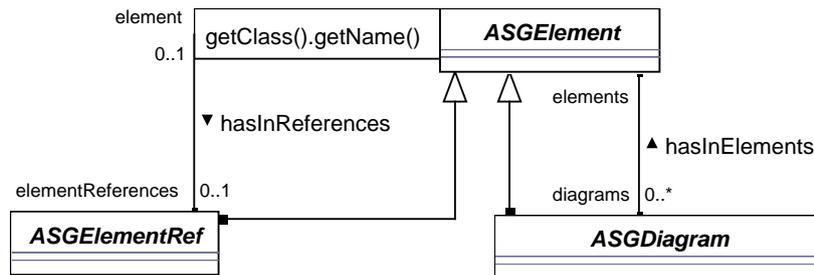


Figure 4.5: Abstract Syntax Graph metamodel [Kla05]

The elements of the Fujaba UML metamodel inherit from the class `ASGElement`¹⁰ and implement interfaces from the generic Fujaba metamodel package. These Fujaba metamodel interfaces (short F-interfaces) represent a more generic access to the metamodel without the specifics of UML. Thus each concrete model element is represented by a Java class, which directly or indirectly inherits from `ASGElement` and implements one or more F-interfaces. The representation by Java classes is mandatory, because story diagrams describe method behavior by means of object manipulation and these objects are instantiated from the metamodel classes.

Fujaba employs a variant of the well-known Model-View-Controller principle to decouple the model representation from the visualization. For each model element class an “unparse module” class exists. During the creation of the model element (either from the user interface or during import or loading) Fujaba searches for the matching unparse module and executes its create method. This method creates the appropriate user interface elements, which visualize the model element. It also attaches appropriate event listeners to the interface objects. Instead of directly using Java’s Swing implementation, Fujaba uses adapter classes¹¹, which contain attributes to link back to the associated model element. This is required to forward changes from the user interface to the model.

Fujaba currently offers two file formats for project files. The deprecated mechanism “FPR” traverses the data and serializes it to a plain text file. The newer format

⁹Abstract Syntax Graph

¹⁰For instance the abstract class `UMLDiagramItem` shown in Figure 4.4 inherits from `ASGElement`.

¹¹Abbreviated with FSA for Fujaba Swing Adapter.

(“CTR”) is based on CoObRA2¹² [Sch07] and stores—also as plain text—a stream of model modifications. In addition to these formats, the Graph eXchange Language GXL¹³ is supported. Combined with an XML stylesheet, im- and export of XMI files is accomplished. Export of both bitmap and vector graphics is available as well.

Although a project file may contain several diagrams, it consists only of one model. It is for instance not possible to have two class diagrams containing classes with the same name but different attributes.

As the main focus of Fujaba is the creation of executable Java programs, it contains an extensive foundation for code generation facilities. The actual code generation is provided by plugins, of which the most commonly used plugin is CodeGen2¹⁴ [GSR05]. It uses a model-driven and template-based approach to code generation, which can be extended for other destination languages or new metamodels.

The user interface of Fujaba and its plugins is described in XML files. Each event definition refers to a java class containing the code to be executed when the event is initiated (e.g. from menus or the toolbar). Plugins consist of a plugin definition XML file, a JAR file containing the classes and additional resources such as user interface and preference descriptions.

4.4 Integrating Stratification into Fujaba

Architecture Stratification is implemented by a set of Fujaba plugins. They extend the user interface to support navigation between strata, annotation of models and automated model transformation.

Development of the main component—SPin—was started in 2005 with the diploma thesis of Felix Klar [Kla05]. It reused Fujaba’s graph transformation capabilities for model transformation and added support for multiple strata. SPin was subsequently improved and adapted to later Fujaba versions, while additional plugins refined the transformation process. An overview is given in Figure 4.6. The upper left part contains the components of Fujaba, the lower part belongs to SPin. To the right the separate code fragment generation plugin Futemplerator is shown.

The remaining chapter describes the current state of the implementation and details a few technical aspects of architecture stratification. The components related to the transformation process (Traceability, Transformation and “Futemplerator”) are described in the following chapter.

¹²Concurrent Object Replication frAamework, see <http://www.se.eecs.uni-kassel.de/~maven/sites/coobra2/> (checked August 2009) for more information.

¹³<http://www.gupro.de/GXL/> (checked August 2009)

¹⁴<http://www.se.eecs.uni-kassel.de/index.php?codegen2> (checked August 2009)

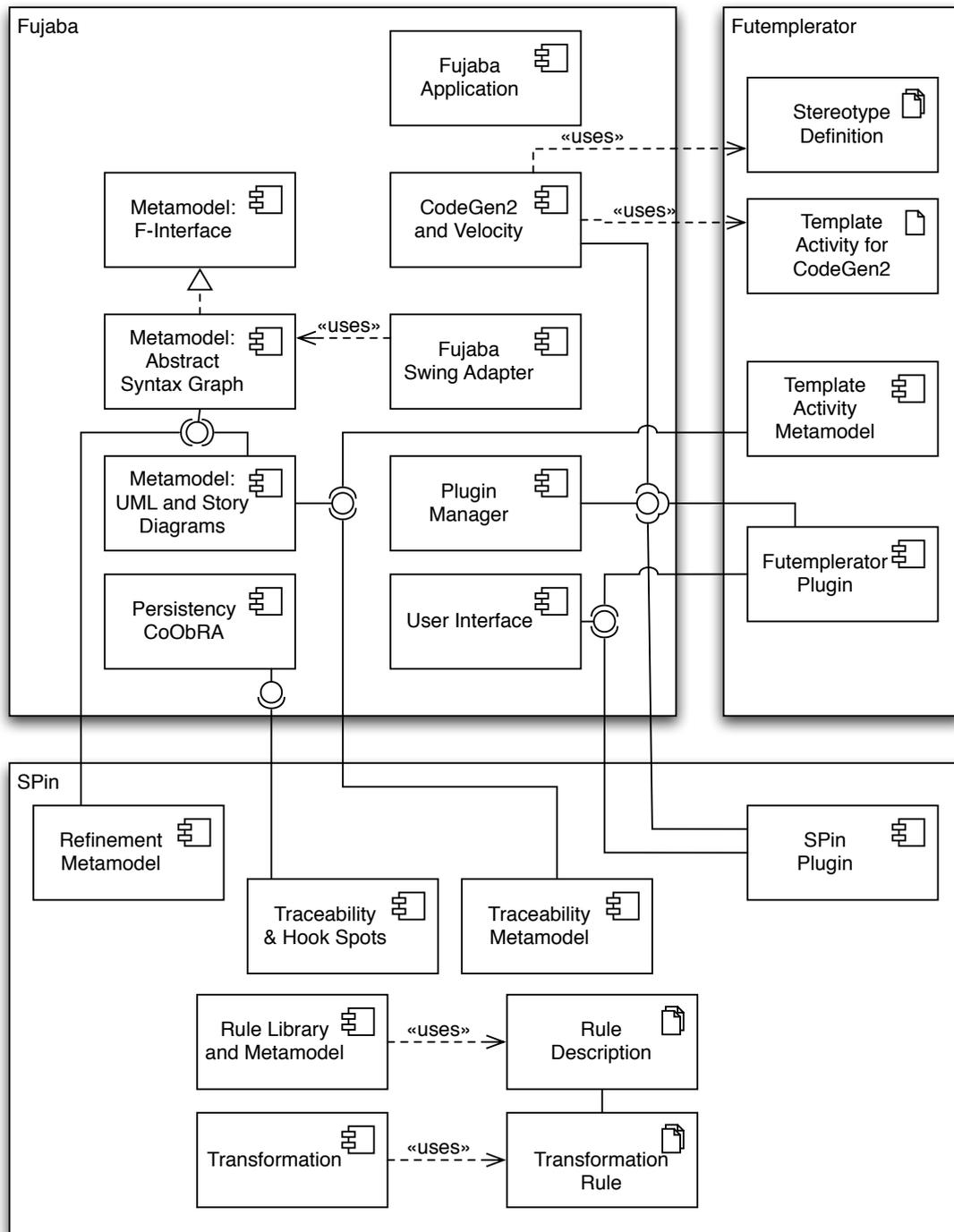


Figure 4.6: Architecture of Fujaba and the plugins implementing stratification

4.5 Extending the Modeling Language

Section 3.4 introduced a model annotation concept based on UML collaboration. It has been implemented by extending Fujaba’s metamodel using the “meta-model extension pattern” [BGN⁺04]. Figure 4.7 shows the relevant parts of the extended metamodel. The added classes are prefixed with an “R”, which indicates “Refinement”.

Shown in the middle is the abstract base class of the SPin metamodel: `RElement`. By inheriting from `ASGElement`, instances of its concrete subclasses `RAnnotation` and `RLink` can be placed into Fujaba diagrams. The parameters are stored by `RParameter` objects, which are connected through an association to `RElement`. Conforming to the “meta-model extension pattern”, `RLink` contains associations to the adapter class `RLinkToASGElement`, which inherits from `ASGElementRef`. Instances of this (and other) adapter classes are stored within the `hasInReferences` association of the `ASGElement` class, thus enabling “cross metamodel” connections from instances of `RLink` to arbitrary Fujaba metamodel elements. This construction makes it possible to extend a metamodel without modifying it.

Fujaba supports structural class diagrams and uses story diagrams to model method behavior. This combination can be seen as a “hierarchical metamodel” with two layers. Because no information is duplicated between those two layers, no synchronization issues exist.

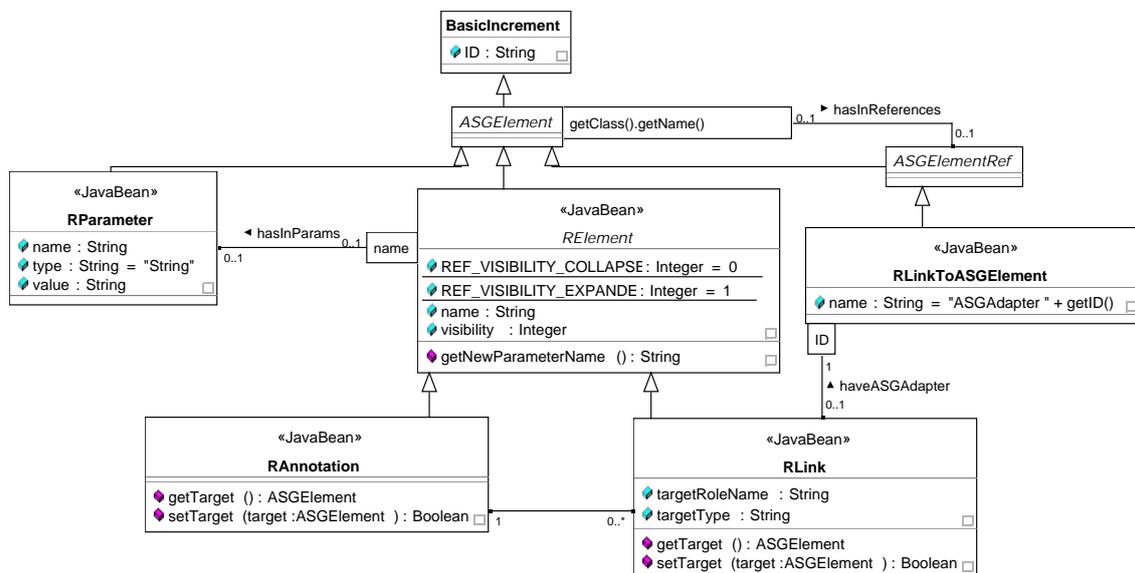


Figure 4.7: The SPin metamodel extension for annotations

4.6 Transformation Rules

As described in Section 3.5, each annotation type relates to a refinement transformation rule, which employs model transformation to implement a concern into the system. In addition, stratification can also be used to reengineer existing systems. For this case additional transformation rules (called abstraction rules) exist, which extract abstract concern descriptions from implementations. This process creates annotations on more abstract strata.

Rules are represented by Java classes and additional metadata is saved in XML structures (see following section for details). The method `apply(annotation: RAnnotation)`¹⁵ of the class is responsible for the transformation. Writing model transformations in pure Java code is obviously a complicated and complex task, so instead more abstract mechanism are used to describe transformations. Details on the model transformation approach can be found in Chapter 5.

4.7 Rule Library

Architecture Stratification is a very flexible approach to model-driven software development, which necessitates an extensive library of annotations¹⁶ for all different kinds of concerns. This section introduces the organization mechanisms of this library, which are needed to describe dependencies between rules and give information on the addressed concerns.

Figure 4.8 shows the model used to describe transformation rules. SPin loads all available rules along with their metadata and provides the user-interface to browse through the library. The `RuleInfo` object, shown to the left, contains several classification schemes:

packageName (attribute of RuleInfo) Mirrors the package name of the associated transformation rule. Used to determine vendor and other packaging aspects like relations to frameworks (e.g. `de.tud.spin.j2ee.beans` for transformation rules related to the J2EE platform).

category (attribute of RuleInfo) Main category for the concern to be implemented by this annotation (e.g. security, persistency).

keywords (association from RuleInfo to String) Additional list of keywords related to this transformation rule.

¹⁵The parameter `annotation` contains the annotation to be refined.

¹⁶Which are, in fact, transformation rules.

4.8 Stratum Handling

SPin uses separate Fujaba project files to store each stratum. As mentioned above, no data is propagated to more abstract strata, so switching to higher strata is simply achieved by loading the appropriate project file¹⁸. To navigate to a lower stratum, SPin “replays” the transformation steps and recreates the intermediate strata. This automatically propagates any changes to these strata. Details on how the transformation is replayed and how changes are propagated are discussed in Section 5.8.

4.9 Code Generation

The most concrete stratum contains a detailed and complete description of the developed application. In order to execute it, code has to be created from the model. For this step, SPin uses the capabilities provided by Fujaba and the aforementioned CodeGen2 plugin. Extensions to this code generation mechanism are discussed in Section 5.7.3. This final code generation step is not to be confused with the generation of code blocks during SPin’s model transformation.

4.10 User Interface

The structural model of an application is created with Fujaba’s UML class diagram editor, which has been extended with annotations. An example is shown in Figure 4.9. It contains a simple address book model and a class representing the user interface. The annotation “ListView” adds a list window to the user interface displaying entries from the address book. “PrintList” and “EditWindow” represent specific actions within the user interface, “Access” adds getter/setter methods to the entry class.

Model annotation is accomplished with SPin’s annotation editor, which is shown in Figure 4.10. The annotation type can be selected from the popup menu. Below the popup menu, the annotation links of the currently selected annotation (in this case “Access”) are shown. The “create links...” button populates the links section with the mandatory and optional links. A separate dialog for link and annotation parameters is available as well. The annotation metadata is shown in the lower section of the dialog. It gives a description on the annotation and the associated concern including required and optional links and parameters.

¹⁸Of course, SPin provides a separate user-interface for strata navigation, which in turn loads the relevant file. See Section 4.10 for more information.

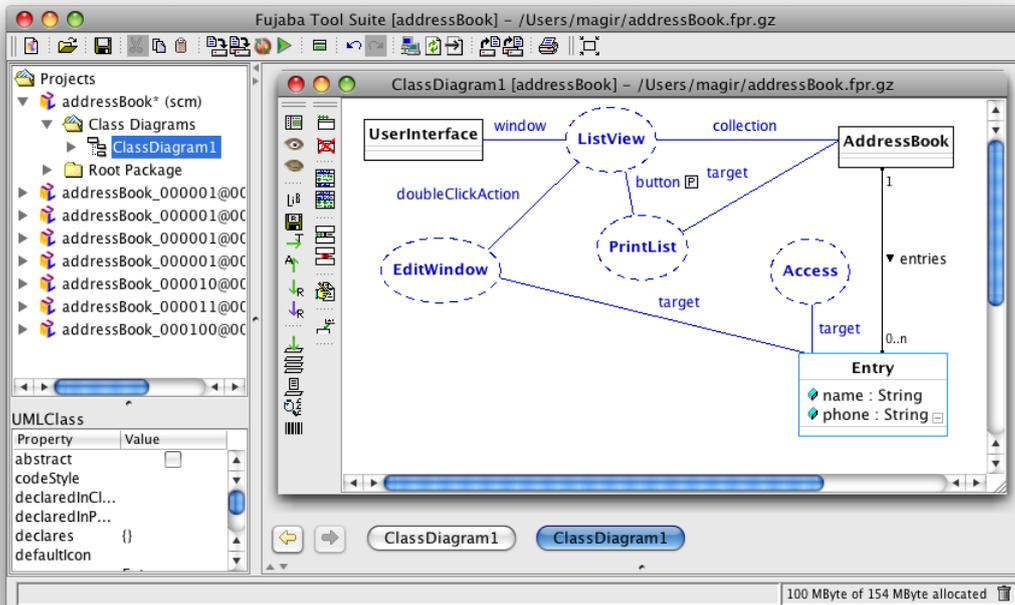


Figure 4.9: Fujaba's class diagram editor with an annotated class diagram

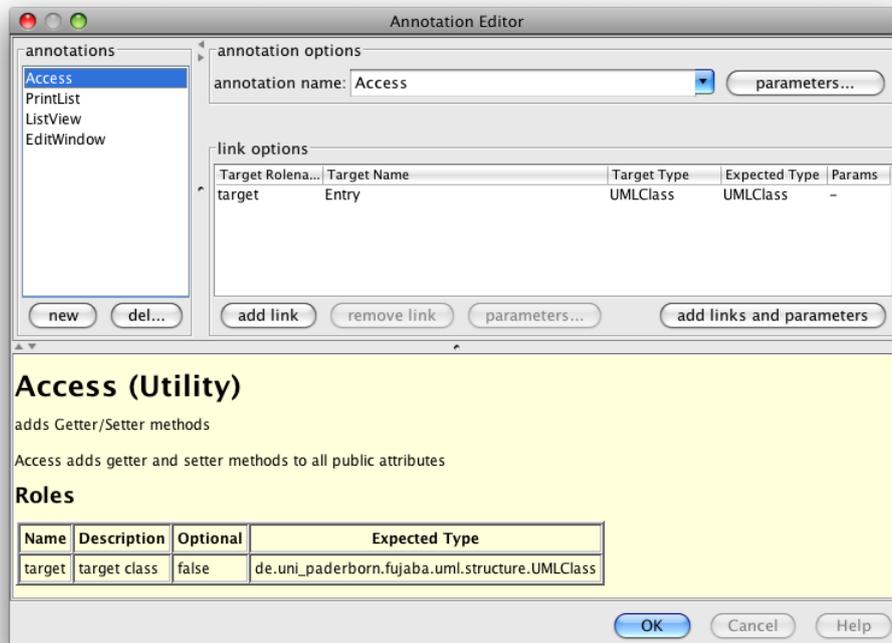


Figure 4.10: SPin's annotation editor

Figure 4.11 shows the SPin strata navigator for the address book application. Each stratum may contain a textual description of the addressed concern, which can be edited from this dialog. Clicking on a stratum automatically either loads the relevant project file for higher strata or replays the transformations necessary to reach lower strata.

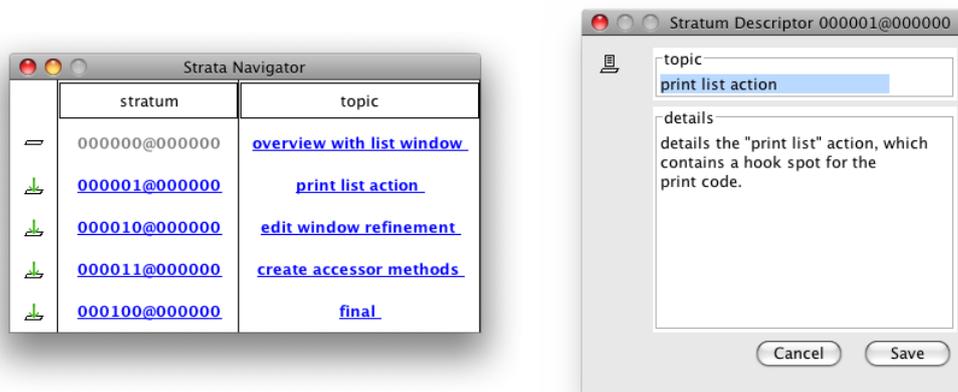


Figure 4.11: SPin's strata navigator and the strata description window

To create a new stratum, the developer selects one or more annotations from the model and chooses the refinement operation from the toolbar. SPin then executes the appropriate transformations in the selected order. If a higher stratum already exists, the developer is asked, whether a new stratum has to be inserted before the adjacent higher strata or whether previous selected annotation refinements are replaced by the current selection. A warning is displayed, if the latter selection leads to data loss in following strata. SPin also provides extensive wizard functionalities for transformation rule developers.

Chapter 5

Model Transformation

The previous chapter gave an overview of the architecture stratification implementation SPin. The missing piece for automated concern implementation is a model transformation facility. In this chapter, current model transformation languages are presented, a suitable language is selected and necessary language extensions are discussed.

Section 5.1 introduces general transformation concepts. In order to describe and classify model transformation languages, Section 5.2 then presents two classification schemes. Based on the described features and classifications, the language requirements are detailed in Section 5.3. In Section 5.4 suitable transformation languages are analyzed.

The selected language is explained in detail in Section 5.5, the integration into SPin is described in Section 5.6. Two important transformation language extensions are presented in Section 5.7 and Section 5.8. First the integration of a code generation language and then the mechanism to enable manual model modifications. The chapter closes in Section 5.9 with a short summary.

5.1 Concepts

Some of the technical terms used in this chapter may not be commonly known and are therefore briefly introduced here.

- A transformation requires an **input (source)** and uses it to create an **output (target)**.

- A **transformation language** is a domain specific language, designed to describe transformations. It consists of a syntax and a semantic description.
- A **transformation rule** expresses a transformation using the transformation language. Rules can be grouped into **rule sets** and may contain several **rule steps**.
- The rules are executed¹ by a **transformation engine**.
- Many transformation languages are **pattern** based. In this case, a transformation rule usually consists of a pattern (or query) and the actual transformation. The input is scanned for the pattern and resulting **matches** are used to create the output.
- Patterns can be displayed in **abstract or concrete syntax**. Concrete Syntax—similar to templates—shows the pattern in the syntax of the model, abstract syntax uses the syntax of the metamodel. An example can be found in Figure 5.1. To the left is a pattern represented in the concrete syntax of UML class diagrams, to the right is the same pattern in abstract syntax using the metamodel of the CASE tool Fujaba.
- Graph-based transformations represent the model by using a **typed, attributed graph**. Depending on the employed metamodel, the type system may support inheritance and other specific features. The graph pattern consists of a typed graph fragment and additional guards, which specify boolean expressions on the graph element attributes. Some engines also support “**negative application conditions**” (NACs), which specify the absence of graph fragments.
- Different techniques are used to define transitivity within the graph. Textual **path expressions** (often using the Object Constraint Language [OMG06]) are one possibility, **recursive matching** techniques another.
- In the context of graph transformation it is often distinguished between the search pattern on the “**left hand side**” (LHS) and the transformation pattern describing the result on the “**right hand side**” (RHS)².

5.2 Model Transformation Classification Schemes

In the context of architecture stratification, transformations are responsible for implementing concerns into a system. Hence, the employed transformation language

¹Whereas “execution” can be seen in different ways, see below.

²Some tools unify both in one diagram, see below.

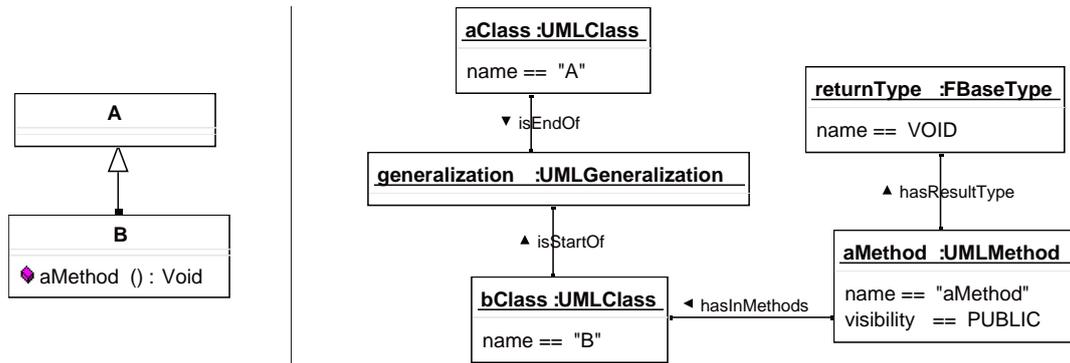


Figure 5.1: A model fragment represented in concrete and abstract syntax

has to fulfill certain requirements. Before stating these requirements, two classification schemes for model transformation languages are introduced. They are used to organize the requirements and categorize the evaluated approaches.

The first classification scheme, outlined by Czarnecki and Helsen [CH03, CH06], uses a feature-model to describe “the different design choices for model transformations” with regard to model-driven development. They identified five main application scenarios (cf. [CH06], enumeration added):

1. Generating lower-level models—and eventually code—from higher-level models
2. Mapping and synchronizing among models at the same level or different levels of abstraction
3. Creating query-based views of a system
4. Model evolution tasks such as model refactoring
5. Reverse engineering of higher-level models from lower-level models or code.

Both the first and last scenario is relevant to stratification. A broad range of tools and languages have been analyzed for the feature-model and helped to define a few major categories. First, a short summary of the relevant criteria from the feature model are given, then the major categories are described.

Transformation Rules This criterion describes, how the transformations are specified.

Domains If both models conform to the same metamodel, a transformation is called “endogenous”, if the metamodels are different “exogenous”.

Patterns Three types can be distinguished: string, term and graph pattern. The latter two can be represented both textually and graphically in abstract or concrete syntax.

Logic Computations of attributes may be expressed declaratively or imperatively.

Parameterization Rules may be parameterizable in several ways. Control flags can be used to add variability, generics and higher-order rules may provide a higher level of reuse.

Rule Application Control A strategy to determine the location within the model, where the rule is applied: deterministic, nondeterministic or interactive. Different kinds of scheduling: implicit (without control constructs) or explicit. The latter can be external with a separate control structure or internal with rules invoking other rules. Rule selection can be explicit, nondeterministic or interactive.

Rule organization Modularization and reuse mechanisms (e.g. inheritance between or composition of transformation rules). Also the organizational structure of the rules (oriented towards the source or target metamodel).

Source-target relationship Some tools use explicit source and target models (sometimes described as “out-place”), others perform the transformation in place, modifying an existing model.

Incrementality Target incrementality enables the updating of existing target models. Source incrementality defines the “amount of source that needs to be re-examined by a transformation when the source is changed”. Preservation of user edits in the target is also considered in this criterion.

Directionality Are the transformations uni-, bi- or multidirectional³?

Tracing Recording of information (e.g. trace links between source and target) during the transformation process. The information is either stored within the model or separately. The creation might be automatic or manual.

Considering these criteria and the evaluated approaches, Czarnecki and Helsen proposed the following major categories:

Model-to-text

Visitor-based The internal representation is traversed and written to an output file.

³Multidirectionality can be emulated by defining unidirectional rules for every direction.

Template-based Text fragments are interleaved with “metacode”, which accesses information from the source.

Model-to-model

Direct-manipulation An API is offered to manipulate the model. The actual transformation aspects have to be implemented manually.

Structure-driven Two distinct phases, one for constructing hierarchical structure of the target model and one for setting attributes and references. The scheduling and application strategy is fixed.

Operational Here, languages with basic transformation support (e.g. extending OCL [OMG06] with imperative constructs) are combined.

Template-based Similar to the model-to-text variant, model templates containing model fragments and metacode to access model information are employed (e.g. by using model annotations).

Relational This category subsumes declarative approaches, which use constraints to specify relations between models.

Graph-transformation-based These languages usually operate on typed, attributed, labeled graphs and use graph-rewriting techniques for the transformation.

Hybrid approaches A combination of the approaches mentioned above, implemented as separate components or at a more fine-grained level within the transformation rules.

A second classification scheme was proposed by Mens and Van Gorp [MG06]. It shares some criteria with the aforementioned publication [CH06] but focusses on issues such as usability and extensibility. The following selection of criteria proved also to be helpful in the context of this dissertation.

Horizontal versus vertical transformations This criterion describes the change of abstraction. In vertical transformations, source and target reside on different abstraction levels, in horizontal transformation they are on the same level.

Syntactical versus semantical transformations Syntactical transformations only change the representation, semantical transformations also the semantics.

Testing and validating Subsumes systematic techniques for debugging, testing and validation.

Non-functional requirements Contains aspects such as usability and usefulness, verbosity versus conciseness, performance and scalability, etc.

5.3 Language Requirements

The two classification schemes, introduced in the previous section, are now applied to stratification. The conclusions are then condensed into seven requirements for a transformation language, which is suitable for concern implementation in the context of stratification.

Applying the criteria of the first scheme [CH06] leads to the following conclusions:

The reference implementation, introduced in Chapter 4, uses static class diagrams for the main model. But as the approach itself is not limited to that, transformation languages should support other modeling languages as well. As long as only one metamodel is utilized for all strata, endogenous transformations are sufficient. If additional DSLs are used on intermediate strata, it is still necessary to transfer any model information from the strata above to the strata below. Thus it becomes necessary, to compose all needed metamodels and merge the complete information in one model. Hence exogenous transformations are—strictly speaking—not required.

As primarily graphical models are used, graph-pattern languages are preferred. The expressiveness of attribute computations is very important but can be achieved with both declarative and imperative logic. Parameterization is also very significant, because the (parameterized) annotations serve as input to the transformation process.

Rule application control is related to the model annotation process. The location and selection of a rule is defined by the annotation and can thus be considered deterministic or interactive, even though the annotation process is separated from the transformation. If the transformation is applied to one location at a time, non-deterministic algorithm would lead to the same result as deterministic approaches. In both cases the affected area is well-defined.

Scheduling control of rule sets is not required as only one explicitly selected rule is applied at one time. Explicit scheduling can also be available *within* a transformation rule, for instance in hybrid approaches, which often combine imperative control elements with declarative transformations. In this case, scheduling is an important aspect to structure the transformation rule. It is also related to rule parameterization, because often parameters lead to “if-else” distinctions, which can easily be handled with control flow constructs.

A major difference between stratification and other model-driven approaches is the rule organization. In stratification the transformation only affects a part of the model, other approaches (e.g. most MDA tools) transform the *complete* model into a different representation. Their transformation rules are often organized by the source metamodel, resulting in one transformation rule per type within the metamodel. Compared to that stratification organizes the rules according to the concerns they solve. This may rule out structure-driven approaches, which are often organized by

source metamodel.

The implementation of a concern affects typically only one part of the system while most parts are copied from the previous stratum during transformation. Although in-place transformations would simplify transformation rules, they are not necessarily the optimal choice, because source and target are two *separate* strata within the stratified architecture. One possible solution is a “deep-copy” mechanism, which copies the model to the next stratum and then applies the transformation in place.

In order to support strata synchronization and the controlled editing of intermediate strata, both “target incrementality” and “preservation of user edits” are important features.

Theoretically, bidirectional transformations can be used to transfer changes back to more abstract strata. However, as each stratum details one concern, any modification to this stratum is related to this concern only, so there is no need to propagate the changes upwards. Consequently, any change, which is not related to this concern should be made directly on the correct stratum. The propagation of new elements to higher strata may pollute these (more abstract) layers with unneeded information and is, therefore, not appropriate. Hence, bidirectional transformations are not required.

This dissertation concentrates on (unidirectional) refinement transformations, which implement concerns within a stratified architecture. The requirements for abstraction rules, which “detect” the presence of concerns within an existing software system, differ in several ways and are thus subject to future research.

Tracing element transformations from source to target can be implemented in different ways. For instance, Borland Together only produces “trace logs”, which contain information on the used transformations but they have no effect during subsequent transformations. Other tools (e.g. GReAT, MOLA) use trace data during the transformation to store auxiliary information, but this data is discarded after the transformation has finished. In order to support target incrementality, the trace information has to be stored persistently⁴. This property of tracing is also required to support preservation of user-edits and is thus mandatory for stratification.

Based on the criteria, Czarnecki and Helsen [CH06] outlined major categories of transformation languages. “Low-level” approaches (direct-manipulation, structure-driven, operational) are not further considered. Although at first sight template-based model-to-model transformation languages seem to be useful, they cannot be used for model transformation easily but rather to generate new models. Owing to the fact that transformations are applied to graphical models, graph-transformation-based approaches seem a natural choice. This also includes hybrid approaches, which employ graph-transformation. Czarnecki and Helsen see deficiencies in current

⁴This is often accomplished by an additional data structure, which connects both models.

graph-based approaches, because these usually consider only graphs with unordered edges, making them unpractical for code models such as abstract syntax trees⁵. Therefore, other mechanisms to generate code fragments have to be evaluated.

Since primarily models are transformed into models, the model-to-text category is not directly relevant to architecture stratification. However it becomes relevant, when the implementation of a concern also includes the creation of code fragments. These code fragments are also stored within the model. Thus, a code generation engine needs to be able to access the model to produce the necessary code blocks and re-integrate the generated code.

The classification scheme by Mens and Van Gorp [MG06] helps to derive a few additional conclusions, which are outlined in the following.

The abstract description of concerns is based on model annotations, which serve as input to the transformation process. The transformation then implements the concern, usually by removing the annotations and adding or modifying model elements. The result of the transformation is a “less abstract” stratum. In the authors’ scheme this is considered a “vertical” transformation.

The second distinction—syntactical versus semantical transformations—cannot be decided easily. On the one hand, all strata describe the same, complete system, on the other hand, the actual implementations—and thus the behavior changing aspects—only appear after the transformation.

The creation of new transformation rules is considered to be part of the stratification process. Thus the language is required to be intuitive and easy to learn, because not only specialized framework and transformation developers but also “regular” system developers may need to create or refine transformations.

The conclusions, drawn from the two classification schemes, lead to the following requirements for a model transformation language for stratification.

1. Expressive attribute computation logic and support for parameterization
2. Explicit control structure within the transformation rule
3. Based on graph patterns, preferably with a graphical notation
4. Intuitive and easy to learn syntax
5. Integrated code generation facility

⁵It has to be noted, that Fujaba and MOFLON [AKRS06] are among the few approaches, which support ordered sets of edges.

6. In-place semantics, which modify a “copy” of the model
7. Support for the preservation of user edits

5.4 Analysis of Transformation Languages

Now that the main requirements are defined, adequate transformation languages are described. The transformation languages marked with an asterisk (*) are part of tools, which have already been discussed in Section 2.3. With the exception of some QVT⁶ implementations, openArchitectureWare, Rational Software Architect and DoME the languages are research prototypes. After a short introduction and analysis of each language (in alphabetical order), the table shown in Figure 5.2 gives an overview.

Atlas Transformation Language (ATL)

[JK06b]

ATL is a hybrid language combining OCL-like constructs with basic control flow mechanisms. It was developed by the ATLAS group⁷ and was submitted to the initial QVT “Request for Proposals” [OMG02]. A comparison between the final QVT standard and ATL was published in 2006 [JK06a]. The textual language employs a syntax similar to OCL to match and create elements within the model. In addition to this declarative syntax, the transformation may also contain imperative parts. With “called rules” other transformations can be executed (similar to procedures), with “action blocks” the control flow of the transformation rule can be modified (“explicit, internal scheduling” according to the first classification scheme). The called rules can also be specified in a native language (e.g. Java). In this case, the complete model is exported, modified externally and imported again. Although ATL does not allow in-place transformations, it provides a special “refinement” mode. Here all elements, which have not been matched by any rule from the applied rule set, are automatically copied to the target model.

The Atlas Model Weaver (AMW) is a graphical editor for defining relationships between two metamodels. Based on these relationships ATL rules are created, enabling a simple transformation between the two metamodels.

Assessment: The mentioned refinement mode seems to be ideally suited for stratification, but actually differs from in-place transformation by only copying elements,

⁶Query, View, Transformations. A standardized model transformation language, see below.

⁷“The ATLAS group is an INRIA research team located at the University of Nantes and linked to the LINA research laboratory.” <http://www.sciences.univ-nantes.fr/lina/atl/AMMA/presentation/> (checked August 2009)

which have *not* been matched during transformations. Moreover, ATL neither offers a graphical syntax nor a built-in code generation facility.

Bidirectional Object-Oriented Transformation Language (BOTL) [BM03]

BOTL is a graph-based declarative bidirectional transformation language. Similar to other graph-based approaches it uses a pattern to define the query in the source. The target pattern is always completely created and then merged with other target patterns as well as the target model. The merging is based on key identifiers within the model. Given the fact that BOTL is a purely declarative approach no control or scheduling language exists. Restrictions within attribute calculations guarantee unambiguous results, independent from rule execution order. Furthermore these restrictions enable bidirectional transformations. The current implementation uses text based descriptions only, even though a graphical syntax exists.

Assessment: The merging of transformation results with the target model enables incremental transformations, but it depends on the stability of the employed key identifiers. This requires careful selection of the attributes, which are used for the identification. Object names are unsuitable, because the renaming of objects may result in the creation of superfluous objects. The restricted attribute computation language and the lack of a control flow language makes the design of transformations rather complicated.

DOmain Modeling Environment (DoME)* [OSBE01]

The primary focus of DoME is the pattern-based implementation of software systems. Initially developed for internal use at Honeywell International it was later released to the public. Since 2003 no updates appeared on the website⁸. Transformations are based on parameterizable patterns, which are instantiated and merged with an existing model. The merging is described by portals: “Through portals, a user completes an instantiation of a pattern or completes the instantiation of a model with elements from within the pattern.”. The patterns are stored in a library and contain the templates for code generation.

Assessment: The pattern applications are stored permanently and can be updated, if the pattern or the parameterization changes. This emulates multi-level modeling but does not allow the model to be viewed on a higher abstraction level. The pattern instantiation process is less powerful than a complete model transformation language, but it offers a good integration with the employed template language.

⁸Please note, that, the website is unavailable from time to time, check <http://www.archive.org> instead.

Fujaba Story Driven Modeling (SDM)*

[NNZ00, FNTZ00]

Fujaba provides a graph transformation language named “story driven modeling”, which is used to model behavior of operations within a software. Implemented by the Universities of Paderborn, Kassel and Darmstadt, this CASE tool forms the basis for the stratification implementation presented in this dissertation (cf. Chapter 4). The applicability for model transformation is demonstrated by Grunske et al. [GGZ⁺05]. SDM employs extended activity diagrams to describe control flow. Different activity types are supported (e.g. declarative graph transformations and Java code), new types can be easily added by using plugins.

Assessment: The main focus of SDM is modeling behavior and not model transformation, therefore some restrictions apply. For instance only endogenous transformations within one metamodel can be performed. The extensibility of the transformation language through the use of plugins is not available in the other languages presented here.

Graph Rewriting and Transformation (GReAT)*[AKK⁺06]

Compared to other graph rewrite languages, the GME plugin GreAT from Vanderbilt University employs concrete syntax (cf. Section 5.1) to describe transformations. It allows an arbitrary number of input and output models and uses an additional non-persistent model, which contains the tracing links and auxiliary elements. Color coding is used to mark created and deleted objects, for attribute assignment a language similar to C++ is available. Each step within the rule has input and output ports, which are connected to form a data flow diagram. When an output port is empty, the transformation ends. Specific operations for sequencing, branching and hierarchical composition enhance the data flow to a full control flow language.

Assessment: Although the concrete syntax is more compact, its use leads to problems, when defining invisible model artifacts or expressing attribute calculations. The input and output ports of the transformation steps are quite useful in practice. In-place or incremental transformations cannot be performed easily and no code generation facility exists.

MModel transformation LAnguage (MOLA)

[KCS05]

In MOLA, which is developed at the University of Latvia, transformation rules consist of an imperative control flow defined by a structured flowchart. The nodes of the flowchart are declarative graph transformation patterns, which use a combined, abstract syntax for LHS and RHS, marking deleted and added elements by using colors. Furthermore a merged metamodel, containing the source and target meta-

model, is required. It also contains—similar to GReAT—auxiliary elements and tracing edges between the metamodels, which can be used during the transformation. The flowcharts offer loop constructs and the ability to call other rules, which enables recursion and reuse.

Assessment: MOLA does not consider any type of code generation. Besides that the language offers a good combination of imperative and declarative constructs, quite similar to Fujaba’s SDM. Incremental transformation—although possible by using the tracing data structure—is not implemented, yet. According to the developers of MOLA, it may require changes to the procedural language or especially designed transformation rules.

openArchitectureWare (oAW)

[HVEK07]

Initiated by the German company B+M Engineering, the now open-source transformation framework has gained wide acceptance in commercial model-driven development projects. It consists of two languages, XTEND for model-to-model transformations and XPAND for model-to-text transformations. XTEND is a textual, functional language, which is not only used for model transformation but also for validation (using an OCL-like syntax) and metamodel extension. Both languages can be combined with a third workflow language. Additionally oAW offers a Java API for in-place model transformations.

Assessment: XTEND does not offer any graphical syntax and the integration with XPAND is only indirectly accomplished using the workflow language. Merging of models is supported, though no true incremental transformation exists. The XTEND language offers simple control flow statements and calling external Java methods for more complex transformations.

Query-View-Transformation (QVT)

[Fon06, Tel08, KE07, Tat06]

In 2002 OMG issued a “Request for Proposals” [OMG02] concerning a transformation language based on the existing OMG standards UML and MOF with a primary focus on MDA. In 2007, the final specification was released [OMG07a]. Depending on the actual implementation, QVT supports an arbitrary number of input and output models. In-place transformations can be performed as well by using the same source and target model. An auxiliary trace data structure can be used to implement incremental transformations.

QVT consists of several parts. The “operational mapping” specifies a textual procedural language, which uses an extended version of OCL to express transformations and control structures. Additional constructs enable the reuse of transformations through composition and extension. Two nearly complete subsets of QVT/oper-

ational have been implemented by Borland (Together [Fon06], commercial) and France Telecom R&D (SmartQVT [Tel08], open source).

The second part of QVT is called “relations”. This purely declarative language defines mappings between models. A graphical syntax is specified, but all known implementations currently use the textual syntax only. When two models are “related”, trace information is created automatically allowing subsequent incremental transformations. Supplementary clauses control the application of relations, however a full control structure is not available.

The execution and thus the semantics of relations is defined by a mapping to a third QVT language named “core”. This imperative language can be seen as an intermediate transformation language used to implement other QVT languages, it is not designed for end users. QVT/relations is available as an open-source implementation from IKV++ Technologies AG (medini QVT [KE07]) and Tata Consulting (ModelMorf [Tat06]). Yet another implementation is planned by the Eclipse Foundation and will be realized by OBEO⁹.

As a fourth language, QVT also allows “blackbox” transformations. They are implemented externally and their only requirement is the ability to deal with MOF/XMI models. Currently no implementations are known.

Assessment: The operational mapping is a very powerful, yet complex, transformation language. Due to its design it is limited to a textual syntax. The automatic population of the trace data structure is only possible with QVT/relations. The generation of code fragments is neither defined by QVT nor available in current implementations. A separate specification for model-to-text transformations has just been released [OMG08].

Rational Software Architect (RSA)*

[SCG⁺05]

The transformation capabilities of IBM’s RSA stem from different previous products together with third party frameworks. The pattern engine, available in previous versions, uses Java for code generation and model transformation. The current release adds a mapping language, which transforms between rather similar models. This mechanism is extended by Java code for more complex transformations. Support for tracing is available but has to be implemented manually. The transformation engine is based on the Eclipse infrastructure and is primarily designed for one input and output model. For model-to-text transformations, the Eclipse JET engine¹⁰ is used.

Assessment: Parts of the transformations are specified on a rather high abstraction

⁹<http://www.obeo.fr/> (checked August 2009)

¹⁰<http://www.eclipse.org/modeling/m2t/?project=jet> (checked August 2009)

level, but the actual transformations are implemented in Java code on a rather low level. Integration of code generation through the JET engine is possible but the integration of both engines is rather low.

Triple Graph Grammars (TGG)

[Sch95]

The primary application for Triple Graph Grammars is incremental model synchronization. TGG rules consist not only of LHS and RHS but also of a persistent third graph, which connects LHS and RHS. The information in this graph can be used for subsequent incremental transformations and information propagation. Specific implementation issues for incremental transformations are discussed by Giese and Wagner [GW06]. TGGs were first described by Schürr [Sch95], implementations now exist for several tools. One example is the MOF-based metamodeling CASE tool MOFLON [AKRS06], developed at the TU Darmstadt. It is based on Fujaba and uses story diagrams as intermediate language to execute model transformations. TGGs are purely declarative and do not provide any control structure, although support for it has been discussed by Van Gorp et al. [GMJ06].

Assessment: Similar to QVT/relations and BOTL, TGG is a purely declarative transformation language without structuring control flow constructs nor direct code generation facility. The graphical syntax and the use of a persistent tracing structure are quite powerful constructs, which are ideally suited for synchronization between strata.

Visual Automated model TRAnsformations (VIATRA2)

[BV06]

This framework, designed at the Budapest University of Technology and Economics (Department of Measurement and Information Systems), uses a textual graph rewriting language for transformation. Matching-patterns can be defined recursively including negation, which is a very powerful concept. Transformation patterns can be described separately or combined with the query pattern. In order to support the reuse of patterns, runtime parameterization and typing of patterns is supported, effectively creating a “meta-pattern” approach. Control flow is defined through an abstract state machine (ASM), which also enables calling native Java code for more complex transformations. Code generation is handled by a template based language extension, which is similar to Velocity¹¹. Support for live transformations¹² [RBÖV08] was added recently.

Assessment: Even though a graphical syntax exists, the current implementation uses the textual syntax, only. The integration of the template language merely

¹¹<http://velocity.apache.org/> (checked August 2009)

¹²Live transformations are automatically triggered by model changes.

supports context transfer towards the template engine and not back to the model. While incremental transformations are possible, they have to be specified manually using negative application conditions in order to prevent creation of superfluous model elements.

Visual Modeling and Transformation System (VMTS) [MLLC06, LLVC06]

VMTS consists of a metamodeling environment and a model transformation facility based on graph transformation. It was developed at the Budapest University of Technology and Economics (Department of Automation and Applied Informatics). The source and target patterns are defined separately and so-called “internal causalities”—specified in a separate dialog—describe the relationships between them (e.g. links, deletions, creations, etc.). Furthermore, “imperative OCL” constructs are used to describe attribute calculations. Transformations are organized using the Visual Control Flow Language (VCFL), which resembles UML activity diagrams (e.g. decision nodes, split and join nodes). Data flow between transformations is accomplished by using “external causalities”, which select elements from the RHS of one rule and pass them on to the LHS of another rule. The transformation language is also used for code generation by creating an intermediate model similar to an abstract syntax tree.

Assessment: The use of separate editor dialogs for relationships and attribute calculations removes important information from the main transformation. Moreover the surrounding control flow is shown in a separate diagram. Code fragment generation in the context of stratification requires different mechanisms, which cannot be implemented using VMTS.

Summary

The table in Figure 5.2 gives an overview of the presented transformation languages. Implementations supporting textual syntax are marked with a “T”, graphical syntax is denoted by a “G”. A combination of both is possible. If the graphical syntax is only described informally with no existing implementation, the “G” is put in parenthesis.

Automatic tracing, which is usually required for incremental transformations, is only possible in relational/declarative approaches. Most other engines support the manual creation of a supporting data structure, albeit not all of them store this structure persistently. An implementation of QVT/relations and QVT/operational in a single engine is not available, yet. It would combine the advantages of relational approaches with imperative constructs.

Besides DoME and VIATRA2, no integration of a code generation facility with a model transformation language is available. Both are not directly capable of trans-

Tool, Institution, URL and publication	platform	syntax	control flow	attr. computation	tracing
ATL/AMW Univ. de Nantes et al. http://www.eclipse.org/m2m/at/ [JK06b]	Eclipse	T/(G)	textual constructs	OCL-like	-
BOTL TU Munchen http://sourceforge.net/projects/boi [BM03]	Java	T/(G)	-	-	automatic
DoME Honeywell International http://web.archive.org/web/*/http://www.hcc.honeywell.com/dome/ [OSBE01]	Smalltalk	G	-	-	-
Fujiaba Univ. of Paderborn, Kassel, Darmstadt http://www.fujiaba.de/ [NNZ00]	Java	G	activity diagrams	Java	-
GME/GREAT Vanderbilt University http://www.isis.vanderbilt.edu/Projects/mobies/downloads.asp#GREAT [AKK+06]	Windows	G	data flow diagrams	similar to C++	manual/live
MOLA University of Latvia http://www.isis.vanderbilt.edu/Projects/mobies/downloads.asp#GREAT [AKK+06]	Eclipse	G	structured flowcharts	-	manual
oAW b+m Informatik (and Community) http://www.openarchitectureware.org/ [HVEK07]	Eclipse	T	textual constructs	OCL-like	-
QVT/operational: SmartQVT France Telecom http://www.operarchitectureware.org/ [HVEK07]	Eclipse	T	-	QVT-OCL	manual
QVT/operational: Together Borland http://www.borland.com/us/products/together/ [Fon06]	Eclipse	T	-	QVT-OCL	manual
QVT/relations: medini QVT IKV++ http://projects.ikv.de/qvt/ [KE07]	Eclipse	T	-	QVT-OCL	automatic
QVT/relations: ModelMorf Tata Research http://www.tcs-trddc.com/ModelMorf/index.htm [Tat06]	Eclipse	T	-	QVT-OCL	automatic
RSA IBM http://www.tcs-trddc.com/ModelMorf/index.htm [Tat06]	Eclipse	G+T	Java	Java	manual
TGG TU Darmstadt http://ibm.com/software/awdtools/architect/swarchitect/ [SCG+05]	Java	G	-	-	automatic
VIATRA2 Budapest University http://www.moflon.org [Sch95]	Eclipse	T/(G)	abstract state machine	-	manual/live
VMTS Budapest University http://dev.eclipse.org/viewcs/indextech.cgi?gmt-home/subprojects/VIATRA2 [BY06]	Windows	G+T	activity diagrams	OCL-like	-

Figure 5.2: Comparison of transformation languages

ferring the generated code back to the model. The partly contradictory requirements of in-place semantics and incremental transformation are also not satisfied by the evaluated tools.

The reference implementation, described in Chapter 4, is based on the CASE tool Fujaba. Despite the fact, that story diagrams do not fulfill all of the stated requirements, Fujaba’s open plugin architecture provides an ideal basis to extend the transformation language in order to satisfy the demands of stratification.

Fujaba already meets the requirements 1 to 4. An expressive, Java-based computation logic and an explicit control structure are available, the transformations are graph-pattern based and use a graphical notation. Finally, the combination employed is easy to understand as it contains only few language constructs. The remaining requirements 5 to 7 are discussed in the following sections.

5.5 Story Diagrams and Model Transformations

Section 4.2 gave an introduction to story diagrams. This section shows, how they can not only be used to describe method behavior but also to describe model transformations. The story diagram itself describes the control flow, while the activities within the diagram (e.g. story patterns or statement activities) specify the object manipulations. The following paragraphs explain the relevant concepts of object manipulation using story diagrams and story patterns.

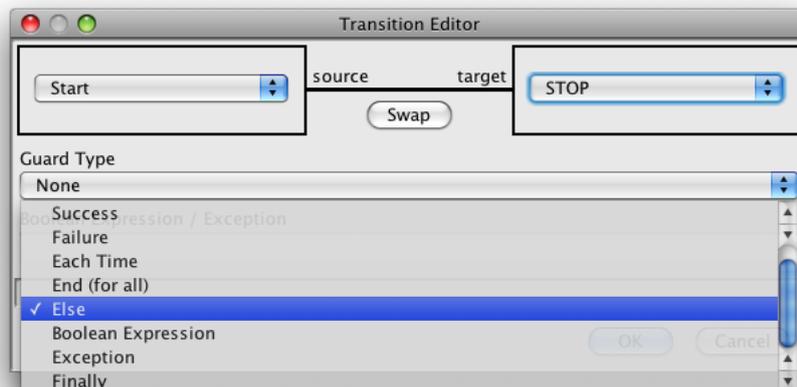


Figure 5.3: Fujaba’s story pattern transition editor

The edges between activities are called transitions and can be edited using the transition editor dialog shown in Figure 5.3. If an activity has more than one outgoing transitions, the guard conditions on each transition are evaluated. The popup menu in Figure 5.3 lists the available guard options. “Success” and “Failure”

are used in conjunction with story patterns. “Each Time” and “End (for all)” are used to construct loops. An example for this can be found in Figure 4.3 on page 60. “Else” and “Boolean Expression” enable branching. “Exception” and “Finally” are available for Java exception handling.

The model transformation is performed by story patterns, which match, modify or delete existing objects or add new ones. Story patterns consist of objects and links, both are specified using editor dialogs, which are described in the following.

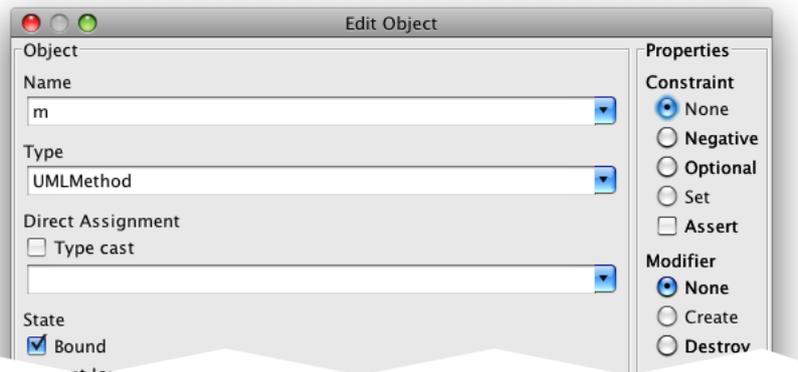


Figure 5.4: Fujaba’s story pattern object editor

The editor dialog for objects is shown in Figure 5.4. All objects within a story diagram are assigned to a local variable, whose name is specified in the name field. In order to reuse an object, which was already assigned to a local variable in a previous story pattern, the state of the object can be set to “bound”.

The three modifiers “None”, “Create” and “Destroy” denote the matching, the creation and the destruction of an object. The “None” modifier can be combined with the “Negative” constraint to express negative application conditions. The constraint “Optional” used in conjunction with “None” or “Destroy” continues the execution of the story pattern even though an object has not been matched. In conjunction with “Create” it creates the object only, if it was not found.

Objects also contain an attribute section, in which boolean guard expressions for the matching can be defined and attribute values set. The syntax is based on Java and regular expressions.

The link editor dialog, shown in Figure 5.5, has options similar to the object editor, although here the link name refers to the association, which has to be used. The dialog lists all available associations existing between the two adjacent types. Path expressions can be specified as well (not shown in the figure).

The objects and links used in story patterns are instantiated from classes and associations, which are contained in class diagrams. This is enforced by Fujaba’s story

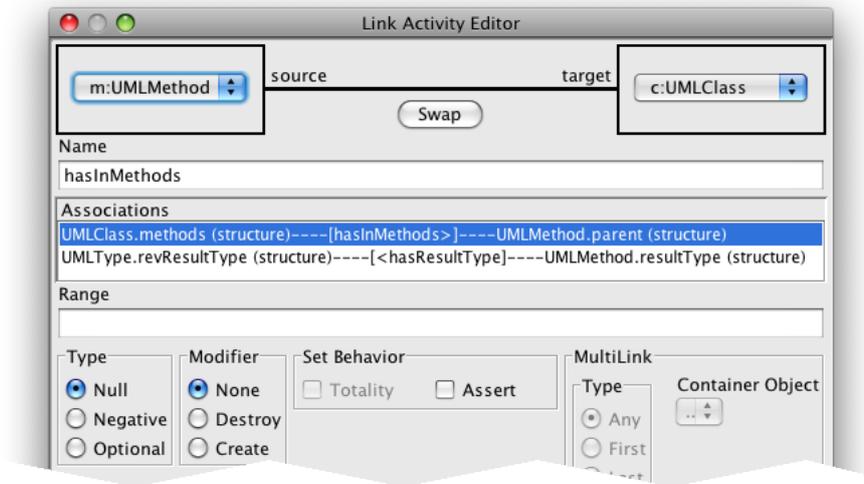


Figure 5.5: Fujaba’s story pattern link editor

pattern editors, which shows only types, attributes and associations available in the underlying class hierarchy.

This relationship between classes and objects is similar to metaclasses (the structural elements of metamodels) and model elements. Thus, if a class diagram depicts a metamodel, story patterns can be used to describe model transformations, which conform to this metamodel.

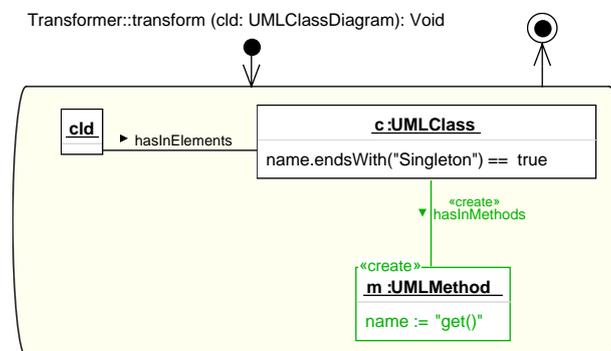


Figure 5.6: Model transformation using a story diagram

To illustrate this approach, Figure 5.6 shows a transformation of a model conforming to Fujaba’s UML metamodel. In order to start the matching process, an initial element has to be given as a parameter, in this case a class diagram, which is assigned to the variable `cld`. The link `hasInElements` and the `UMLClass` object `c` check for the presence of a class within the diagram. The guard attribute checks (by using Java code), whether the name of the class ends with “Singleton”. If this is the case, a new method `m` with the name “get()” is created and added to the class `c`.

In addition to graphical model transformation using story patterns, Fujaba’s state-ment activities can be used to describe transformations textually using Java code. This alternative can be used for transformations, which are not easily specified graphically. It also enables access to external Java libraries.

5.6 Integration into SPin

The previous section outlined, how story diagrams can be used to describe model transformations. In this section, the integration of this approach into SPin is de-scribed. First the handling of strata using project files is discussed, then the support for reconstructing class diagrams from Java classes using introspection. The section closes with a description of the tool support for editing and exporting transformation rules.

5.6.1 Strata and Project Files

SPin stores each stratum in a separate project file. Instead of a “deep-copy” algo-rithm, which copies the contents of a stratum to the subjacent¹³ stratum, the project file is copied on file-system level and after that the transformation is performed in place. This meets the “in-place semantics” requirement, stated in Section 5.3.

SPin automatically creates new project files as needed. The employed naming scheme also enables the insertion of new strata without the need for renaming files or central registries. Each project file name within a stratified architecture consists of the original project name, a high and a low bit-field and a version number. An example is shown in Figure 5.7.

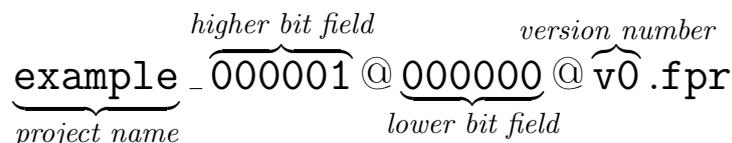


Figure 5.7: SPin’s project file name structure

The following rules are evaluated to determine the project’s file name during trans-formation:

1. If a new stratum b is added below the most concrete stratum a , the higher bit-field of a is increased by one and assigned to b .

¹³The adjacent, less abstract

2. If the stratum b has already been created before, only its version number is increased by one.
3. If a stratum b has to be inserted between strata a and c , the bit-fields of a are concatenated and added to the concatenated bit-fields of c . The sum is divided by 2 (by shifting the bit-field to the right) resulting in the new bit-field of b ¹⁴.

Using this algorithm, the strata can always be sorted by their concatenated bit-field. The implementation currently uses 6 bits for both the high and low bit-field providing room for at least $2^6 = 64$ strata. These stratum identifiers are also necessary to provide traceability support for stratification, which is explained in Section 5.8.

5.6.2 Metamodel synchronization

Fujaba does not offer a metamodel interchange format. Its ASG and UML metamodels are “hardcoded” into the application and any metamodel extension is based on Java classes, as well. These Fujaba specific classes can be generated automatically from class diagrams, which is useful when developing Fujaba plugins.

In order to allow automated type checking in story diagrams, the necessary metamodel elements have to be present in the current project. As the original class diagram may not be available, SPin uses a different mechanism to create the required metamodels. The responsible `VMSynchronizer`¹⁵ class employs the Java Reflection API to infer classes, attributes and associations from the metamodel classes exported by Fujaba.

Its functionality is useful during the modeling of applications, when reference classes (e.g. from the Java class library) are needed. Figure 5.8 illustrates this using an example. To the left, a class diagram is shown, which contains an interface from the Java API (`javax.swing.tree.TreeModel`)¹⁶ and the implementing class `MyTreeModel`. To implement the interface correctly, certain methods have to be added. Instead of checking the Java documentation, the developer chooses “sync with VM” from the context menu and the relevant methods are added automatically. The result can be seen on the right of Figure 5.8¹⁷

¹⁴Example: $a = 010@000, c = 011@000. a + c = 101000$. After shift right: $b = 010@100$.

¹⁵The name is derived from the Java Virtual Machine and the “synchronization” with the metamodel.

¹⁶The Fujaba-specific stereotype `<<reference>>` disables code generation for this class.

¹⁷After the synchronization, a separate function can be used to add unimplemented methods to the class `MyTreeModel`.

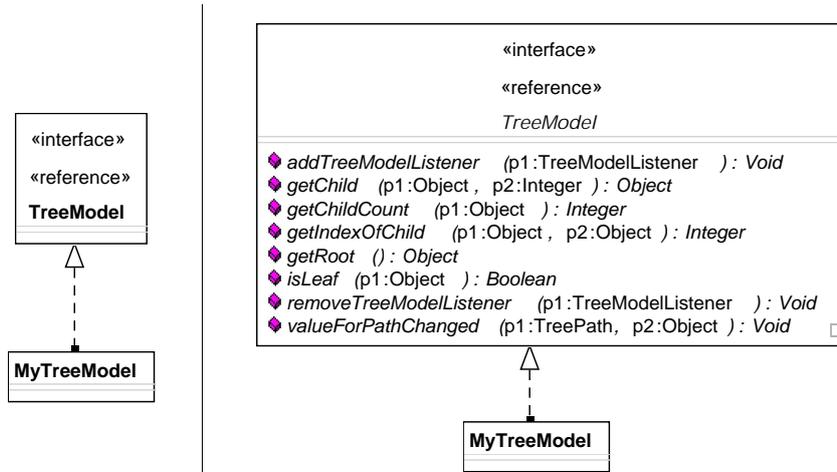


Figure 5.8: Example of the VMSynchronizer

SPin uses this technique to create a class diagram containing the Fujaba UML metamodel and the SPin refinement metamodel. After this, automatic type checking is available within the story patterns. This simplifies the creation of transformations and avoids typing problems.

5.6.3 Transformation Rules

The SPin plugin for Fujaba provides several utility functions to handle transformation rules. Rules are stored in Fujaba projects, and the rule creation is simplified by a setup wizard. After starting the wizard from the class diagram context menu, a name for the rule needs to be supplied. SPin creates the necessary metamodels (see above), a class diagram containing the transformation rule class and a story diagram for the method `apply(annotation: RAnnotation)` of that class (cf. Section 4.6). The diagram already contains the basic matching-pattern for the rule and three example activities. One for textual transformation using Java code, one for code generation using Futemplator (cf. Section 5.7) and a story pattern. The context menu provides additional options to add links and parameters to the matching-pattern. An example rule with one annotation link is shown in Figure 5.9. As the rule implements a concern, the concern description using the annotation is not needed anymore and is consequently removed from the model. This is denoted by the red “destroy” label.

The rule can be modified as needed and exported using a toolbar icon. The Java code is created, compiled and added to the rule library. It is available immediately for transformation. To test a transformation rule, an annotation with the necessary links and parameters has to be added to a new class diagram. If an XML file

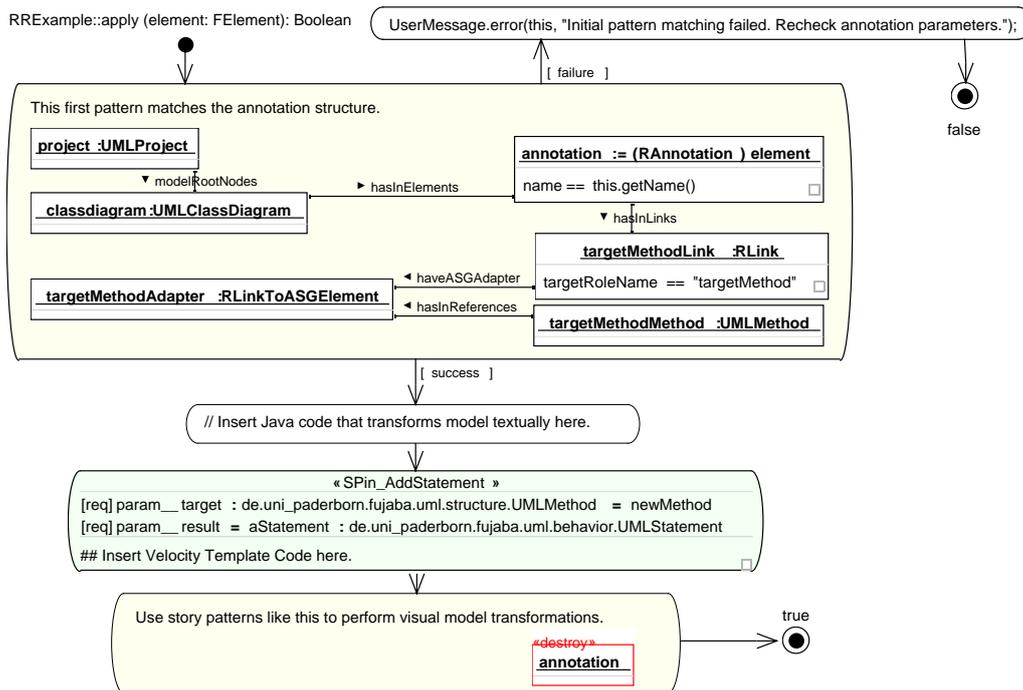


Figure 5.9: An example transformation rule with one annotation link

describing the rule metadata (cf. Section 4.7) is available, the link and parameter names and types can be filled in automatically. By choosing “refine” from the context menu of the annotation the transformation is executed. Debugging is currently limited to the source code level using standard Java IDEs (e.g. Eclipse).

5.7 Integrating Code Generation into Story Diagrams

In this section requirement 5 (“Integrated code generation facility”, cf. Section 5.3) is discussed. This section is based on the paper “Integrating Template based Code Generation into Graphical Model Transformation” [Gir08]. First the need for an integrated code generation facility is motivated by an example. Then criteria for a code generation language are discussed, followed by a description of the integration into the main transformation language. The section closes with the description of some language extensions and a discussion on performance issues.

5.7.1 Motivation for a Code Generation Facility

To model method behavior, Fujaba supports plain Java code blocks within activity diagrams. These statement activities also forms the basis of the hook spot concept,

described in Section 3.6. Generated code is separated from editable code by placing it in separate statement activities within the same activity diagram. This enables a clean separation of generated and hand-written code without the need of artificial code comments used by other round-trip capable modeling tools¹⁸.

The implementation of a concern not only involves the transformation of the model but also the generation of code blocks. The following example is related to the user interface of an application and implements—based on a given data model class—a Java Swing dialog, which displays appropriate edit fields for the attributes of the class.

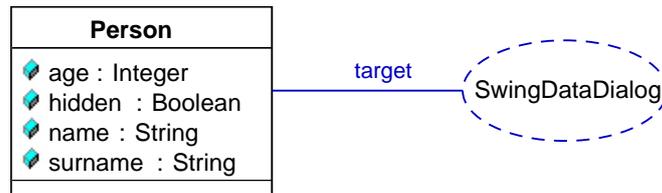


Figure 5.10: Swing Data Dialog: The annotated class diagram

The class diagram in Figure 5.10 shows the data model class “Person” and a “SwingDataDialog” annotation, which represents and parameterizes the concern. The annotation link “target” tells the annotation for which class a dialog has to be implemented.

The concern is implemented by executing the transformation associated with the annotation “SwingDataDialog” resulting in the diagram shown in Figure 5.11. On the right side the transformed class diagram, on the left the generated code of the created methods is shown. As can be seen, two classes, an association, a generalization and several methods along with their method bodies have been created. In order to describe this transformation, a language is needed, which supports the transformation of primarily graphical models and the ability to generate source code blocks.

As can be seen in this example, concern implementation primarily involves the creation of *new* code fragments, which are connected to methods within the model. If a method already contains code blocks, new blocks are inserted before or after one of the existing blocks. The transformation of existing code is not within the focus of this thesis. Possible applications are discussed in Section 7.2.

Figure 5.12 shows an excerpt of the story diagram describing the transformation. The first story pattern within the diagram matches the structure shown in Figure 5.10. When the transformation is initiated, it finds the annotation object within the class diagram and binds the variable “targetClass” to the destination of the

¹⁸These tools insert special markup code with “do not edit” comments into the code in order to prevent the user from editing this regions.

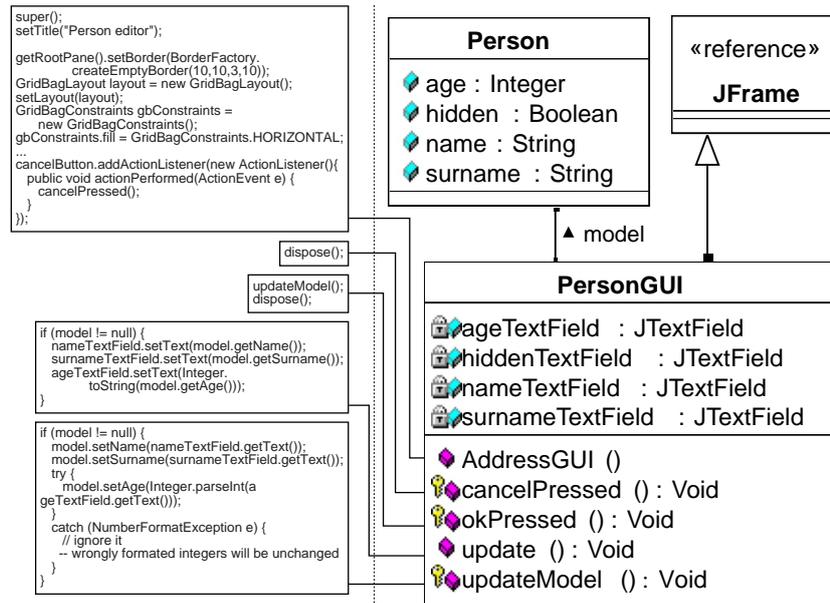


Figure 5.11: Swing Data Dialog: The result after the transformation

annotation link “target” (in this case the class “Person”)¹⁹. In the following, the state of a transformation rules—which contains all bound objects—is called the “context”. The next story pattern, marked with a double border (denoting a “for each” activity), searches for all attributes within “targetClass”. For each element found, it executes the third story pattern, which creates a new private attribute and attaches it to the class “gui”²⁰.

To fully implement the concern, code for the construction of the dialog and its behavior has to be generated as well. Although in principle possible the generation of these code fragments using story patterns is rather cumbersome. Thus, different means for code generation are needed. Two different approaches can be envisioned:

- The code generation is separated from the model transformation. This approach is taken by most model-driven development tools such as openArchitectureWare. A common technique with these tools is to apply several model transformation and use a final code generation steps to generate the executable code. Fujaba already supports this procedure with its CodeGen plugins.
- An alternative is the integration of a code generation facility into the main transformation language. The transformation tools DoME and VIATRA2—described in Section 5.4—follow this approach. This enables the direct transfer

¹⁹The intermediate object “targetAdapter” is needed for technical reasons.

²⁰The creation of the class “gui” is not shown in this diagram.

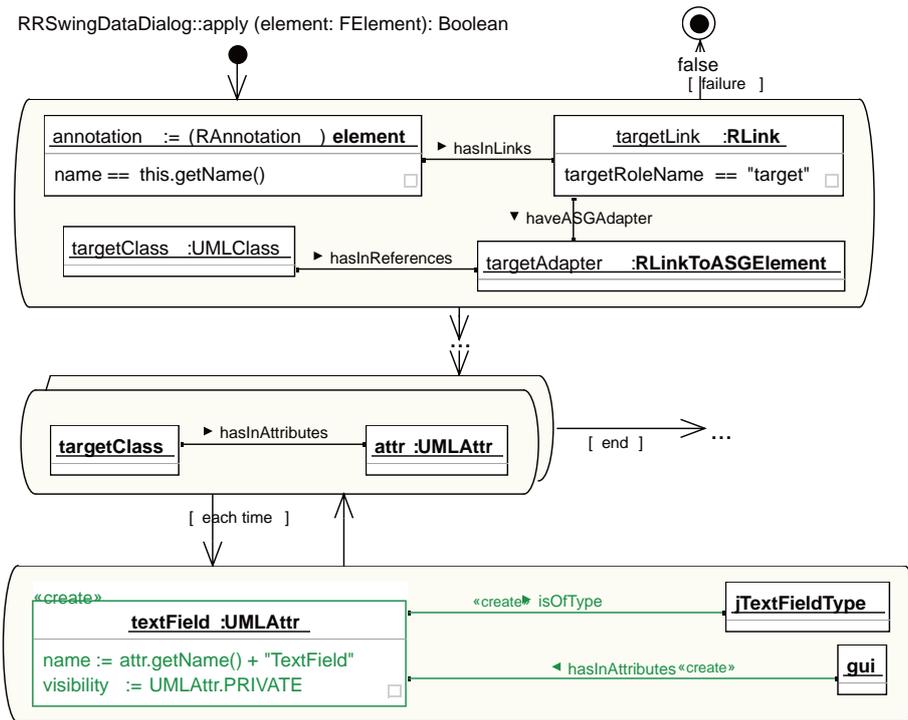


Figure 5.12: Swing Data Dialog: An excerpt of the transformation rule

of the transformation context between the model transformation and the code generation.

Although stratification also uses multiple model transformations, code generation is not only required in the last step. Each concern implementation transforms the model and also generates code. Hence, an integrated approach is preferred. The described context transfer removes the need to rematch any elements created during model transformation within the code generation in order to attach code fragments to them or obtain parameters from the annotation.

A simple solution for the integration of code generation into story diagrams is the use of statement activities, which contain Java code. Using this approach, the code has to be created "manually". Figure 5.13 shows a statement activity, which creates the code for the method `updateModel()`. The activity is simply included into the control flow of the main transformation shown in Figure 5.12.

This approach has several disadvantages: The multi-line string has to be concatenated manually, quotes and line ends have to be escaped and the mixture of the executing Java code and the Java code to be generated is confusing. In the following section the selection of a more suitable language is discussed. After that, the integration into story diagrams is presented.

```

String modelSetCode = "";
for (Iterator i = targetClass.iteratorOfAttrs(); i.hasNext(); attr = (UMLAttr) i.next())
{
    if (attr.getAttrType().getName().equals("Integer"))
    {
        modelSetCode += "try {\n model.set" + attr.getName() +
            "(Integer.parseInt(" + attr.getName() + "TextField.getText()));\n"
            + " } \n catch (NumberFormatException e) {\n" +
            " // ignore it -- wrongly formatted integers will be unchanged\n";
    }
    else if (attr.getAttrType().getName().equals("String"))
    {
        modelSetCode += "model.set" + attr.getName() + "TextField.getText();\n";
    }
}

UMLStatementActivity updateModelMethodBody =
    ActivityDiagramHelper.insertStatementAtStartOfMethod
    (updateModelMethod, "updateModelMethodBody", modelSetCode);

```

Figure 5.13: A statement activity responsible for code generation

5.7.2 Criteria for a Code Generation Language

As argued in the previous section, concern implementation mainly requires code *generation* and not *transformation*. Transformation of existing code would require tokenization and a more detailed representation of the code (e.g. by using abstract syntax trees) and is out-of-scope for this thesis. As a consequence, the textual representation is sufficient for stratification. Concern implementations can be seen as codified best practices and patterns. The created code fragments can thus be taken from previous implementations and only minor changes are required to adapt this code to the situation at hand.

Template based approaches are ideally suited for this situation as they produce parameterized text blocks. This is accomplished by mixing the output text with control elements, which are evaluated during runtime and insert computed results into the output.

Depending on the control elements needed and the type of the output text (structured XML, unstructured plain text, source code, etc.) different approaches exist. In his bachelor thesis Bausch evaluated existing template languages according to the following criteria [Bau07]:

Integrability Availability of an API to control the template execution from Java in order to incorporate it in the transformation rule. Easily integrable into Fujaba.

Context Transfer Access to Java runtime objects (either direct or via introspection). The parameters of the graphical transformation rule should be easily transferable to the template. The template output needs to be transferred back to the rule.

Control Elements Support for if/else constructs, loops and macros.

Syntax Simple markup model²¹. Suitable for Java code generation.

The evaluation started with a collection of 19 languages, presented by the web site java-source.net²². Some languages were discarded, as their XML based syntax is unsuitable for code generation or the template engine was not available for download anymore. The popular engines JET and Xpand2 were added to the list. Table Figure 5.14 gives an overview of the evaluated languages.

A full evaluation revealed, that JET and Xpand2 were not suitable for integration into fujaba, because of their tight coupling with Eclipse. The remaining languages are primarily geared towards generation of web pages. One exception is Velocity, which already proved its capabilities for Java code generation, for instance in Fujaba itself, where the plugin CodeGen2 [GSR05] uses Velocity to generate the executable code for Fujaba models. This existing implementation simplified the integration into Fujaba, hence Velocity was chosen.

Velocity is an interpreted template language. Markup is needed only for variables and control elements, the remaining text is copied verbatim to the output including line breaks. Listing 5.1 shows the Velocity template creating the same code as the statement activity in Figure 5.13.

```

1  if (model != null) {
2  #foreach( $attr in $targetClass.iteratorOfAttrs() )
3  #if( $attr.AttrType.Name == "Integer" )
4      try {
5          model.set${attr.Name.substring(0,1).toUpperCase()}${attr.Name.substring(1)}(
              Integer.parseInt("${attr.Name}TextField.getText()));
6          }
7          catch (NumberFormatException e) {
8              // ignore it — wrongly formatted integers will be unchanged
9          }
10 #elseif( $attr.AttrType.Name == "String" )
11     model.set${attr.Name.substring(0,1).toUpperCase()}${attr.Name.substring(1)}(${
              attr.Name}TextField.getText());
12 #end
13 #end
14 }
```

Listing 5.1: The Velocity template equivalent to Figure 5.13

Control elements start with a #, variable references with a \$. Attribute values are determined by using Java introspection. Calls to Java methods (including side effects) are possible, as well. Compared to Figure 5.13 the template provides cleaner separation of fixed and variable text.

²¹Template languages employ markup characters, which separate control elements from remaining text.

²²<http://java-source.net/open-source/template-engines> (checked August 2009)

Template Language URL	Integrability	Context Transfer	Control Elements	Syntax	Note
Velocity http://velocity.apache.org/	++	++	++	++	direct access to the Java VM via reflection
FreeMarker http://www.freemarker.org/	+	++	++	0	focus on HTML and XML generation
WebMacro http://sourceforge.net/projects/webmacro	0	++	+	0	based on servlets, reflection available
Jamon http://www.jamon.org/	+	++	++	0	template is compiled to java code
Tea http://teatrove.sourceforge.net/	+	+	++	+	compiled to java code but separate VM, focus on web pages
jxp http://jxp.sourceforge.net/	0	+	+	+	similar to java server pages, reflective access possible but template I/O only via file system
BTE http://ostermiller.org/bte/		0	+	0	no direct context transfer possible
JDynamiTe http://dynamite.sourceforge.net/	+	0	0	0	no direct context transfer possible, no control structures
String/Template http://www.stringtemplate.org/	+	0	0	+	no direct context transfer possible
JET http://www.eclipse.org/modeling/m2t/?project=jet	0	++	+	+	integrated with eclipse
Xpand2 http://www.openarchitectureware.org/	0	+	++	++	integrated with eclipse

Figure 5.14: Comparison of templates languages

5.7.3 Integrating Templates into Story Diagrams

Similar to statement activities, templates are integrated into story diagrams as a new activity type. The Fujaba plugin “Futemplerator” (**Fujaba Template generator**)—developed as part of a bachelor thesis [Bau07]—extends the Fujaba metamodel for story diagrams with a new template activity and adds the visualization and code generation to it.

In order to execute transformation rules, the associated story diagram is transformed into executable Java code. This step is performed by the Fujaba plugin CodeGen2 [GSR05]. Futemplerator provides the necessary files for CodeGen2 to enable the transformation from template activities to Java code.

In addition to the Velocity template, the new activity supports the specification of template parameters. Figure 5.15 shows such a template activity. The upper compartment contains a popup menu where a stereotype can be selected. This template stereotype describes further processing of the output generated by the template. It also defines a set of template parameters, which are displayed in the middle compartment.

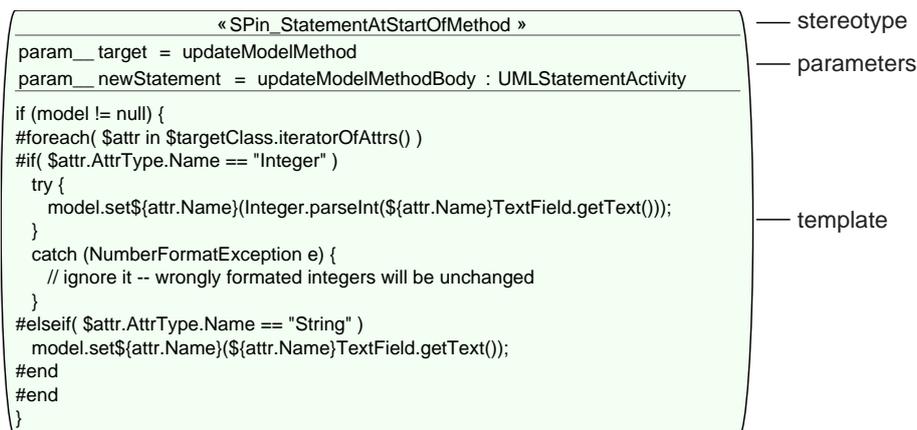


Figure 5.15: The template activity equivalent to Figure 5.13

The template stereotype `SPin_StatementAtStartOfMethod` for example inserts a new code block at the beginning of the method referred to by `param__target`²³. The new code block is assigned to a local variable whose name is defined by `param__newStatement`. This enables further use of the created code block in the remaining transformation rule. For instance the stereotype `SPin_StatementAfterCodeBlock` inserts a statement after an existing block. In order to reference this block, a reference to the aforementioned code block is needed.

Futemplerator already provides several stereotypes for different purposes, more defi-

²³The prefix "param__" was chosen to prevent name clashing with other variables.

nitions can be added easily. Information on the extensible stereotype concept follows in Section 5.7.5.

5.7.4 Context Transfer between Model and Template

In comparison to languages, which separate model transformation from code generation by providing two *separate* languages, the integrated approach—described in the previous section—enables co-evolution of model and code. An important aspect of this integration is the ability to transfer the context of the transformation (e.g. model elements bound or created during the transformation) to the template engine and attach the created code fragments to the model. As the graphical model transformation is executed by the Java Virtual Machine (JVM), a mechanism is needed, which transfers the context or state of the JVM to the Velocity Template Engine (VTE). The VTE evaluates the template and returns the created code to the JVM, which integrates it into the model.

Velocity is an interpreted language and is executed separately with no direct access to the JVM. Hence, the context of the graphical transformation has to be fed manually into the the VTE context. In addition, the parameters defined in the middle compartment of the template activity are transferred.

Referring back to the example in the previous section consider the middle story pattern within the transformation rule in Figure 5.12 on Page 98. The matched attribute `attr` is reused in the template activity in Figure 5.15 on Page 102 to determine the type and the name of the attribute.

The graphical transformation description is translated to executable Java code using the CodeGen2 plugin. The plugin ensures, that all objects of the story diagram have a local variable name and are added to a list. When a template activity is reached, the variables of the list are copied to the VTE context. After the template evaluation, the variables are retransferred back to the JVM. The latter step allows back-propagation of template evaluation side effects (e.g. creation of new elements).

This process is visualized in Figure 5.16. It shows the behavior of the code, which is created for a template activity. This code is inserted into the executable transformation rule²⁴. The diagram is divided into three “swim lanes” for data residing in the file system, the Java Virtual Machine and the Velocity Template Engine. The arrows denote control and data flow within the diagram, the rectangular pins on the left side of an activity show “data-in” and “-out” ports. In addition to the UML 2 notation, activities have a step counter in the upper left corner.

In the first step, macro definitions are taken from the file system and prepended to the template code from the template activity. The use of macros is discussed in

²⁴This is accomplished by adding an additional code template to CodeGen2.

Section 5.7.6. After the initialization of the VTE in step 2 its context is populated with local variables and template activity parameters. It is followed in step 4 by the template evaluation and then the retransferral of the context back to the JVM. The result of the evaluation is processed by template stereotype definitions, which are described in the next section.

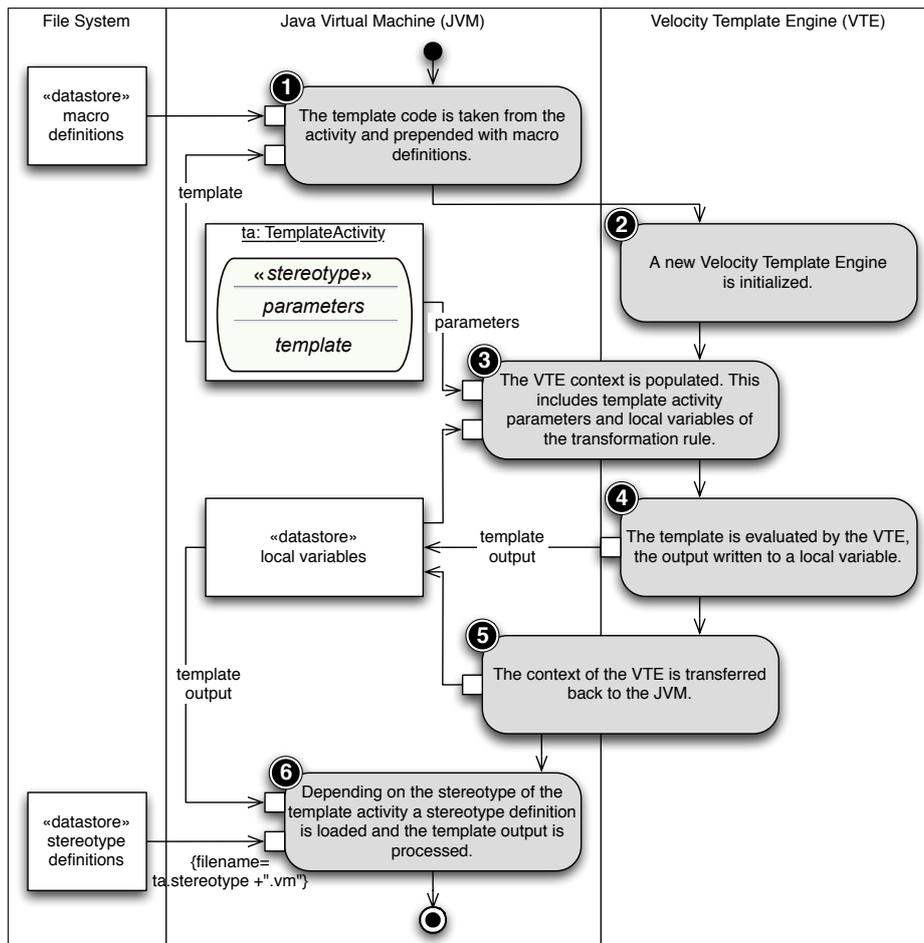


Figure 5.16: Control and data flow within a template activity [Gir08]

5.7.5 Template Stereotypes

The example in Section 5.7.1 already illustrated the use of template stereotypes. Their application is not limited to code generation, they can also be used for the generation of other artifacts (e.g. build scripts or configuration files). As the list of template stereotypes is extensible, additional functionalities can be implemented easily.

```

1 #set( $result = $imports.addToImports("java.io.FileWriter") )
2   File futemplator__filewriter = new FileWriter("${param__filename}");
3   futemplator__filewriter.write(futemplator__templateOutput);
4   futemplator__filewriter.close();

```

Listing 5.2: Stereotype template: Filewriter.vm

Each stereotype is defined by two files: A Velocity template and an XML parameter definition file. The list of available stereotypes can be extended by providing a simple Fujaba plugin containing these files and a few lines of code, which attach an additional search path to Futemplator. The new stereotypes appear in the popup folder at the top of each template activity. During the creation of the transformation rule the template associated with the selected stereotype is parsed (step 6 in Figure 5.16). Listing 5.2 shows an example stereotype template, which writes the output from step 4 to the file specified by `param__filename`.

When the stereotype template shown in Listing 5.2 is *parsed*, `param__filename` is replaced by the parameter value and the result is inserted into the model transformation code. During *execution* of the model transformation, this code instantiates a `FileWriter` and writes the output (contained in the variable `futemplator__templateOutput`) to the file.

A stereotype template is supplemented by an XML file describing the mandatory and optional parameter names and types along with a help text displayed as a tool tip within the template activity. When the stereotype is selected, the parameters are added automatically to the template activity. Listing 5.3 shows the parameter definition file for the template from Listing 5.2.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <fpd:params xmlns:fpd="http://futemplator.sf.net/..." >
3   <fpd:param name = "filename"
4             type = "java.lang.String"
5             default = "&quot;output.txt&quot;"
6             description = "String specifying the name of the output file." />
7 </fpd:params>

```

Listing 5.3: Stereotype parameter definition file: Filewriter.params

The implementation of the stereotype concept is very flexible and allows other applications outside the actual code or text generation. For instance the stereotype “UserMessage” can be used to create messages during transformation. These messages are linked to model elements and can remind the developer to finish parameterization of created objects (e.g. fill in annotation parameters or add manual written code to hook spots).

5.7.6 Template Macro Definitions

Step 1 in Figure 5.16 mentions the inclusion of macros. Velocity supports the definition of macros, which can be used similar to functions. Futemplerator automatically includes a set of global macros and macros specific to certain stereotypes in the template code. For instance, the stereotype templates for SPin provide macros, which ease the development of transformation rules. As an example, the macro in Listing 5.4 helps to add import statements to a class. Macros are called from templates by prepending their name with a # (e.g. #addToImports("java.util.Iterator")). The example also shows additional uses of Velocity in addition to plain code generation. Its ability to call Java methods from a template is used to add the import entries to a model class.

```
#macro( addToImports $text )
#set( $result = $umlFactory.addToImportedClasses( ${param_target}, $!text )
)
#end
```

Listing 5.4: Macro definition: addToImports.vm

5.7.7 Measuring Transformation Performance

The previous sections presented two alternatives for code generation: compilable statement activities (shown in Figure 5.13) and interpreted template activities (Listing 5.1). Bausch [Bau07] compared both alternatives in a synthetic benchmark, which measured the execution time of n activities within a story diagram. The overhead generated by the Velocity engine resulted—for very large n —in a rather poor performance of the template activities.

In order to verify the feasibility of template activities for model transformation a more realistic comparison was undertaken. Two transformation rules, which create n methods and add them to an existing class were built. One uses statement activities and the other template activities to create the code added to the methods.

Figure 5.17 shows the execution times for each activity type, when creating n methods on a Core 2 Duo 2.33 GHz machine. The time includes the complete transformation process (loading the transformation rule, copying the stratum, transforming the model).

A single transformation rule seldom contains more than a dozen template activities. Even when considering that several transformations have to be re-executed when the developer switches to a lower stratum, the total count of code generation does not exceed 1000. Thus the chosen sample sizes are sufficient. Within the complete transformation process, the delay resulting from the use of interpreted template activities

n	100	200	500
statement activity	1,49 sec	5,18 sec	35,11 sec
template activity	2,32 sec	5,80 sec	37,42 sec

Figure 5.17: Execution time of statement activity vs. template activity

is negligible. Bausch [Bau07] also conducted an analysis of memory consumption, which did not reveal any specific differences between the two approaches.

5.8 Preservation of User Edits

The last remaining requirement from Section 5.3 (support for the preservation of user edits) is discussed in this section. One way to achieve it, is the use of target incremental transformation, which update an existing model without destroying any manual modifications.

The existing problems with target incremental transformations are analyzed first. This leads to the use of traceability concepts in model transformation, which are discussed in Section 5.8.2. Here existing solutions from other transformation languages are presented. After detailing the special properties for stratification in Section 5.8.3, the implemented solution is described in Sections 5.8.4 to 5.8.6. Related work is discussed in Section 5.8.7.

5.8.1 Target Incremental Transformations

As already stated in the summary of Section 5.4, target incrementality prohibits the prerequisite for “in-place semantics”. If a model is modified in-place, the source information is destroyed, making incremental transformations impossible. SPin resolves this by duplicating the project file. The source information is retained, but with this solution each subsequent transformation duplicates the project anew, which in turn prevents target incrementality.

The table in Figure 5.18 lists the different model modifications and how target incremental transformations have to deal with them. It has to be noted, that any combination of these modifications is possible as well. When modifications are only applied to the source model (type SM) or the transformation rule (type TR), the target model can be created from scratch. If the target model is modified as well (type TM), incremental transformations are necessary. Type TR is not considered by current transformation engines, but is highly relevant for model-driven approaches, where updated transformations are required to deal with new framework revisions of the destination platform.

type	applied modification	transformation
SM	source model	update target model
TR	transformation rule	update target model
TM	target model	detect and resolve conflicts

Figure 5.18: Table of target incrementality requiring scenarios

For type TM two scenarios have to be taken into account. The first is *model synchronization*, where the transformations are usually applied in both directions and any changes are propagated to the other model. This often implies, that both models are on the same abstraction level and the transformation can thus be classified as horizontal according to the scheme described in Section 5.2. The second scenario entails *stepwise refinement* with vertical refinement transformations. Here, the modifications in the target model are not transformed back to the source model (cf. Section 5.3).

Independent of the scenario the changes applied to the target model fall into two categories. The first category contains *modifications* of generated elements. Here, conflicts during incremental transformations are to be expected and they are usually handled by defining a dominant model. If the source is dominant, the transformation overwrites the modifications. If the target is dominant, the modifications are retained. Both variants may lead to inconsistent models.

Changes from the second category are *model amendments*. If these new model elements are part-of or related-to generated model elements, they may also be affected by incremental transformations. For instance when the transformation removes or renames elements in the target model.

As argued in Section 3.6, stratification supports “controlled editing” of strata. This prohibits—with the exception of hook spots—the editing of generated elements and only allows the addition of new elements. Thus, changes in the target model only belong to the second category.

The main challenge in implementing incremental transformations is to detect the changes in the source model and to update the right target elements accordingly. In order to find the right elements, some kind of connection between source and target has to exist. This intermodel relationship is often called traceability. The following section describes this concept and its application to model transformation.

5.8.2 Traceability and Model Transformation

Traceability is often related to requirement traceability, which—according to Gotel and Finkelstein [GF94]—“...refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction.” Applied to software

artifacts in general, traceability helps to relate elements within and between these artifacts. A discussion of the different applications of traceability, associated problems and implementation techniques can be found in “Model traceability” [ARNRSG06].

In the context of model transformation, traceability information connects the source with the target model. The granularity of this information can range from coarse (rule level, e.g. which transformation was applied) to fine (element level, e.g. which source element is related to which target element). This information can be stored permanently or only during rule application. In order to implement target incremental transformations, fine grained, permanent relationships are required.

The trace information can be represented in different ways. An overview is given by Espinoza et al. [EAG06] and Aizenbud-Reshef et al. [ARNRSG06]. The transformation languages, described in Section 5.4 employ different techniques to create this data. The automatic generation of traceability information is only available in declarative languages such as QVT, TGG and BOTL. In other languages it is necessary to create the traceability information manually using the transformation language.

Only a few transformation languages currently support incremental transformations. VIATRA2 [BV06] enables incremental updates using live transformation. A similar approach—with a planned implementation for Fujaba—is described by Varró [Var04].

Triple Graph Grammars [Sch95] use a separate graph to store traceability information, which enables bidirectional incremental transformation. The QVT/relations implementation from IKV++ [KE07] supports the QVT feature “incremental update”²⁵.

Within stratification, incremental transformation is primarily needed to support preservation of user edits. This goal is also targeted by other languages. Tratt proposes PMT²⁶ [Tra06] as a “change propagating” language, which is able to deal with modifications of the target model.

BOTL [BM03] and M2ToS²⁷ [Rei05] simulate incremental transformation by *merging* the transformation result with the target model. In order to identify elements from the result with elements from the target model they rely on key attributes. These attributes are typically defined in the metamodel. If their values are equal, the

²⁵As defined in the QVT specification [OMG07a]: “Once a relationship (a set of trace instances) has been established between models by executing a transformation, small changes to a source model may be propagated to a target model by re-executing the transformation in the context of the trace, causing only the relevant target model elements to be changed, without modifying the rest of the model.”

²⁶“Propagating Model Transformation language”

²⁷german: “Modell-zu-Modell-TransfOrmations-Sprache” (model-to-model transformation language)

element from the transformation result is merged with the target model element. The main drawback of these approaches lies within the selection of key attributes. If—for instance—element names are chosen, the renaming of elements produces duplicate elements.

5.8.3 Stratification Traceability

The definition of stratification traceability adheres to the more general notion to “follow the life of a requirement” [GF94]. As defined in Section 2.1 a concern is a reification of one or more requirements. Each stratum introduces a concern implementation, which is described by one or more annotations. The implementation manifests itself in the generation of model elements, it can thus be said, that the annotation “induces” model elements. Hence, this relationship is called “inducement”. The following cases for “inducement” can be distinguished:

concern → **annotations** A stratum is related to a concern, the concern is described by annotations.

concern → **concerns** Concern implementations may “reveal” sub-concerns, which are added as annotations.

annotation → **generated model elements** Model elements created by a model transformation are induced by the annotation, which triggered the model transformation.

annotation → **hook spots** Hook spots are created by annotations and are thus induced by them.

annotation → **other model elements** New model elements are always part of a specific system concern. In order to integrate them into the architecture, they are linked from and therefore induced by an annotation (cf. Section 3.6).

In consequence all elements of a stratified architecture are induced by concerns. Adding this information to the architecture enables traceability from concern to implementation artefacts. This traceability also enables incremental transformation and preservation of user edits. With it the model transformation is able to propagate changes correctly to the next stratum without destroying the additional information on it.

5.8.4 Enabling Incremental Transformation in Stratification

With the special properties described in the previous section, target incremental transformations and traceability can be achieved. The technical solution is de-

scribed in this section. In Section 5.8.2 two approaches for the realization of target incremental transformations have been described. The first is model synchronization using incremental model transformation. All transformations respect the established information between source and target and only propagate the changes. The second approach is model transformation combined with model merging. The transformation is always executed completely, the target is created from scratch and is merged with the previous target from the previous transformation run.

The available inducement information, described in the previous section, and the restriction to controlled editing, described in Section 3.6, enables a combined approach: The established traceability information can be used to perform a *controlled merge* of a fully executed transformation with new model elements of the subjacent stratum.

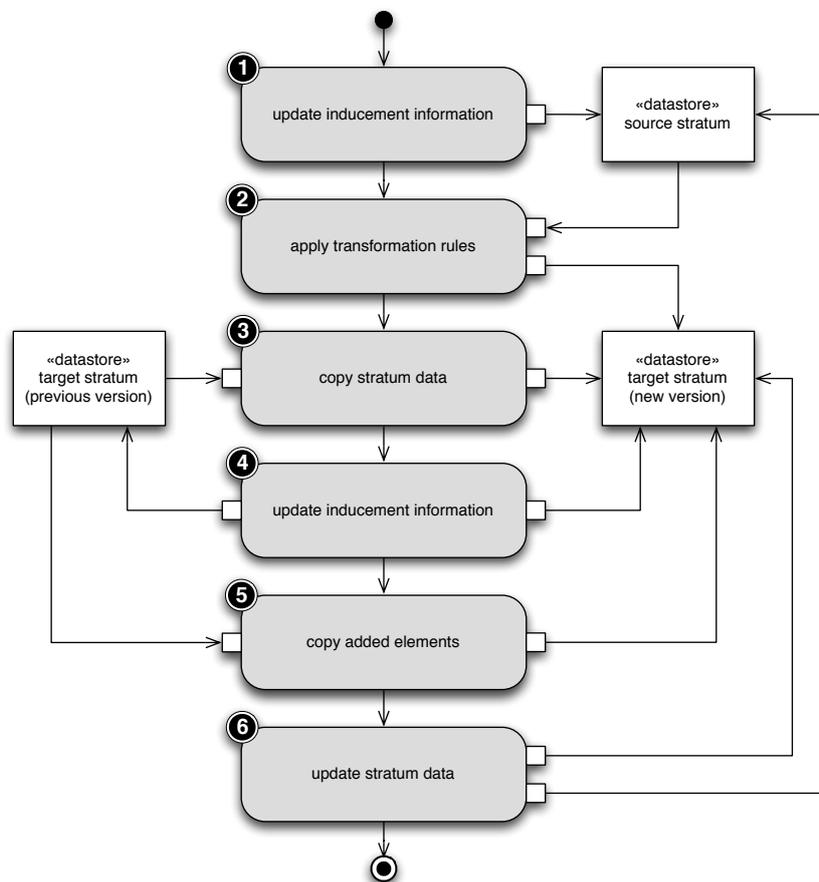


Figure 5.19: Control and data flow during a transformation with controlled merge

Figure 5.19 illustrates the control and data flow during a transformation with controlled merge. The notation is similar to Figure 5.16 on Page 104. In step 1 the inducement information in the source stratum is updated. This ensures, that all

elements within the stratum contain valid inducement relationships. Step 2 then executes the in-place transformation and saves the result as a new project file—the new target stratum. Stratum data (e.g. the name and description of the stratum) is copied from the previous version in step 3. In order to notice any manual modifications applied to the previous target stratum and the modifications executed by the current transformation, the inducement information on both the previous and the new target stratum is updated in step 4. With the help of the inducement relationships any elements manually added to the previous target stratum are copied to the new version (step 5). Finally, in step 6, the stratum data is updated, which ensures that the source and the new target are linked together.

5.8.5 Stratification Identifiers

Instead of representing the inducement relationships by using an additional persistent intra- and intermodel data structure, a less coupled approach was taken. To each model element a unique identifier is attached, which contains the identifiers of “inducing” elements.

As described in Section 5.8.3 the creation of each element can be traced back to a concern. By assigning a unique identifier to each model element, this trace can be established. This “stratification identifier” (SID) needs to have the following properties:

1. The SID is unique within the entire stratified architecture.
2. An element appearing on several strata has always the same SID.
3. The SID of an element never changes.
4. Any references to other elements are represented using their SID.

The first three properties are mandatory in order to support the fourth. Instead of an additional data structure, the direct representation of relationships using SIDs simplifies data keeping. The second and third property can be achieved by computing and assigning the SID when an element is first created.

Some transformation languages (e.g. M2ToS) rely on key attributes to identify elements. However, simple choices like the name of an element are not sufficient, because changing the name attribute would violate property 3. A different approach is a global ID registry, which assigns unique identifiers to each new object. Most modeling environments follow this approach to identify elements internally.

Two problems arise, when this technique is applied to the stratification approach described in this dissertation. Firstly, the registry needs to be able to assign the

same ID to an element, which exists on two strata (in order to achieve property 2). Secondly, because strata are recreated from scratch during transformation, the registry needs to assign the same ID to a recreated object thus making the ID “reproducible”.

These problems are solved by the combination of the SID data structure with an adapted transformation process. An SID consists of a type, a “global” and a “local” part. An overview of these elements is given in Figure 5.20. As can be seen, two cases are considered. On the left SIDs for generated elements, on the right SIDs for manually added elements are described.

type	generated		manually added	
	without hook spots	with hook spots	linked from annotation	placed within diagram
global part	stratum identifier		-	
	SID of annotation, which triggered transformation		SID of linked annotation	-
local part ²⁸	RID (set by transformation rule)		link name	parent + LID

Figure 5.20: Parts of a stratification identifier (SID)

SID: Type

The type is used to differentiate between generated elements, manually added elements and hook spots. During transformation, generated elements are created automatically. Manually added elements and the contents of hook spots have to be copied from the previous version of the stratum after the transformation has finished.

SID: Global Part

The global part describes the inducement relationship with a concern, which is represented by a stratum using one or more annotations. The stratum is specified using the structure from Section 5.6.1, which was designed to achieve property 3. If the element was induced by an annotation, the SID of this annotation also belongs to the global part. This applies to two situations: Either the element was *generated* by the transformation rule of the inducing annotation or the element was *manually added* and linked to an annotation.

²⁸RID: refinement identifier, LID: local identifier

SID: Local Part

The local part identifies the induced element within a set of elements. As argued in Section 3.6, added elements are always related to a concern and therefore connected to an annotation using an annotation link. However, this connection may be indirect (e.g. a class with attributes, where only the class is connected to an annotation). In this case, the local part consists of the SID of the parent object (e.g. the class containing the attributes) and a unique “local identifier”. This identifier—also called LID—can be computed and reproduced easily due to the locality of the name space (e.g. only elements within the class are affected). For links between elements (e.g. associations) the SID of the linked elements are used.

For elements generated by a transformation rule the local part is set by the rule during the refinement transformation. This “refinement identifier” is also called RID. How the transformation rule achieves the necessary properties “reproducibility” and “uniqueness” is described in the following section.

5.8.6 Integration into the Transformation Process

Stratification identifiers are automatically assigned after the transformation process. They rely on the refinement identifiers created by the transformation rule. The following example illustrates this process.

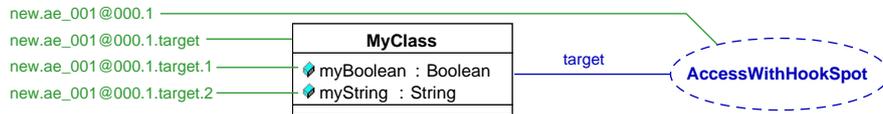


Figure 5.21: Implementing access methods: The topmost stratum

Figure 5.21 shows a class with two attributes annotated with “AccessWithHookSpot”. Before the transformation, SIDs are assigned to all manually added elements. They are shown in green on the left side of Figure 5.21. The first part is the type, “new” denotes manually added elements. The global part consists of the stratum identifier (here `ae_001@000`)²⁹. The local part for the annotation is a LID (e.g. 1). As the initial stratum is never rebuilt and any assigned SIDs are retained in the project file, using consecutive numbers is sufficient here. The class `myClass` is linked by the annotation, thus the SID of the annotation is added to the global part. The local part consists of the link name (here `target`). The attributes are indirectly linked and thus their local part consists of the parent element `myClass` and LIDs (e.g. 1 and 2).

²⁹ae is the project name (`access example`), the high and low field is shortened to 3 bits to simplify the presentation of SIDs.

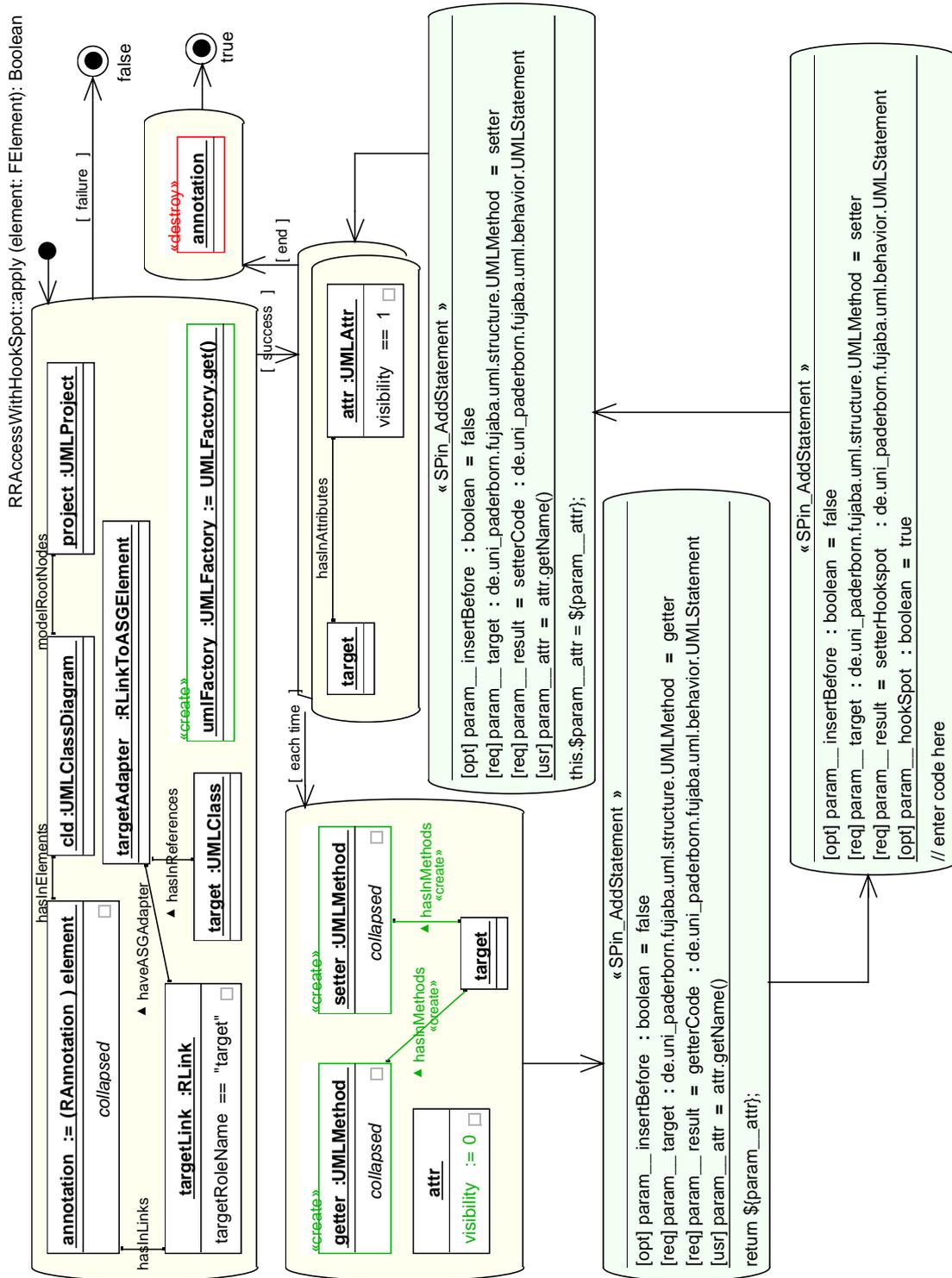


Figure 5.22: Implementing access methods: The transformation rule

The corresponding transformation rule for the annotation is shown in Figure 5.22. It iterates over all attributes found in the linked class, creates a getter and setter method and generates code for both methods. In order to execute custom code before the value is set within the setter method a hook spot is inserted (cf. template activity in the lower right of Figure 5.22).

The subjacent stratum revealing the generated access methods is shown on the left side of Figure 5.23. The activity diagram of the generated `setMyString` method is shown to the right.

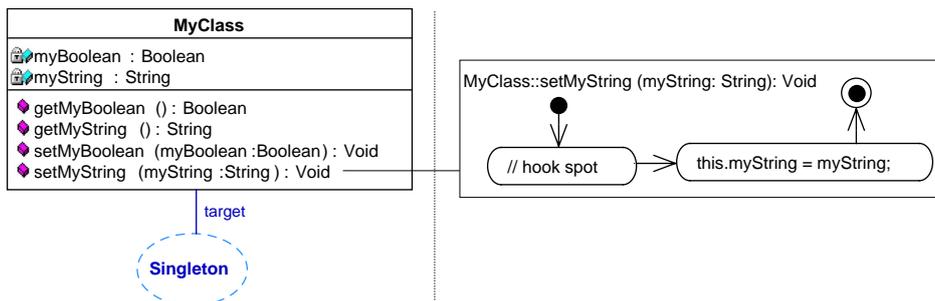


Figure 5.23: Implementing access methods: Second stratum and generated method

To compute the RIDs for these methods two simple solutions can be envisioned:

1. Enumerating the methods (e.g. 1, 2, 3, 4)
2. Using the name of the methods (e.g. `getMyBoolean`, `getMyString`, ...)

The first variant fails when the creation order of elements changes or elements are added or removed. The second variant fails, if attributes are renamed. Both issues can be resolved by using identifiers from the transformation rule itself. In Listing 5.5 an excerpt from the executable transformation code is shown. This code is generated by Fujaba from the story diagram shown in Figure 5.22. All objects within a story diagram have a name, which is used for the local variable within the generated code. As can be seen in line 23, the created `set` method is assigned to a local variable named `setter`, which is defined in line 2 of the generated code. The name of the local variable is reproducible during subsequent transformations thus satisfying the first requirement.

The loop from line 8 to 33 is executed for all attributes and thus each newly created `set` method is assigned to the same local variable `setter`, which prevents uniqueness. The solution is already shown in Listing 5.5. In line 5 an `RIDGenerator` is initialized with a stack. When a loop is entered, the iterated element is pushed on the stack (cf. line 15). After a new object is created, an RID is assigned automatically (line 24). Here the SIDs of the elements from the stack are added to the newly created

RID, which ensures both reproducibility and uniqueness. The automatic assignment of RIDs for story pattern and template activity stereotypes has been integrated into the code generation process for story diagrams by adapting CodeGen2 velocity templates.

```

1 public boolean apply (FElement element ) throws Exception , Error {
2     UMLMethod setter = null;
3     ...
4     java.util.Stack<FElement> spin__ridPrefixes = new java.util.Stack<FElement>();
5     RIDGenerator spin__ridGen = new RIDGenerator((RAnnotation)element ,
6         spin__ridPrefixes);
7     ...
8     fujaba__IterTargetToAttr = target.iteratorOfAttrs ();
9     while ( fujaba__IterTargetToAttr.hasNext () )
10    {
11        try
12        {
13            attr = (UMLAttr) fujaba__IterTargetToAttr.next ();
14            // check object attr is really bound
15            JavaSDM.ensure ( attr != null );
16            spin__ridPrefixes.push(attr); // RID support
17            // attribute condition visibility == 1
18            JavaSDM.ensure ( attr.getVisibility () == 1 );
19            // story pattern
20            try
21            {
22                fujaba__Success = false;
23                ...
24                setter = project.getFromFactories( UMLMethod.class ).create ();
25                spin__ridGen.generateRID(setter , "setter", false); // generate RID
26                ...
27                // create link
28                target.addToMethods (setter);
29                fujaba__Success = true;
30            }
31            ...
32        }
33        spin__ridPrefixes.pop(); // RID support
34    }
35    ...

```

Listing 5.5: Excerpt of Java code for the transformation rule from Figure 5.22

In the example the local part for the method `getMyBoolean` consists of the SID of `myBoolean` and the local variable name `getter`. A readable representation of the SID is shown in Figure 5.24.

The SID of the hook spot differs in the type (`modifiable`) and the local variable name, which is `setterHookspot`. After retransformation, the contents of the hook spot are available under that SID in the previous version of the stratum and can be copied to the current version retaining any manual code changes.

The SID assignment process and the im- and export of manually added elements has been implemented as part of the diploma thesis of Haha [Hah08].

The presented naming concept and its implementation enables the reproducible and

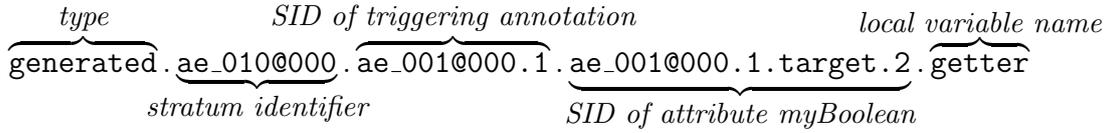


Figure 5.24: Implementing access methods: SID of the method `getMyBoolean`

unique creation of identifiers during the transformation. By linking manually added elements to generated elements, a traceability relationship is built, which in turn allows the reintegration of these manually added elements after retransformation.

5.8.7 Related Work

Although already a number of transformation languages with traceability support exist, none of them supports both traceability between and within models. Triple Graph Grammars [Sch95] use a middle graph to permanently link models. ATL [Jou05] and QVT [OMG07a] use transformation traces to store information. There are similarities of the local and global naming concept for stratification identifiers to traceability “in the large” and “in the small” as considered by the Global Model Management [BFB07] approach from the ATLAS group. An overview on concepts for traceability in the MDS context is given by Espinoza et al. [EAG06]. Vanhoff et al. [VBJB07] discuss a different approach to extract traceability information from model transformations.

Tratt [Tra06] proposes a “change propagation” language, which connects source to target elements using computed identifiers. These identifiers are based on a unification of participating source elements with further input from the transformation rule. During the transformation, the identifier is used to find the target object and propagate any changes to it. This is different from SPin’s approach, where the target model is always created from scratch. Albeit similarities in the naming concept exist, the underlying transformation languages are different and SPin also considers loops and dependencies within the model.

5.9 Summary

This chapter described a model transformation language suitable for architecture stratification. After outlining two classification schemes, requirements on a transformation language have been compiled. An analysis of existing languages showed, that “story driven modelling” (SDM) satisfied most of these requirements. SDM uses Java to describe attribute computations and the integration with the anno-

tation concept and the transformation rule library allows flexible parameterization. The available control structure elements enable the definition of transformation rules in an “imperative” manner. The notation is based on a graphical abstract syntax in the Fujaba metamodel. By using abstract syntax, it can easily be applied to other modeling languages and offers intuitive inclusion of attribute computation.

SPin’s integration of story driven modeling as a concern implementation technique also enabled the necessary in-place semantics and the support for preservation of user edits. By enhancing SDM with a template based code-generation language, now not only the model can be transformed but also code can be generated.

Chapter 6

Case Study

In this chapter, the concepts of architecture stratification are evaluated using a case study. An existing application was chosen, which contains both high-level concerns¹ (like the integration of an application framework) and low-level parts (e.g. the implementation of design patterns or the connection to a certain database). The chosen “Java Pet Store” from SUN’s BluePrints Program² already served as a guideline for several demonstration projects and is ideally suited to demonstrate the stratification approach.

The chapter begins with a description of the original Java Pet Store followed in Section 6.2 by an overview of implementations for other frameworks or in other languages. The Pet Store architecture is described in Section 6.3, and Section 6.4 discusses its stratification process, which is based on a study thesis by Schulz [Sch08]. The chapter closes with an evaluation in Section 6.5.

6.1 The Java Pet Store

Designed as sample application for J2EE technology it served as a running example in the J2EE book “Designing Enterprise Applications with the J2EE™ Platform, Second Edition” [SSJ02]. A detailed description of its architecture can be found in the online published appendix titled “Sample Application Design and Implementation”³ [Mic02].

The case study is based on version 1.4 of the Java Pet Store. The most recent release, 2.0, is a complete rewrite and focused on different architectural aspects

¹A high-level concern pertains to an abstract concept within an application.

²<http://java.sun.com/reference/blueprints/index.html> (checked August 2009)

³http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/sample-app/sample-app1.3.1.pdf (checked August 2009)

(namely service oriented architectures). As the following section shows, the old version has been already analyzed in other contexts and is therefore better suited for this case study.

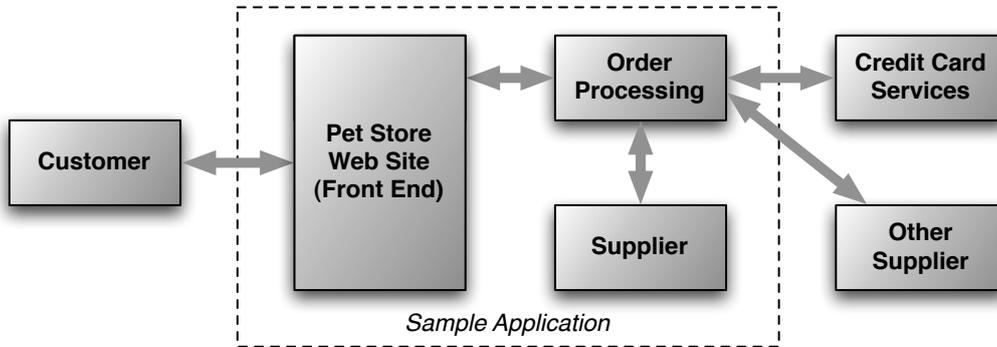


Figure 6.1: Main components of the Java Pet Store (adapted from [SSJ02])

The Pet Store is a typical web-based e-commerce application for selling animals to customers over the internet. In addition to the web shop it provides an administration interface and connectivity to external users for maintenance and inventory tasks. It follows the well-known model-view-controller design and is based on SUN's J2EE technology. Figure 6.1 shows the main components of the web shop. The customer interacts with the front-end using the pet store web site, which communicates with the order processing. This back-end talks to both internal and external suppliers and the credit card verification service.

Figure 6.2 shows a screenshot of the web front-end. It provides the following functionalities:

- browse catalog (searching, browsing categories, product details)
- account management (sign on/off, update account, update personalization)
- shopping (update cart, submit order)

The back-end applications answer the requests from the web server by querying the database and handling purchases.

6.2 Other Implementations

Since its publication the Pet Store served as an example for various projects: Oracle used it as a performance benchmark for J2EE applications⁴. Several J2EE applica-

⁴http://www.oracle.com/technology/sample_code/products/rdb/index.html (checked August 2009)

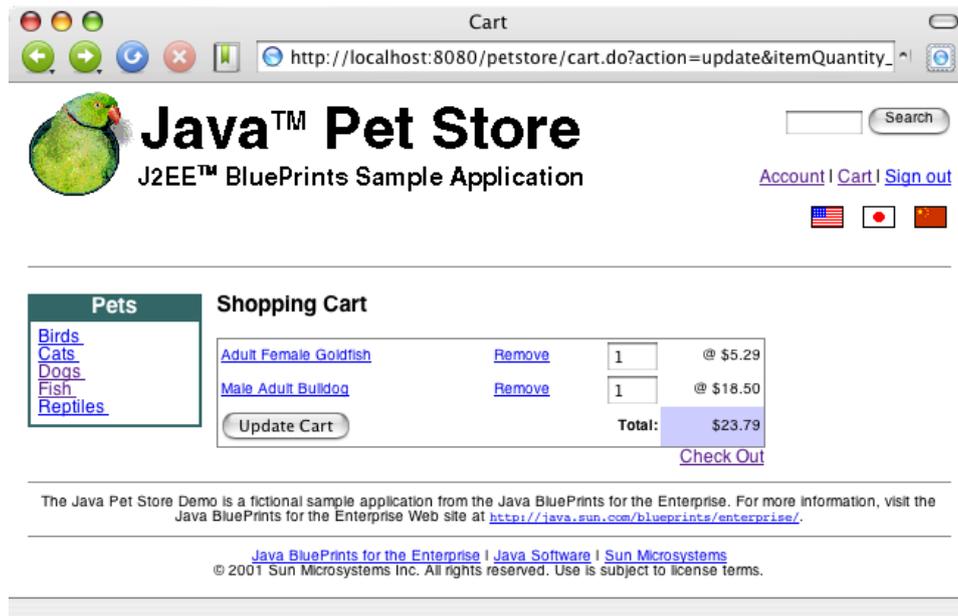


Figure 6.2: Screenshot of Pet Store front-end running in a browser

tion server vendors used it to certify their implementations. A Microsoft-sponsored project re-implemented it in .NET⁵. This implementation inspired Thomas Gil to create an improved version also based on .NET⁶. It was updated later with support for aspect oriented programming using Aspect DNG⁷. Based on Clinton Begin’s implementation for the iBATIS Data Mapper framework⁸ an example application for the J2EE Spring framework⁹ was developed by Jürgen Höller.

Paulo Merson gave an example from the Pet Store in “Representing Aspects in the Software Architecture - Practical Considerations” [Mer05], which showed how aspects can be represented graphically using stereotyped UML class diagrams.

Of greater interest are model-based reimplementations. The Middleware Company (a now discontinued subsidiary of Techtargent network¹⁰) initiated a case study for the comparative analysis of a traditional versus a model-based reimplementation of the Java Pet Store. The results were published online, comparing OptimalJ [HR03] or IBM Rational Rapid Developer [Her03b] against traditional approaches.

Although to be taken with a grain of salt, the model-driven approach was in both studies noticeably faster compared to the “traditional” method. A following study

⁵<http://msdn2.microsoft.com/en-us/library/ms954626.aspx> (checked August 2009)

⁶<http://dotnetguru.org/articles/PetShop/PetShopDNG/us/PetShopDNG.htm> (checked August 2009)

⁷<http://aspectdng.tigris.org/> (checked August 2009)

⁸<http://ibatis.apache.org/index.html> (checked August 2009)

⁹<http://www.springframework.org/> (checked August 2009)

¹⁰<http://www.techtargent.com/>

[HT04] also analyzed the maintainability of the model-driven approaches. Here, a set of typical enhancements was added to the application. Here too the team, which used the model-driven tool OptimalJ, finished earlier.

The initial specification [DE03] for both studies written by The Middleware Company was also used by Outsourcercfé to demonstrate the usability of their code generation product “JavaGen”¹¹. A similar approach was taken by Sygel and their WMEE tool¹² and the open source MDA tool openmdx¹³.

These reimplementations often only remodel the Pet Store from the requirements perspective, but differ quite often extensively from the original implementation. The reimplementations presented in this chapter stays as close to the original as possible.

6.3 The Pet Store Architecture

The Pet Store structure is based on a four-tier application. The client tier is entirely located in the browser and communicates via HTTP with the web tier. The EJB tier is connected to the database tier using JDBC¹⁴. As can be seen in Figure 6.3 the front end web tier is connected to the back end and the order processing using JMS¹⁵. This separation of the business logic from the persistency provides more flexibility and allows distribution of components across several computers. For performance reasons it is recommended to locate the front and back end within the same application server.

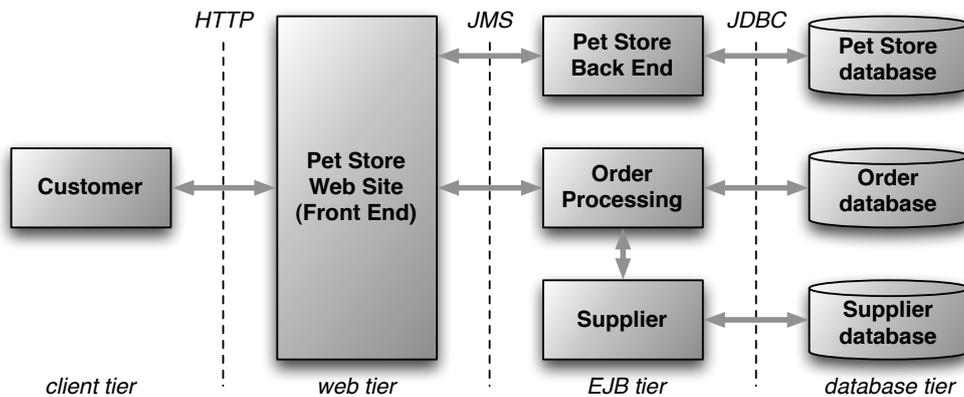


Figure 6.3: Four tier architecture of the Pet Store

¹¹<http://www.javagen.com/> (checked August 2009)

¹²<http://www.sygel.com/sygelsite/en/Products/Petstore.html> (checked August 2009)

¹³<http://web.archive.org/web/20060515225843/http://www.openmdx.org/openmdx/core/1.7/Petstore/html/c12.htm> still contains the documentation.

¹⁴Java DataBase Connectivity

¹⁵Java Messaging Service

6.3.1 The Client Tier

The client tier is represented by the browser application within the client's computer. This part does not contain any active components and is only responsible for rendering the HTML pages delivered by the web tier and for submitting forms¹⁶.

A screenshot of the Pet Store interface is shown in Figure 6.4. The presented shopping cart includes a form element for changing the amount of ordered items. It is submitted to the server using HTTP when the user updates the cart.



Figure 6.4: Screenshot of Pet Store shopping cart [SSJ02]

6.3.2 The Web Tier

The front end of the Pet Store is based on SUN's Web Application Framework (WAF) [Mic02]. The WAF follows the "Model 2" architecture with a front controller and presentation elements. The front controller dispatches requests to the application logic and the presentational elements create the result to be returned to the client. This approach enables the construction of web applications using the Model-View-Controller architecture design pattern. The basic control flow through the framework is shown in Figure 6.5. As can be seen, the framework also bridges the gap to the EJB tier. The inner elements belong to the framework, the outer elements (shown in green) are the configuration points of the framework. The injection of the configuration points is handled using configuration files (e.g. the Request Map shown in the figure).

The framework employs a standard processing pipeline for requests, which starts

¹⁶Basic form input validation is accomplished using client-side Javascript code.

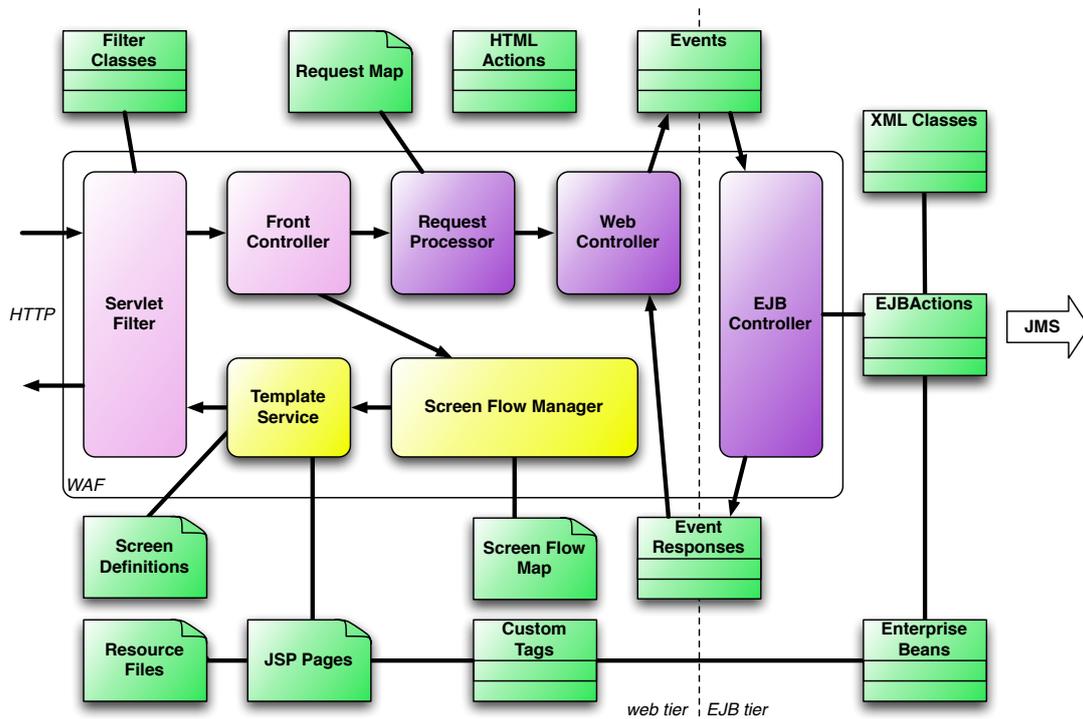


Figure 6.5: Web Application Framework control flow (adapted from [SSJ02])

with the servlet filter. The Pet Store provides a filter for encoding foreign characters and a sign-on filter, which handles authentication. After that, the WAF front controller forwards the request to the request processor. Here the HTML parameters are parsed and an action class is selected (e.g. `OrderHTMLAction` or `SignOffHTMLAction`). This class optionally generates events, which are routed to the EJB tier. The event responses are used to parameterize the following HTML content, which is based on Java Server Pages (JSPs) selected by the screen flow manager. In order to create consistent layout, the WAF template mechanism is used.

6.3.3 The EJB Tier

This tier contains the business logic of the application. By separating it from the web tier, it can be reused in other application contexts (e.g. account management or billing). As the name implies, this tier is based on Enterprise Java Beans (EJBs). Entity Beans are used to represent business data, Session Beans control user interaction, and asynchronous message passing via JMS is implemented using Message-Driven Beans.

The Web Application Framework integrates enterprise beans using EJBAction classes.

For instance in the Pet Store the class `CartEJBAction` handles web tier events, which pertain to updating the shopping cart. To ensure data consistency, the EJB tier uses declarative transaction control, which is defined in the deployment descriptors.

As can be seen in Figure 6.3 both order processing and the supplier are located in the EJB tier. To decouple both components and allow distribution, XML-based messaging with JMS is used here as well. After an order has been received, it is forwarded to the appropriate supplier. When the order has been dispatched, a message is sent back to the order processing system.

6.3.4 The Database Tier

Each EJB tier application is connected to a database using JDBC. The Pet Store database contains the user accounts and the pet store item catalog. Orders are stored in the Order database, and the supplier database holds the inventory of items.

For instance, the pet store catalog is implemented as a session bean on the EJB tier, which accesses the Pet Store database on the database tier. The customer information is stored in entity beans, which use J2EE Container Managed Persistence (CMP), deployment descriptors and database specific Data Access Objects (DAOs) to connect to the database.

6.3.5 Packaging, Deployment and Administration

The Pet Store runs as a web container within a J2EE application server. To deploy the application to the server, the components have to be packaged correctly, which is done by an Ant¹⁷ build script.

To administer the running application a Java Swing front end is supplied, which connects to the application server using JMI.

6.4 Stratifying the Pet Store

In order to create a stratified version of the Pet Store, the existing concerns and their dependencies have to be identified first. According to the dependencies and architectural considerations, the stratum hierarchy is constructed. For each concern transformation rules have to be designed while at the same time the sample application is constructed. This process and the resulting stratified architecture are presented in the following subsections.

¹⁷<http://ant.apache.org/> (checked August 2009)

6.4.1 Determining System Concerns

A stratified architecture describes a system on multiple layers with decreasing abstraction where at each layer a specific concern is made explicit. The four-tier architecture—described in the previous section—provides a basis for the determination of system concerns. For example:

Application Infrastructure The main concern is responsible for the creation of a four-tier application infrastructure and ensures the availability of required model elements (e.g. necessary data models).

User Interface The web tier needs to create web pages, which are rendered by the client browser.

Framework Integration The Web Application Framework—used by the web and the EJB tier—needs to be configured and instantiated.

Persistency A persistence mechanism is required to store the catalog and the account data.

Application Server Integration Based on data model annotations, deployment descriptors for the entity beans have to be created.

As described in Section 3.4, during the refinement of a concern, sub-concerns are revealed. By analysing the functional and technical elements of the Pet Store, the concerns shown in Figure 6.6 have been identified. The figure also shows the dependencies between the concerns.

The “Web Shop” concern builds the foundation of the four-tier architecture by creating the business logic integration (concern “Control and Business logic”) and the persistence infrastructure for the integration of a database. The web tier is generated and the integration of the WAF is prepared (“Web Application Framework”).

The web tier of the Pet Store is based on the web application framework, hence the user interface of the front end application needs to be integrated with it. This is described by the “User Interface To WAF” concern. A similar integration is required by the business logic. “Session Controlling” is a sub-concern of “Web Tier”.

The persistence is based on enterprise beans and thus on deployment descriptors. Both are handled by the concerns “EJBs” and “Session Beans”. The integration of the Web Application Framework is finalized by the generic concern “Framework Inheritance”, which connects the classes of the application to the framework classes. The final concern “Packaging and Deployment” prepares the application for deployment.

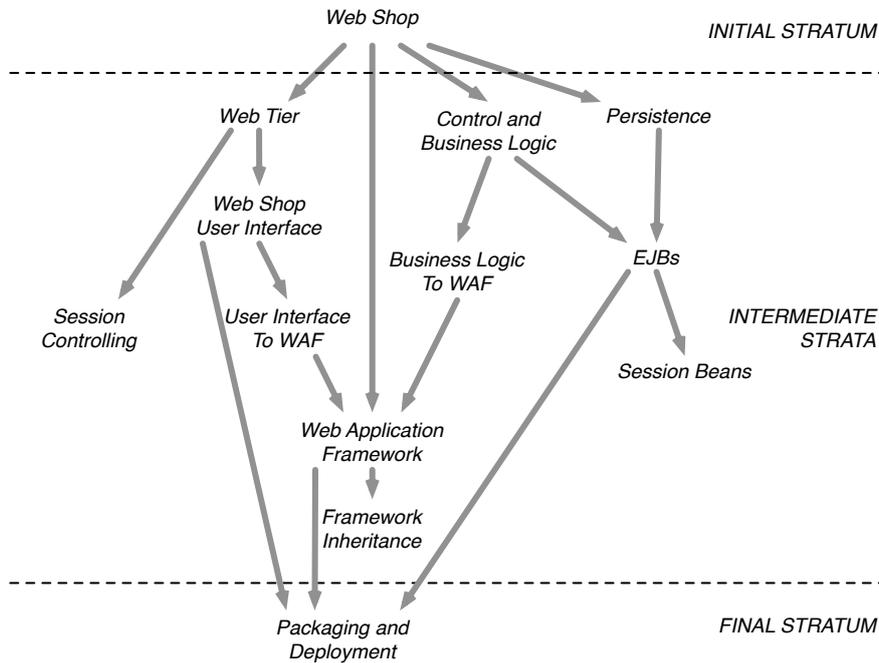


Figure 6.6: Pet Store Concerns ordered by dependency

6.4.2 From Concerns to Stratification

Based on the concerns and their dependencies a linear stratification order can be determined. The initial stratum—shown in Figure 6.7—contains the basic data models for a pet store and the “WebShop” annotation, which represents the infrastructure for a web shop system.

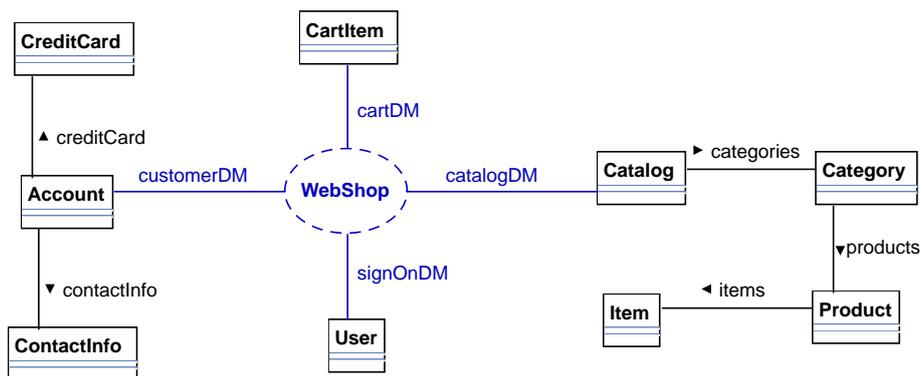


Figure 6.7: Pet Store: initial stratum

The annotation is linked to four data models: One for an account, one for a user registered within the system, one for a shopping cart item and four classes repre-

sending a catalog and the items contained therein. Class attributes (and operations) have been left out to simplify the presentation.

By refining the WebShop annotation, the second stratum is created (Figure 6.8). The annotation “WebShopParameterRepository” stores parameters, which are then available on subsequent strata. This approach can be used by following annotations in order to access parameters such as the database type or name of the store. The other annotations, which have been created on the second stratum, will be refined on subsequent strata. It can be decided, whether first the web tier or the parts belonging to the EJB tier (e.g. persistence) are refined.

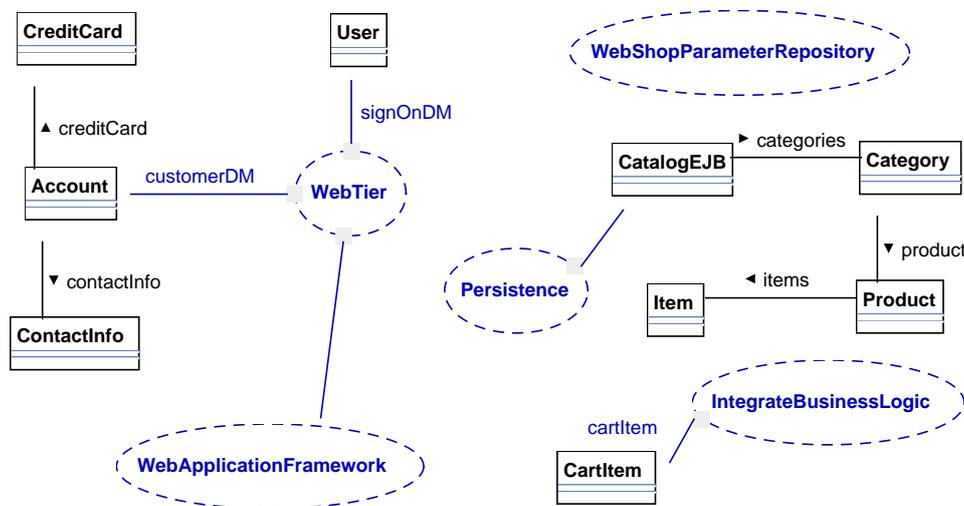


Figure 6.8: Pet Store: second stratum with refined WebShop

In the example, the web tier has been chosen. After refinement, the stratum shown in Figure 6.9 is created. A few classes have been added, they will be part of the framework instantiation. “PetStoreKeys” defines class and file names, which integrate the cart and customer classes into the framework. “CreateUserFlowHandler” adds the functionality to return to a certain web page after a new user has been created. The “ServiceLocator” is responsible for connecting the web with the EJB tier.

The inheritance relationships to the framework classes are handled by the helper annotation “FrameworkInheritance”. The annotation “WebApplicationFramework” will create the necessary class references while “WebShopUserInterface” is responsible for adding the JSPs and screen definitions.

On the following stratum (Figure 6.10), the “WebShopUserInterface” annotation has been refined. The created external files (JSP templates, screen definitions and other auxiliary files) are not shown in the model. The integration into the Web Application Framework is defined using the “WebShopUserInterfaceToWAF” annotation.

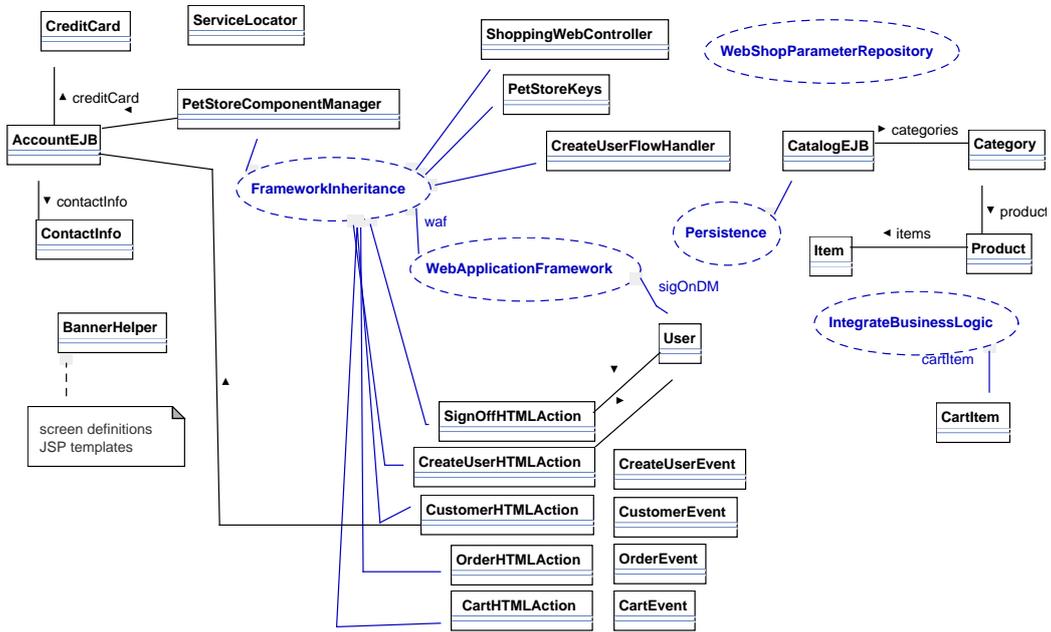


Figure 6.11: Pet Store: fifth stratum with refined WebShopUserInterfaceToWAF

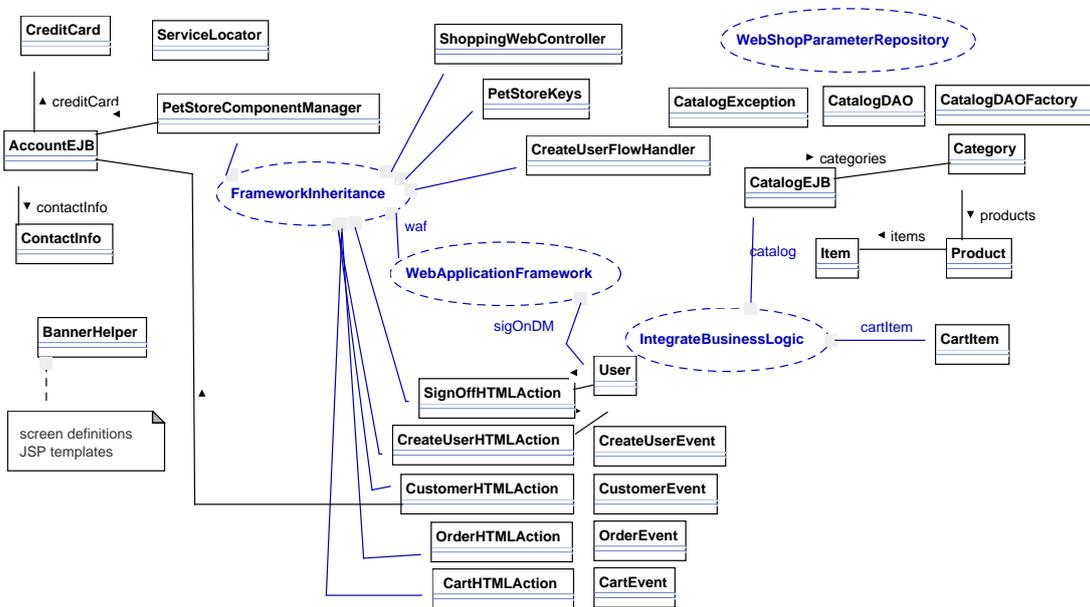


Figure 6.12: Pet Store: sixth stratum with refined Persistence

transactions on the beans to database calls. The SQL templates are generated as text files and stored in the file system next to the project files.

The EJB actions reacting on the events from the web tier are created by the “IntegrateBusinessLogic” transformation. It also generates annotations for EJBs, the session controlling and the integration of the logic into the framework. The resulting stratum is shown in Figure 6.13.

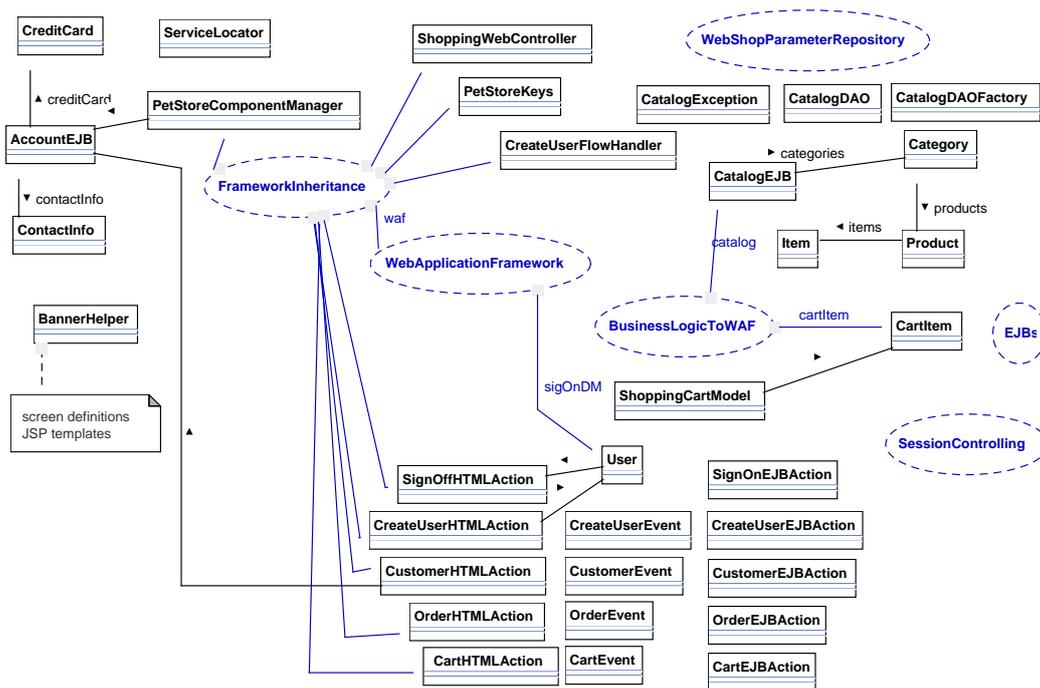


Figure 6.13: Pet Store: seventh stratum with refined IntegrateBusinessLogic

With the refinement of “SessionControlling” a different annotation mechanism can be seen in Figure 6.14. The classes “CatalogEJB”, “ShoppingClientFacadeLocalEJB” and “ShoppingCartLocalEJB” have been marked with the stereotype “J2EE.Session-Bean”. These markings are detected by the “EJBs” annotation and a matching “SessionBean” annotations is created.

The remaining annotations are resolved on strata 9 to 14. Details on these refinements can be found in the study thesis by Schulz [Sch08]. The final step before packaging and deployment is the refinement of the helper annotation “Framework-Inheritance”. The resulting model fragment with the created reference classes and inheritance relationships is shown in Figure 6.15.

After the system has been completely refined, a “Packaging” annotation creates the Web Archive files, which can be deployed on a J2EE application server.

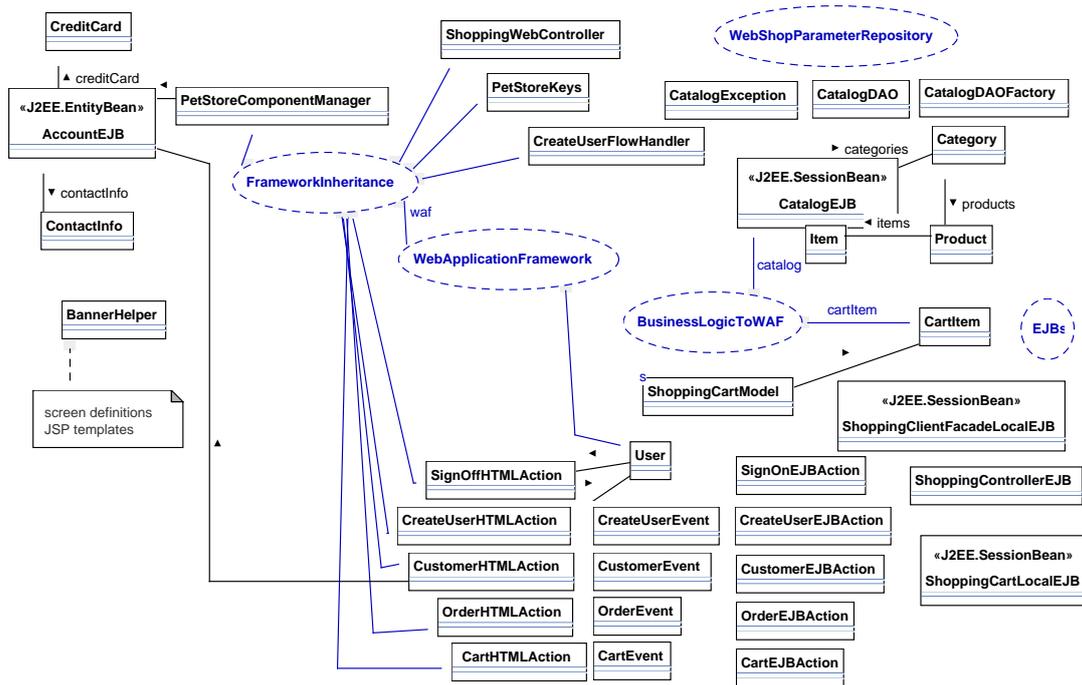


Figure 6.14: Pet Store: eighth stratum with refined SessionControlling

6.4.3 Transformation Rules

This section illustrates the applicability of the transformations by presenting some of the concepts used within the transformation rules.

Abstract Syntax versus concise Models

One disadvantage of using abstract syntax to describe model transformations is the “talkativeness” required by the metamodel. For instance to create a method in metamodel syntax a `UMLMethod` object is required, a new `UMLType` has to be created for each parameter, the visibility has to be assigned and thrown exception classes have to be added.

To simplify this process and make the transformation rules more readable, some “syntactic sugar” has been added. An example is shown in Figure 6.16.

This story activity adds an operation and an attribute to a class and creates a generalization to `java.io.Serializable`. The `UMLClass` object in the lower left has a special attribute named `additionalStereotype`, which sets the appropriate stereotype without the need to create and attach a `UMLStereotype` object. The special attribute `decl` in the upper right corner parses the given string and creates the necessary type objects and sets additional attributes (e.g. name and visibility).

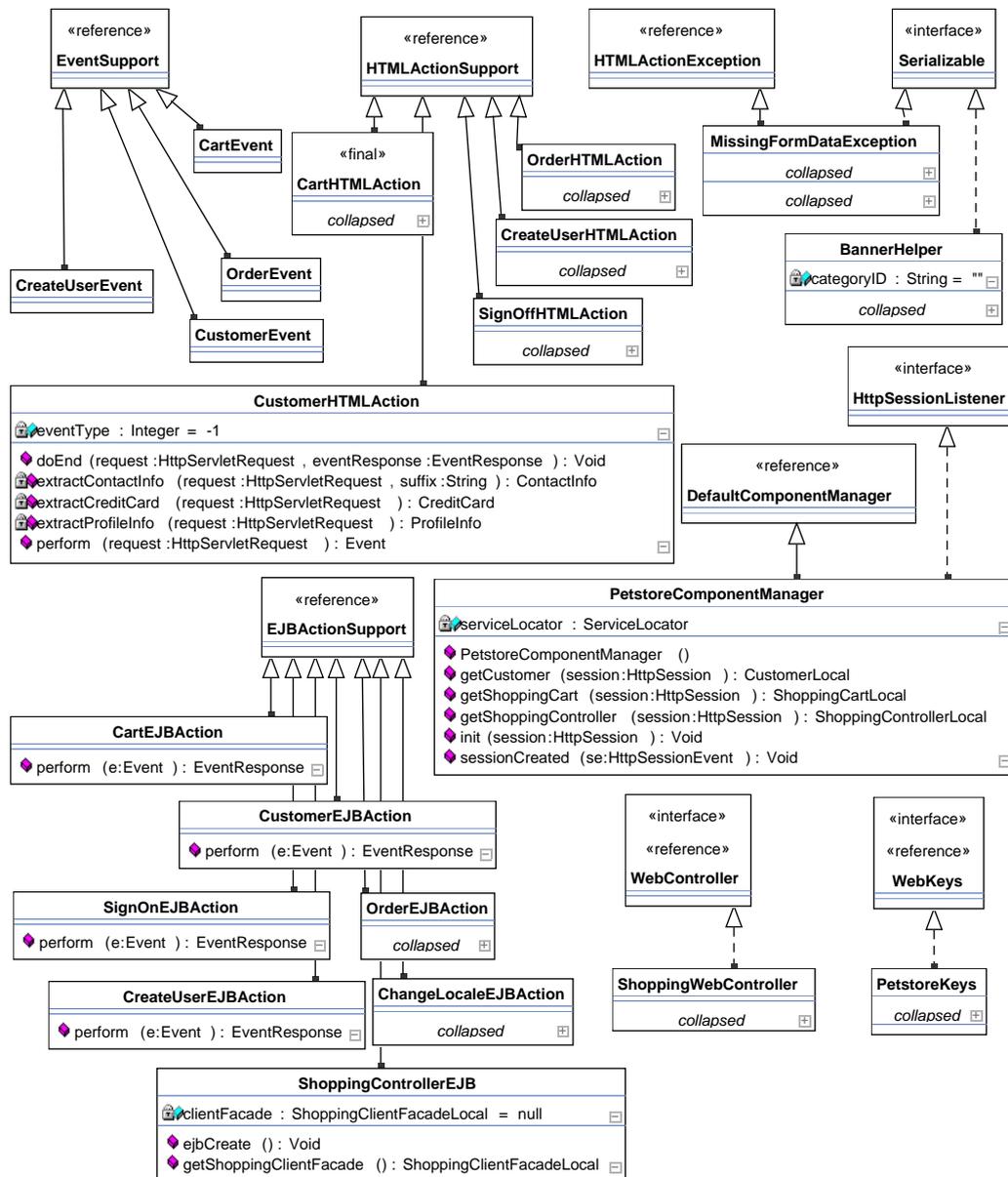


Figure 6.15: Pet Store: result of “FrameworkInheritance” refinement

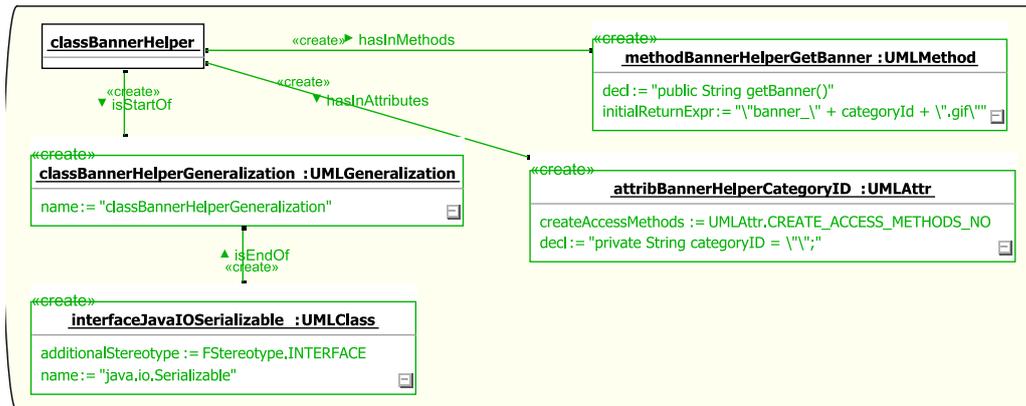


Figure 6.16: Pet Store: syntactic sugar within transformation rules

`initialReturnExpr` finally sets the return expression of the method body associated with the created operation.

Mass creation of Methods

Initially, Futemplerator (Section 5.7) was primarily geared towards code generation. The extensible stereotype concept however enabled additional usage scenarios. For instance the transformation rule “WebTier” uses the stereotype “MethodMaker” to create several methods at once using a single template activity containing multiple method declarations (Figure 6.17). The Velocity template associated with this stereotype parses the activity and creates the necessary objects and attributes.

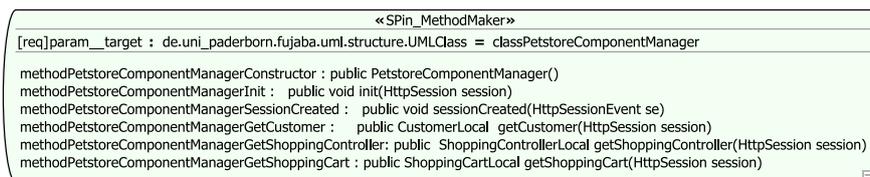


Figure 6.17: Pet Store: template activity “MethodMaker”

Creating external Files and User Notifications

In addition to Java code, the Pet Store requires several Java server pages and configuration files for mappings. These files are also created by template activities. An example from the rule “WebShopUserInterface” is shown in Figure 6.18.

The “UserMessage” stereotype, shown on the right in Figure 6.18, displays a message in the Fujaba console view. The message entry informs the user that the transfor-

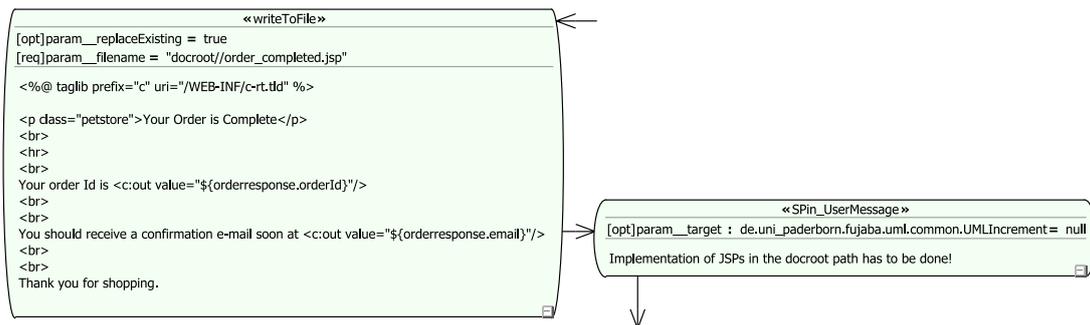


Figure 6.18: Pet Store: template activity “WriteToFile” and “UserMessage”

mation requires manual edits. It optionally links to a specific model element, which can be used to navigate to hook spots or other model elements.

Annotation Parameters

An important aspect of transformation rules is their adaptability. Both annotations and annotation links can be parameterized using a parameter editor. An example is shown in Figure 6.19 with three string parameters attached to an annotation link.

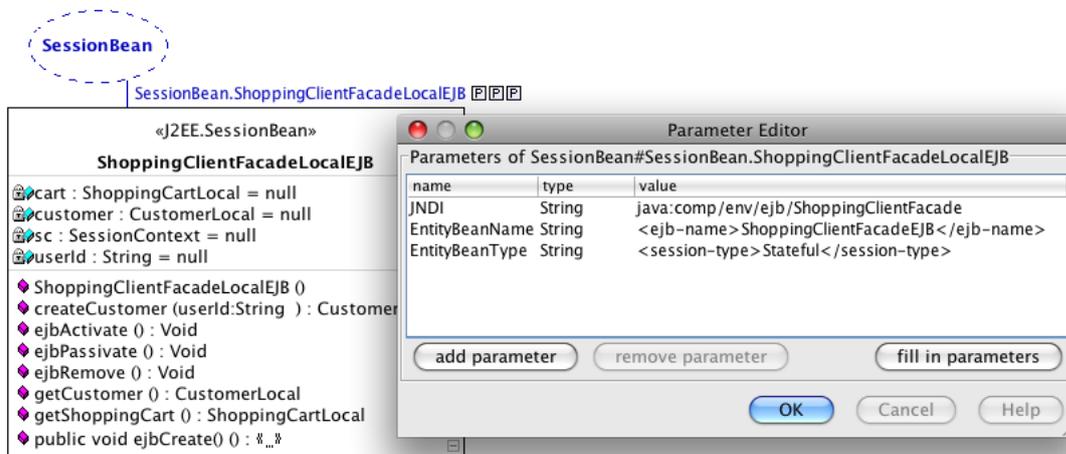


Figure 6.19: Pet Store: SessionBean annotation with parameter editor

The associated transformation rule, presented in Figure 6.20, iterates over all links of the annotation (activity 2) and associates the three parameters with `RParameter` objects (activity 3). For each link an inheritance relationship to `javax.ejb.SessionBean` is created (4). Then template activities are used to add imports (5) and set some comments (6). If the `RParameter` “entityBeanName” is set, it is used to create

JavaDoc comments for later processing by XDoclet¹⁹ (7). An `ejbCreate` method is created, if it does not exist, yet (8) and finally each remaining method is marked with an XDoclet comment (9).

Customization using Hook Spots

Some transformation rules create hook spots, which enable the developer to add hand-written code at predefined spots within a method (cf. Section 3.6 for details). For instance the transformation rule for “WebShopUserInterfaceToWAF” (Figure 6.11) adds HTML action classes with “perform” methods, which prepare the request object and contain a hook spot for the actual action handling. This code is then added by the developer, and SPin ensures that it is reinserted at the correct location during re-transformation. The hook spot activity is marked with an open lock in Figure 6.21.

6.5 Evaluation

Although the Java Pet Store is an artificial application, developed to demonstrate SUN technologies, it contains several aspects, which are present in real-world systems. Java Enterprise applications are quite common and are often used in combination with an MVC-Framework such as WAF. In addition to source code files, other artefacts are necessary for the configuration of the system or to represent the user interface.

The stratification of the Pet Store addresses and separates these concerns. The first step was to identify the concerns, which will be addressed on the different strata. The goal was to rebuild the application using stratification instead of building a similar application from scratch as it was done by some of the implementations described in Section 6.2.

This approach required a detailed analysis of the existing application in order to detect all dependencies and to build transformation rules, which re-created the existing classes. This abstraction process within stratification revealed several problems, which are discussed in Section 7.2. In retrospect building a new application from scratch, which carries the some functionality as the Pet Store and integrates off-the-shelf components, would have been sufficient.

Stratification is based on a strict hierarchical structure where on each stratum only one concern is implemented. The Pet Store contained circular dependencies, which complicated structuring the concerns. As a result, some transformation rules not

¹⁹<http://xdoclet.sourceforge.net/xdoclet/index.html> (checked August 2009)

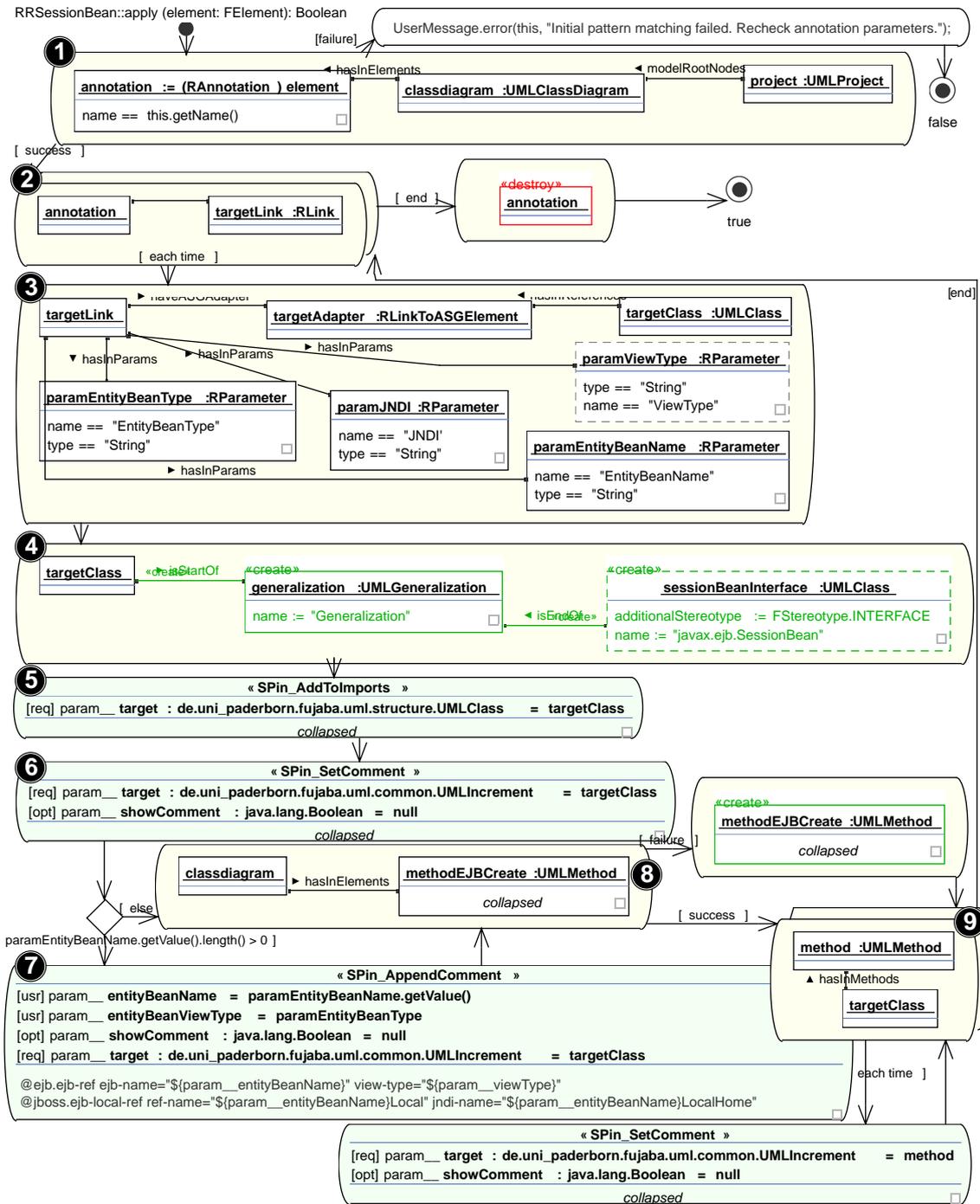


Figure 6.20: Pet Store: SessionBean transformation rule

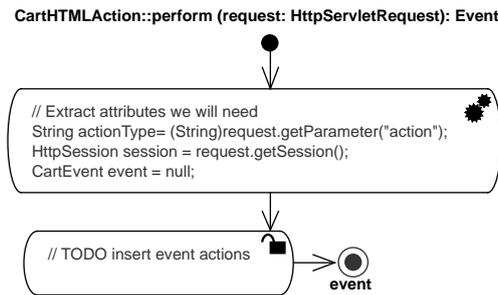


Figure 6.21: Pet Store: generated `perform` method with hook spots

only require the previous execution of other rules, but also create parts for succeeding transformations.

As can be seen in the example, some concerns consist of sub-concerns, which become visible when the transformation rules create the matching annotations. This “chain” of concerns is enforced by the transformation rules and helps to deal with the dependencies between the different concerns.

The close connection between transformation rules sometimes restricts their reusability. In addition, reusability is more limited when the rule is designed for a certain environment. This happens more often on less-abstract transformations, which do not reveal sub-concerns, but transform the model and create actual code.

Here a good balance has to be found between these two types of transformations. The concept of “abstract annotations” described in Section 3.6 has not been used in the example, but may provide some help to structure the transformation rules for a certain application area.

The Web Application Framework provided a good basis for the construction of the Pet Store. Framework integration and parameterization is easily described using SPin’s concern notation. Transformation rules create the configuration files, the glue code and the hot spots where the developer adds the application specific code.

Stratification helps to understand the architecture of an application by adding one concern at the time while still showing the complete model. This helps to see how the concern integrates into the system, but for larger applications the model size may become a problem. Possible solutions for this are discussed in Section 7.2.

An advantage of model-driven software development is the separation of the application developer from the transformation developer. Creating transformation rules requires deeper knowledge of the involved technologies and dependencies. The result, however, can be used for several applications and the application developer does not have to deal with the underlying technology. Instead he is guided by the annotations and fills in the necessary parameters and code fragments.

Chapter 7

Conclusion

The final chapter summarizes the results of this thesis and gives an outlook to future work areas.

7.1 Summary

The goal of this thesis was to concretize the concept of architecture stratification and to show its applicability for software development. In Chapter 2 current trends in software development are analyzed. A comparison of existing tools revealed that most of the trends focus on specific areas and refrain from combining them in order to achieve higher productivity.

Architecture stratification embraces these trends by offering a concern-oriented approach to model-driven software development, which enables the stepwise implementation of aligned and crosscutting concerns. The graphical transformation language is suited for the transformation of small and large models and the generation of textual artefacts such as code or configuration files. Finally, parameterizable transformation rules, hook spots and the ability to create sub-concerns, which in turn trigger transformations, show similarities to software product lines.

The approach was implemented on top of the open source CASE tool Fujaba. By extending Fujaba and its story driven modelling capabilities a suitable model transformation system with the following features was created:

Rule orientation Within a stratified architecture, model annotations form the connection to the model transformation system. Each annotation within the model is related to one transformation rule. The annotation serves as a well-defined entry point into the model.

Parameterization The transformation rule is parameterized by the annotation.

Annotations may link to other models elements and both the links and the annotation itself can be parameterized using primitive types¹.

Rule metamodel A transformation rule is enriched with metadata, which describes the annotation parameters, the purpose of the transformation rule and relationships to other transformations. This information is available within the annotation editor and helps to correctly parameterize annotations.

Visual control flow The overall control flow of the complete transformation rule is described using UML activity diagrams. This provides a visual overview of the complete transformation.

Visual, metamodel-based declarative model transformation The model transformations are described using story patterns, which are a form of UML collaboration diagrams. A story pattern describes both the search pattern and the changes to be applied. Each objects within the pattern is an instance of the underlying metamodel. SPin provides the UML metamodel for class diagrams and Fujaba uses it to enable “model completion” for transformation rule developers. However, this approach is not limited to class diagrams—other metamodels can be added.

Integrated, template-based code generation In addition to Fujaba’s built-in story patterns, support for code generation has been added. Code generation elements are integrated into the overall control flow using template activities. They are able to access the objects of other story patterns within the same transformation rule.

Extensible transformation concept The Fujaba architecture enables the integration of other transformation facilities, examples can be found in Section 7.2. Java code fragments can be added to the overall control flow providing direct access to story pattern and template activity objects.

Unique identifiers To each element created by the transformation a unique identifier is assigned. By combining transformation rule information with stratification data, the transformation engine ensures, that if a transformation is applied a second time, the same identifier is assigned to the same element. Using this feature, SPin is able to allow manual additions to the model providing more flexibility for the application developer.

The case study in Chapter 6 shows the applicability of the approach and the developed software system. A real-world application was re-constructed using stratification. The resulting architecture highlights the different concerns of the application using 14 strata.

¹e.g. strings, numbers, booleans

7.2 Future Work

The concept of architecture stratification was initially described by Atkinson and Kühne [AK00]. This thesis builds upon their work and elaborates several aspects of stratification. However, there are still open questions and research areas, which should be addressed in the future.

7.2.1 Stratification

With each stratum in a stratified architecture abstraction decreases and the model becomes more concrete. The described implementation demonstrates the approach using class diagrams but it can be applied using other modeling languages as well. This requires a single metamodel, which supports the description of a software system on several levels of abstraction. As this may pose a limitation to modeling, the use of several metamodels within the same stratified architecture has to be researched. For instance the use of domain-specific languages on selected strata may help to specify certain concerns. The dependencies and possible transformations between these metamodels and their effect on stratification has to be investigated.

As demonstrated in the case study, transformation rules can be parameterized for different destination platforms. To achieve even more platform independency, different metamodels for the destination platforms may be required. As a result, the strict linear structure of a stratified architecture can be extended to a tree-like system with different destination platforms as leafs.

For component-based systems and in environments with distributed development teams, mechanisms for the integration of several stratified architectures have to be found. This leads to another area: the model size. Currently, transformation rules mainly *add* new elements to the model, which may lead to unmanageable model sizes. Each stratum focusses on one concern and usually affects only a fraction of the system leaving the remaining parts untouched. In order to focus on the affected areas, several views per stratum can be used. Especially in the case of crosscutting concern the affected areas are scattering across the system. Here, algorithms for selective hiding of irrelevant elements can help to regain overview of the architecture.

This thesis does not address the reengineering aspect of stratification. In order to create a stratified architecture for an existing system, the concerns have to be detected manually. After that, corresponding transformation rules have to be built by hand. For the automatic detection of concerns, more complex algorithms are required, which also take the behaviour of the system into account. One promising approach is edge-filtering [PL07].

Section 3.6 describes the concept of a “chain of concerns” which guides the creation of a stratified architecture. This approach can be combined with mechanisms from software product lines requiring a more detailed dependency management for trans-

formations and the ability to exchange annotation parameters. The current solution, which removes an annotation after it has been transformed into an implementation, may not be suitable in this scenario.

7.2.2 Transformation

Stratification uses concern-based stepwise refinement to get from an abstract model to an executable solution. The concern implementation is described through transformation rules. As a consequence, more complex concerns result in more complex rules, which implement them. The visual presentation of transformations helps to organize the transformation description, but may result in diagrams too large to handle efficiently. Hence, mechanisms to alleviate these effects have to be researched. Possible solutions are splitting of rules into parts (useful for reusing parts of a transformation), hierarchical transformations similar to VIATRA2 or the data flow presentations used in GReAT (cf. Section 5.4).

SPin already provides tools for the construction of transformation rules. Further help can be provided by creating transformation rules from existing models. This requires a process for selecting input and output model elements and defining the variation points within them.

Traceability within SPin is based on unique identifiers, which are assigned to all model elements. The created “traceability chains” are broken, if model elements are removed during transformation. Further research may also address issues arising from changing models and the resulting conflicts during retransformation. The migration of existing transformation rules also falls into this context.

The current transformation system uses story diagrams to describe model transformations and velocity templates for the generation of code and other textual artefacts. Especially in the context of other metamodels the graphical transformation of behaviour has to be investigated. Behaviour representation can be based on abstract syntax trees, which are close to the code, or more abstract metamodels, such as state diagrams. An extension for ASTs within Fujaba is discussed by Pieter Van Gorp et al. [GSMD03]. Aspect oriented techniques may be of relevance to describe concern implementations using join points and advices.

As mentioned in the previous section, future research may address reengineering of existing architectures by applying stratification. In this context bidirectional transformation languages are highly relevant. A promising candidate—the declarative Triple Graph Grammar [Sch95]—has already been mentioned in Section 5.4. Active research towards incremental implementations [GW06] and the addition of control structures [GMJ06] may provide necessary building blocks for a transformation language, which describes abstraction and implementation of a concern using a single transformation rule.

Bibliography

- [ADP⁺05] Joao Paulo Almeida, Remco Dijkman, Luis Ferreira Pires, Dick Quartel, and Marten van Sinderen. Abstract Interactions and Interaction Refinement in Model-Driven Design. In *EDOC '05: Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference*, pages 273–286, Washington, DC, USA, 2005. IEEE Computer Society.
- [AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An Overview of CaesarJ. *Transactions on AOSD I, LNCS*, 3880:135–173, 2006.
- [AK00] Colin Atkinson and Thomas Kühne. Separation of Concerns through Stratified Architectures. *International Workshop on Aspects and Dimensions of Concerns, ECOOP 2000, Cannes, France*, 2000.
- [AK03] Colin Atkinson and Thomas Kühne. Aspect-Oriented Development with Stratified Frameworks. *IEEE Software*, 20(1):81–89, 2003.
- [AKB99] Colin Atkinson, Thomas Kühne, and Christian Bunse. Dimensions of Component-based Development. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 185–194, London, UK, 1999. Springer-Verlag.
- [AKK⁺06] Aditya Agrawal, Gabor Karsai, Zsolt Kalmar, Sandeep Neema, Feng Shi, and Attila Vizhanyo. The Design of a Language for Model Transformations. *Journal on Software and System Modeling*, 5(3):261–288, September 2006.
- [AKRS06] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference. LNCS*, volume 4066, pages 361–375, Berlin, Germany, 2006. Springer-Verlag.

- [ALS06] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 122–131, New York, NY, USA, 2006. ACM Press.
- [ARNRSG06] Neta Aizenbud-Reshef, Brian T. Nolan, Julia Rubin, and Yael Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.
- [BA99] C. Bunse and C. Atkinson. The Normal Object Form. Bridging the Gap from Models to Code. In Robert B. France and Bernhard Rumpe, editors, *Proceedings of UML '99*, pages 675–690. Springer-Verlag, 1999.
- [BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley, New York, NY, USA, 2004.
- [Bac78] Ralph-Johan Back. On the Correctness of Refinement Steps in Program Development. Ph.D. thesis, Åbo Akademi, Department of Computer Science, 1978.
- [Bac98] John Backus. The History of Fortran I, II, and III. *IEEE Annals of the History of Computing*, 20(4):68–78, 1998.
- [Bac02] Ralph-Johan Back. Software Construction by Stepwise Feature Introduction. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 162–183, London, UK, 2002. Springer-Verlag.
- [Bau07] Daniel Bausch. Integration einer Template-Sprache in das Story-Driven-Modeling-Framework von Fujaba. Bachelor thesis, Technische Universität Darmstadt, September 2007.
- [BC04] Elisa Baniassad and Siobhán Clarke. Theme: An Approach for Aspect-Oriented Analysis and Design. *26th International Conference on Software Engineering (ICSE 2004)*, pages 158–167, 2004.
- [BFB07] Mikael Barbero, Marcos Didonet Del Fabro, and Jean Bézivin. Traceability and Provenance Issues in Global Model Management. *3rd ECMDA-Traceability Workshop*, 2007.
- [BFS06] Eric Bodden, Florian Forster, and Friedrich Steimann. Avoiding Infinite Recursion with Stratified Aspects. In Robert Hirschfeld, Andreas Polze, and Ryszard Kowalczyk, editors, *NODE/GSEM 2006: Erfurt, Germany. LNI*, volume 88, pages 49–64. Gesellschaft für Informatik, 2006.

- [BGN⁺04] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tich, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool integration at the meta-model level: the Fujaba approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):203–218, 2004.
- [BHS07] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. *Pattern Oriented Software Architecture: On Patterns and Pattern Languages*. John Wiley & Sons, Inc., 2007.
- [BM03] Peter Braun and Frank Marschall. Transforming Object Oriented Models with BOTL. *Electronic Notes in Theoretical Computer Science*, 72(3):103–117, 2003.
- [BM06] Thomas Büchner and Florian Matthes. Introspective Model-Driven Development. In Volker Gruhn and Flávio Oquendo, editors, *Software Architecture, Third European Workshop, EWSA 2006, Nantes, France, September 4-5, 2006, Revised Selected Papers. LNCS*, volume 4344, pages 33–49. Springer-Verlag, 2006.
- [BMPP89] Friedrich Ludwig Bauer, Bernhard Möller, Helmut Partsch, and Peter Pepper. Formal Program Construction by Transformations-Computer-Aided, Intuition-Guided Programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, 1989.
- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
- [Bos98] Jan Bosch. Design Patterns as Language Constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.
- [Bro87] Frederick P. Brooks Jr. No Silver Bullet - Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.
- [Bro97] Manfred Broy. Towards a Mathematical Concept of a Component and its Use. In *Software- Concepts and Tools*, volume 18, pages 137–148, 1997.
- [BSR04] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):1278–1295, 2004.
- [BV06] András Balogh and Dániel Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1280–1287, New York, NY, USA, 2006. ACM Press.

- [CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming - Methods, Tools and Applications*. Addison-Wesley, New York, NY, USA, 2000.
- [Cer04] Gary Cernosek. Next-generation model-driven development. <ftp://ftp.software.ibm.com/software/rational/web/whitepapers/rsa-cernosek-wp.pdf> (last checked July 2008), Dec. 2004.
- [CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLA 2003 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.
- [CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
- [CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Using Feature Models. In Robert Nord, editor, *Proceedings of the Third Software Product-Line Conference. LNCS*, volume 3154, pages 266–283, Berlin, Germany, Sept 2004. Springer-Verlag.
- [Cho57] Noam Chomsky. *Syntactic structures*. Mouton, The Hague, 1957.
- [CHT81] Thomas E. Cheatham, Jr., Glenn H. Holloway, and Judy A. Townley. Program Refinement by Transformation. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 430–437, Piscataway, NJ, USA, 1981. IEEE Computer Society.
- [Com06] Compuware. Compuware OptimalJ: How transformation patterns transform UML models into high-quality J2EE applications. <http://frontline.compuware.com/javacentral/members/resources/javacentral.Patterns.41.pdf> (last checked July 2008, registration required), 2006.
- [Coo07] Steve Cook. Interactive Television Applications using DSL Tools, 2007. Model-Driven Development Tool Implementers Forum 2007. <http://www.dsmforum.org/events/MDD-TIF07/> (last checked June 2008).
- [CRS+05] Ruzanna Chitchyan, Awais Rashid, Pete Sawyer, Alessandro Garcia, Mónica Pinto Alarcon, Jethro Bakker, Bedir Tekinerdogan, Siobhán Clarke, and Andrew Jackson. Survey of Analysis and Design Approaches, 2005. <http://www.comp.lancs.ac.uk/computing/aop/papers/d11.pdf> (last checked June 2008).
- [DE03] Salil Deshpande and Will Edwards. The Middleware Company Application Server Platform Baseline Specification. The Middleware Company. <http://web.archive.org/web/20031003120719/http://>

- www.middleware-company.com/casestudy/specification.pdf (last checked Nov. 2007), May 2003.
- [Dij82] Edsger W. Dijkstra. On the role of scientific thought. In *Selected writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, New York, NY, USA, 1982.
- [EAG06] Angelina Espinoza, Pedro P. Alarcon, and Juan Garbajosa. Analyzing and Systematizing Current Traceability Schemas. *30th Annual IEEE/NASA Software Engineering Workshop SEW-30 (SEW'06)*, pages 21–32, 2006.
- [FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language and Java. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 296–309, London, UK, 2000. Springer-Verlag.
- [Fon06] Choong Koon Fong. Quick Start Guide to MDA - A Primer to MDA Using Borland Together Technologies. http://www.borland.com/resources/en/pdf/products/together/together_mda_quickstart.pdf (downloaded July 2008), Sept. 2006.
- [FS03] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, New York, NY, USA, 2003.
- [GF94] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An Analysis of the Requirements Traceability Problem. In *Proceedings of the First International Conference on Requirements Engineering, 1994.*, pages 94–101, 1994.
- [GGZ⁺05] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Dániel Varró. Using Graph Transformation for Practical Model Driven Software Engineering. In *Model Driven Software Engineering*, pages 91–118. Springer-Verlag, 2005.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, New York, NY, USA, 1995.
- [Gir08] Martin Girschick. Integrating Template based Code Generation into Graphical Model Transformation. In Thomas Kühne, Wolfgang Reisig, and Friedrich Steimann, editors, *Modellierung 2008, 12.-14. März 2008, Berlin, Germany, Proceedings. LNI P-127*, pages 27–41. Gesellschaft für Informatik, March 2008.

- [GKK06] Martin Girschick, Thomas Kühne, and Felix Klar. Generating Systems from Multiple Levels of Abstraction. In D. Draheim and G. Weber, editors, *Conference Proceedings Trends in Enterprise Application Architecture 2006, LNCS 4473*, pages 127–141, Berlin, Germany, 2006. Springer-Verlag.
- [GLZ06] Jeff Gray, Yuehua Lin, and Jing Zhang. Automating Change Evolution in Model-Driven Engineering. *Computer*, 39(2):51–58, 2006.
- [GMJ06] Pieter Van Gorp, Olaf Muliawan, and Dirk Janssens. Integrating a Declarative with an Imperative Model Transformation Language. 1st Triple Graph Grammar Workshop - TGG' 06, Bayreuth, Germany, 2006.
- [GS04] Jack Greenfield and Keith Short. *Software Factories*. John Wiley & Sons, Inc., 2004.
- [GSMD03] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML Refactorings. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings. LNCS*, volume 2863, pages 144–158, Berlin, Germany, 2003. Springer-Verlag.
- [GSR05] Leif Geiger, Christian Schneider, and Carsten Reckord. Template- and modelbased code generation for MDA-Tools. In Holger Giese and Albert Zündorf, editors, *3rd International Fujaba Days 2005*, Paderborn, Germany, September 2005.
- [GV07] Iris Groher and Markus Völter. XWeave: Models and Aspects in Concert. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 35–40, New York, NY, USA, 2007. ACM Press.
- [GW06] Holger Giese and Robert Wagner. Incremental Model Synchronization with Triple Graph Grammars. In Oscar Nierstrasz, John Whittle, David Harel, and Gianna Reggio, editors, *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genoa, Italy. LNCS*, volume 4199, pages 543–557, Berlin, Germany, 2006. Springer-Verlag.
- [Hah08] Anouar Haha. STrac – Ein Werkzeug zum automatisierten Im- und Export von Modellfragmenten. Diplomarbeit, Technische Universität Darmstadt, April 2008.

- [Her03a] Javier Hernandez. Model based multi-tier & multi-platform application generation. http://www.care-t.com/_downloads/whitepapers/WP-N-tier.pdf (last checked July 2008), 2003.
- [Her03b] David Herst. Model Driven Development for J2EE with IBM Rational Rapid Developer. The Middleware Company. http://www.tspllc.com/RRD_Productivity_Study.pdf (last checked Nov. 2007), November 2003.
- [HHHL03] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic Design Pattern Detection. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 94–103, Washington, DC, USA, 2003. IEEE Computer Society.
- [HO93] William H. Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428, New York, NY, USA, 1993. ACM Press.
- [HR03] David Herst and Ed Roman. Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach - Productivity Analysis. The Middleware Company. http://www.omg.org/mda/mda_files/MDA_Comparison-TMC_final.pdf (last checked Nov. 2007), June 2003.
- [HT04] David Herst and Owen Taylor. Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach - Maintainability Analysis. The Middleware Company. <http://www.compuware.no/optimalJ/Middleware2.pdf> (last checked Nov. 2007), January 2004.
- [HVEK07] Arno Haase, Markus Völter, Sven Efftinge, and Bernd Kolb. Introduction to openArchitectureWare 4.1.2, 2007. Model-Driven Development Tool Implementers Forum 2007. <http://www.dsmforum.org/events/MDD-TIF07/> (last checked June 2008).
- [Inc08] No Magic Inc. MagicDraw data sheet. <http://www.magicdraw.com/files/brochures/a4/MagicDrawDataSheet.pdf> (downloaded July 2008), 2008.
- [Int05] Interactive Objects Software GmbH. ArcStyler - The leading platform for Model Driven Architecture. http://www.interactive-objects.com/fileadmin/pdf/products/ArcStyler5_Whitepaper_220205.pdf (last checked July 2008), 2005.
- [JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

- [JK06a] Frédéric Jouault and Ivan Kurtev. On the Architectural Alignment of ATL and QVT. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1188–1195, New York, NY, USA, 2006. ACM Press.
- [JK06b] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference.LNCS*, volume 3844, pages 128–138, Berlin, Germany, 2006. Springer-Verlag.
- [Joh97] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, 1997.
- [Jou05] Frédéric Jouault. Loosely Coupled Traceability for ATL. ECMDA Workshop on Traceability http://www.sintef.no/content/page1____6259.aspx (last checked June 2008), 2005.
- [KCS05] Audris Kalnins, Edgars Celms, and Agris Sostaks. Model Transformation Approach Based on MOLA. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, 2005.
- [KE07] Jörg Kiegeand and Hajo Eichler. Enabling comprehensive tool support for QVT, 2007. Eclipse Modeling Symposium 2007 <http://www.eclipsecon.org/summiteurope2007/index.php?page=detail/&id=2> (last checked July 2008).
- [KGK06] Thomas Kühne, Martin Girschick, and Felix Klar. Tool Support for Architecture Stratification. In Heinrich C. Mayr and Ruth Breu, editors, *Modellierung 2006, 22.-24. März 2006, Innsbruck, Tirol, Austria, Proceedings*, pages 213–222. Gesellschaft für Informatik, 2006.
- [KKG05] Felix Klar, Thomas Kühne, and Martin Girschick. SPin – A Fujaba Plugin for Architecture Stratification. In Holger Giese and Albert Zündorf, editors, *3rd International Fujaba Days 2005*, pages 17–23, September 15-18 2005.
- [Kla05] Felix Klar. SPin – Ein Werkzeug zur Realisierung von Architektur-Stratifikation. Diplomarbeit, Technische Universität Darmstadt, April 2005.
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, USA, 1997.

- [KML⁺04] G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits. Composition and Cloning in Modeling and Meta-Modeling. *IEEE Transactions on Control Systems Technology*, 12(2):263–278, 2004.
- [LLVC06] László Lengyel, Tihamér Levendovszky, Tamás Vajk, and Hassan Charaf. Realizing QVT with Graph Rewriting-Based Model Transformation. *Electronic Communications of the EASST*, 4, 2006. GraMoT 2006 Workshop Proceedings.
- [Mer05] Paulo Merson. Representing Aspects in the Software Architecture - Practical Considerations. Early Aspects Workshop at OOPSLA 2005 <http://www.cse.cuhk.edu.hk/~elisa/EA/wp.html> (last checked June 2008), 2005.
- [MG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.
- [Mic02] SUN Microsystems. Java Pet Store - Sample Application Design and Implementation, 2002. http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/sample-app/sample-app1.3.1.pdf (last checked Nov. 2007).
- [MJ03] Joaquin Miller and Jishnu Mukerji, editors. *MDA Guide Version 1.0.1*. Object Management Group, Needham, USA, 2003. <http://www.omg.org/cgi-bin/doc?omg/03-06-01>.
- [MLLC06] Gergely Mezei, László Lengyel, Tihamér Levendovszky, and Hassan Charaf. A Model Transformation for Automated Concrete Syntax Definitions of Metamodeled Visual Languages. *Electronic Communications of the EASST*, 4, 2006. GraMoT 2006 Workshop Proceedings.
- [MO04] Mira Mezini and Klaus Ostermann. Variability Management with Feature Oriented Programming and Aspects. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 127–136, New York, NY, USA, 2004. ACM Press.
- [MS07] Mia-Software. MIA-Generation - User Guide - Getting Started. http://www.mia-software.com/fileadmin/fichiers/documentation/manual_en.pdf (downloaded July 2008), 2007.
- [Nei80] James Neighbors. Software Construction Using Components. Ph.D. thesis, University of California at Irvine, 1980. ICS-TR-160.

- [NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA Environment. In *ICSE '00: Proceedings of the 22nd international conference on Software Engineering*, pages 742–745, New York, NY, USA, 2000. ACM Press.
- [NSW⁺02] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards Pattern-Based Design Recovery. In *Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348, New York, NY, USA, May 2002. ACM Press.
- [OMG02] OMG. *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*. Object Management Group, Needham, USA, 2002. <http://www.omg.org/docs/ad/02-04-10.pdf> (last checked July 2008).
- [OMG05a] OMG. *Unified Modeling Language: Infrastructure version 2.0*. Object Management Group, Needham, USA, July 2005. <http://www.omg.org/spec/UML/2.0/> (downloaded June 2008).
- [OMG05b] OMG. *Unified Modeling Language Specification Version 1.4.2*. Object Management Group, Needham, USA, 2005. <http://www.omg.org/spec/UML/ISO/19501/PDF>, also available as ISO/IEC 19501.
- [OMG05c] OMG. *Unified Modeling Language: Superstructure version 2.0*. Object Management Group, Needham, USA, July 2005. <http://www.omg.org/spec/UML/2.0/> (downloaded June 2008).
- [OMG06] OMG. *Object Constraint Language (OCL), version 2.0*. Object Management Group, Needham, USA, May 2006. <http://www.omg.org/docs/formal/06-05-01.pdf>.
- [OMG07a] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Object Management Group, Needham, USA, July 2007. <http://www.omg.org/docs/ptc/07-07-07.pdf>.
- [OMG07b] OMG. *Meta Object Facility(MOF) 2.0 XMI Mapping Specification, v2.1.1*. Object Management Group, Needham, USA, Dec 2007. <http://www.omg.org/spec/XMI/2.1/PDF> (last checked June 2008).
- [OMG08] OMG. *MOF Model to Text Transformation Language, v1.0*. Object Management Group, Needham, USA, Jan. 2008. <http://www.omg.org/docs/formal/08-01-16.pdf>.
- [OSBE01] David Oglesby, Kirk Schloegel, Devesh Bhatt, and Eric Engstrom. A Pattern-based Framework to Address Abstraction, Reuse, and Cross-domain Aspects in Domain Specific Visual Languages. In *OOPSLA*

- 2001 Workshop on Domain Specific Visual Languages*, Jyväskylä, Finland, 2001. Jyväskylä University Printing House. <http://www.isis.vanderbilt.edu/oopsla2k1/Papers/papers.htm> (last checked July 2008).
- [Par76] David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.
- [PK07] Risto Pohjonen and Steven Kelly. Interactive Television Applications using MetaEdit+, 2007. Model-Driven Development Tool Implementers Forum 2007. <http://www.dsmforum.org/events/MDD-TIF07/> (last checked June 2008).
- [PL07] Niklas Pettersson and Welf Löwe. A Non-conservative Approach to Software Pattern Detection. *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 189–198, 2007.
- [Pre94] Wolfgang Pree. Meta Patterns—A Means For Capturing the Essentials of Reusable Object-Oriented Design. *Lecture Notes in Computer Science*, 821:150, 1994.
- [Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. *Lecture Notes in Computer Science*, 1241:419–444, 1997.
- [RA06] Mauro Regio and Víctor García Aprea. Guidance Automation Toolkit and Domain-Specific Language Tools for Visual Studio 2005: Integration Scenarios. [http://msdn.microsoft.com/en-us/library/aa905334\(printer\).aspx](http://msdn.microsoft.com/en-us/library/aa905334(printer).aspx) (last checked August 2009), Nov. 2006.
- [RBÖV08] István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. Live Model Transformations Driven by Incremental Pattern Matching. *First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008 Proceedings. LNCS*, 5063:107–121, 2008.
- [Rei05] Clemens Reichmann. Grafisch notierte Modell-zu-Modell-Transformationen für den Entwurf eingebetteter elektronischer Systeme. Ph.D. thesis, Universität Karlsruhe (TH), 2005.
- [RGF⁺06] Y.R. Reddy, S. Ghosh, R.B. France, G. Straw, J.M. Bieman, N. McEachen, E. Song, , and G. Georg. Directives for Composing Aspect-Oriented Design Class Models. *Transactions on Aspect-Oriented Software Development I. LNCS*, 3880:75–105, 2006.
- [Roy87] Winston W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society.

- [Roz97] Grzegorz Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.
- [SCC06] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional Software. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 451–464, New York, NY, USA, 2006. ACM Press.
- [SCG⁺05] Peter Swithinbank, Mandy Chessell, Tracy Gardner, Catherine Griffin, Jessica Man, Helen Wylie, and Larry Yusuf. *Patterns Model-Driven Development Using IBM Rational Software Architect*. IBM Corp., 2005. <http://www.redbooks.ibm.com/redbooks/pdfs/sg247105.pdf> (last checked July 2009).
- [Sch95] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163, London, UK, 1995. Springer-Verlag.
- [Sch06] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.
- [Sch07] Christian Schneider. CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrenkonzepten. Dissertation, Universität Kassel, 2007.
- [Sch08] Axel Schulz. Architekturstratifikation am Beispiel des Java Pet Store. Study thesis, Technische Universität Darmstadt, Dezember 2008.
- [SK04] Shane Sendall and Jochen Küster. Taming Model Round-Trip Engineering. Proceedings of Workshop on Best Practices for Model-Driven Software Development, 2004.
- [SOEB03] Kirk Schloegel, David Oglesby, Eric Engstrom, and Devesh Bhatt. Composable Code Generation for Model-Based Development. In *Software and Compilers for Embedded Systems, 7th International Workshop, SCOPES 2003, Vienna, Austria, September 24-26, 2003, Proceedings, LNCS*, volume 2826, pages 211–225, Berlin, Germany, 2003. Springer-Verlag.
- [SRF⁺05] Devon Simmonds, Raghu Reddy, Robert France, Sudipto Ghosh, and Arnor Solberg. An Aspect Oriented Model Driven Framework. In *EDOC '05: Proceedings of the Ninth IEEE International EDOC*

- Enterprise Computing Conference*, pages 119–130, Washington, DC, USA, 2005. IEEE Computer Society.
- [SSJ02] Inderjeet Singh, Beth Stearns, and Mark Johnson. *Designing Enterprise Applications with the J2EE Platform, Second Edition*. Addison-Wesley, New York, NY, USA, 2002.
- [SSK⁺07] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, and G. Kappel. A Survey on Aspect-Oriented Modeling Approaches, 2007. http://publik.tuwien.ac.at/files/pub-inf_4920.pdf (last checked June 2009).
- [Sta94] The Standish Group International, Inc. CHAOS Report 1994. http://web.archive.org/web/20060327172832/http://www.standishgroup.com/sample_research/PDFpages/chaos1994.pdf, 1994.
- [Ste06] Friedrich Steimann. The Paradoxical Success of Aspect-Oriented Programming. *Proceedings of the 2006 OOPSLA Conference. ACM SIGPLAN Notices*, 41(10):481–497, 2006.
- [SWZ99] A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES approach: language and environment. In *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*, pages 487–550. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.
- [Tat06] Tata Research Development and Design Centre. ModelMorf: A model transformer (web page), 2006. <http://www.tcs-trddc.com/ModelMorf/ModelMorf.htm> (last checked July 2009).
- [Tel08] France Telecom. SmartQVT documentation. http://smartqvt.elibel.tm.fr/doc/fr.tm.elibel.smartqvt.doc/SmartQVT_documentation.pdf (downloaded July 2008), Apr. 2008.
- [TN86] Hirotaka Takeuchi and Ikujiro Nonaka. The New New Product Development Game, 1986. *Harvard Business Review*, Jan-Feb 1986.
- [TOHSMS99] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. *21st International Conference on Software Engineering (ICSE'99)*, pages 107–119, 1999.
- [Tra06] Laurence Tratt. A change propagating model transformation language. Department of Computer Science, King's College London, Technical Report 06-07, August 2006.

- [Var04] Gergely Varró. Towards Incremental Graph Transformation in Fajaba. In Holger Giese, Andy Schürr, and Albert Zündorf, editors, *Fajaba Days 2004*, Darmstadt, Germany, 2004.
- [VB05] Markus Völter and Jorn Bettin. Patterns for Model-Driven Software-Development. In Dietmar Schütz Klaus Marquardt, editor, *Proceedings of the 9th European Conference on Pattern Languages of Programs, 2004*, pages 525–560. UVK Verlagsgesellschaft mbH, Konstanz, Germany, 2005.
- [VBJB07] Bert Vanhooff, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. Traceability as Input for Model Transformations. *ECMDA Traceability Workshop 2007*, 2007.
- [Wir71] Niklaus Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.
- [Wit96] J. Withey. Investment Analysis of Software Assets for Product Lines, 1996. Technical Report CMU/SEI-96-TR-010 <http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.010.html> (last checked June 2008).
- [WJ04] Weerasak Witthawaskul and Ralph Johnson. An Object Oriented Model Transformer Framework based on Stereotypes. 3rd Workshop in Software Model Engineering (WiSME 2004). <http://www.metamodel.com/wisme-2004/papers.html> (last checked July 2008), 2004.
- [WO06] Lothar Wendehals and Alessandro Orso. Recognizing Behavioral Patterns at Runtime using Finite Automata. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 33–40, New York, NY, USA, 2006. ACM Press.

Curriculum Vitae

- 1995 Abitur Schuldorf Bergstraße, Seeheim-Jugenheim, Germany
- 1996 - 2002 Studies in Computer Science, graduated as Diplom-Informatiker
- 2002 - 2010 PhD student in the department “Metamodeling and its Applications” lead by Prof. Dr. Thomas Kühne, Technische Universität Darmstadt, Germany
- since 2008 Senior Software Engineer at Capgemini sd&m, Offenbach, Germany