M.Sc. Thesis

Developing Interacting Domain Specific Languages

November 2007

Sander Mak
sam@solcon.nl

Supervising professor	:	Prof. Dr. S. D. Swierstra	Center for Software Technology,
Supervisor UU	:	Dr. B.J. Heeren	Dept. of Information and Computing Sciences,
Supervisor TUD	:	Dr. E. Visser	Universiteit Utrecht,
INF/SCR-07-20			Utrecht, The Netherlands.

Abstract

Domain specific languages (DSLs) and model driven software development (MDSD) both aim at raising the abstraction level for programmers, thereby enhancing both quality and productivity in software development. Many approaches exist to create and use DSLs. Unfortunately, almost all of these DSL based solutions depend on having a single, monolithic model of the domain as basis. From a software engineering point of view, this approach does not scale very well.

In this thesis project, we explore a road less travelled. Instead of starting from a monolithic model, we propose to improve this practice by splitting up different concerns over different domain specific languages. Consequently, the models expressed in these languages must together form a complete application, and therefore need to interact. This cross-language modularity provides more possibilities for reuse and allows DSLs to be narrower and more focused. Examples of this idea are occasionally found in model driven software development approaches. However, the interaction is often implicitly handled and poorly formalized. We have studied these issues from a more traditional programming language and compiler based point of view.

A case study was performed, entailing the creation of two textual domain specific languages for technical domains (and the design for a third). The first language, DomainModel, is geared towards modeling persistent data models, and targets the existing Java Persistence Architecture framework. The second language, WebLayer, is concerned with creating a web-application around such a data model, and targets the JBoss Seam framework. This prototype forms the basis of our study on the interaction amongst those languages. Through this case study, we investigate whether such DSL interaction is feasible and practical, and what the design issues are.

Contents

1 Introduction **1**
1.1 Setting the scene . . . 2
1.2 Models and abstraction . . . 3
1.3 Challenges in DSL development . . . 5
1.4 Research questions . . . 6

2 Modeling software **7**
2.1 Libraries and frameworks . . . 7
2.2 4GL languages . . . 8
2.3 Embedded DSLs . . . 8
2.3.1 Language assimilation . . . 9
2.3.2 Natural embedding . . . 10
2.3.3 Syntax macros . . . 10
2.3.4 Concluding remarks . . . 11
2.4 Language oriented programming . . . 11
2.4.1 Intentional programming . . . 11
2.4.2 Language workbenches . . . 12
2.5 Model Driven Architecture . . . 13
2.6 Our approach . . . 14

3 DomainModel DSL **15**
3.1 Language description . . . 15
3.1.1 Syntax . . . 15
3.1.2 Types and annotations . . . 16
3.2 Implementation . . . 19
3.2.1 Java Persistence Architecture . . . 19
3.2.2 Translating concepts . . . 20
3.2.3 Translating concept members . . . 22
3.2.4 Equals and hashCode implementation . . . 25
3.2.5 Semantic checks . . . 26
3.3 Concluding remarks . . . 27

4 WebLayer DSL **29**
4.1 Target libraries . . . 30
4.2 Language Description . . . 31
4.2.1 Text elements . . . 33
4.2.2 Iterative constructs . . . 34
4.2.3 Input elements . . . 35
4.2.4 Actions and forms . . . 36
4.2.5 Page and session variables . . . 37
4.3 Implementation . . . 38
4.3.1 Semantic checking . . . 39
4.3.2 Specializing generic constructs . . . 40
4.3.3 Translating pages . . . 41
4.3.4 Page navigation and data-flow . . . 44
4.3.5 Translating page elements . . . 45

4.3.6 Action language . . . 48
4.3.7 Session variables and validators . . . 49
4.4 Issues . . . 50
4.5 Concluding remarks . . . 51

5 Interaction aspects 55
5.1 Interaction between DSLs . . . 55
5.1.1 Motivation . . . 56
5.1.2 Intended usage scenario . . . 57
5.1.3 Separate compilation and interface files . . . 58
5.1.4 Interface characteristics . . . 61
5.1.5 Issues . . . 62
5.1.6 Dependencies . . . 63
5.2 Interaction between DSL and user-written code . . . 63
5.2.1 Motivation . . . 64
5.2.2 Extended types . . . 64
5.2.3 Comparison . . . 67
5.2.4 Inlined Java annotations . . . 68
5.3 Concluding remarks . . . 69

6 BusinessRules DSL 71
6.1 Language Description . . . 71
6.2 Interface . . . 73
6.3 Concluding remarks . . . 74

7 Related work 75
7.1 Model composition . . . 75
7.2 Ordina Software Factory . . . 76
7.3 openArchitectureWare . . . 79
7.4 DSLs for the web . . . 80
7.4.1 Links . . . 81
7.4.2 bigwig/JWIG . . . 82
7.4.3 WASH/WebFunctions . . . 82
7.4.4 WebObjects . . . 82
7.5 Active Libraries . . . 83

8 Conclusion 85
8.1 Reflection . . . 85
8.2 Future work . . . 87

Bibliography 87

A Implementation details 89
A.1 Syntax definitions . . . 89
A.2 Generated code . . . 91
A.2.1 DomainModel . . . 91
A.2.2 WebLayer . . . 94

B Model driven development environments 99
B.1 JetBrains Meta Programming System . . . 99
B.2 OptimalJ . . . 101

Chapter 1

Introduction

Software engineering as a discipline finds itself in a permanent state of flux. Especially when it comes to programming languages, their design and application are subject to numerous studies in both academic and corporate environments. Programming languages can be categorized into two large groups: general purpose languages (GPLs) and domain specific languages (DSLs). The former are languages that can be used to create arbitrary computer programs, whereas the latter are focussed on expressing programs on a single, smaller domain. It is the latter group that is of prime interest in this thesis.

We observe a surge of interest in high-level software development. Whether it is touted as Model Driven Software Development, Model Driven Architecture, Domain Specific Languages, or using another fashionable term, many companies are willing to invest. Mainly, the industry's need for faster turnaround times on software is the driving force behind this heightened awareness. However, oftentimes these initiatives are oblivious to decades of existing research on raising the abstraction level in software development. By contributing a comprehensive survey of existing ideas and approaches in the introductory part of this thesis, we hope to alleviate this problem. Both conceptual foundations and actual implementations are treated.

Our survey (Chapter 2) shows that many approaches to high-level software modeling exist. From now on, we refer to the collective of these approaches as *model-driven software development* (MDSD). In this thesis we want to explore a road less travelled. Many of the MDSD solutions depend on having a single, monolithic model as basis. From a software engineering point of view, this does not scale very well. Instead of starting from a monolithic model, we propose to improve this practice by splitting up different concerns over different domain specific languages. This cross-language modularity provides possibilities for reuse and allows DSLs to be narrow and focused. Consequently, the models expressed in these languages must together form a complete application. The goal of this thesis is to create and research a prototype MDSD environment following this idea, thereby studying and describing the interaction that is necessary between DSLs.

Organization

The first part of this thesis is dedicated to the introduction of the goals and possibilities of software development at high abstraction levels. In the remainder of this chapter, we introduce the key concepts and ideas of model-driven software development. Chapter 2 continues by exploring existing strategies to implement these concepts, concluding with a short description of our approach. If the reader is already familiar with model-driven software development at large, all but the last section of Chapter 2 can be safely skipped. Chapters 3 and 4 consequently describe the individual DSLs we have developed for our prototype. In Chapter 5, interaction between the DSLs is scrutinized. Chapter 6 introduces another DSL which has been investigated, building on the introduced interaction mechanism. The remaining chapters compare our work to existing work.

1.1 Setting the scene

Ever since the inception of computers there has been a constant evolution in the means to program them. Starting from processor specific machine code, programming languages have evolved to higher levels of abstraction in small steps. Along the way, focus has shifted from programming the computer itself to programming a certain high-level task. The how and when of the execution of the aforementioned machine codes have become side issues to this task, instead of leading the development. Currently, the most prevalent languages are the so-called third generation languages. These *general purpose* languages (GPLs), such as C++ or Java or Haskell, give ample means to construct software (almost) independent of the machine it runs on. However, in the face of the ever-increasing complexity of software (Dijkstra even coined the term *software crisis* [**?**]) another step in the evolution of programming languages is imminent. The designation *general* purpose language indicates that these third generation languages are suited to describe a very large class of tasks. This enormous flexibility turned out to be as much of a weakness as it is a strength. Still, computational steps (albeit at a higher level) are the units of composition in these languages. The steps of an algorithm, for example, are easily mapped onto these units. The development of large applications, however, reveals a disparity between the actual development means and the high-level requirements provided by the outside world. Yet it is this outside world that normally commissions the development of software.

Several solutions to this problem emerged, mostly out of necessity. One of them is the advent of constructs for reusability. Collections of common tasks, called libraries or frameworks are developed. The idea is to implement a specific task and do it well, and in such a way that it can be reused at a later time. Abstractions of a certain domain are captured in such a library. A related concept is *component* software [**?**], which also enables and promotes this reuse engineering technique. Still, these reusable elements are used and combined within languages that do not carry the intentions of a programmer very well. In practice this has already led to frameworks that are large and unwieldy, in which domain concepts are obfuscated by implementation issues. Performing maintenance on such systems, and assessing the reliability , or correspondence with the underlying ideas is hard. Moreover, not every desirable aspect of domain specific extensions can be captured in reusable libraries. It is, for example, in most languages impossible to add arbitrary semantic checks at compile time in user-definable libraries. Furthermore, the syntax, paradigm, and conventions of a general purpose languages impose restrictions on the forms which libraries and their usage can take. More often than not, these restrictions prohibit the *right* abstraction mechanism to be implemented, and the programmer unfortunately has to settle for *good enough*.

Standard practices (such as design patterns) and reference architectures are other aids for the programmer to tame the development complexity. From a technical engineering point of view these tools are very valuable. Unfortunately, these vocational approaches still do not bridge the gap between the abstractions of a domain and the code that implements software for this domain in a automatable manner. Expertise when to use what pattern can generally not be expressed at the code level.

All these issues lead to the realization that a more fundamental change in the software engineering toolset is necessary, if we want to continue the upward spiral of abstraction in programming. There is a large movement in computer science that approaches this problem by creating domain specific languages. A domain specific language is a language that is geared to a specific task or application domain. The major idea behind DSLs is that much can be gained if a programmer can describe software in a way that is close to the domain as it exists in the programmers mind:

Quality By abstracting away from low-level details, the potential for errors related to the act of programming itself decreases dramatically. Furthermore, domain-specific semantic checks can be performed automatically.

Productivity By restricting a programmer to a certain domain, development within this domain becomes faster and requires less effort.

Clarity Mapping ideas or requirements to an implementation becomes easier. Also, intentions of an implementation are better identifiable in the actual (DSL) source code, rather than in (most likely out-of-sync) documentation and comments.

Optimization Optimization can be delegated to the DSL implementation, leveraging domain specific knowledge.

As stated in the last point, in some cases, performance can also be improved by using a DSL. For example, optimizations can be applied because of assumptions that may be made for a certain domain, that would not hold in the general case (i.e. it will never be implemented in a GPL compiler). Given these prospective benefits, it should not come as a surprise that there is a large interest from the academic field as well as the corporate field for domain specific languages. The vast amount of available work shows that there is no single right answer to the question of how domain specific languages should be created and used. There are many different approaches to construct and implement DSLs. Even the concept of what a DSL is, is not always clearly definable. A configuration file with a specific structure constitutes a DSL for some, whereas others only use this term for a full-blown language with a dedicated compiler. The existing approaches can be roughly put into three categories:

1. Embeddings of languages in general purpose host languages.
2. Stand alone languages.
3. Visual modeling languages.

Generally, the term domain specific language refers to the second approach: stand alone languages. All of the ideas enumerated above have in common that they try to model (part of) the application at a higher level than possible in low-level GPL code. These models of a specific domain are generally translated into GPL code that implements it, either by code generation or using other techniques. Chapter 2 examines each of these approaches.

1.2 Models and abstraction

Abstraction has been the key word in the preceding introduction, but it is a very general term:

"The act of abstraction is the obtainment of the general in favor of the specific. Our human civilization rests on the power of abstraction insofar as the volume of specific cases handled by an abstraction is typically much greater than the overhead of the abstraction mechanism itself. This is especially true in software so the ability to abstract is key to any encoding." (C. Simonyi [?]).

There are many ways of encoding software, but as we have already seen, one is prevalent: general purpose language code. In this section we are going to examine many approaches that try to abstract away from this notion. An intuitive classification of these approaches will help in determining where they fit in the complete spectrum of model-driven engineering efforts. Figure 1.1 provides such a classification. For each column, one has to consider what exactly *can be* a model, and what the arrows between code and model mean. For now we will settle on the definition of *model* as an abstract representation of a program in a certain domain, and *code* as the implementing source code in a general purpose language. The arrows indicate an automated (i.e. not involving human intervention) connection between code and model, or vice versa.

On the far left, we find the currently prevalent practice: a program, its characteristics, and underlying ideas are determined solely by the code that was used to implement the program. There might or might not exist any corresponding documentation, but even if it does exist, there is no direct, tangible connection to the code. Guarantees whether these documentational artifacts are valid or up-to-date, cannot be given. Essentially, there is no model, except for the mental model in the programmer's mind. Following the reasoning of the preceding introductory section, this approach does not scale well in the light of ever growing (both in size and complexity) software projects. Software development possibly involves many people who need to understand the application and work on it effectively. On the code level, abstractions can only be expressed in terms of available constructs and concepts in a language. So, in Java one has to encode domain abstractions in terms of classes and methods, Haskell requires the programmer to encode abstractions in terms of (higher-order) functions, data types, and type classes, and so on.

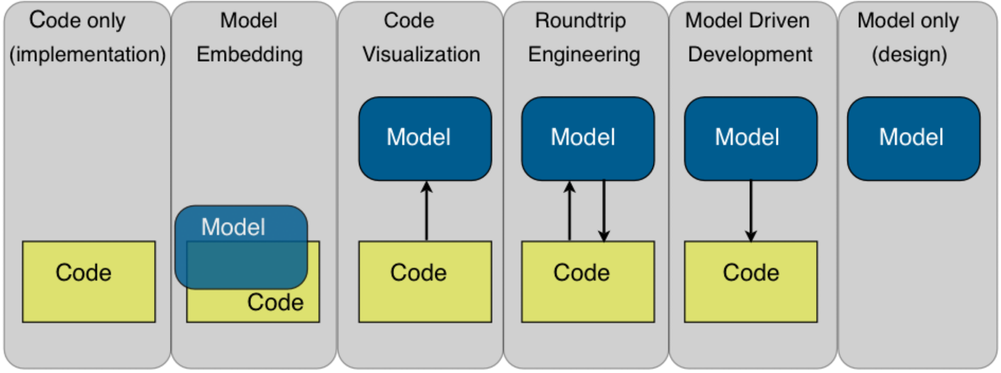


Figure 1.1: Comparison of software development methodologies

The second column, *model embedding*, indicates a next step in the growing need for abstraction: enhance the language by allowing models to be embedded inside a general purpose language. This is particularly appealing since only select parts of the application can be *modelised*, whereas other parts can be described in low-level GPL code. Section 2.3 explores the design limits of this approach, for the feasibility of this approach highly depends on the embedding method and target language. Generally, the level of abstraction of embedded models is modest compared to the approaches that follow.

The third column, *code visualization*, depicts the school of thought that code and the model should be separate entities. Moreover, the model describes the application as a whole. In this way, development teams should really reap the rewards of higher productivity, maintainability, and quality. However, code visualization cannot achieve this, since it relies on the code to present and manipulate the model. Therefore code visualization can aid program *understanding* but not program *construction*. An alternative use of this class of tools is to extract models from legacy code, and use these models as starting point for any of the remaining approaches.

Moving on to *roundtrip engineering*, we now identify a bi-directional relationship between the model and code. This implies that development can take place on both the high abstraction level, and the low implementation level. It is not hard to see that this approach is made or broken by the quality of the synchronization between these two levels. More concretely, code generation (arrow down) must respect changes made by programmers in underlying code. Conversely, the model has to reflect changes in the underlying implementation (arrow up), in order to be able to keep developing on the model level. In practice, this problem turns out to be very hard. Integration between low-level GPL code and code following from the model therefore also has the focus in this thesis proposal, though not necessarily in a bi-directional setting.

The penultimate column, *model driven development*, depicts the inverse of code visualization. Developing programs using this approach, is more of a blackbox activity. A program is encoded by creating a high-level model, whereupon this model is translated to an executable implementation. Full generation of applications from models puts a heavy burden on the modeling capabilities. It might very well be the case that developing programs using this approach provides less freedom. However, this depends very much on the quality and scope of the models that can be used, and of course on the desired degree of freedom. Our research prototype fits into this category, where domain specific languages assume the role of models.

In the last position, we find the *model only* approach. This has been the state of model driven engineering for a long while. Generally, software development is started in good style by creating a model first. However, the mapping from model to implementation is then performed manually. While the upfront design steers developers in the right direction, this is not as optimal as either of the previous two approaches. Experience gained by performing many of these mappings cannot be captured for later use in an automated fashion, as with the previous two approaches.

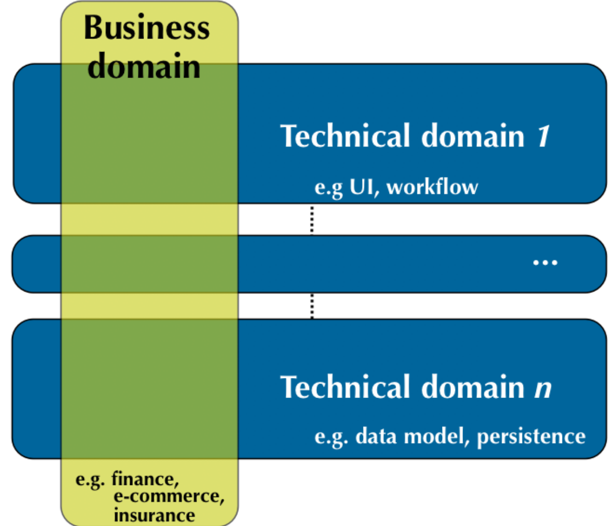


Figure 1.2: Relation between technical and business domains

An interesting link to 'Functional Design Patterns', studied by several UU Master's students [**?**, **?**] exists. In this research, structured documentation and specification of possibilities for reuse are explored, based on functional and architectural similarities between programs. However, while useful, still such an approach does not provide a generative connection from the model (functional design patterns) to an implementation. Moreover, opportunities to create such a connection are only touched upon, and not actually realized. We think our work and approach, introduced in more detail in Section 2.6 is exemplary of how information contained in a functional design pattern can be materialized as concrete software engineering tools.

One way of describing a model is to use a domain specific language. The language definition defines the restrictions imposed on the model, and each program written in the DSL constitutes a model instance. Another way of describing models is, for example, by creating diagrammatic representations. The central theme is that models can drive software development. What the shape, form, or abstraction level of models is, and how expressive these models are, what the scope of models is, is all subject to many factors. Models are depicted as monolithic entities in Figure 1.1, but this need not be the case at all. In fact, our research rejects the notion of a single model approach, as will be discussed in the following section as well as in our approach in Section 2.6. Also, the extent of the connection between model and code is an important variable to observe when comparing various approaches. Many approaches are hybrids between the described categories. Nevertheless, the categorization presented here is meant to give a solid basis for the assessment of various tools, languages, and techniques in the field of model driven software development.

1.3 Challenges in DSL development

We have chosen to express high-level models using domain specific languages. Chapter 2 gives an overview of the alternative approaches, and places this choice in a wider context. Many interesting issues exist in the area of DSL design and development. In the design process, non-trivial choices are to be made, each of which can have an effect on the usability or even viability of a DSL. These effects are sometimes obvious, but oftentimes not so. This already becomes clear when trying to give a basic definition of what constitutes a domain. Traditionally, a domain is often linked to a real world subject matter, e.g. the domain of creating software solutions for banking products or insurance brokers. Another view on domains pertains to focusing on technical areas that are part of the software engineering process. These areas typically cross cut the former real world domains, as depicted in Figure 1.2. A domain specific language implementation can be made or broken by the choice of the domain concerned.

By their nature, domain specific languages come in multitudes. You most probably need several

languages or models to create a complete software solution. This is especially true when the concerning domains are technical domains rather than real world problem domains, which is the premise of the remainder of this thesis. Having multiple DSLs raises the issue of how these languages should interact. This interaction can be described on several levels. Conceptually, a DSL must be able to reference an abstraction in a related DSL. Ideally, the modularization between interacting DSLs should follow the principle of *separate compilation*. This notion is explored in detail in Chapter 5. Furthermore such interaction should feel natural to the DSL user.

Apart from these meta-issues, finding the right abstractions for the design of a DSL is currently more of an art than it is a standard engineering practice. In this regard we will explore the similarities between library design and domain specific language design in Section 2.1. This issue becomes even more interesting when we take into account the notion of a *family* of interacting languages.

The advent of domain specific languages is based on two important principles in language design: abstraction and restriction. A general purpose language allows its users to define any program, but at increasing costs. The complexity of achieving a task increases with the complexity of this task. Domain specific languages aim at raising the abstraction level, thereby lowering the complexity of achieving a specific task. On the other hand, other tasks might become harder or even impossible to achieve in a DSL. While the imposed restrictions lead to improved productivity, several downsides can be identified as well:

- Programmers can grow frustrated when a DSL restricts them by its design.
- The domain might evolve in unforeseen ways, leaving the programmer with insufficient means to express himself.
- Certain tasks are inherently better suited to a solution expressed in a GPL.

A route that is often taken to remedy these problems, is to start modifying the code that was generated from a domain specific language. Unfortunately, this is not a solution at all, since it leads to several new problems. First of all, any changes to the generated code will be lost when the code is regenerated, after a change at the model level. Second, a programmer has to be familiar with every detail of the translation scheme in order to edit the generated code in a meaningful way. However, the original problems cannot be ignored, for they prohibit the adoption of DSLs. We think a good DSL design therefore also involves creating well-defined extension points from the DSL to the target language. Our prototype contains such extensions, and as such these will serve as vehicle to investigate this type of interaction. Furthermore, we believe that having multiple interacting DSLs, all using a well defined interface mechanism, opens up the way for new DSLs to be *plugged in*. In this way, new domain abstractions can be introduced, if doing so is justified.

1.4 Research questions

We observe that domain specific languages can improve the software engineering process, with respect to the challenges outlined in this section. Using several domain specific languages together to model complete applications is not unprecedented. However, the interaction aspects seem to be under exposed. Summarizing, we can identify two research themes, with corresponding research questions:

1. Interaction between DSLs
    - What interaction patterns can we identify?
    - How can we implement DSL interaction?
    - How does interaction affect the design of a DSL?
2. Interaction between DSL and host language code
    - When and how should interaction between DSL and GPL code be implemented?

By creating a prototype environment we want to answer these research questions. Furthermore, we firmly place our work into a wider context, which we believe is an important contribution. The field of MDSD research is wide and fragmented, yet this thesis aspires to provide a solid overview.

Chapter 2

Modeling software

Before moving on to the details of our prototype (Chapter 3 and further), we provide a survey of principal ways to raise the abstraction level in software development. We will see that, although the described approaches share this common objective, vastly different techniques, methods, and ideas can be identified. The last section of this chapter (Section 2.6) introduces the concrete setup and design goals of the prototype we developed during the course of this thesis project.

2.1 Libraries and frameworks

We have seen that models can abstract from code in several ways. However, we also noted that the code itself can contain more abstract notions, in the form of libraries and frameworks. A library is a reusable component in a language, providing functions that perform certain (domain specific) tasks. A framework is defined in the same way, only at a higher level. Frameworks usually guide the programmer towards using a certain architecture, possibly utilizing and integrating several libraries. We will now explore the link between domain specific languages and libraries and frameworks.

Libraries themselves form a prime target for (embedded) domain specific languages. By extension, we believe that a framework is a prime target for multiple interacting DSLs. When the usage of a library becomes too complex or too verbose, (E)DSLs (which are discussed in Section 2.3) can provide a significant improvement. Often, libraries have implicit semantics layered over the bare semantics of the language it is implemented in. This can be hidden state, unexpected semantics pertaining to the ordering of function calls, or complex rules with respect to the initialization of libraries. Learning to use a library in some cases is equally hard as learning a new (E)DSL, without getting the advantages.

However, caution should be taken when designing a DSL with a specific library implementation in mind. Details from the underlying library should *leak* through to the higher layer as little as possible. A famous example is an SQL query with the clause `WHERE a=b AND b=c AND a=c`, which runs faster than with the equivalent (at least to the SQL *user*) clause `WHERE a=b AND b=c`. This behavior is due to the way the query is translated to an execution plan using indices in the DBMS. A good abstraction layer is intuitive, however, in some cases having to know what is going on at a lower level is unavoidable.

Frameworks encode domain specific knowledge (i.e. best practices and architectural choices) in a GPL. A trend that can be discerned is that more and more of these frameworks rely on external specifications to be instantiated. Most Java and .Net frameworks rely on these external files, predominantly in XML format. These configurations are then interpreted at deployment time creating an instance of the framework, often extensively using reflectional facilities of the language. It is clear that domain specific languages can play a big role in eliminating these sort of constructions. That is, by letting the programmer express the variability of a framework in a domain specific language, which can then generate optimized code for that specific instance. This transition can almost be equated to the shift from an interpreted to a compiled programming language.

Program families are another important notion when talking about libraries and frameworks. A program family denotes a set of programs that have enough in common to regard them as a common system with controlled variations. Frameworks exploit the same sort of commonality between programs.

Analyses to assess whether programs can be grouped into a program family, or supported by a framework, are quite mature. These analyses also form the foundation of DSL design (e.g. commonality analysis [?]). A natural conclusion is that DSL design, framework design, and program families are closely related concepts. And indeed, this is shown by case studies [?]. An interesting observation that we would like to add, is that frameworks typically encode layered architectures. Each layer is responsible for a small aspect of the complete application. These aspects, in turn, are often supported by a library. This maps nicely onto our classification of technical domains (Figure 1.2, page 5). Hence, if libraries map to DSLs, then frameworks potentially map to our proposed multiple DSL approach. Our prototype confirms this thought, as we will show in this thesis.

2.2 4GL languages

A first DSL based answer to the so-called software crisis, as sketched in Section 1.1, came in the mid-eighties. Many so-called fourth generation languages (4GL) emerged. A good overview from this era on the design goals of 4GLs is given by Alan Tharp [?]. One of the most ambitious objectives is that 4GLs should be *simple, English-like, non-procedural languages.* Programming by writing natural text was hailed as the ultimate goal. Furthermore, these languages are intended to reflect high-level domains, corresponding to business domains in our definition from Section 1.3. 4GLs are an early instance of the 'model driven engineering' approach (i.e. the fifth column of Figure 1.1 on page 4). Some examples of 4GLs are:

Progress 4GL Language to define data-entry (and related) applications.

Oracle Forms Language to create management reporting applications on top of Oracle databases.

Mathematica Language for mathematicians.

These are examples of 4GLs that are still in use. However, many 4GLs did not survive the test of time. We can identify several reasons for the demise of these languages. First of all, the focus has been on business domains. Effectively, this means that a 4GL language should take care of the complete implementation, from storage to logic, as well as presentation, and everything in between. Especially in the eighties, creating such a language with an appropriate mapping to an implementation was highly non-trivial. Library support for all sorts of tasks in the underlying implementation language was not as excellent as it is nowadays. Also, even when creating a language from the business perspective, changes in technology can still have a profound impact. Think, for example, of moving from console applications to graphical applications, to web-applications, etc. To facilitate these changes, the languages had to be changed and extended as well. The danger of this approach is that 4GL languages gradually turn into general purpose languages. All of the examples mentioned above succumbed to this temptation to various degrees, thereby surrendering at least some of the advantages of having domain specific languages.

The most important downside of nearly all 4GLs is the fact that they are strongly connected to software components of the vendors of the 4GL. This point is made explicit by observing the examples, where two already contain a direct reference to large database companies. Vendor lock-in is not only present in the target languages and platforms, but also in development environments. We can conclude that the lack of an open environment works detrimental for all but the largest vendors. Applicability and development of 4GLs depend too much on solitary, external parties to really gain acceptance as a good software engineering practice.

2.3 Embedded DSLs

An embedded DSL is the counterpart of the stand-alone DSL (such as a 4GL language). The class of embedded DSLs (EDSL) consists of languages defined *within* a general purpose language. In this way, the embedded language inherits the infrastructure of the *host*-language. Creating an EDSL therefore eliminates some of the start-up costs attached to stand-alone DSLs. Another advantage is that the power of the host-language can be mixed arbitrarily with the DSL embedding. The degree of success of an embedding highly depends on how amenable the host-language is to these embeddings. There can be many inhibiting factors for creating embedded domain specific languages, depending on the approach

taken and the host-language concerned. We will consider these issues and evaluate them in this section. The approaches examined in this section all constitute an instance of the second column of Figure 1.1.

When a stand-alone DSL partially implements functionality for a certain domain, the question arises how this can be incorporated into a complete software product. One approach is to reference artifacts generated from this DSL (e.g. invoke a parser generated by Yacc), or to derive from them (e.g. using subclassing). The disparity between domain specific definition in the DSL and usage in the actual code can lead to confusion or inconsistencies. If a language is directly embedded, the interplay between the domain specific concepts and the host-language is much clearer. Note that when a (combination of) stand-alone DSL(s) caters for complete program generation, this problem is also avoided.

Three different embedding strategies can be distinguished. One strategy is to create a new language (syntax) definition for the embedded language combined with the host-language. Consequently, this hybrid syntax is parsed and translated into a pure host-language program, in pre-processor style. Section 2.3.1 explores this strategy.

The second strategy is to fit the embedding in the existing syntax of the host-language [**?**], an approach often taken by domain specific languages for Haskell and Ruby. Limiting factor in this approach are the features offered within a language. Operator overloading, as a single example, allows for more natural and expressive embeddings. A language such as Java lacks many of these desirable features, generally making it a bad choice for this type of embedding, whereas the two mentioned language have excellent provisions. Section 2.3.2 treats this type of embedding.

As a third strategy, we identify embeddings through syntax macros. This strategy is related to the first one, but in this case the host-language itself provides the environment for the assimilation of an embedding. This last embedding strategy is handled in Section 2.3.3. We will now proceed with exploring each of these strategies in more detail.

2.3.1 Language assimilation

Language assimilation in the case of domain specific languages means that domain specific abstractions within a general purpose language are reduced to general purpose abstraction implementing the desired functionality. In effect, a language $\mathcal{L}$ is extended to language $\mathcal{L}'$ by adding domain specific extensions. Then, a pre-processor assimilates $\mathcal{L}'$ sources into $\mathcal{L}$ sources, which in turn can be compiled by the standard $\mathcal{L}$ compiler.

An example of this approach is the MetaBorg project [**?**, **?**]. The approach taken by MetaBorg is to create a language definition for a domain, and compose it with the syntax definition for the target language. This is done by exploiting the modular Syntax Definition Formalism [**?**], in which arbitrary language declarations can be mixed. The next step is to write an *assimilator* for this mixed language definition, which translates constructs of the domain specific embedding to constructs in the target language. In this translation, semantic checks can be performed to ensure safety. Also, this checking phase offers large potential for user-friendly error reporting. The translation itself is described with term rewrite rules in Stratego/XT [**?**]. One of the advantages is that the embeddings are completely independent from the language features provided by the host-language. As long as the combined syntax can be parsed, and the domain specific constructs can be expressed in host-language constructs, all is well. Parsing hardly is a problem due to the generalized LR parser that is used. A prime example of the MetaBorg approach is the embedding of a Swing User-interface Language in Java [**?**].

If a syntax definition for the host-language is available in the SDF format, creating an embedding following the MetaBorg approach is surprisingly straightforward. However, a drawback of this approach is the lack of compositionality of the embeddings. For each combination of domain specific language embeddings, a composed syntax definition must be created. Furthermore, the assimilations potentially influence each other, introducing non-determinism in the absence of a clear ordering with arbitrary combinations of embeddings. Also, assimilated code may conflict with assimilated code of another embedding.

2.3.2 Natural embedding

With a natural embedding we mean that the domain language can be expressed within a general purpose language, without extending the syntax of this language. Features such as operator overloading/introduction and higher-order functions can make this approach feel sufficiently natural for a domain. Essentially, natural DSL embeddings are libraries disguised as languages. It is clearly the most economical way of introducing a domain specific language, since one can focus on the domain only, without having to worry about the infrastructure. Compilers, IDEs, and other tools can still be used, even though they are not aware of the domain specific embedding. Especially in the Haskell community, many of these embedded languages [?] are developed. These languages are sometimes also called combinator libraries, because often they rely on introducing new operators to express domain constructs. Advantage of having such an embedding is that the full power of the host-language can be combined with the domain abstractions. On the other hand, this can be a danger as well, since domain specific programming relies on restriction as much as it relies on domain expressivity.

A downside of the natural embedding approach is that errors are reported directly by the host compiler, which has no knowledge of domain specific constructs. There are exceptions, such as the Helium compiler[1] which can incorporate specialized typing rules and error messages for domain specific constructs. However, such a mechanism is generally not available. Furthermore, the resulting syntax often is not the most desirable syntax when looking from the domain perspective. A striking example of this argument is the usage of regular expressions in Java. Regular expressions can be regarded as a domain specific language for string matching and manipulation. Java supports regular expressions through an API, which accepts a regular expression described in a string. The format is almost the same as for Perl regular expressions (arguably a de facto standard), but at the same time you are confined to the way strings are handled on the Java platform. This means that many symbols that are used in regular expressions, need to be explicitly escaped. Most unfortunately, the escape symbol of regular expressions is one of these symbols. This leads to the undesirable situation in which a programmer is doing some domain specific programming (namely creating a regular expression), but meanwhile all the peculiarities of the host-language have to be considered as well. Even worse, regular expressions inside a string are not checked for validity until the actually run-time API call. So, even though the *'natural'* embedding is possible in this case, it is not always the best choice. The suitability of this approach therefore highly depends on the capabilities of the host language.

2.3.3 Syntax macros

The main idea behind syntax macros is that a programmer can instruct a compiler to translate an input pattern to an output pattern, without requiring any changes in the compiler or the compilation pipeline. There are many ways such functionality can be implemented, ranging from very safe but limited to principally unsafe but very expressive.

The archetypical example of extending a language through macro facilities is provided by (Common) Lisp. This language has a very minimalistic syntax, but allows for powerful extensions as it is a *programmable programming language*. In Lisp, the code itself can be generated and altered at run-time, because data and code share the same representation (it is a *homo-iconic* language). The concrete Lisp syntax resembles the abstract syntax very closely. Ultimate expressivity is gained by this mechanism. Macros work on the internal program structure. Using functions and macros, Lisp can be extended to cover a specific domain. However, the power of Lisp has proven to be too much for many. In practice, this freedom of unstructured extension did not serve domain specific development very well.

Also, Lisp macros allow for the definition of local transformations in a natural way. Creating transformations that use global information is much harder. Having the ability to specify transformations using global information is, however, very important when high-level domain specific languages are concerned.

At the other end of the spectrum, far less powerful macro approaches can be identified. Macros in C are nothing more than a textual search and replace action on C-sources. A similar example is the

[1] Helium is a compiler for a subset of Haskell.

templating system in C++. Macros in this templating system are expanded without any type checking, deferring these checks to the compilation of the expanded code. Naturally, this leads to very confusing error messages in terms of the (generated) expanded code. This is a problem that is strongly related to the one-way nature of these macro expansions. A macro facility for Haskell which overcomes this shortcoming, by providing a two-way mechanism, is created by Rommers [?]. This is a powerful way of extending syntax in a type safe manner, but it requires a custom Haskell compiler. As such, it voids the advantage of lower startup costs by using existing infrastructure for the translation of an embedding.

All in all, most macro facilities offered by general purpose languages (if any) are too lightweight for an integrated domain specific programming approach. Either the translation lacks safety, or requires a large effort to ensure safety. Some of the most popular languages (e.g. Java) do not even offer a macro facility.

2.3.4 Concluding remarks

Embedding DSLs is a convenient method to lift code to a more abstract level at minimal costs, using any of the three approaches. The trade-off between these approaches mainly consists of finding the right balance between implementation effort, usability, and safety. We observe that natural embeddings are mostly used for domains that are quite *narrow*. A small, convenient abstraction is injected into a GPL, e.g. parser combinators or a query language. While this raises the abstraction level of parts of a program, it does not completely cover the goal of raising the abstraction in application development. This can be seen as a blessing or a curse. On the one hand, the full power of the host language is available around the embedding. On the other hand, this may exactly be something that needs to be avoided to guarantee quality and productivity improvement in some settings.

Depending on the environment and approach taken, combining different EDSLs is potentially problematic. This holds in particular for approaches that use pre-processors to translate the embedding, i.e. not for natural embedding. Conversely, in the case of natural embeddings it is generally not possible to perform specialized semantic checking on embedded languages, since the compiler generally is unaware of the extensions. A sufficiently expressive type system helps in this regard. However, the more expressive such a general purpose type system is, the harder it is to provide legible (or even domain specific) error messages.

2.4 Language oriented programming

Language oriented programming (LOP) is another idea to bring specialized languages into software development. The central idea is to integrate (domain specific) language development into every software development process. It should be as easy to define a new language for use within a project as it is to add a new class or module in present practice. This differs from the previous approaches we have seen, where DSL are developed independently of actual development projects, to be used many times over. It is not very surprising that none of the current LOP approaches entirely succeed in the aforementioned goal, but the direction is clear.

2.4.1 Intentional programming

The term intentional programming (IP) [?] is first used by Simonyi at Microsoft Research in the mid-nineties. It forms the basis for most of the current work on LOP, as it was the first instance of a language oriented approach. Simonyi observes that languages as they exist are fundamentally flawed in several ways, including:

- There is a mismatch between the (low) level of programming languages and the (high) level of development goals.
- Existing languages are by default not compatible with each other.
- Domain experts cannot take part in the actual development process in any shape or form, because of the language barrier.

Most of his observations amount to the same conclusion: general purpose languages are not very well suited for a programmer to express his intentions in. Undoubtedly the intentions get obfuscated by having to mix them with trivialities and implementation details. Simonyi postulates that *'every good comment in source code indicates a shortcoming in the language'*. Much less can a domain expert without traditional programming abilities participate in the development process, beyond voicing his intentions in a way that is not machine processable.

Intentional programming envisions a development environment in which there is no single traditional programming language. Instead, many small languages may be developed in order to enable quick construction of the application at hand. This language development is split into two phases:

1. The programmer and/or domain expert create the appropriate abstractions.

2. The programmer links these abstractions (together forming a 'language') into the intentional programming environment.

After these steps, a program can be developed in terms of the abstractions of a specific domain. However, it is still up to the programmer to create a consistent mapping that translates these abstractions in a meaningful way. In order to make these languages interoperable they all map to a standardized, common representation. This representation is called the intention tree, which is a normalized representation of computations. This tree can have many views, at various levels of abstraction. Editing a program involves editing the tree, or rather one of its synchronized views. In fact, each of the aforementioned languages are particular views on the intention tree. An editor that works on the structure of those trees or views is an essential part of IP. The notion of editing program code as unstructured text is abandoned, thereby avoiding the difficulties of parsing a mixture of textual representations. By working in such a structured editing environment, the well-formedness of programs is forced by construction. This contrasts with the traditional compiler approach, where well-formedness is forced by corrective error messages. The front-end part of a compiler (that does the parsing and builds an AST) can be omitted in the IP environment, since the programmer is constructing the AST directly in the structured editor. Whether this approach really fits the programmer's perspective remains to be seen; editors that force the programmer into a certain mode of operation can be experienced as a nuisance. This complete departure from current practice could be a severe liability to the adaptation of this new technique.

The first prototype of IP was constructed at Microsoft Research. Unfortunately, this work was shelved after several years because it was too disruptive for Microsoft's .Net strategy, which promotes a more traditional paradigm. Currently, Simonyi's company Intentional Software is working on a new implementation. A paper re-iterating the principles of IP and containing a concrete example was presented at OOPSLA 2006 [?]. Besides this paper, not much information is available on this project and it remains to be seen whether the ambitions of IP can be materialized.

2.4.2 Language workbenches

In his article 'Language workbenches: the killer-app for domain specific languages?', [?] Martin Fowler explains a vision akin to what intentional programming tries to achieve. However, his description is implementation agnostic. In fact, the intentional programming environment *is* an implementation of a language workbench, albeit an ambitious one.

In the description of language workbenches, however, healthy skepticism is exercised towards the *lay*-programmers argument. This is the idea that a domain expert without programming knowledge will be able to write his own programs. While it offers a nice perspective, this goal might be unattainable. Fortunately, this does not obliterate the need for language workbenches (or domain specific languages for that matter) at all. Having a programmer work on a level closer to the understanding of a domain expert is just as valuable, giving all the advantages discussed in the introduction of this proposal.

A strong emphasis is put on the fact that evolution now can take place on two axis: a program can evolve, but so can the languages that were used to implement it. This calls for a strong notion of refactoring, to reflect changes of a *meta*-language in the development of applications in a language

workbench. This danger is also described by Klint et al. [?, ?]. We have surveyed a concrete implementation of a language workbench call the 'JetBrains Meta Programming System'. A comprehensive description of this environment can be found in Appendix B.1.

2.5 Model Driven Architecture

Model Driven Architecture (MDA) itself is not an environment to create (visual) DSLs. It is a standardization effort by the Object Management Group[2]. The idea is to create standards which all meta-modeling environments should adhere to, in order to ensure interoperability between model driven development environments and their input models. A meta-model is a model that describes well-formedness for model instances (i.e. a model conforms to a meta-model, as a source file conforms to a grammar). MDA describes the following ingredients for the model driven approach:

Platform Independent Model The actual high-level description of an application (UML).

Platform Definition Model A model of a specific architecture (e.g. CORBA, .Net or a web-environment).

Platform Specific Model Executable description of an architecture instance.

Given a Platform Independent Model, which corresponds to a meta-model, and a Platform Definition Model, a Platform Specific Model has to be generated. How this is achieved, is left as an exercise to the implementor of the tool. The many acronyms associated to MDA already indicate that this standard is quite heavyweight. Even more standards than mentioned here are bundled into MDA. In practice, not many actively used MDSD environments make use of the complete set of MDA standards. This can be mostly attributed to the (uncalled-for) complexity.

The modeling languages and environments based on and related to MDA are mostly of visual nature. A prevalent visual modeling language is UML. However, many tools also employ their own visual language, since UML as a general purpose modeling language can suffer the same problems as general purpose programming languages. In particular, the complexity and sheer size of the language in practice makes it hard to succinctly express domain concepts. Furthermore, UML does not natively support any module system. Consequently, managing models suitable for code generation for complete applications is cumbersome at best. On a more practical note, version control of UML models is a complex endeavour.

Still, many approaches do use UML, since a vast amount of editors is available for this generic language. Building a custom visual editing environment is costly, but generally deemed worthwhile because of the abovementioned issues.

Some concrete modeling environments that follow the MDA ideas (but do not necessarily implement the associated standards) are:

- Eclipse Modeling Framework (EMF), an open source effort. The Graphical Modeling Framework provides means to create custom graphical editors. Compatible with (but not built on) UML.

- Microsoft DSL tools, a proprietary visual DSL development environment. Discussed as related work in Section 7.2.

- OpenArchitectureWare, a tool to transform and check EMF/GMF (and various other types of) models. Discussed in Section 7.3.

These projects form the current state of the art in graphical modeling coupled with code generation, and also have a track record (though short, compared to the approaches in previous sections) of actual usage in production environments.

[2]OMG is a consortium of corporate and academic members that also standardized UML and CORBA.

2.6 Our approach

A prototype has been implemented to research whether creating multiple interacting DSLs is feasible, practical, and desirable. The prototype is comprised of three technical domains, each having their own domain specific language. The choice of domains for our prototype is largely immaterial, since we are interested in the principles behind the language design and interaction rather than the domains themselves. However, to avoid a contrived case study, we selected domains that benefit from a DSL approach regardless. Concretely, we model web-application development for the Java platform. The existing libraries and frameworks for this domain show that it is extremely difficult to concisely and accurately express the concepts of this domain through a natural embedding (i.e. native Java libraries). This can be attributed mainly to Java's narrow set of general abstraction facilities, as was discussed in Section 2.3 and Section 2.1 of this chapter. Therefore we develop several stand-alone DSLs, meanwhile showing how this improves the ability to model web-applications.

A well known and prevalent architecture for web-applications is the three layer (or tier) model [?]. It consists of a *data*-layer at the base, which we model with a DSL called *DomainModel* (Chapter 3). The top layer is concerned with *presentation* aspects, such as navigation between pages, presenting forms for data entry etc. In between these two layers resides the *business* logic of web-applications. Business logic is quite a fuzzy term, we define it as the code responsible for data manipulation, thereby enforcing business rules and policies. The DSLs for these layers will target existing `Java` libraries.

We restrict the scope of the DSLs to modeling *data-intensive* web-applications. A precise, agreed-upon definition of data-intensive web-application is not directly available. An intuitive definition suffices: data-intensive means the web-application is structured around a data model and consists of an interface to manipulate the underlying data-model, according to a set of pre-defined rules. The paper 'Design principles for data-intensive web site' [?] contains an excellent exploration of the design space for such applications. Practical examples that spring to mind are internal data-processing applications of companies to support tasks such as managing customer information, or a student administration system. These types of applications are built in large quantities on a daily basis, and therefore constitute an excellent *program family* to be supported by DSLs.

In particular, we stress the fact that it is not the intention to create production-quality, all encompassing implementations of the DSLs. Rather, we are interested to see how modular or independent our DSLs can be, and how we can shape the interaction between different DSLs and between DSL and host language code.

Chapter 3

DomainModel DSL

The objective of the DomainModel DSL (which we will refer to as *DomainModel* from now on) is to provide a language to specify persistent data models for arbitrary domains. Such a domain model forms the foundation for any type of application that works on domain data, whether it is a desktop application, command line tool, or web application. In this context, the adjective persistent means that instances of a domain model must be storable in a non-volatile environment, such as a database or filesystem.

Since our languages are targetted at the object-oriented language Java, *DomainModel* follows the object-oriented paradigm as well. Prime goal is to provide a language that provides a flexible way of modeling a domain, whilst keeping the user oblivious to the implementing machinery aiding the persistence of the domain model. From a modeling perspective, *DomainModel* bears some resemblence to UML class diagrams. This language independent, visual formalism is widely known in the data modeling world. Even though *DomainModel* is not a visual language, some elements of UML class diagrams can be distinguished, which we elaborate upon later. In general, it is good practice to leverage existing formalisms or known mechanisms for a domain in a relevant DSL. In this way, users of a DSL can reuse their intuitive knowledge of a domain when using or learning the DSL. From a functional point of view, however, UML class diagrams are mostly used as formal input for database schema design. Our focus, on the other hand, is on generating an implementation that handles the persistence of an object-oriented domain model, from the code level to the database level.

The *DomainModel* language will be introduced by describing its structure and by inspecting a concrete example of a domain model. Subsequently, we analyze the implementation of the DSL, where the semantics of the language is solidified by giving the translation to Java.

3.1 Language description

3.1.1 Syntax

We introduce the *DomainModel* language by analyzing its concrete syntax. Figure 3.1 shows a condensed version of the concrete syntax in EBNF notation. Bold, quoted symbols denote a terminal whereas italic names indicate the non-terminals of the language. Definitions of trivial non-terminals (i.e., *ucase_ident* and *lcase_ident*) are left implicit, as is the definition of *java_annotation*. The latter is introduced later, during the description of the implementation in Section 3.2. The actual syntax definition (in SDF format) used in the implementation can be found in Appendix A.1.

Domain constitutes the root element of a *DomainModel* definition, containing the name associated with this domain model definition and a list of concepts defined within this domain model. A concept definition describes the structure of entities that can exist in the domain model. On this abstraction level, one can view concepts as being equivalent to value-types[1], since a data structure is defined, without any possibility to define operations on this data structure. Each concept describes the structure of an entity in our domain model. This description is comprised of concept member definitions, where

[1]Also known as *structs*.

domain	::= **'domainmodel'** *lcase_ident* *concept* ⋆	domain definition
concept	::= **'concept'** *ucase_ident* **'{'** *member*⋆ **'}'**	concept definition
member	::= *lcase_ident association type* { '('*annotation* {',' *annotation* }⋆ ')' }**?**	concept member
association	::= '→'	reference
	| '◇'	composition
	| '**::**'	built-in
type	::= *builtin_type*	*DomainModel* type
	| *ucase_ident*	concept type or extended type
	| *enum_type*	enumeration type
	| '[' *ucase_ident* ']'	list type
builtin_type	::= **'String'**	*DomainModel*
	| **'Integer'**	built-in types
	| **'Boolean'**	
	| ...	
	| **'Date'**	
enum_type	::= **'{'** *enum_dec* {',' *enum_dec*}⋆ **'}'**	enumeration type
enum_dec	::= *quoted_text* **':'** *all_ucase_ident*	single enum value
annotation	::= **'unique'**	*DomainModel*
	| ...	annotations
	| **'name'**	
	| *java_annotation*	concrete Java annotation

Figure 3.1: Simplified syntax definition for the *DomainModel* language

such a definition contains the name, type, and meta data of the member. A type either refers to a built-in *DomainModel* type or to a user-defined concept, or lists of these, or defines an enumeration type. Furthermore, a concept member can be either a *value-type* (e.g., String, Int), a *reference* (to another concept) or a *composition* with another concept. The semantics of these modifiers will be discussed shortly. Last, a concept member has optional annotations providing meta data. Figure 3.2 lists the available *DomainModel* annotations.

3.1.2 Types and annotations

We now move on to a concrete example of a domain model, presented in Figure 3.3 on page 17, in order to explain the concepts behind this high level domain modeling language. In the example, a concise domain model for a blog is defined. It consists of four concepts:

User Central concept in this domain. The `User` concept has a list of `BlogEntry` concepts.

BlogEntry Describes a blog posting, which can contain tags and replies in addition to the actual posting.

Tag Encapsulates the name of a tag.

Reply Describes a reply to a blog post, linked to a certain `User`.

Annotation	Description
unique	Every instance of a Concept must have a distinct value for the *member* carrying this annotation: $\forall\ c1,c2 :: Concept : c1.member == c2.member \Rightarrow c1 == c2$
name	Member is used in canonical representation of instances of the enclosing concept.
required	A non-null value must be assigned to this member before persisting.

Figure 3.2: Available *DomainModel* annotations

```
domainmodel blog

concept User {

  name        :: String       (unique, name)
  email       :: Email
  blogEntries <> [BlogEntry]

}

concept BlogEntry {

  title    :: String  (name, required)
  abstract :: Text
  contents :: Text    (required)
  date     :: Date    (name)
  tags     -> [Tag]
  replies  <> [Reply]
  category :: {"Technical" : TECH, "Other" : NONTECH}

}

concept Tag {

  tagName :: String

}

concept Reply {

  contents :: String
  user     -> User   (name)
  date     :: Date   (name, @Column(name="reply_date"))
  level    :: { "Nice reply!" : GOOD
              , "Average"     : AVERAGE
              , "Not good."   : BAD }

}
```

Figure 3.3: Concrete example of *DomainModel* definition

Observing the concept definitions, we see that most concept members are value-types, (using '`::`' syntax). Furthermore, concept members with a type referring to (a list of) other concepts are either a reference ('`->`') or a composite ('`<>`') member. The latter means, that the lifecycle of the member is tied to the lifecycle of the entity (concept instance) that contains this composite member. In other words, when an entity is deleted, this deletion cascades to all composite members. In our example, deleting a `User` entity entails deleting all its blog postings as well, since the list `blogEntries` is defined as a composite member within the `User` concept. Note that this is a transitive process, deleting each `BlogEntry` of a user also deletes its composite replies, and so forth. When defining a reference member, however, the lifecycle of the member is independent of the enclosing concept instance. Note that all members with a *DomainModel* built-in type by default behave as composite members, since there is no notion of an independent entity for these types. Still, there are use cases for having references to these built-in value-types. This is exemplified by the `tags` of a `BlogEntry`, where tags should be shared between blog posts. By wrapping the built-in `String` type in a concept definition (thereby promoting it to an entity), we can model this.

The reference/composite distinction is somewhat similar to the distinction between primitive and reference types in `Java`. However, under the surface every member is implemented by a reference type, as can be seen in the description of the implementation in Section 3.2. Later, we will see (Section 4.3.2) that the reference/composite designation can also be used to steer more user-interface oriented concerns, in the WebLayer DSL.

Regarding the types used in the example, we can distinguish the four different alternatives as portrayed in Figure 3.1. Besides the value-types, familiar to many programming languages (such as String and Integer), there are also *extended* types available in the *DomainModel* language. An example of such a type is found in the `email` member of the `User` concept. The type of this member is not drawn from the list of built-in types (as given in Figure 3.1). Rather, it is an *extended* type, meaning that it derives from a built-in type while it adds additional semantics. In this case, `Email` derives from `String`, and adds data validation to ensure that only valid email addresses are stored in this member. An elaborate discussion of how these extended types are defined is provided in Section 5.2.2, since it constitutes an instance of interaction between DSL and host language code. For now we only assume their existence.

The enumeration type, as defined for the `level` member in `Reply`, defines three valid values for this type. An enumeration is useful for situations in which the domain designer wants to restrict the values of a member by design to a small number of options. The quoted text of an enumeration type member specifies the user-friendly rendering of the enumeration value, whereas the identifier specifies the programmatic name of the member.

In the example, we can also see the usage of *DomainModel* annotations. The `name` member of a `User` is declared to be unique, meaning that no two instances of the `User` concept can have the same name. Furthermore, the `name` member has a name *annotation* as well. Note that the first `name` is a user-defined identifier, whereas the second name is a *DomainModel* construct (compare to `title` member of `BlogEntry`). This annotation indicates that this field is the visual identifier for instance of this object. Whenever a string representation of the instance is necessary, the string representation of the contents of this field will be given. It is possible to annotate multiple fields with name, leading to a concatenated string representation of all annotated members. In the case of the `BlogEntry` concept, the annotated member is a String itself. However, this is not mandatory, as can be seen in the `Reply` concept. There, the `name` member with type `User` is annotated with name. The mechanism works transitively, i.e., the actual representation is delegated to `User`. Hence, the `name` of the user is part of the string representation of `Reply`, as is the date. If no name annotation is provided, a default implementation is generated for the concept. The details of this default are described in the implementation section. Members with any type can have the name annotation, with one exception: list types. The name annotation is intended to create a short, meaningful string representation of a concept instance, and lists are not likely to contribute to this goal. Note that the name annotation is most useful when at least one of the name annotated members is defined to be unique as well, to ensure that the string representation indeed is a good identifier for the instance.

All in all, creating a domain model is structurally quite similar to creating an object model in, for example, `Java`. However, the constructs offered by *DomainModel* allow the programmer to think only

in terms of concepts, members and associations. In the example, no low-level details with regard to how the model is stored are exposed. Also, identity management and linking of instances of concepts (as in primary/foreign keys) is kept implicit. The DSL definition coupled with the compiler encapsulates all these details, as described in the following section. We also note that inheritance between concepts, while possible in the target library, is not implemented in *DomainModel*. This is purely a practical choice to restrict the implementation effort and we see no inherent obstructions to adding this functionality later, if desired.

3.2 Implementation

We provided a description of a fully declarative language for specifying persistent domain models in the previous section. In this section, we will look into the implementation of the compiler for this language. By inspecting the transformation steps from *DomainModel* source to Java output, we can establish a better frame of reference for the semantics of the language. First, we will briefly introduce the target API and library.

3.2.1 Java Persistence Architecture

With the advent of object-oriented languages, the mismatch between data storage facilities and object-oriented data usage in programs grew. Persistent data storage is predominantly done in relational databases, for they provide many desirable properties with respect to safety and data integrity. Relational database systems have been thoroughly studied and developed. Many libraries emerged, trying to solve the disparity between object-oriented data models and relational storage. A detailed study of such object-relational mapping schemes is well beyond the scope of this thesis. We suffice with the observation that these schemes are practically usable, though not all theoretical problems of the OO/relational mismatch can be solved consistently. Hibernate is an O/R mapping library for Java, which grew to be a *de facto* standard in Java application development. The basic idea is that objects are extended with O/R mapping hints, and the library takes care of the actual object persistence. The following contributions of the Hibernate library can be identified:

- Generic solution to the O/R mismatch.
- Transparent support for different relational backends.
- Automatic generation of database schema's for domain models.
- Caching layer for objects, to minimize stress on database.
- Transactional safety on the object level.
- Fully transparant optimistic locking implementation using object versioning.
- Database connection pooling.
- Lazy loading of collections inside objects.
- An object query language.

Since these facilities are advantageous to almost any data-intensive program, the approach taken by Hibernate was standardized in 2006 and called Java Persistence Architecture (JPA). Simultaneously, the O/R mapping hints could also be provided through Java annotations, instead of through a proprietary Hibernate XML format. The resulting API is normalized and managed by the Java-community, and implementations (such as Hibernate) all conform to this API. Therefore, *DomainModel* emits code that leverages the JPA API, while using Hibernate as specific implementation provider on the deployment side. This way, the *DomainModel* compiler does not depend on a (theoretically unreliable) third-party, but uses features specified by Java itself. Such third-party dependencies should always be minimized in a generative setting, though this is not always feasible.

3.2.2 Translating concepts

The *DomainModel* compiler is implemented using rewrite rules, defined in the strategic rewriting language Stratego [?]. We have set up an infrastructure in which we can traverse the abstract syntax tree of a domain model, and transform input terms into Java code. This Java code in the rewrite rules can be expressed in concrete syntax, yet the transformation takes place using the abstract syntax. By virtue of this mechanism (as described by Bravenboer et al. [?]), the syntactical validity of the target code is verified by the compiler of our infrastructure. This approach contrasts with many other code generation infrastructures, where emitting code is equivalent to building an unstructured string representation. In this section, examples of such rewrite rules are given to illustrate the mechanisms used to implement the *DomainModel* compiler.

As is normal for a compiler, the *DomainModel* compiler works in distinct phases:

1. Semantic checking phase, verifying (amongst other things) that:
   - every type used is a legal *DomainModel* type or is defined in the domain model, and
   - every member in a concept has a legal name.
2. Code generation phase, where:
   - each concept is mapped to a Java class, and
   - an XML configuration file is created for deployment, and
3. Additionally, an interface file is emitted.

The semantic checks will be revisited later in this chapter. In this section we focus on the code generation phase, while deferring the explanation of the interface file mechanism to Chapter 5. We illustrate the translation scheme of the *DomainModel* compiler by looking at the translation of the `BlogEntry` concept as found in Figure 3.3. A condensed version of the resulting Java code can be found in Listing A.1 on page 91 and further. The omitted class members (as indicated in the comments in Figure 3.4) pertain to infrastructural necessities. For example, the optimistic locking mechanism of the JPA library is activated, and a domain-specific implementation overriding the standard `equals` method is provided. Some of these class members are introduced in Section 3.2.4. For a complete overview of the generated class members we again refer to Listing A.1.

Code generation is driven by a syntax-directed traversal of the input AST. A domain model contains multiple concept definitions, each of which is translated into a Java class, using the transformation rule as presented in Figure 3.4. In this rule, italic identifiers indicate meta-variables of the transformation, and a star indicates that a variable is a list. For brevity, capitalization of the contents of a variable is assumed to be performed when the identifier is capitalized, rather than by calling an auxiliary strategy to perform this operation. The code, resulting from the application of this transformation rule to the `BlogEntry` concept from Figure 3.3, is given in Appendix A.2.1.

In order to control the name space in which the code is generated, the package qualifier begins with the meta-variable *prefix*, which can be set using a compiler flag. Also, the name of the domain model (*dm_name*) is incorporated into the package structure. Then, a class is introduced that will implement the *DomainModel* concept we are translating. The annotation `@Entity` is an indicator for the JPA library that this class represents a persistable entity. Next, an empty default constructor is generated, as demanded by JPA. Furthermore, we also generate a constructor accepting values for all concept members, assigning them to their respective fields. These fields and their respective `get/set` methods are computed by mapping the rule `translate-member` over the list of members in the concept. The result of this translation (*translated_members*) is then spliced into the class we are generating. We describe this rule in the subsequent section.

In short, each concept member is translated into a:

1. private class member variable of the correct type, and
2. `get/set` method, annotated with the appropriate JPA annotations.

```
translate-concept :

Concept(conceptname , members*)
  -> compilation-unit
     |[
         package prefix.dm_name.domainmodel;

         import java.util.*;
         import javax.persistence.*;

         public class Conceptname implements java.io.Serializable {

            public Conceptname() { }

            public Conceptname(fs*) { as* }

            private Long id;

            @Id @Generated
            public getId() { return id; }

            protected setId(Long id) { this.id = id; }

            translated_members*

            // other class members omitted for brevity

         }
     ]|
where translated_members* := <map(member-to-classbodydecls)> members*
      (fs*, as*) := <map(member-to-formalparam-and-assign); unzip> members*
```

Figure 3.4: Transformation of a concept

Also, an `id` member variable and corresponding `get/set` methods are added, representing the system identity of the object. The annotations `@Id @Generated` instruct the JPA implementation to use this field as the primary key in the database representation of the object, using an automatically generated key. This mechanism can be used to track the immutable identity of objects without depending on (volatile) data in user-defined properties, nor on the memory location. The latter is the default Java mechanism for tracking object identity, however, the memory location is different each time an object is retrieved from storage and therefore is not adequate. This phenomenon will be elaborated upon further when looking at the generated `equals` implementation.
The comments in Figure 3.4 indicate that there still is more to the translation of a concept. Additional methods that are generated for a concept are discussed in Section 3.2.4 and further.

Previous versions of Hibernate persistence necessitated the implementation of a Hibernate specific interface, or the usage of post-compilation bytecode enhancement in combination with an external XML mapping file in order to make an object persistable. With the advent of JPA, however, the class only has to adhere to the JavaBean[2] principle to achieve this. That is, a property of an object is formed by a getter/setter method pair, and a default (empty) constructor for the object is present. In this way, data is encapsaluted in private fields and access is mediated through the corresponding getter and setter, which can be intercepted by the JPA implementation. The O/R mapping information can be provided through Java annotations rather than through an XML file. Consequently, the generated code only depends on the standardized `javax.persistence` API.

[2]Specification available at: `http://java.sun.com/products/javabeans/`

```
translate-member :

ConceptMember(membername, NativeType(), dm_type, dm_annotation*)
  -> class-body-dec*
     |[
         type _membername;

         @Basic
         annotation*
         public type getMembername() {
            return _membername;
         }

         public void setMembername(type _membername) {
            this._membername = _membername;
         }
     ]|
where annotation* := <translate-annotations> dm_annotation*
    ; type        := <translate-type> dm_type
```

Figure 3.5: Transformation of a concept member

3.2.3 Translating concept members

The translation of a concept member depends on its type and the kind of association. A transformation rule for concept members with a built-in type is given in Figure 3.5. In the `BlogEntry` example, the first four members match this rule. While its structure is directly derived from the implementing Stratego rule, many details (mostly infrastructural) have been omitted for the sake of clarity. First thing to notice is that the member variable is prefixed with an underscore. By doing this, we do not have to disallow the usage of reserved `Java` keywords as *DomainModel* identifiers. There are over 50 reserved keywords, some of which are entirely plausible when modeling a domain (e.g. `abstract` when modeling publications, or `class` when modeling a school administration). However, the name exposed by the `set/get` methods can still be the unmodified *DomainModel* identifier, since it is part of a larger name. Furthermore, a translation from the *DomainModel* type to a corresponding `Java` type is performed using the strategy `translate-type`. Figure 3.6 defines the mapping function $\mathcal{T}$ that is being applied in this translation. Native, built-in types (e.g. String, Integer) are translated directly to their `Java` counterpart. A special case is the mapping of *extended* types. The mapping for these types is provided by the user at compile time. Specifics of this mechanism are described in Section 5.2.2. Effectively, the mapping function's domain is augmented with mappings found in the extended type definitions. In Figure 3.6, `Email` constitutes such a mapping. Types referring to another *DomainModel* concept are translated to the type of the corresponding newly created class. A *DomainModel* list type is translated to the `Java` collections List type, parameterized with the translation of the element type. Note that our DSL definition is more restrictive than the mapping function indicates: the type argument of a list type cannot be another list type, as encoded in the syntax definition. This choice was made since the O/R mapping does not support directly nested lists, and because nesting can still be achieved by wrapping the nested list in an entity. In fact, only other concepts may be used as parameter.

Furthermore, the annotations to instruct the JPA library are placed on the getter method. In this case, the annotation `@Basic` is provided, indicating that a mapping to a native type in the database should be performed. Note that the actual mapping to a database type is left to the library, since it also depends on the database used. When translating a composite member with a type referring to another *DomainModel* concept, the `@OneToOne` annotation is generated instead. In the case of a reference member, the `@OneToMany` annotation is emitted (which subsumes the `@OneToOne` mapping).

An example translation of a reference member with type `[Tag]`, can be seen in Listing 3.1, where the generated code for concept member `tags` is given. We can see that the JPA annotation in this case

```
private List<Tag> _tags;

@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
public List<Tag> getTags()
{
  return _tags;
}

public void setTags(List<Tag> _tags) { this._tags = _tags; }

public void addToTags(Tag b_0)
{
  if(this._tags == null)
  {
    this._tags = new ArrayList<Tag>();
  }
  this._tags.add(b_0);
}

public void removeFromTags(Tag c_0)
{
  if(this._tags != null)
  {
    this._tags.remove(c_0);
  }
}
```

Listing 3.1: Translation of `tags` concept member

is `@ManyToMany`, indicating that an intermediate table will be used by the JPA library to model the list relation. Within this annotation, we specify that events that update list members directly cascade to these members. Note that removal from the list, however, does not cascade to a delete since we are dealing with a reference member. If this member was specified to be a composite, `CascadeType.REMOVE` would have been present as well.

In addition to the getter/setter and member field, two extra methods are generated for members with a list type. The list is lazily initialized upon addition of the first element using the addToTags method, thus avoiding `NullPointerException`s in code that uses this concept. Removing an element is also safe with respect to null pointers when using the generated removeFromTags method. While this defensive coding style is desirable, oftentimes it is not used because of the verbosity, or simply because of ignorance. Since we are in a generative setting, we can easily adopt these kind of best practices, providing robust code.

Another variation on the presented mapping is the case of an enumerated type. The translated code for the `category` member is provided in Listing 3.2. Both the programmatical identifiers (`TECH, NONTECH`) and the user-friendly labels are used in the translation. Again, the rule for translating a member with an enumerated type is an extension to the rule presented in Figure 3.5, since a type declaration for a type safe `Java` enumeration is generated as well. This enum declaration follows a well known pattern, in which the user-friendly representation is stored as a member variable (in this case the string `label`) of the enum. The possible enum instantiations are being fixed in the declaration, as can be seen on the third line of Listing 3.2. Values of this enum type can now be referred to as `Enum_category.TECH` or `Enum_category.NONTECH`. The private constructor precludes any other instantiations at a later time. Furthermore, two ways of retrieving the user-friendly label of an enum value are generated, one overriding the standard `toString` method, and `getLabel`, following the JavaBean getter idiom. Note that the annotations on the getCategory method are different as well, since we have to instruct the JPA library to map values of the enum to their ordinal (integer) value for storage.

Translating annotations

So far, we have emitted JPA annotations based on the type and association of the member. These annotations are used to steer the O/R mapping process. Additionally, annotations given in the *DomainModel* program must be translated. This translation corresponds to the application of the `translate-annotations` strategy in Figure 3.5. The (optional) list of annotations on a concept mem-

```
public enum Enum_category
{
  TECH("Technical"), NONTECH("Other");

  private String label;

  Enum_category (String label) { this.label = label; }

  public String getLabel() { return label; }

  @Override public String toString() { return label; }

  public Enum_category[] getValues() { return Enum_category.values(); }
}

private Enum_category _category;

@Basic @Enumerated(EnumType.ORDINAL) public Enum_category getCategory()
{
  return _category;
}
```

Listing 3.2: Translation of `category` concept member

$$
\begin{array}{lcl}
\mathcal{T}[\![\text{ String }]\!] & \mapsto & \texttt{java.lang.String} \\
\mathcal{T}[\![\text{ Date }]\!] & \mapsto & \texttt{java.util.Date} \\
\mathcal{T}[\![\text{ Boolean }]\!] & \mapsto & \texttt{java.lang.Boolean} \\
\mathcal{T}[\![\text{ Integer }]\!] & \mapsto & \texttt{java.lang.Integer} \\
\mathcal{T}[\![\text{ Long }]\!] & \mapsto & \texttt{java.lang.Long} \\
\mathcal{T}[\![\text{ Double }]\!] & \mapsto & \texttt{java.lang.Double} \\
\mathcal{T}[\![\text{ Email }]\!] & \mapsto & \texttt{java.lang.String} \\
\mathcal{T}[\![\textit{ Type }]\!] & \mapsto & \textit{package}\texttt{.domainmodel.}\textit{Type} \\
\mathcal{T}[\![\text{ [} \textit{Type} \text{] }]\!] & \mapsto & \texttt{java.util.List.} < \mathcal{T}[\![\textit{ Type }]\!] >
\end{array}
$$

Figure 3.6: Available types and their mapping

ber may contain both special purpose *DomainModel* annotations, as well as inlined Java annotations. The latter are introduced into the *DomainModel* language to allow for the customization of generated code. Users can add specific annotations to fine-tune deployment parameters. An example can be seen in the domain model given in Figure 3.3, where the `date` member carries the Java annotation `@Column(name="reply_date")`. By default, the JPA library uses the member name for the corresponding database column. If, for example, we have reuse an existing database schema, this annotation can steer the mapping in the right direction. The possibility to inline Java annotations is implemented by importing the syntax definition for these annotations into our DSL syntax definition. This is possible since we can use the Java 1.5 syntax of JavaFront in a modular fashion, as can be seen in Appendix A.1. Thus, it is possible for the user of *DomainModel* to fine-tune the generated code, without having to edit the generated source files.

Translation of the annotations takes place in two steps:

1. Each annotation is either
   - mapped to a corresponding Java annotation using the function $\mathcal{A}$ (Figure 3.7),
   - or copied verbatim if it is a Java annotation.
2. The resulting list of Java annotations is post-processed.

Post-processing the list entails first eliminating any duplicate annotations, and occurrences of $\perp$. Duplicates can arise when the user provides the same Java annotation in the *DomainModel* source as the one that will be generated based on the type and association of the concept member concerned. Second, the annotations may be of the same type, but have different contents, e.g. `@Column(unique=true)` is

$$
\begin{array}{lcl}
\mathcal{A}[\![\text{ unique }]\!] & \mapsto & \texttt{@Column(unique=true)} \\
\mathcal{A}[\![\text{ required }]\!] & \mapsto & \texttt{@Column(required=true)} \\
\mathcal{A}[\![\text{ name }]\!] & \mapsto & \bot
\end{array}
$$

Figure 3.7: Mapping annotations

generated, and `@Column(name="cname")` is provided by the user. In this case, the bodies of the annotations will be merged (i.e. `@Column(unique=true, name="cname")`). Annotations that are not part of the domain of $\mathcal{A}$ result in a compile time warning. This is the reason for `name` to be in the domain, even though it maps to $\bot$, which is filtered out afterwards.

```
@Override public String toString()
{
  StringBuilder name = new StringBuilder();
  if(_title != null)
  {
    name.append(_title.toString());
  }
  name.append(" ");
  if(_date != null)
  {
    name.append(_date.toString());
  }
  return name.toString();
}
```

Listing 3.3: Implementing `toString` using name annotations

One exception in the annotation translation scheme is the name annotation. For this annotation, there is no local, one-to-one mapping. Instead, we need to collect all members that carry the name annotation and use these to implement the `toString` method. Figure 3.3 shows how a string representation is build from the annotated members, in this case `title` and `date`. For each of these members, their `toString` method is transitively called and the result appended to the string representation. Spaces are interspersed when multiple members are part of the string representation. If none of the concept members carries a name annotation, a default is generated in which the name of the enclosing concept and the identifier (the automatically generated `id` member) are combined.

Our *DomainModel* implementation shows that having a well defined, and preferably small and controlled extension point in a DSL to the target language can be a good interaction pattern. Precautions have to be taken, however, to eliminate conflicts between generated and inlined code. Section 5.2.4 investigates this in more detail. Another danger is that the DSL is too tightly coupled to the target language. Since the Java annotations that can be inlined in *DomainModel* only pertain to deployment and configuration issues, which have good defaults if omitted, we believe this represents a fair trade-off between higher coupling and control over the generated code.

3.2.4 Equals and hashCode implementation

Object identity is an important issue in Java programming. Many standard library implementations depend on the correct implementation of the `equals` method, which is responsible for establishing the external identity of objects. A correct implementation adheres to the (informal) contract defined in the language's documentation. In short the implementation must be:

reflexive, symmetric, transitive and *consistent*.

The latter property is a bit awkward compared to the other well defined properties. Consistency in this context means that multiple calls over time should return the same value, given that none of the values used in the `equals` body have changed (in other words, it should be a true deterministic *function*). A thorough description of all the problems associated with this contract is given by Stevenson et al. [?].

For example, the implementation of collections in the Java standard library all hinge on the correct implementation of the `equals` and `hashCode` methods. If the contract is broken, undefined behavior surfaces, such as objects that seem to be magically dissappearing from lists and so on. However, the correctness of implementations for these functions cannot be verified by the Java compiler, and requires the programmer to guarantee the correctness. Since the default implementation (provided in the universal super class `java.lang.Object`) uses the reference comparison (the == operator), it satisfies the contract trivially and is sufficient most of the times.

```
public @Override boolean equals(Object o)
{
  if(o != null && o instanceof BlogEntry)
  {
    Long own_id = this.id;
    Long other_id = ((BlogEntry)o).getId();
    if(own_id != null)
      return own_id.equals(other_id);
    else
      return this == o;
  }
  else
    return false;
}

@Override public int hashCode() {
  return getId() != null ? getId().intValue() : super.hashCode();
}
```

Listing 3.4: `equals`/`hashCode` implementation for `BlogEntry`

However, this standard approach fails in the setting of the persistence library we are using. Objects are instantiated by the library, based on persistent information stored in the database. We can retrieve the same object multiple times in a program. Semantically, these objects are equal, but when using the standard `equals` method, they are deemed to be different since they are in separate memory locations. In order to prevent the aforementioned unintended behavior, we generate an alternative implementation as shown in Figure 3.4. This approach is better suited to our intended domain. The idea embodied in this implementation is that the JPA library assigns an immutable, unique identifier to each concept. We use this knowledge to base the object equality on this identifier. However, simply delegating to the `equals` implementation of the identifier (which is of type Long) will not suffice. The problem is, that the identifier only is assigned to a concept when it has been saved to the database by the library. When a new concept instance is constructed, and it has not been persisted yet, the identifier value is null. Comparing null values using `equals` always returns false, which would lead to the odd situation that a new object is not equal to itself (no reflexivity). Therefore, we revert to the reference comparison implementation in the event that the identifier of the concept is null, which can be seen in the inner `else` branch in Figure 3.4.

The `hashCode` method is used to efficiently store and retrieve objects in hashing data structures. It must return a native `int` that represents the hash of the object. It should satisfy the contract `o1.equals(o2)` $\Rightarrow$ `o1.hashCode() == o2.hashCode()`. The hash need not be unique (but the better the hashing scheme, the better the performance of data structures). We use the identifier's value, converted to a native `int`, as hash value. Since a `long` consists of 64 bits as opposed to 32 bits for an `int`, the hash will eventually wrap to zero. However, this is not a problem. Again, we must revert to the original hashing strategy when the identifier of the object has not been assigned yet.

It is clear that using the JPA library correctly spans beyond just knowing and calling the API. We are able to encode specific knowledge with respect to the handling of object identity into the DomainModel DSL compiler, using properties that result from our translation scheme.

3.2.5 Semantic checks

The presented transformations in the previous sections all work by the assumption that the input model is well-formed. However, this well-formedness does not come for free: it is checked in a separate phase,

before the actual compilation. In this phase, we check the following constraints:

1. A concept name may not clash with the name of a *DomainModel* built-in or extended type.
2. Each type used, must be defined (either as built-in or as concept type).
3. Each concept name must be unique, and within a concept each member name must be unique.
4. A member name may not be 'id' or 'versionNum' (since these are used in translation scheme).
5. Unknown annotations are ignored, and a warning is emitted.

These checks are fairly standard, and resemble the traditional semantic checking phase of a GPL compiler. However, we are also able to report more domain specific issues to the user of *DomainModel*. These issues pertain to the usage of annotations on concept members. For example, having a unique annotation on a concept member with an enumeration type is suspicious, since this limits the amount of possible instantiations of a concept to the number of values defined in the enumeration type. Furthermore, we can disallow the usage of the name annotation on a concept member with list type. And, the usage of a unique contraint on such a member is prohibited, since the library does not support this (but the JPA library can checks this at run time only). Whereas the checks enumerated earlier come for free in the `Java` compiler, more domain specific checks such as described cannot be given at compile time. We believe this is a strong feature of DSLs.

3.3 Concluding remarks

With *DomainModel* we implemented a completely declarative language, which is compiled to a fully functional persistent domain model, using the standardized `Java` persistence API. Technical, domain-specific knowledge with regard to the usage of this API is encoded in the *DomainModel* compiler, and virtually no implementation details are visible at the language level. One exception is the possibility to inline `Java` JPA annotations in *DomainModel* code, for users who are familiar with the compilation target. This allows for fine- tuning of the generated code, without having to alter it after generation. However, in typical cases this feature is not needed.

Having a terse, declarative language at a high abstraction level brings many advantages. First of all, when looking at the raw lines of code for our example, we observe a major increase in lines of code, going from the *DomainModel* code to the implementing code. Typically, we observed this increase to be at least a factor ten. Developing and refactoring a domain model in `Java` code necessitates coordinated changes in several locations to change a single aspect (i.e., a concept member name), whereas in *DomainModel* all information is in a single location. Besides the raw line count, another advantage is that our compiler generates default implementations for standard functions (such as `equals/hashCode`) using knowledge of the application domain. Many programmers are not aware of the issues involved, and consequently encounter erratic behavior due to not addressing the issues. The usage of the JPA library is abstracted away from, leaving the programmer with the task to actually model the domain data without worrying about what mapping hints and configuration should be provided.

Furthermore, we can identify mistakes in metadata at compiletime, that would otherwise lead to run-time errors by the JPA library. This semantic checking phase is implemented independently from the generation phase, allowing for reuse when another backend needs to be implemented. Alternative backends, such as native Hibernate or a `.Net` equivalent[3] can be implemented following the same relatively straightforward translation scheme. In the beginning of this chapter we also identified some overlap with UML class diagrams. Theoretically, it is possible to use a UML class diagram editor as an alternative front-end to our compiler. This would involve engineering a way to include the appropriate metadata for concept members in these diagrams, as well as serializing diagrams to a format processable by our compiler. One advantage would be that such an editor creates a graph, whereas our language is mimicking a graph-like structure (as does almost any programming language) using symbolic references. On the other hand, issues like version management of models become harder, whereas in our textual language any source control tool can be used.

[3] This would require a (currently unavailable) JavaFront like infrastructure for a `.Net` language in Stratego though.

Having different backends for the *DomainModel* language raises the question of what the semantics of the language are. In this chapter we introduced the semantics by describing intentions and effects of language constructions, as well as by outlining a translation scheme towards the Java language (leveraging JPA libraries). The Java language has a well understood operational semantics [?], yet it is the knowledge encoded in the libraries we target that really forms the basis of our understanding. However, we are not able to formalize these high-level semantics, neither for the JPA library nor for our DSL.

Chapter 4

WebLayer DSL

In this chapter we are moving to the top layer of a typical web-application stack, which is the presentation layer. A typical web-application stack in Java web development is depicted in Figure 4.1, and is loosely based on the Model-View-Controller architecture. With the previously described *DomainModel* DSL, we are able to model the database layer of the application stack (Model) in a high-level manner. We wish to achieve the same for the remaining layers. For the time being we will disregard the middle layer, and focus on the notion of web-applications that purely manipulate data. A large class of applications can suffice with solely basic CRUD[1] actions, without having to invoke any specialized business logic. In Chapter 6 we return to the middle layer. Our goal in this chapter is to define a language in which we can express the views necessary to manipulate a domain model created using the *DomainModel* DSL. Combining these two languages, it must be possible to define a complete and directly deployable web-application.

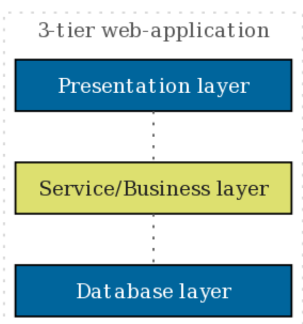


Figure 4.1: Typical web-application architecture

Typical concerns in the presentation layer relate to showing data, editing and adding data as well as defining flow between screens in a web-application. An important aspect of page flow, besides simple navigation, is data flow between different pages. Ultimately, every web-application runs on top of the HTTP protocol, regardless of the language it was created in. Since the HTTP protocol is a stateless protocol, offering only a request/response cycle, state and data flow are inherently difficult to control. One of the main features of the *WebLayer* language is that it abstracts over common solutions (or rather, workarounds) and their intricacies. Moreover, the complex and brittle usage patterns of web development libraries for the Java platform in general will be addressed in the implementation of *WebLayer*.

[1] Create, Retrieve, Update and Delete

4.1 Target libraries

Before proceeding with the language description, we first inspect the libraries the *WebLayer* language is going to target. There is a rich variety of web frameworks and libraries available. We deliberately forgo the basic libraries such as Java Servlets, since these are too low-level. These standards still form the basis of the current flock of web development libraries. Generating towards them would constitute replicating all efforts that went into the more high-level libraries.

The libraries we elected to use are:

- Java Server Faces (JSF)
- Facelets
- Ajax4JSF (A4J)
- JBoss Seam (Seam)

Each of these libraries is actively supported and widely used in the Java community. For the presentation layer, JSF is quickly becoming a standard. JSF defines a component based mechanism for creating web pages. The library components basically wrap around HTML elements, for example, by providing a systematic way of creating databindings between Java objects and HTML form elements. Databinding is based around a *lifecycle* (Figure 4.2 [?]) that is applied for each request to the JSF web-application.

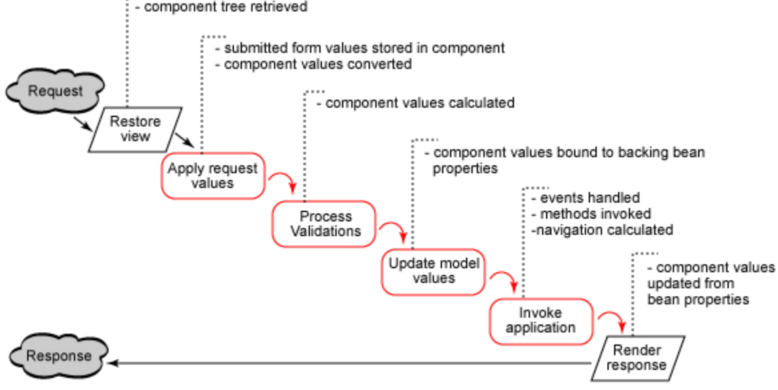


Figure 4.2: JSF lifecycle definition

Without going into the details of this lifecycle, we can already establish that development using JSF is both powerful and intricate. Every phase of this lifecycle is extensible and adaptable. However, due to this flexibility in its design, using the JSF framework turns out to be complex. Some properties, forced by its operational model (such as: every request must be a POST request in the application), also limit the flexibility and applicability a great deal. The Facelets libraries extends the capabilities of the JSF framework, providing for example a templating mechanism and adding missing components. Furthermore, the strong coupling between JSF and Java Server Pages[2] is abandoned in favor of an interpreted, XML based approach involving custom tag libraries. Ajax4JSF supplies an implementation of the asynchronous in-page messaging paradigm, while still trying to fit in with the JSF lifecycle.

Integrating the aforementioned libraries, which together form just one possible permutation out of many alternatives, is no small task. JBoss Seam recognizes this deficiency and provides a glue layer between combinations of these (and other) libraries. Using Seam reduces boilerplate code that would otherwise be necessary, and introduces an alternative means of relating Java code and view code expressed using Facelets and JSF tags. Still, creating a basic application requires a large effort, involving activities ranging from selecting actual library distributions (easily more than 20 `jar` files in a basic application) to setting up configuration files for these libraries. All this before even a single line of code

[2]The optionally compiled, dynamic server side scripting language for Java.

can be written and executed. Then, development commences, revealing another problematic issue: all of the discussed libraries introduce many dynamic (interpreted) mechanisms on top of normal, statically type safe Java code. Section 4.3 inspects these mechanisms when discussing the implementation of the *WebLayer* language, but obviously this is not a desirable situation.

In the light of the complications discussed, an interesting aspect of Seam is that it comes bundled with an application called `seam-gen`. Using `seam-gen`, a developer can set up a skeleton for a Java web-application by answering a few questions. Furthermore, entity classes (similar to *DomainModel* output) can be generated by reverse engineering an existing database scheme. This generator was created to give developers a headstart, addressing some of the aforementioned issues. However, it only addresses the initial setup, since it is a one-time generation approach. Actual development takes place by editing the generated skeleton. There is no model involved in the usage of `seam-gen`. Still, the existence of such a basic generator in itself is an indicator that a high-level, generative approach seems to be unavoidable in the context of current web development frameworks.

In our opinion, the libraries listed in this section represent the state of the art in current Java web-application development. All the issues discussed, do not pertain to these libraries exclusively. Each of the existing alternatives exhibits similar problems, most even to a higher degree. It should be noted that there are sound reasons to use these libraries, despite their shortcomings. In particular, these libraries are mature, run on a scalable architecture and have proven themselves in production environments. Our motivation to develop the *WebLayer* DSL is to create a prototype that abstracts over the technical details as discussed, providing an expressive high-level language for the presentation layer of web-applications. In doing so, it is not our goal to unleash the complete power of all underlying libraries in the DSL. Rather, the prototype should show that it is beneficial to create a language that captures principled and common usage patterns of these libraries, and the ideas they embody. Moreover, it must be able to leverage models created using *DomainModel* as seamlessly as possible, thereby relieving the DSL user of having to know any implementation details of the interaction between the frameworks involved.

4.2 Language Description

Again, we first inspect the concrete syntax of the *WebLayer* language. A definition is provided in Figure 4.3, following the conventions introduced in Section 3.1.1. The actual syntax definition used in the implementation is provided in Appendix A.1.

weblayer	::= **'weblayer'** *lcase_ident* *import*⋆ *body_elem*⋆	weblayer preamble
import	::= **'using'** *lcase_ident lcase_ident*	import DSL statement
body_elem	::= *page* | **'var'** *ucase_ident lcase_ident*	top-level elements
page	::= **'initial'**? *ucase_ident* '(' { *param* {',' *param* }⋆ }? ')' **{** *page_elem*⋆ **}**	page definition
param	::= *ucase_ident* *lcase_ident*	formal parameter

Figure 4.3: Condensed syntax definition for the *WebLayer* language

A *WebLayer* definition consists of a header, specifying the name of the module and import statements. These statements specify which other DSL definitions we link to. The first identifier refers to the name of the DSL, and the second to the name of the actual definition. Although the syntax definition of imports is completely free-form in order to allow for future extension, currently only `domainmodel` and `businessrules` are recognized as DSL identifiers by the *WebLayer* compiler, though the latter is

described in Chapter 6 but not implemented. Following the import statements, page definitions may be provided. Page definitions model the contents of distinct pages in the web-application. Also, *session* variables may be declared at the top level, which are further explained in Section 4.2.5. Each page definition has a name, zero or more *page parameters*, and a body consisting of *page elements*. The optional **initial** keyword is used to designate the page that will be shown as the first page of the web-application. Page parameters are used to model data-flow throughout the application. A parameter is scoped over the page it is declared in, and can have any concept type introduced in the *DomainModel* module that is imported. Obviously, the initial page of an application may not depend on incoming page parameters, and there can be only one initial page in a *WebLayer* module. The non-terminal *page_elem* in Figure 4.3 is deliberately left unspecified, as it has many alternatives. Each of these alternatives is introduced by example in the remainder of this language description.

Figure 4.23 (page 53) shows a page generated using *WebLayer*. It is taken from a larger *WebLayer* module that builds upon the `blog` domain model, which was introduced in the previous chapter (Section 3.1.2). The source code for this page is shown in Figure 4.4. The first two lines of code form the header of the module, with the `using` clause indicating that the `blog` *DomainModel* definitions are imported. On the generated webpage shown in Figure 4.23, a single `BlogEntry` is shown and users can add replies to the entry. It is instructive to compare the *WebLayer* source and the resulting page, to see how the definitions match the output, from top to bottom. This page will be our running example for the introduction of the *WebLayer* language constructs and their implementation. Note that in each subsequent figure containing EBNF rules for the *page_elem* non-terminal, we are extending it rather than redefining it.

```
weblayer blog

using domainmodel blog

page ViewBlog(BlogEntry be, User u){

  var Reply r

  header(be.title + " (written on " + be.date + ")")

  text(be.contents)

  table Tag t in be.tags { "Assigned tags" -> t.tagName }

  "Reply to this post:"
  form( input(r)
        action("Add reply", r.user = u; be.replies.add(r); be.save())
      )

  text("Replies for post " + be.title + " :")

  for Reply r in be.replies { show(r) }

  navigate("Home", Blog(u))

}
```

Figure 4.4: *WebLayer* page definition

page_elem	::= **'text'** '(' *text_expr* ')'	normal text
	| **'header'** '(' *text_expr* ')'	header text
	| **'navigate'** '(' *text_expr* ',' *nav_binding* ')'	page navigation
	| **'show'** '(' *qualified_ident* ')'	generic show
	| **'show'** '(' *key_value+* ')'	show
text_expr	::= *text_expr_part* {'+' *text_expr_part*}⋆	list of text parts
text_expr_part	::= *quoted_text*	literal text
	| *qualified_ident*	reference
quoted_text	::= ' " ' *text* ' " '	quoted text
nav_binding	::= *upper_ident* '(' *qualified_ident* {',' *qualified_ident*}⋆ ')'	page reference
key_value	::= *quoted_text* '->' *page_elem*	key-value pair

Figure 4.5: Textual page elements

4.2.1 Text elements

Text is the basic building block of web pages. Hence, *WebLayer* offers constructs to add both static and dynamic blocks of text to a page. The corresponding constructs are presented in Figure 4.5. In its simplest form, literal text can be placed on a page:

```
text("Reply to this post:")
```

In Figure 4.4 we can see that in the case of literal text, the enclosing `text` element is optional, since the standard interpretation of a text expression directly embedded in a page definition is a `text` element containing this expression. Alternatively, dynamic elements can be added to a text element. This can be achieved by using *fully qualified* identifiers (fqi). These identifiers take the form of `x` (just a single variable name), or `x.y.z`, where `x` is a variable, and `y` is a concept member of the concept type of `x`. Consequently, `z` is a concept member of the concept type of `y`. Fully qualified identifiers can be used in many other language constructs as well, as a means of navigating through a concept to indicate the member of interest. Furthermore, a text element can be composed of different parts, concatenated using the + operator. A combination of literal and dynamic text parts looks like this:

```
header(be.title + " (written on " + be.date + ")")
```

Note that we use the `header` element rather than the `text` element, as it prints the expression in a larger font. The type of the fqi expressions is immaterial in the context of a text element, since any type in our system can be automatically coerced to a string. In the case of a concept type, this coercion entails the usage of the `name` annotation, as introduced in Section 3.1.2.

A special case of text on a web page is a hyperlink. In *WebLayer*, navigation between pages is handled by the `navigate` language construct. This construct abstracts away from URLs, request parameters, and other low-level implementation details. It requires a descriptive text expression and a destination page:

```
navigate("Home", Blog(u))
```

Note that the destination page may have page parameters, as in this concrete case. The actual parameters (in this case `u`, which itself is a page parameter of the `ViewBlog` page) must be provided in that case. In Section 4.3.1, we will see that these parameters are all checked at compile time. So, in effect, we have strongly typed data-flow between the pages of a *WebLayer* application.

The last element for displaying text on a page is somewhat more advanced. The `show` construct comes in two versions (a pattern that also applies to several other constructs that will be introduced),

page_elem ::=	**'table'** *param* **'in'** *qualified_ident* **'{'** *key_val*⋆ **'}'**	table with selector
	| **'table'** *param* **'{'** *key_val*⋆ **'}'**	table
	| **'for'** *param* **'in'** *qualified_ident* **'{'** *page_elem*⋆ **'}'**	for-loop with selector
	| **'for'** *param* **'{'** *page_elem*⋆ **'}'**	for-loop

Figure 4.6: Iterative page elements

a *generic* version, and a normal version. An example of the usage of the generic `show` is:

```
show(r)
```

In this case, `r` is a variable of type `Reply`, which is brought into scope by a `for` loop (to be introduced in the next section). This concept instance will be shown in a structured fashion (see the lower part of Figure 4.23 for the result), following a *membername : value* pattern for each member of the concept. The information necessary to expand this expression is derived from the *DomainModel* definition of `Reply`. The name of the field is the same as the concept member identifier. The value is coerced to a string, with one exception: if the member is a composite concept, its members are recursively expanded as well. As such, `show` is a powerful construct that leverages type information from the *DomainModel* DSL. Should this default handling of concepts not be satisfactory, there also is the normal version of show:

```
show( "Reply :" -> r.contents
      "Date :"  -> r.date
      "Good? :" -> header(r.level)
    )
```

Note that the above piece of code is not reflected in our example, since the generic `show(r)` is used instead. The normal version of `show` accepts one or more *key-value* pairs (see Figure 4.5 for the definition). This allows the user to fine-tune the names of the fields, and the exact representation of the value. On the righthand side of the arrow, an arbitrary page element may be provided. In this case we mostly leverage the default interpretation of `text` that was mentioned earlier, i.e., a text expression (and an fqi is in itself a valid text expression) is automatically interpreted as a `text` element. Only the last value is wrapped in a `header` element, so it will be displayed more prominently. A call to the generic show on the righthand side is also possible, to automatically show a member with a concept type (both a reference and a composite!). Using this version of `show` also allows the user to show only a projection of a concept, or to combine fields of several concepts into a single structured display block. In the example above, the `user` member was deliberately left out.

4.2.2 Iterative constructs

In Figure 4.6 we present the iterative page elements that are part of the *WebLayer* language. Generally, either recursion or looping is necessary to effectively process sequences in a language. Since our language does not allow for any user-defined function-like abstractions, the latter was elected.

DomainModel allows lists of concepts to be defined as concept member. Combined with the fact that the database itself contains a collection of each concept, this forms the basis for the iterative page elements. The first element, and one that is prevalent in web pages, is the table. A table page element consists of the `table` keyword followed by a parameter binding and an optional selector expression. The identifier introduced in the parameter binding may shadow other variables with the same name. The body of `table` reuses the key-value notation of the `show` element:

```
table Tag t in be.tags { "Assigned tags" -> t.tagName }
```

This line produces a table that lists the `tagName` member for all tags in `be.tags` (`be` is a page parameter) in a single-column table with "Assigned tags" as header. The variable `t` is scoped over the body of the `table` construct, iterating over the elements of the `tag` member for every row. The

page_elem ::=	**'edit'** '(' *qualified_ident* ')'	data entry widget
	| **'input'** '(' *qualified_ident* ')'	generic input
	| **'input'** '(' *key_value+* ')'	input

Figure 4.7: Input page elements

alternative version of `table` (Figure 4.6) neither has the `in` keyword, nor a selector expression. Instead, it ranges over all available concept instances of the type declared in the parameter binding, and it acts as a universal quantifier over that type. This can be useful to, for example, list all users in a system.

Not all information that is iterative fits the table representation. Therefore, a second iterative construct is introduced, the `for` loop:

```
for Reply r in be.replies { show(r) }
```

Using `for`, the user is not forced into a row/column structure. Rather, the body contains other page elements (in this example, a single generic `show` element), which are repeated for every element in the collection the `for` loop is iterating over. As with `table`, the `for` construct can also be used without a selector expression, following the same semantics. Figure 4.23 shows the result of the above line of code, showing only a single `Reply`, since `be.replies` contains only one `Reply` for the `BlogEntry` we are viewing on this page. Also note that the parameter binding of this `for` loop shadows the local page variable with the same identifier `r`.

One observation is that in the `table` and `for` elements with a selector, the type annotation for the iterator variable is redundant. Since all expressions are typed (Section 4.3.1 elaborates on this), this type can be inferred from the type of the selector expression. However, the annotation is essential in the iterative constructs without a selector. For the sake of consistency, we have chosen to require the annotation in all cases. In the description of the implementation of the iterative constructs, an additional argument in favor of type annotation over inference will be discussed.

4.2.3 Input elements

Besides showing data, we also want to receive input through our web-application. To this end, two new page elements will be introduced. The first is the `edit` element. It takes an fqi as argument, and results in a data entry widget on the page. The form of the widget depends on the type of the argument. For example, a String argument results in a textbox, an argument with an enumeration type results in a combo box containing the alternatives of the enumeration, and an argument with type Date produces a textbox with an associated date-picker, and so on. This makes `edit` another generic construct. In Section 4.4 we discuss the drawback of this generic approach. Also, a data entry widget is nothing on its own: it needs to be submitted back to the server to be handled appropriately. The next section addresses these concerns.

The second page element for data entry is the `input` element. It is the counterpart of `show`, allowing for structured editing of concepts. As with `show`, there is a normal version, accepting key-value pairs, and a generic version that presents a complete edit form for a concept:

```
input(r)
```

The generic `input` construct presents the name of each member of the concept, and a suitable input widget. Figure 4.23 shows the result of the above line of code, a labeled input form for a new `Reply` concept.

An interesting question is what the input widget for a concept member with a list type looks like. Our example page does not contain such a field, but Figure 4.8, taken from another page of the same blog application, shows what it looks like. The list of concepts is visualized by printing the names of the concepts, with the names being composed from their `name` annotated members. Each entry also has a delete button to remove the entry from the list. Elements can be added by selecting a concept from the combo box, which is then automatically added to the list. Section 4.3.5 discusses the implementation

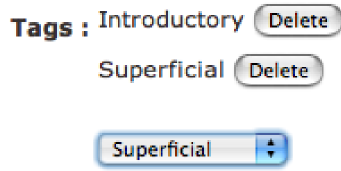


Figure 4.8: Editing a `tags -> [Tag]` member

behind this mechanism, since it poses some additional challenges with respect to form handling. This is also the reason that editing of composite lists (rather than reference lists) is not implemented in the prototype.

4.2.4 Actions and forms

So far, we have seen the *WebLayer* page elements for showing and entering data. The missing link between these elements is how the input data is processed, and possibly stored in the database. For this purpose, we introduce the last two page elements in Figure 4.9.

page_elem	::=	**'action'** '(' *text_expr* ',' *actions* ')'	action button
		| **'form'** '(' *page_elem*⋆ ')'	form grouping element
actions	::=	*action* {';' *action* }⋆	separated action list
action	::=	*qualified_ident* '=' *qualified_ident*	assignment
		| **'redirect'** '(' *nav_binding* ')'	page redirect
		| *qualified_ident* '(' { *qualified_ident* { ',' *qualified_ident* }⋆ }? ')'	call

Figure 4.9: Data processing page elements

The first element, `action`, encodes data processing logic. It takes a text expression and an *action language* block as parameter:

```
action("Add reply", r.user = u; be.replies.add(r); be.save())
```

The `action` element itself is mapped to a button, with the text expression as label. When clicked, the following happens:

1. Data is submitted (exactly what is submitted will be discussed further on)
2. Input data is validated with respect to the:
   - `required` annotation of the concept member (if present)
   - custom validation logic, if the member is an extended type (see Section 3.1.2 and Section 5.2.2)
3. If the validation fails, the form is presented again (reflecting any previously entered data)
4. If the validation succeeds, the action code is executed

An example of a validation failure is shown in Figure 4.10. An empty form was submitted, but the `user` field has a `required` annotation on it, so the form is presented again, providing an error message below the field that failed to validate. When all fields are valid, the action block is executed. An action block is comprised of statements, separated by a semi-colon. There are three types of statements in the very basic, sequential action language that is implemented in *WebLayer*. The first type, the assignment, is necessary to link concepts to each other and to manipulate data. In the example, the first statement is an assignment that assigns the page parameter `u` to the `user` member of the `Reply r` that is being

Figure 4.10: Required field not set

edited. The second type of statement, `redirect`, allows an action to redirect the user to a different page than the page the action code originated from. In the current state of the action language, the `redirect` statement only makes sense as the last action in an action block. If omitted, the action by default presents the same page after it has successfully completed. The last type of statement is the call. The example contains two call statements. The first call, `be.replies.add(r)` adds the submitted reply to the `replies` member of the current `BlogEntry be` we are viewing on the page. Conversely, a `remove` call is also available. The second call, `be.save()`, flushes the current state of the `BlogEntry be` to the database. Since the reply was added to its `replies` member, it is transitively saved as well. A `delete` call is available as well. Note that is possible to have an action that has no associated input elements, for example, a delete button in a row of a table that deletes an element from a list. The *WebLayer* action language is very basic, and has only a few built-in calls that can be performed. However, for basic CRUD applications this model suffices. In Chapter 6 we investigate a mechanism to increase the capabilities of the action language in a manner that fits our multiple DSL approach, based on the call statements.

One last remaining problem is that we want to logically group what data is submitted when a button is clicked (i.e. an action is performed). A naive implementation would be to wrap the generated page completely in an HTML form element. However, this means that whatever button is clicked, every input element is submitted. This, in turn, means that all elements are validated as well, even though the action may pertain to a certain group of elements (i.e. resulting from a single `input` element) only. However, we cannot infer a smaller scope for the enclosing form element without loss of generality. Therefore, a `form` page element is introduced into the *WebLayer* language, allowing the user to indicate the correct grouping. It logically groups the page elements that belong to a single 'unit' within a page:

```
form( input(r)
      action("Add reply", r.user = u; be.replies.add(r); be.save())
    )
```

Now, only the fields generated by the `input(r)` element are submitted when the "Add reply" button is clicked. Though *WebLayer* allows forms to be nested (since they are page elements themselves), ultimately, the underlying markup language (HTML) does not. In the case of nested forms, only the outer `form` element is significant, since HTML is defined that way.

4.2.5 Page and session variables

Having surveyed all page elements, there are still two *WebLayer* language constructs left: local page variables and session variables. First we look at page variables. The input elements introduced in Section 4.2.3 all accept an fqi to indicate what concept (member) to edit. So far, the only possibility we have seen for data to be available in a page, is through page parameters. However, what if we want to edit (create) a new concept on a page? An option would be to introduce a new page element (e.g. something like `create(Concept)`). However, it would work nearly the same as the `input` element, the important difference being that a type is explicitly provided rather than inferred from the argument. Recognizing this fact, we opted to implement not a different page element, but a page variable declaration:

```
page ViewBlog(BlogEntry be, User u){
   var Reply r
   ...
```

This construct introduces a local variable of type `Reply` to the page. All fields of this variable are set to their initial (empty) value. Now, we can just refer to this identifier in an input element (e.g., `input(r)`, or `edit(r.contents)`). Local variables may be passed to other pages, but only if they have been committed to the database first. This restriction stems from the way data-flow between pages is implemented, which will be discussed in further detail in Section 4.3.4. Having this construct also allows us to let the language grow toward variable declarations with initializers (e.g. `var Reply r =` *some query*), should a query language be embedded at some point. Such an embedding is briefly discussed in the concluding remarks of this chapter.

While data-flow using page parameters is a useful feature, sometimes it can become cumbersome having to *thread* a variable through multiple pages. In this case, session variables offer an alternative. It allows variables with a global scope (though bound to the session of a particular user) to be introduced. Session variables are declared on the top-level of a *WebLayer* module (between page definitions), using the same syntax as for page variables. Initially, a session variable is also a transient, empty concept instance. Then, actions may assign values to the session variable. Its value is preserved as long as the session is alive on the server, or a new value is assigned to the variable. A session variable may not be passed as parameter to a page (which is not necessary anyway, since session variables are accessible in any page).

4.3 Implementation

In the introduction of this chapter we already introduced the target of the *WebLayer* compiler, the Seam framework. In this section we provide a detailed account of how the compiler is implemented. To start a *WebLayer* project, a *skeleton* is provided. It provides the user with a directory containing all static configuration files necessary to compile, build, and even deploy a DSL project consisting of *DomainModel* and *WebLayer* sources. Furthermore, a build file is provided, that given the locations of the sources and the DSL compilers, automates these tasks.

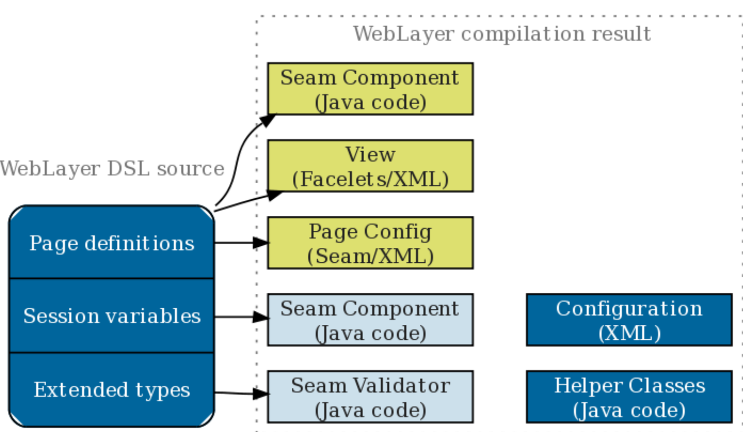


Figure 4.11: Page generated using *WebLayer*

Before moving on to the implementation details, we provide a high-level overview of what the *WebLayer* compiler generates. The relation between a *WebLayer* source file and the emitted files by the compiler is visualized in Figure 4.11, where an arrow indicates a direct '*generates*' relation between the definition and output file (e.g., every single page definition results in a single Seam Component, and likewise for session variable declarations).

As is the case with the *DomainModel* compiler, there is no one-to-one correspondence between the DSL source and a single output file. Instead, every page definitions translates into three files, a

```
type-check-page-elem :

r@Repeat(ParamBinding(type, ident), selector, elems) -> annotated
where in-tc-context(
         {| TypeOf :
           where(  t := IteratorScope(type)
                 ; rules(TypeOf : ident -> t) )
                 ; Repeat( ParamBinding(is-defined, id)
                         , is-listtype(|type)
                         , map(type-check-page-elem))
         |}
       | [<pp-page-elem> r]
       )
     ; ?annotated
```

Figure 4.12: Type checking rule of `for` construct

Seam Component, an XML file containing markup (Facelets/XML) code, and an XML file containing configuration to link both of the preceding files. Furthermore, each top level session variable declaration translates into a Seam Component. Also, every extended type (originating from the import of a *DomainModel* definition) is translated into a Seam Validator. Last, various configuration files and helper classes are output by the compiler, visualized as compilation results without an attached arrow in Figure 4.11. These differ from the other compilation results in that they do not correlate to individual page definitions (or other definitions in the body of a *WebLayer* module), but rather need to be generated when compiling a module. In other words, the same amount of helper classes and XML configuration files is emitted, regardless of the actual number of definitions in the body of the module.

In the remainder of this implementation section, Seam Components and the rest of the generated artifacts are introduced. Furthermore, we explain the role and contents of most of the generated files, as well as how the compiler achieves the translation. Before moving on to the actual code generation phase (in Section 4.3.3), we first look at the preparatory machinery of the compiler.

4.3.1 Semantic checking

After parsing the *WebLayer* source, the first stage of the compiler is concerned with importing other DSL modules mentioned in the `using` clauses. In practice, this means the compiler imports concept definitions from a separate *DomainModel* module, and information on how to use these definitions. The mechanism that enables this interaction is described in Chapter 5. For now, we just observe that these definitions are available throughout the compilation process, while the *DomainModel* source module need not be parsed and processed again (it can be compiled separately).

These definitions are very important for the second stage of the compiler, the type checking phase. During this phase, the module is checked for well-formedness, which includes (but is not limited to) checking the types of all expressions and page elements. This is achieved by performing a syntax-directed traversal of the input AST. This traversal is encoded in Stratego *rewrite* rules. This might seem odd for a checking phase, but this approach was chosen for two reasons:

1. It allows us to rewrite the checked term to a term with a type annotation.
2. We can decorate the AST with generated (unique) identifiers, which is necessary for the translation of some constructs, as we will see later.

An example of a type checking rewrite rule is given in Figure 4.12. There is a lot going on in this rule, and we try to convey the intuition behind it. The rule matches a `for` page element with selector (which is parsed into a `Repeat` node with three children). The result of the rule is the *annotated* meta-variable, which contains the result of applying the type checking constructs in the `where` clause. This type checking is performed inside a (higher-order) strategy `in-tc-context`, which keeps track of the current element being checked, in a stack-like manner. The list argument after the vertical bar contains a pretty-printed string of the `for` element we are currently checking. This way, we are able to create

error messages that contain information from the surrounding context, making it easier for the user to spot the error. Unfortunately, it is not easily possible to include location information in error messages when rewriting the AST.[3] This would allow for more concise error messages, as the 'context-stack' currently can grow quite large when an error is detected in a deeply nested page element.

Inside this strategy, we start a new dynamic rule scope (between the `{| |}` delimiters) for `TypeOf` rules. This means, that any `TypeOf` rule introduced within this scope is invalidated when returning outside the scope delimiters. Dynamic rules (i.e. rules created at run-time) are used to record the types of the identifiers we encounter. Using the dynamic rule scope we implement the shadowing of identifiers. Also note that not only the type of the identifier is recorded in the righthand side of the `TypeOf` rule that is introduced, but also the fact that the variable stems from an iterative construct (i.e. `IteratorScope(..)`. This information is necessary when translating expressions accessing this identifier. The two other scopes we identify are page scope (for page parameters and local page variables) and session scope (for session variables).

The actual checking is performed through the `Repeat` congruence strategy with its nested strategies. The `is-*` strategies perform checks in accordance with their names, and add an error message to a collection (again using dynamic rules) of error messages if the check fails. Note that the type checking phase tries to find as many errors as possible at once, it is not *fail-fast*. Finally, the `map(type-check-page-elem)` strategy recursively continues the traversal. Note that no specific type annotation is added to the `Repeat` AST node, but the recursive invocation may return annotated nodes. In particular, fully qualified expression are resolved using the imported *DomainModel* definitions, and their type is appended to the AST node.

There is a `type-check-page-elem` definition for each distinct page element. Some are more straightforward than this rule (e.g. in the generic `show` the argument only has to be a concept type, and needs to be annotated), others are more intricate (e.g. `action` blocks have much more constraints). However, the principle behind each of these rules is the same. At the end of this phase, the AST is either correct and annotated, or an error has been detected and the compiler will report these.

Going back to the rule in Figure 4.12, we can see that the `is-listtype` check gets the *type* metavariable (containing the type annotation of the parameter binding) as an extra argument. It is used to verify that the type annotation is the same as the inferred element type of the selector expression. We already established that this annotation is superfluous in this particular construct. However, the selector expression might also become a database query in a future version of *WebLayer*. This is a worthwile addition, which unfortunately could not be realized within the confines of this thesis project. When a query is present as selector rather than an fqi, it is questionable whether inference of the right type for the iterator variable is always possible.

4.3.2 Specializing generic constructs

Now that we have a typed AST, the next step is to prepare this intermediate format for the code generation stage. This means bringing all constructs into their normal form, also known as *desugaring*. Doing so entails traversing the AST once more. The iterative constructs without a selector are assigned a special selector value indicating all concepts should be iterated over. Moreover, the generic versions of `show` and `input` need to be expanded. These constructs need to be unfolded following the structure of the type of their argument expression, in order to present all the concept members, or an appropriate edit form for a concept, respectively. We call this process *unfolding*, or *type-directed* desugaring. Towards the end of this section we briefly discuss the relation to generic programming.

Figure 4.13 shows the rules for unfolding the generic `input` construct, and `DesugarShow` works in a similar fashion. The rule `DesugarInput` rewrites a generic `input` construct to the normal `input`, containing key-value pairs of labels and `edit` elements (the *key_values* variable). This is achieved by mapping `member-to-keyval-input` over the members of the concept type. This map is implicitly performed by `concept-to-keyvals`, which deconstructs the appropriate concept type into a list of its members, based on the type of *ident*. Note that we use concrete *WebLayer* syntax between the `|[ ]|` brackets in the transformation rules, allowing for a more intuitive definition of the rules. There is a second `member-to-keyval-input` rule that matches only on concept members with a composite association (explained in Section 3.1.2). These members are not rewritten to an `edit` element, but rather to another generic `input` for this member. This means, that a nested form will be created for

[3]This situation has been improved during the course of our work.

```
DesugarInput :

pelem |[ input(ident) ]| -> pelem |[ input(key_values) ]|
where <is-id> ident
    ; key_values := <concept-to-keyvals(member-to-keyval-input)> ident

member-to-keyval-input(|ident) :

ConceptMember(_, name, _, _, _) -> keyval |[ label -> edit(qid) ]|
where label := QuotedText(<capitalize> name)
    ; qid   := <extend-id(|name); post-tc-annotate-id> ident
```

Figure 4.13: Specializing generic `input` construct

such a composite member. This works since the `DesugarInput` is applied repeatedly (until it fails) to a term and its subterms in a topdown manner. This application is guaranteed to reach a fixed-point if and only if there are no concepts with mutually recursive composite members. This condition must be enforced by the *DomainModel* compiler, since it constitutes an inconsistent situation, considering the semantics of composite associations. Intuitively, it corresponds to saying two distinct physical objects are inside each other, which is also impossible.

Another interesting aspect is that such a rewritten generic `input` has not been typed by our type checker, since it is a result of the transformation step after type checking has been performed. However, we do not have to perform a complete type check of the AST after the transformation, since the type can be incrementally computed during the desugarings (as witnessed by the invocation to `post-tc-annotate-id`).

The type-directed desugaring described above resembles the specialization of functions based on type arguments that is performed by many generic programming languages, such as Generic Haskell or Clean. In fact, the iData project [?] (written in Clean) also boasts its ability to derive HTML forms from a Clean data type. However, the aforementioned languages allow their users to write such type-directed functions, whereas our specialization is built-in to the compiler and is applicable only to specific language constructs. As such, Haskell's `deriving` clause, which instructs the compiler to derive some predetermined functions for arbitrary data types, is closer to what we implemented in *WebLayer*.

4.3.3 Translating pages

We now turn to the last stage of the compiler, the code generation phase. Throughout this section and the subsequent sections, we refer to listings of code generated by the *WebLayer* compiler, which are provided in Appendix A.2.2 (starting on page 94). In this section, we look at the three generated files when translating a page definition, as shown in Figure 4.11. We use the running `ViewBlog` example to explore the translation scheme. In this section we look at the result of translating a single page definition. The subsequent sections provide a discussion of the implementation of page navigation, the translation of individual page elements, the action language, and the remaining constructs, respectively.

Listing A.4 and Listing A.5 show the Seam component generated from the `ViewBlog` page. A Seam component is a Java class annotated with an `@Name("`*ident*`")` annotation, where *ident* is a unique identifier linking to a single instance of this class. Instantiation of components is managed by the Seam framework itself. References to instances of components can be obtained through *dependency injection*, or by retrieving them (by name) from a Seam component manager class. Dependency injection is a run-time lookup mechanism, allowing the programmer to write variable declarations like `@In(name="`*ident*`") ComponentType member_name` for a class member. The framework consequently wires the right component instance into `member_variable`, relieving the programmer from obtaining a reference to the correct object instance, as would be necessary in plain Java. Configuration of this wiring is typically done in an external (XML) file. However, this mechanism is not statically checked, and can lead to unexpected failure when, for example, a typo is made in *ident*. Dependency injection is used in our page components to inject an `EntityManager` and a `FacesMessages` component, as can be seen in the first lines of the `ViewBlogComponent` class in Listing A.4. The former is necessary to ma-

nipulate JPA entities (i.e. concepts originating from *DomainModel*), whereas the latter is a component that manages error messages that need to be displayed on (possibly different) pages. Besides these two variable declarations, the contents of the Seam component for a page can be roughly divided into code supporting:

1. Initialization
2. Page parameters
3. Iterative constructs
4. Actions
5. Editing of lists

This structure of components is also reflected in the transformation rule for translating page definitions. This rule is shown in Figure 4.14. Assignments to the (italic) meta-variables using auxiliary rules are left out for conciseness (hence the ellipses following the `where` clause). We discuss the initialization part of the page component in this section, the latter four parts of a page component are treated throughout the ensuing sections. Figure 4.14 shows that a page component is annotated an `@Name` annotation, making it a Seam component, and an `@Scope(ScopeType.CONVERSATION)` annotation. This last annotation indicates the scope of the component instance. These scopes are specific to Seam. The conversation scope is a subscope of the well-known session scope, that is, a user can have multiple conversations within a single session (the proliferation of tabs in browsers led to the introduction of this scope). Assigning this scope, rather than the more narrow page scope, allows us to cache state between views of different pages.

Every page component has an `initialize` method. An external configuration file (in the case of our example: `ViewBlog.page.xml`, presented in Figure A.3) makes sure this method is called on each page view of the associated page. The `initialize` method is annotated with `@Begin(join=true)`, indicating that Seam should start a new conversation context when executing this method, or that if a conversation exists for this user, Seam should merge this component into the existing context.

As can be seen in Listing A.4 and Listing A.5, there are many class member variables with associated `get` methods (following the JavaBean standard). Note that all references to artifacts generated by the *DomainModel* compiler are fully qualified. Chapter 5 explains how we obtain these references without duplicating knowledge from the *DomainModel* compiler. Every page component has an `initialized` (boolean) member, with `false` as its default value. Within the `initialize` method, member variables relating to page parameters (e.g., those in *`page_param*`*) are assigned their initial values. In Section 4.3.4 we will see the exact mechanism behind the page variables. Furthermore, the member variables containing lists for the iterative constructs in a page definition (if any) are initialized. The last statement sets `initialized` to `true` as a signal for other parts of the code that the page component is ready for use.

Besides the component, a file containing the markup of the page is generated as well. This file contains Facelets and JSF code. Facelets is an XHTML-like templating mechanism to create views for JSF based web-applications. JSF is the component framework used by Seam to render actual elements on the screen, and to bind them to `Java` code. From now on, we refer to this file as the viewcode. Figure A.1 and Figure A.2 show the viewcode for the `ViewBlog` page. Viewcode is structurally much closer to the page definition than the Seam component. When a page in the web-application is viewed, the viewcode is parsed and interpreted by Seam and its underlying libraries. It contains many references to Seam components (in particular the page component for that page) which are resolved dynamically. Ultimately, pure HTML and JavaScript is emitted by JSF to the browser, which in turn can be several orders of magnitude larger than the generated viewcode itself.

The top-level structure of the viewcode is shown in the transformation rule in Figure 4.15. Note that we use concrete syntax for `XML` in this rule, between the `%> <%` delimiters. The document type declaration and additional namespace declarations are omitted, the complete header can be seen in Figure A.1. In the top-level element a `template.xhtml` file is referenced. This file resides in the non-generated part of the web-application skeleton we provide, and can be edited freely by the DSL user.

```
page-to-component :

PageDecl(_ , name, param*, elem*)
  -> compilation-unit
    |[
      package ~id:Id(<Package>);

      // Import statements omitted for the sake of brevity

      @Name("~component") @Scope(ScopeType.CONVERSATION)
      public class ~id:Id(component) implements Serializable {

        @In private EntityManager em;

        @In private FacesMessages facesMessages;

        ~*page_param*

        public @Begin(join=true) void initialize(){
           try {

             ~*initialize_pageparam*
             ~*initialize_iterator*
             initialized = true;

           } catch (NullPointerException npe){
               // Exception code omitted
          }
        }

        public boolean initialized = false;

        public boolean getInitialized() {
          return initialized;
        }

        ~*iterator_member*

        ~*action_method*

        ~*listedit_method*
      }
    ]|
where ...
```

Figure 4.14: Transformation of a page definition to Seam component

```
page-to-view :

PageDecl(_, _, _, elem*) ->
%>
 <ui:composition xmlns="http://www.w3.org/1999/xhtml" ... template="template.xhtml">

   <!-- global error and validation messages for page  -->
   <ui:define name="globalMessages">
     <h:messages globalOnly="true" styleClass="errors"/>
   </ui:define>

   <!-- page element content -->
   <ui:define name="content">
     <div class="section">
       <s:div rendered="#{!<% !component %>.initialized}">
        <p> The page could not be rendered due to missing or
            invalid information in the request parameters. </p>
       </s:div>
       <s:div rendered="#{<% !component %>.initialized}">
         <% !elem_xml* :: *  %>
       </s:div>
     </div>
   </ui:define>

 </ui:composition>
<%
  where component := <Component>
      ; elem_xml* := <map(page-elem-to-xml); separators> elem*
```

Figure 4.15: Transformation of a page definition to view code

All generated content is embedded in this template, allowing a consistent style to be applied to each page.

Furthermore, two sections are defined (using Facelets' `ui:define` tags). The first one displays any messages that are stored in the `FacesMessages` component (also mentioned in the description of the page component above). The second defines the actual page following from the page definition. However, it this section is rendered conditionally. The `rendered` attributes contain an EL[4] expression (between the `#{ }` delimiters) checking whether the page component has been initialized. The expression works because there is an `initialized` property on the page component with a corresponding `getInitialized` method. If `initialized` is `false`, an error message is rendered instead of the page elements, avoiding any errors that might arise from using an un-initialized page component. EL expressions form the (statically unchecked) connection between the viewcode and Seam components, and as such they play a central role in the translation scheme.

The actual page elements are translated by mapping `page-elem-to-xml` over the page elements in the AST, a strategy that will be discussed in Section 4.3.5. Afterwards, the strategy `separators` is applied, which intersperses line-break elements, such that all page elements are put beneath each other.

4.3.4 Page navigation and data-flow

Navigating between pages is conceptually very simple in *WebLayer*. Pages link to each other by name, and concepts may be passed as parameters. The implementation of this data-flow between pages is handled by the *WebLayer* compiler. In general, there are three ways to achieve data-flow between different pages of a web-application. One way is to put the data into the session storage mechanism, and retrieve it from the (transient) session on a different page. The second way is to submit the data

[4]Expression Language, defined in the JSF standard, and extended by Seam.

itself to a different page. This means that the data has to be encoded in the URL (in the case of a GET-request) or has to be submitted as form data (for POST-requests). The last way is to store the data in the database, and retrieve it in the next page.

We chose a combination of the last two approaches. A drawback of the first approach is that a page relies on transient data to be present in the session storage, without this fact being represented in the URL. In other words, it is not possible to create an external link to such a page, since sessions expire and data may be gone. However, sending back and forth the complete set of data between the client and server (which is implied by the second approach) is not very appealing either. Therefore, we chose to use the identity of concepts as bridge between different pages. That is, whenever a concept is passed to another page, this is achieved by encoding its id into the URL. Then, the receiving page uses this id to retrieve the correct concept from persistent storage. This entails that the state of a page is encoded in its URL, which in turn means that external links can be created. However, this comes at a cost: only data that has been persisted can be used in this approach. The first approach of data-flow, storing data in the session, is also available to the *WebLayer* user by means of session variables. Session variables make it possible to exchange transient data between pages, at the cost of having hidden state, precluding direct links that completely restore the state of the web-application.

The implementation of the combined scheme is to create a class member in the page component for each page parameter (the concept itself), and one for its associated id, and the appropriate `get` methods to be able to reference the members through EL expressions. Furthermore, the id member is annotated with `@RequestParameter("`*name*`")`, which instructs Seam to automatically retrieve the value for this member from a GET-parameter called *name* in the requesting URL. An example would be `http://../ViewBlog.seam?blogEntry_beId=1`, which produces the value `1` for the id member. Hence, data-flow is achieved by constructing URLs containing the right parameters with the correct ids. Then, within the `initialize` method these ids are used to lookup the correct concepts from the database and assign them to the class members. In Listing A.4 we show the members generated for the first `BlogEntry be` page parameter of the `ViewBlog` page.

However, we cannot guarantee that in all situations the GET-parameters for a page are present. For example, if an action is invoked on a page (i.e., a button is clicked and the form is submitted) a POST-request is issued, which does not include any of the GET-parameters that were present in the original URL. Therefore, we introduce a caching mechanism in the page component (which itself outlives a single page view) for the ids that are passed. In the `initialize` method we assign the current id (if it is present) to the caching member. Furthermore, the `get` method for the ids revert to returning the cached value if no actual id is present.

To see what navigation looks like on the implementation level, we look at the translation of the `navigate` statement of the example page definition (Figure 4.23). In the viewcode, navigation is represented using a Seam `link` tag:

```
<s:link value="Home" view="/Blog.xhtml">
 <f:param name="user_uId" value="#{ViewBlogComponent.user_uId}"/>
</s:link>
```

We use the `s:link` with nested `f:param` tags rather than a normal (X)HTML anchor tag because Seams needs to insert additional, administrative information (e.g., the current conversation id) into the actual generated link. The value of the parameter is extracted from the page component by using an EL expression, which may return the actual or cached id, since the EL interpreter accesses members through their `get` method.

4.3.5 Translating page elements

The translation of the `navigate` statement is already an example of how individual page elements are translated by the compiler. In this section we will lift out some more transformations that are exemplary for how the compiler works, without attempting to give an exhaustive overview. Page elements are translated to viewcode only, except for iterative constructs and edit fields for concept members with a list type. We will show how these constructs need additional support from the page component. Another exception is the local page variable, which needs nearly the same code as the page parameters

(discussed in the previous section), but without the associated id members, since a local variable's value is not passed as page parameter.

We will first look at the simplest of page elements, the `text` element. Every page element has its own matching definition of the `page-elem-to-xml` rule. In Figure 4.16 the definition of this rule for the `text` element is provided.

```
page-elem-to-xml :

TextExpr(contents) ->
  %><h:outputText value="<% !contents-value %>" escaped="true" /><%
where contents-value := <textExpr-to-elstring> contents
```

Figure 4.16: Transformation rule for `text` page elements

The transformation is straightforward: an `outputText` tag is emitted, with an EL expression as value binding. Creating this expression is delegated to a strategy which handles the fact that a text expression can either be quoted text, or an fqi, or a combination (recall Figure 4.5). Most interesting is the second case, where an fqi must be translated to the correct EL sub expression binding, for example, to a member of a page parameter. Also note that we explicitly set the escaping of this `outputText` component to `true`. It is possible to just output text without using the `outputText` component in viewcode, though doing so makes the application vulnerable to cross-site scripting attacks[5]. Thus, we enforce the best practice of escaping all dynamic data before rendering it within the compiler.

```
qualId-to-elstring :

QualifiedId(var, qualId) ->
  <concat-strings> [<scope-resolver> var, ".", <qualId-to-elstring> qualId]
```

Figure 4.17: Translating *WebLayer* identifiers into EL expressions

We already established that the translation between an fqi and its corresponding EL expression is a recurring theme in all `page-elem-to-xml` rules of the *WebLayer* compiler. Therefore, this translation is centralized into a single location in the *WebLayer* compiler. Figure 4.17 shows the recursive case of this transformation rule. The `scope-resolver` strategy is an auxiliary strategy that acts on the annotated scope information of identifiers (if any), provided in the type checking phase. Each of the three different scopes (page parameter and local variable, or iterator, or session) results in a slightly different EL expression. For example, a page variable identifier must point to the correct member of the page component, and a session identifier must point to the corresponding session component (to be explained in Section 4.3.7). Listing A.4 and Listing A.5 contain many examples of these different EL expressions.

```
page-elem-to-xml :

Iterator(ParamBinding(type, SimpleId(name)), _, key_vals){list_ident} ->
 %><h:dataTable value="<% !list %>" var="<% !name %>">
       <% !columns :: *  %>
   </h:dataTable><%
where columns := <map(key_val-to-column)> key_vals
    ; list    := <concat-strings> ["#{", <Component>, ".", list_ident, "}"]
```

Figure 4.18: Transformation rule for `table` elements

To further illustrate the translation of page elements, we take a look at the `table` element. Figure 4.18 shows how the `Iterator` AST node, associated with a `table` with selector expression, is

[5]Cross-site scripting attempts to inject malicious code into a webpage.

translated to viewcode. The corresponding JSF element is the `dataTable`, which automatically iterates over the binding provided in `value` property. The `var` property contains the iterator variable name, that may be used in EL expressions in the nested column definitions. Column definitions are represented by nested `column` elements, which are generated by the `key_val-to-column` rule (definition not shown). Returning to the value binding, we see that *list* is an EL expression, referring to some member of the page component. Interestingly, the selector expression (second child of the `Iterator` AST node) is not being used in this EL expression. This is due to the fact that a JSF `dataTable` cannot work on arbitrary lists: the elements of a list must be pre-processed to conform to JSF's `ListDataModel` interface. Fortunately, Seam provides us with an annotation (`@DataModel`) that transforms any list transparantly to the right format. However, this does mean that we need to create an explicit class member containing the list to be processed in the page component.

Since the list member and the `dataTable` value binding are in two disparate files, and the `table` construct itself does not have a distinguishing identifier, we use a compiler generated unique identifier to link these two locations. The translation rule shows an example of how a unique identifier (meta-variable *list_ident*), generated during the semantic checking phase, is being used in the EL expression for the list value. This same identifier is used in the page component for the list member and its `get` method. In the generated code of the `ViewBlog` page, `iterator3` is the unique identifier for the `table` element containing the tags of the `BlogEntry`.

The final page element we elaborate upon is the `edit` element. We note that the generic constructs have already been desugared to their normal forms in this stage, leaving `edit` as the only primitive for the input of field data. The `page-elem-to-xml` rule for `edit` constructs delegates the actual construction of the viewcode to a rule (`qid-to-inputfield`) which extracts information from the type annotation of `edit`'s fqi argument.

```
qid-to-inputfield :

qid ->
  %><s:decorate>
      <h:inputText id="<% !ident %>" required="<% !required %>"
                   value="<% !binding %>">
        <% !validator :: * %>
      </h:inputText>
    </s:decorate><%
where <type-of(?NativeType("String"))> qid
    ; ident     := <qualId-getlast> qid
    ; binding   := <qualId-to-elstring> qid
    ; required  := <get-required> qid
    ; validator := <get-validator> qid
```

Figure 4.19: Transformation rule for `table` elements

A different rule is available for each of the distinct value types. Figure 4.19 shows the (slightly adapted) rule for plain string arguments. The first action in the `where` clause is to establish whether the type of the argument indeed is a string, otherwise a different rule applies. A JSF `inputText` element is constructed, with the name of the concept member as its systematic id (*ident*, a required property with `true` or `false` (depending on the information of the imported *DomainModel* definition), and a value binding EL expression (*binding*). Last, a validator may be attached to an input field. Validators are created by *WebLayer* for *DomainModel* extended types, and can be linked to an input field so that the custom validation logic is applied by JSF. The *validator* variable is a list, since we either want one `validator` tag, or zero. In this particular rule, `get-validator` will always return an empty list, since we are translating a concept member with string type. We included the validator part to illustrate the mechanism, though.

Other `qid-to-input` rules exist to handle different types of input widgets. One more interesting issue arises when `edit`'s argument has a list type. Figure 4.8 already showed our solution, but the implementation is somewhat problematic. Originally, we anticipated list editing to be a straightforward desugaring from `edit(list_member)` to (for example) a `table` element listing the current contents, and

```
page-elem-to-xml :

ActionBlock(actiontext , actions){actionmethod} ->
   %><h:commandButton value="<% !descr %>" action="<% !action_expr %>" /><%
where descr       := <textExpr-to-elstring> actiontext
    ; params      := <infer-el-params; separate-by(|", "); concat-strings> actions
    ; action_expr := <concat-strings> [ "#{", <Component>, "."
                                      , actionmethod , "("
                                      , params , ")}" ]
```

Figure 4.20: Transformation rule for `action` elements

actions to add and remove elements from the list. However, implementing this idea has an unfortunate side-effect. Since nested forms do not exist in HTML, everytime an action fires to manage the list (which itself is a part of larger input form) all fields of the outer form are submitted as well. This means, that validation is performed for these fields, and the page component is updated with (possibly empty) values, even though this is not intended. Therefore, we had to find a way to bypass the JSF mechanism for our list editing mechanism. The Ajax4JSF library provided us with methods to bypass the described problems. In particular, it allows parts of forms to be submitted without refreshing the page, and bypassing the JSF processing. However, in doing so, it could no longer be implemented as a desugaring to native *WebLayer* constructs. Instead, many specific Ajax4JSF tags must be used, and even a helper method for each list to be edited must generated in the page component. In Figure A.4 we show the viewcode generated for `edit(be.tags)`, which is the expression that results in the screenshot in Figure 4.8. These problems are also the reason why we only implemented editing of reference lists, and not composite lists.

4.3.6 Action language

In the language description we have seen the three different types of statements of the *WebLayer* action language. Now we will take a look at the translation of the `action` element and the action statements contained therein. We first inspect the viewcode generated for an `action` element, as shown in Figure 4.20.

A JSF `commandButton` tag is used to implement actions. The descriptive text is translated using the rule we saw earlier for text expressions. We will discuss the *`params`* meta-variable later. The `action` property binds to a method of a page component, rather than a member variable, using an EL expression. In this method, the action statements themselves are translated. Again, we link the viewcode and the method in the page component using a generated identifier (*`actionmethod`*).

In the case of our example page, the generated method is named `action3`, and can be seen in Listing A.5. The translation of the action statements is embedded in a `try-catch` block, which handles validation errors that may occur upon persisting a concept, and detects concurrent modification (by virtue of the optimistic locking scheme).

We now examine the translation of the statements in the `"Add reply"` action on the example page of Figure 4.4. Each distinct type of action statement is translated using its corresponding `action-to-java` rule, of which one example is given in Figure 4.21. The first statement, `r.user = u`, is translated to a `Java` statement consisting of a sequence of linked `get` calls, and a `set` call for the last element of the fqi on the righthand side. The argument of this `set` call is the translation to `get` calls of the lefthand side. Note that this translation to `get` calls is more or less the page component's dual of creating an EL expression in viewcode. The second statement, `be.replies.add(r)`, is translated to the corresponding `addToReplies` method of the `BlogEntry be` concept, translating the argument to a sequence (of one in this case) of `get` calls.

The last statement, `be.save()`, is translated to three successive `Java` method calls. The `persist` call is performed on the `EntityManager` class, which is injected into the page component by Seam. It forms the gateway to the database, and a `persist` call forces the concept passed as parameter to be updated with the new values. However, it may also be the case that the parameter is a new concept (not yet in the database). In fact, that is the case in our example, where the `Reply r` that is passed originates from a local page variable. This means that the concept does not have an id value yet, since

```
action-to-java :

Call((qid, "save"), [])
  -> bstm*
    |[
       em.persist(get_expr);
       em.flush();
       em.refresh(get_expr);
    ]|
where get_expr := <component-expr> qid
```

Figure 4.21: Transformation rule for `save` calls

this is automatically generated by the persistence layer. In turn, this means that the concept cannot be used as parameter to another page, since data-flow relies on the id value. Therefore, two additional statements are generated to explicitly push the change down into the database (`flush`), thereby forcing an id to be assigned, and to refresh the object reference (`refresh`) to reflect this new id value. Now, the concept may safely be passed to another page.

This brings us to the last type of action statement, `redirect`, which is not present in the example. A `redirect` statement is translated to a `Java` `return` statement, which returns a string representing the destination page. Seam is configured to interpret this return value of the action method, and present the user with the correct page. If no `redirect` statement is provided in the action code, the same page is shown. This default is implemented by a `return` statement after the main `try-catch` block of an action. Listing A.5 shows the structure of this return statement: a URL is created, with all the corresponding request parameters for the page parameters of the destination page. Note that cached values of the ids are used to construct this URL. Before the actual `return` statement is executed, we also set the `initialized` variable of the component to `false`, in order to force the elements on the page to be reloaded. We do not know what is changed by the action, so we make the conservative assumption that all data needs to be reloaded on the next page view. Also, local page variables are initialized to an empty instance again.

In the translation of action statements outlined above, we assume we can access the used variables through appropriate `get/set` calls, or by accessing the right session component. However, not every variable is in the scope of the page component, and thus in scope of the action method. In particular, iterator variables introduced by the iterative page elements are not statically available in the page component. Still, these may be used in action statements. Therefore, we implemented a method which locates the usage of variables with iterator scope, and passes these as arguments to the action method from the viewcode to the page component. This process is somewhat reminiscent of creating a lexical closure. It is performed by the `infer-el-params` strategy in Figure 4.20, leading to zero or more parameters in the *params* variable. Furthermore, the action method uses the same inference results and accepts the correct concepts as parameters. It follows that assigning to an iterator variable is not possible. In the concrete example, no iterator variables are used in the action statements.

Implementing a small action language in *WebLayer* was a deliberate choice. An alternative approach is to embed an existing GPL (in our setting `Java`) as action language. This would be more convenient, since no consideration has to be given to a translation scheme. However, doing so would mix two levels of abstraction in a source file, and hook our DSL design to `Java` by definition. Whereas currently the DSL user keeps thinking in terms of concepts and some pre-defined actions, with such an embedding the user must think in terms of the implementation of concepts. In our opinion, it is not acceptable to let the user deal with the peculiarities of using, for example, the `EntityManager` protocol and `get/set` methods.

4.3.7 Session variables and validators

Up until now, we have focused on the three arrows originating from 'page definitions' in Figure 4.11. In this section we briefly look at the two remaining arrows, originating from 'session variables' and 'extended types'. In the language description we have seen that session variable declarations may occur at

the top level of a *WebLayer* module. In the compilation phase, each session variable declaration is translated into its own Seam component. This component has session as its scope type, and contains a single member of the corresponding concept type of the declaration, and `get` and `set` methods. As with local page variables, the member is initialized with an empty concept instance, so it can be used in `input` and `edit` elements. Every session variable component implements our `weblayer.support.SessionVar<T>` interface (where `T` is instantiated to the type of declaration), allowing action code to handle session variables generically, without having to know the name and specifics of the class for this session variable. An `@AutoCreate` Seam annotation on session components makes sure that Seam creates these components lazily, when they are accessed for the first time.

Last, we look at validator components. The *WebLayer* compiler creates one for every extended type imported from a *DomainModel* definition. Listing A.6 (page 98) shows the validator component for the extended type `URL`. The main goal of a validator component is to attach validation error messages to specific fields. We have seen how the validator tags are placed on fields. These link to a validator component, through the name given in the `@Validator` annotation. The `validate` method is invoked by JSF upon form submission. Inside this method, we cast the value to the right type, delegate the actual checking to the extended type implementation, and raise an exception if an error message has been returned by this implementation. This message is then displayed below the offending field, similar to Figure 4.10.

4.4 Issues

In the previous sections we have seen both the design of the *WebLayer* language and most of its implementation. During the description of the implementation, we already elaborated on some problematic issues and design choices in the compiler. In this section we want to discuss the issues that arose during the creation and use of *WebLayer*.

Early versions of the *WebLayer* compilers did not have the type checker we described in this chapter. This revealed a large flaw, in our opinion, of the implementing libraries we use as target for our DSLs. Namely, it is very easy to create code that compiles, but it is comparatively hard to create code that does not exhibit run-time errors. Especially the run-time resolving of EL expressions is the culprit, and the interpretation of the different annotations to a lesser degree. This observation led us to write a type checker, which also attaches scope information to identifiers. Consequently, the code generation phase did not have to anticipate inconsistent input anymore, and the extra information made the translation to EL expressions more straightforward. Having this type checker also allowed us to implement the generic constructs, which shows that a typeful approach to DSL development can actually bring more benefits than improved safety alone.

Another issue that emerged during the development of *WebLayer* is that of layout and styling. The *WebLayer* language allows the user to express the structure of a page in terms of page elements. How these elements are composed on the actual page, and what their appearance must be, cannot be expressed in the DSL. Instead, page elements are automatically placed beneath each other, and default CSS class names are assigned to the elements. Also, we only distinguish between header and normal text, whereas more individual control might be desirable. While the styling is flexible in the sense that user can edit the CSS stylesheet of the skeleton, the vertical layout of page elements is fixed.

We considered ways of improving this situation, for example, by letting the user indicate custom regions in a page (e.g., through named `div` elements which can be uniquely referenced in CSS). However, this clutters up the page definitions, and still does not allow the DSL user to override standard elements. The latter could be alleviated by introducing more types of page elements, for example, a dedicated `editXX` component for each available widget in HTML forms. The generic `edit` could be maintained as well. Nevertheless, growing *WebLayer* to include all this variability also introduces more potential for user errors. This is a classical trade-off, and we believe that in the case of a language like *WebLayer*, it can be beneficial to consolidate many layout and styling policies into the compiler. There are DSLs that try to get the best of both worlds, such as WebRB [**?**], a visual web DSL based on relational algebra components mixed with UI components in a unified editor. However, this approach has the weakness of requiring the graphical designer and the application programmer to be the same person. Our approach of embedding the generated page into a freely editable template allows for a

separation between these roles.

A different problem that arose during the implementation of *WebLayer* is an issue with the composability of page elements. Several elements, such as the `table` and `for` elements, may contain contain nested page elements. This is a useful and intuitive abstraction. However, the translation scheme and implementation of `table` elements poses a restriction. We have seen that a table is bound to a member in the page component with a special annotation to prepare the list for usage in the viewcode. Also, an iterator variable is brought into scope. The problem is, that this (dynamic) variable has no direct (static) counterpart in the page component (also described in Section 4.3.6). Therefore, list members of the concept this iterator variable is pointing to, cannot have the appropriate annotation on them. Whenever a `table` element is nested within another `table` element, and the inner table iterates over a list member of the outer table's iterator variable, this will fail. Knowing the translation scheme, it is clear why. Nevertheless, a DSL user should not need to know about the internals, and should be able to nest arbitrary page elements in a table. Interestingly, the tag that implements the `for` loops does not need a pre-processing annotation, and therefore does not exhibit the same problem.

The `form` element also has nested elements, though this page element was not anticipated in early designs of *WebLayer*. Rather, our initial design was to always implicitly pair an action with a single input statement. Soon it became clear that this was a bit too restrictive. What if, for example, we want to have two different actions associated with the same input, or if we want to have two generic `input` elements associated with one action? Ultimately, in the implementation this all comes down to the question: what is part of the same HTML form in a page? For a while, it seemed feasible to infer such grouping from the proximity of certain elements in the page definition. However, to do so in an unambigious and general manner was hard, and attempts were unsuccesful. Therefore we introduced this form grouping explicitly into *WebLayer*. It allows arbitrary combinations of `input` and `action` elements, at the expense of having to always use a `form` element for them to work, even if we only want to have a single `action`.

The last issue we want to discuss is more related to an implementation quirk of JSF, and as such illustrates how a framework can interfere with DSL abstractions. We have seen how the validation of extended types is applied at the corresponding input field on a form. We recall that the same validation is performed in the *DomainModel* layer. Of course this second validation pass is guaranteed to succeed, since it is the same code as invoked by *WebLayer* for that field. However, a problem arises when such a field is left blank, and it has not been marked as `required` in the *DomainModel*definition. JSF does *not* invoke validation for empty fields. If the extended type validation rejects empty values, the validation will still fail in the *DomainModel* layer, throwing an exception which we catch in *WebLayer* code. Unfortunately, this validation step in *DomainModel* is global to a single concept, and therefore JSF cannot trace this failure back to the empty field, thus showing the error message above the form instead of adjacent to the offending field.

This behavior can be easily prevented by annotating such concept members with `required`, though it is not obvious that this is necessary from the user's perspective.

4.5 Concluding remarks

In this chapter we have explored the design and implementation of the *WebLayer* language and compiler. We have shown that clear and concise abstractions for presentation elements, which can be used in conjunction with *DomainModel* concepts, are compiled to a deployable web-application. In particular, the closeness of mapping[6] between *WebLayer* page definitions and the resulting page is much higher than between the Seam framework code and the resulting page. No library specific artifacts are present on the language level. For example, the structure of pages and data-flow between the pages are directly discernable in the *WebLayer* source, rather than hidden in Java's GPL abstractions. We have also shown that many problems of the underlying libraries and frameworks, especially relating to the dynamic nature of many constructions, can be alleviated through a typeful DSL approach.

Besides improving quality of code, *WebLayer* also improves productivity. In order to show the order of magnitude, we present some metrics for the running example of this chapter, the blog. This sample

[6] An important *cognitive dimension* [**?**] for assessing the abstraction level of a programming language.

	DSL Source	Generated code	
DomainModel	45		
Java		801	
XML		22	
WebLayer	110		
Java		841	
XML/Facelets		501	
XML		48	
Totals	**155**	**2213**	$\sim\ \times 14$

Figure 4.22: Metrics for blog example application

application consists of five pages and four *DomainModel* concepts, from which we have seen one page definition (Figure 4.4) and one concept definition (Figure 3.3). Figure 4.22 shows the lines of code count of both the source modules and the target files that are generated by the *WebLayer* and *DomainModel* DSL compilers. The factor 14 increase in lines of code is typical, according to our experiences with different sample applications. It should be noted that the Java lines are much more involved to write, compared to *WebLayer* code. On the other hand, writing the same application from scratch in the target frameworks might yield a smaller codebase, since generated code invariably contains some dead code. Nevertheless, we crafted our code generation after real usage patterns of the frameworks, therefore the disparity between generated code and hand-written code is not significant in our opinion.

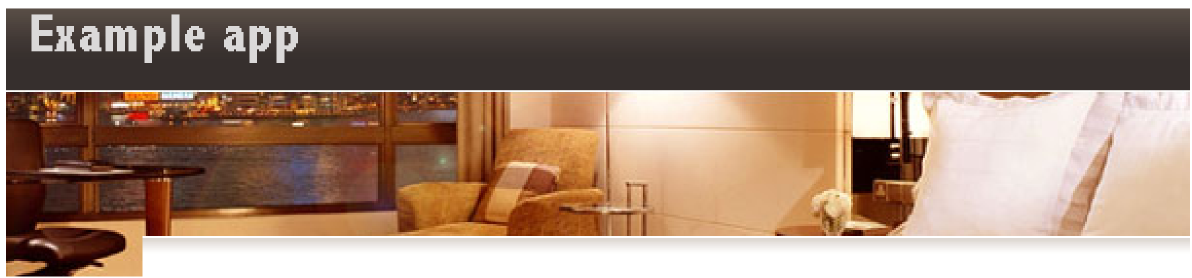
In addition to gaining quality and productivity through the use of DSLs, we think this case study also shows the synergy that can exist between DSLs. For example, the generic `input` and `show` constructs leverage compile time capabilities that cannot be replicated when only using the target frameworks.[7]

A large drawback of our approach is the fact that when developing a stand-alone DSL, as opposed to an embedded DSL, we cannot reuse facilities of a host-language. In other words, type checking and general abstraction mechanisms need to be implemented from scratch for our DSL. The same holds for a module system, which is not implemented for this very reason. While writing such facilities by hand allows us to choose abstractions that exactly fit the domain, and write very specific semantic checks, it still requires a considerable effort. On the other hand, we also experienced that embedding a web-application DSL in Java (which is ultimately what Seam tries to accomplish) does not lead to satisfactory results.

With respect to the implementation, we have found the Stratego/XT toolkit to be valuable for this type of DSL development. Especially the ability to use concrete syntax for Java code, XML markup, and even our own DSL syntax in rules, helped a great deal while developing the compiler. Tasks that were not primarily syntax-directed, which are often hard to express in rule rewriting systems, were accomplished using Stratego's generic traversals.

The *WebLayer* language is far from being complete (since it was not our goal to create a single production quality DSL), but we believe the implementation is easily extendable in its current form, and has a clear structure. Every compiler phase is decoupled and implemented in relative isolation. Extending the type checking and code generation rules to handle more page elements is fairly straightforward, because of the separation between viewcode and page component code generation. Some ideas that were not implemented due to time constraints were, for example, embedding a (possibly already existing) query language for *DomainModel* concepts, and abstraction mechanisms to reuse parts of page definitions.

[7]The best frameworks can do there is to use run-time reflection on classes to simulate this behavior.



Title of post (written on 2007-10-25 00:00:00.0)

Insightful blog post goes here.

Assigned tags
Introductory
Superficial

Reply to this post:

Contents :

Date :

User : Please select :

Level : Please select :

Add reply

Replies for post Title of post :

Contents : First reply!

Date : Oct 26, 2007

User : User 1

Level : average

Home

Figure 4.23: Page generated using *WebLayer*

Chapter 5

Interaction aspects

In the previous chapters we have provided a detailed account of the individual building blocks for our prototype of DSL driven web development. However, the mechanism through which these various parts work together has been left implicit. In this chapter, we discuss the various types of interaction that can be observed in our implementation. We differentiate between two types of interaction:

1. Interaction between two distinct DSLs.

2. Interaction between DSL code and user-written `Java` code.

Note that a third type of interaction can be established as well, which is cooperation between modules written in the same DSL. Nevertheless, our DSLs currently do not exhibit this kind of interaction. We reflect upon this choice, and possibilities in this regard in the concluding section of this chapter. The first mode of interaction is investigated in Section 5.1, where the cooperation between *DomainModel* and *WebLayer* and between *WebLayer* and *BusinessRules* is elucidated. The second mode of interaction is investigated in Section 5.2, by looking at the mechanism used to introduce extended types in the *DomainModel* DSL. Since interaction between host language code and DSL is also employed by several other model-driven engineering approaches, a comparison with these approaches is provided as well.

5.1 Interaction between DSLs

The premise throughout this thesis has been that the development of DSLs should mostly focus on technical domains. In our case, these domains relate to the three distinct layers in typical web-application development. Since we did not develop a single monolithic DSL to model this domain, a complete application is created by compiling the separate definitions in the distinct DSLs. However, while the DSLs are distinct, they are not isolated. In order to create meaningful applications, DSLs must be able to reference definitions in other DSLs. In particular, *WebLayer* definitions may use *DomainModel* and *BusinessRules* definitions, by importing them through `using` clauses. Figure 5.1 provides a schematic representation of how our DSLs cooperate to create a complete web application in the `Java` target environment.

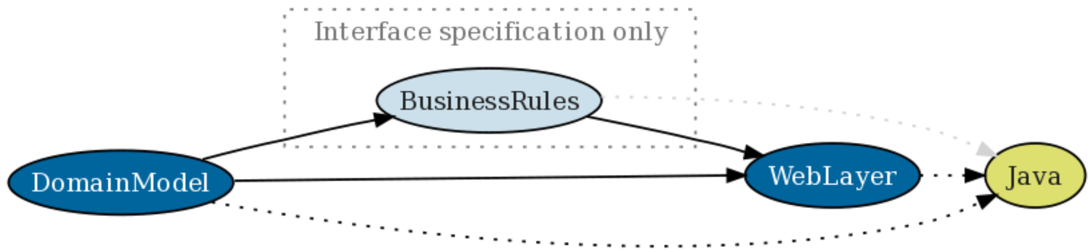


Figure 5.1: Flow between DSLs

In this figure we can clearly identify the hierarchical relationship between the DSLs. In our particular case, we have a non-cyclic directed graph. More concretely, this means that both *BusinessRules* and *WebLayer* can import *DomainModel* models, and *WebLayer* can import *BusinessRules* models, but not the other way around. We note that the composition of the DSLs matches the architectural layering that is present in typical web application development projects. This is an important observation, since it indicates that other layered architectures possibly can benefit from having multiple interacting DSLs as well. The subsequent section eloborates upon this thought.

So far, we have seen how code is generated from models expressed in our DSLs. While the link between the various models is clear on the DSL level, we avoided the issue of how the generated code is linked. An important observation in this regard is that we want the generated code to behave and look like code as if it were hand written. Especially the performance of applications may not be negatively affected by the fact that it was generated from DSLs. In particular, we want to stress the fact that the generated code from the DSLs must link statically. This rules out any interaction mechanism based on runtime mediation between the generated artifacts. In Section 7.1 we discuss and compare these contrasting alternatives in more detail.

Since we want to statically link code from different generators, it is clear that these generators need to be aware of each other. Specifically, the *WebLayer* compiler needs to know about the JPA classes generated by *DomainModel* and about the methods implementing *BusinessRules* constructs. We approach this problem by introducing *interface files*. Section 5.1.3 discusses the implementation. Our goal is to minimize the amount of sharing of inner details of the DSL compilers, while still being able to create code that statically links. Sections 5.1.5 and 5.1.6 deal with issues encountered while researching the interaction aspects. Czarnecki et al. also raise the issue of identifying interaction between DSLs compilers in 'Generative Programming' [**?**] (p. 157). However, no concrete suggestions are made as to how this problem should be approached. Some related work in this area exists, which will be discussed in Section 7.

5.1.1 Motivation

Before moving on to the implementation of interaction between our DSLs, we first want to revisit the motivation for having multiple DSLs to begin with. Furthermore, Section 5.1.2 discusses the situation in which we envision our system of interacting DSLs to thrive.

It should be clear that creating a stand-alone DSL is a serious investment and is typically more involved than creating actual applications. Therefore, it is not normally deemed feasible to implement DSLs by, and for a single company. In the introduction (Section 2.2), it already became clear that large monolithic DSLs, as exemplified by 4GL languages, are not the way to go. However, it is too easy to abandon (stand-alone) DSL development altogether based on this negative legacy. By working with multiple interacting DSLs, a more agile path emerges. DSL implementations can focus on one (technical) aspect of the complete picture, and do it well. This decreased granularity in our opinion provides several advantages:

1. Flexible composition of languages.
2. Reuse of languages.
3. Gradual introduction of DSLs in a project.
4. Separate compilation of DSL models.
5. Separation of concerns.

The combined effect of these advantages in our opinion allows for DSL development in smaller settings. The first point stems from the modularity that is promoted by the idea of interacting languages. In our particular case, creating a typical web-application involves writing code in all three DSLs. However, if we only want an application that performs rudimentary data manipulation (i.e. CRUD), then we can suffice with writing just *DomainModel* and *WebLayer* code. Furthermore, specialized DSLs for business rules and calcultions can be developed, as we have seen in Chapter 6. When the generator of such a DSL conforms to the *BusinessRules* interface, *plugging* in the DSL is as easy as importing the right *BusinessRules* module in the *WebLayer* program.

A special case of the aforementioned flexible composition of languages, is their reuse in different settings. As an example to the second point, we consider the use case in which we want to create a DSL for adminstrative desktop applications (i.e. a desktop oriented alternative to *WebLayer*). Obviously we need to create a DSL that accurately captures different screens, menu structures, dialogs, and other concepts that exist in the realm of desktop applications. However, the remaining concerns, that is the data model and business rules, do not change for desktop applications[1]. Thus, we can reuse the language definitions and compilers of the existing languages. The hypothetical new language can import (using the existing interface definitions) models expressed in these languages, allowing the implementor to focus solely on issues pertaining to the domain of desktop application GUIs. Note that this reuse is possible by virtue of the stringent separation of concerns imposed by our multiple DSL model. In ordinary web-development, ideally the data layer and business rules layer are equally well decoupled from the presentation layer. Unfortunately, in practice it is all too common for business logic to leak into the domain model or into the presentation layer. These inadvertant strong couplings then prevent reuse of the created artifacts.

The third advantage listed is that DSLs can be gradually introduced into existing projects or software development methodologies. Transitioning from a traditional development environment, using 3GLs, to an environment in which DSLs are used can be a daunting task. Completely switching to a DSL based practice at once may present a large risk to companies, even if the languages suit their practice perfectly. When dealing with multiple interacting DSLs, however, it is also possible to develop only parts of projects with a DSL. For example, *DomainModel* might be used to model and generate JPA classes. These classes can then be integrated into the rest of the code. In the case of *DomainModel* some knowledge of the translation scheme is necessary to be able to achieve this. However, most of this knowledge relates to standardized JPA usage, and is knowledge that should be present at any rate. Of course, when a language has a dependency on another language (as is the case with, for example, *WebLayer*), it cannot be used solitary.

The fourth advantage given, separate compilation of models in different DSLs, has a more technical nature. In the next section we introduce separate compilation. For now, we suffice with remarking that separate compilation in general is a desirable property for a programming language. As a last advantage, having multiple interacting DSLs allows for split development of parts of an application. Several model-driven engineering approaches are based on the premise that there is a single, monolithic model. As a result, traditional methods of software engineering (i.e. source control, joint development) break down. Our approach allows different developers to work on different parts of an application, without getting in the way of (or being hampered by) fellow developers.

We believe our approach of multiple interacting DSLs can be applied to other layered architectures, besides web-applications, as well. This belief is strengthened by similar observations from the EDSL community. For example, Hudak mentions that the root of a good (E)DSL implementation is having 'layers of abstractions rather than a monolithic structure' [**?**]. In his case, this layering happens through the facilities offered by a single host-language.

A more directly related observation in this regard is the approach to device-driver development taken by Merillion et al. [**?**]. Here, three layers of abstraction are also distinguished. A DSL is only developed for a single layer, however, so interaction aspects in this project are unfortunately left unattended. Therefore, we believe our approach constitutes a natural continuation of existing ideas in this area.

5.1.2 Intended usage scenario

It is evident that companies are eager to reap the benefits of using DSLs, in particular increased productivity and quality. In our opinion, having multiple interacting DSLs helps this cause, by lowering the barrier for using and creating DSLs. We envision that lead developers or designers play a key role in the adoption of DSL based development. These persons are responsible for setting the standards, creating the architectures, and so on, that are to be used by the companies' developers. Usually, such persons also create customized frameworks, libraries, and even guidelines to be used throughout a business, as they have the overview over the complete spectrum of applications that need to be developed.

[1] One exception may lie in database configuration issues.

This knowledge can be encoded in, and its use automated by DSLs. Such languages can be built in conjunction with existing languages for technical domains. For example, the *DomainModel* will appeal to a large range of development projects, since its implementation is fairly general. In *WebLayer*, on the other hand, many choices are made that are likely to make sense only in the context of a specific company policy. Therefore, such a language would be developed by a lead developer, possibly reusing existing DSLs such as a *BusinessRules* implementation and the *DomainModel* DSL. By developing small, focused DSLs these lead developers can enforce architectural styles, best practices and boost productivity for specific program families.

5.1.3 Separate compilation and interface files

In order to appreciate the interface file mechanism used to allow interaction between our DSL implementations, we first introduce the general concept of separate compilation. In general purpose languages, compiling a source file of an application requires knowledge from other source files, referenced within the file to be compiled. For example, functions are called that reside in other compilation units, or externally defined types are used. Separate compilation entails that a source file (or compilation unit) should be compilable *without* needing other source files. By ensuring this property, only those source files that have changed need to be recompiled. Without separate compilation, compilation times would increase dramatically, since each source file would have to be compiled again upon a single change. However, *some* information is still necessary for successful compilation. A solution employed by, for example, the `C` compiler is to use header (`.h`) files. Those files contain function signatures and type definitions and provide the necessary information to be used for compilation. These header files must be written by the user though. Haskell (specifically the GHC compiler) works similar, only the interface (`.hi`) files are generated by the compiler. These files essentially contain a compiler readable description of the contents of the generated binary [?]. `Java`, on the other hand, extracts the desired information from compiled class files. Separate compilation also allows for a form of source protection: compiled artifacts (e.g. libraries) can be distributed, and can be used in other programs without having to disclose the original source code.

No universally accepted, formal definition of separate compilation exists. Cardelli [?] reduces the separate compilation problem to the following judgment: $\Gamma \vdash \texttt{S} : \tau$, meaning a source file `S` can be type-checked and assigned type τ using an environment Γ. Header files and interface files are an instance of such an environment. Typically, type checking is only one part of the problem, albeit the most intricate one.

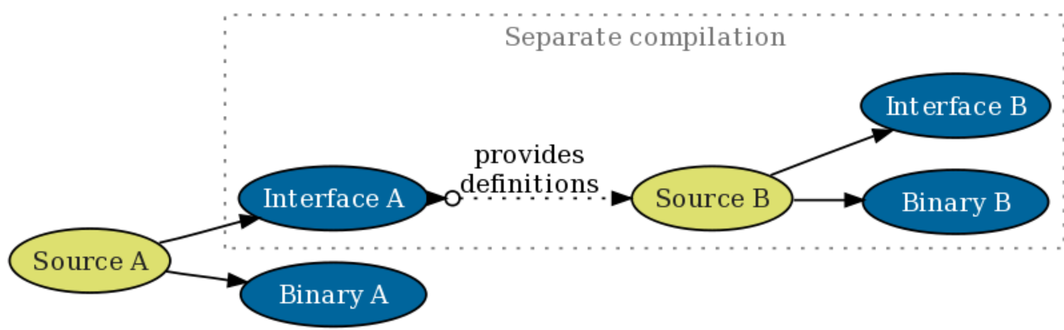


Figure 5.2: Separate compilation

Figure 5.2 presents a diagram showing the essentials of separate compilation by example. A compiler normally translates source code into an executable binary (e.g. 'Source A' compiles into 'Binary A'). The solid arrow represents the *generates* relation in this diagram. Now, if we want to write code ('Source B') that uses definitions from 'Source A', having separate compilation means that the compiler does not need the actual 'Source A' file in order to compile 'Source B'. Instead, the compiler uses a descriptive interface file that was also emitted when 'Source A' was compiled (indicated by the dotted line). Thus, the two source files can be compiled in separate passes, where the second pass does not

redo work done in the first pass. As seen in the discussion of interface files in the beginning of this section, it might also be the case that interface and binary are the same physical file.

We now turn to our domain specific languages, in order to see how separate compilation can be introduced. First of all, our goal is slightly different. We want to establish separate compilation between models expressed in different DSLs, as opposed to separate compilation of modules within a single language. For example, we look at our two DSLs *DomainModel* and *WebLayer*. Now, we want to be able to compile a *WebLayer* source (that uses *DomainModel* definitions) without actually needing the *DomainModel* compiler.

Furthermore, our DSL compilers do not translate a single source file into a single binary file. Rather, a source file is translated into multiple files, in particular `Java` and `XML` files. Therefore, we consider the aggregation of these files (resulting from a single compilation pass) as the equivalent to 'the binary', in the sense as it was used in the introductory part of this section. As an aside, we note that the `Java` files can be either in source or bytecode format in this context.

If we have a binary, produced by one of our DSL compilers, and want to reference its definitions in another DSL, we have two options:

1. Extract the information from the binary.
2. Communicate the information in an interface file.

These options correspond to the solutions as found in implementations of separate compilation in general purpose languages. Since the compilation result in our case comprises many files (in differing formats), the first option is rather unattractive. Implementing the first option almost amounts to creating a reverse engineering tool for our DSL. In itself that might be useful, but for the purposes of separate compilation this is overkill. Hence, we are left with the second option of emitting and reading interface files. This choice also conveniently allows us to regard both generated `Java` files and their compiled bytecode counterparts as unit of distribution (binary).

Figure 5.3 shows a concrete instance of an interface file emitted by the *DomainModel* compiler. The file is the result of compiling the *DomainModel* example program of Figure 3.3 on page 15. Formatting is added manually, and some parts are truncated (indicated by ellipses) for the sake of brevity. The interface is output in the concrete syntax of ATerms [?], a canonical format for representing tree-structured data (comparable to `XML`).

Looking at the contents of the interface, we observe that it is comprised of a top-level `DomainModel` term, containing three nested terms. The meaning of these terms can be described as follows:

1. **Name** Declares the name of the compiled domain model. This corresponds to the name declaration on the first line of the *DomainModel* source file.
2. **Extended types** Contains a list of extended types found during the compilation (Section 5.2.2 introduces the mechanism behind extended types).
3. **Concepts** Contains a list of actual concept definitions from the corresponding source file.

Clearly, the third term in the interface file closely resembles the (parsed representation of) the original source. Superfluous details are removed, but since *DomainModel* already is a terse declarative language, there are not many details to be omitted. In particular, only *DomainModel* annotations (e.g. `unique` for the `name` member of the User concept) of concept members are preserved, not the inlined `Java` annotations. These annotations are very implementation specific, and tied to the `Java` language. Therefore it is not interesting to promote them as part of the model.

Furthermore, additional information is added to the interface. Most notably, fully qualified identifiers for each of the generated artifacts are provided. This allows the compiler that uses the interface definitions to reference these artifacts directly by using the fully qualified identifier. No assumptions have to be made on the package and/or class structure of the code generated by the *DomainModel* compiler. This form of loose coupling is essential to allow the implementation of the first compiler to change, without breaking compatibility with the second (importing) DSL compiler. We also note that the representation of this fully qualified identifier is a plain string. This means it needs to be interpreted, in our case in the context of `Java` types and methods. Should a back-end for another language

```
DomainModel(
  "blog"
, [ ("Text", "String",  [( "validate"
                         ,  "org.blog.domainmodel.validation.Text.validate" )])
  , ("Email", "String", [( "validate"
                          , "org.blog.domainmodel.validation.Email.validate" )])
  ]
, [ Concept(
      "User"
    , "org.blog.domainmodel.User"
    , [ ConceptMember( Native(), "name", NativeType("String")
                     , "java.lang.String", ["unique"] )
      , ConceptMember( Native(), "email", ExtendedType("Email")
                     , "java.lang.String", [] )
      , ConceptMember( Composite(), "blogEntries"
                     , ListType(ConceptType("BlogEntry"))
                     , "List<org.blog.domainmodel.BlogEntry>", [] )
      ]
    )
  , Concept(
      "BlogEntry"
    , "org.blog.domainmodel.BlogEntry"
    , [ ConceptMember( Native(), "title", NativeType("String")
                     , "java.lang.String", ["required"] )
        <...>
      , ConceptMember( Native(), "category", EnumType(["TECH", "NONTECH"])
                     , "org.blog.domainmodel.BlogEntry.Enum_category", [] )
      ]
    )
  , Concept(
      "Tag"
    , "org.blog.domainmodel.Tag"
    , [ConceptMember( Native(), "tagName", NativeType("String")
                    , "java.lang.String", [] )
      ]
    )
  , Concept(
      "Reply"
    , "org.blog.domainmodel.Reply"
      [  <...>  ]
    )
  ]
)
```

Figure 5.3: Interface file for *DomainModel*

be implemented, however, we can use the same interface format, and use a string representation suitable to the alternative language.

Another difference between the plain source and the emitted interface file, is that disambiguation of type names has been performed. In the *DomainModel* language, a type name (uppercase identifier) of a concept member can either be a:

- native, built-in type, or an
- extended type, or a
- type name referencing another concept.

The interface file explicitly differentiates between these possibilities by emitting respectively `NativeType`, `ExtendedType`, or `ConceptType` terms, containing the type name. Moreover, when reading an interface file, the well-formedness of the corresponding binary can be safely assumed. An interface file is only emitted after succesfully compiling a DSL source file, thus semantic checks (name resolution, type checking) have been performed. In the case of *DomainModel*, this effectively means that every concept is well-formed, and that all extended types that are referenced are defined.

It might be tempting to think that distributing a DSL source file is just as convenient as emitting an interface, since they are so similar. However, this entails that every DSL compiler that wants to reference another DSL's artifacts, must duplicate (or invoke) the other compiler's front-end (parsing and semantic checking). In effect, the interface file mechanism can be viewed as a caching mechanism for these checks[2]. Moreover, information that resides in distinct source files, such as extended type definitions, is also aggregated in a single interface file.

Currently, the only thing that needs to be shared by the DSL compiler implementations is the ATerm signature (comparable to an XML schema definition) of the interface format. Obviously, the compiler that wants to import another DSL's constructs must be able to read the interface file and disclose the information therein contained.

5.1.4 Interface characteristics

In the previous section we dissected one particular interface. However, it would be presumptuous to think that everything is said with the description of this interface. An important observation is that the interface between *DomainModel* and *WebLayer* has an explicit and an implicit part. The interface file (of which an example was given in Figure 5.3) encodes the explicit part. However, there are many implementation related facts that are not reflected in the interface file. Still, this knowledge is necessary to use the generated code. Examples of such implicit facts are:

- Every concept has an (automatically administered) `id` member.
- Each concept member is accessible through a `set/get` method.
- Every concept employs optimistic locking.
- Each concept member with a list type has additional methods to manage the list.

Interestingly, all of the above sentences contain the universal quantifier in some way. That is precisely what sets these implementation details apart from the information contained in the interface file. In the interface file, we only record the variabilities of the model. For example, which concepts and extended types are defined, what is their structure, and so on. The implicit interface consists of how the artifacts, generated according to these variabilities, can be used. This is based on static conventions. So, while we decoupled the implementation of the two DSL compilers to a large extent, there still is an invisible coupling in the form of the implicit interface. This invisible coupling is of concern when designing a new DSL that should interact with existing DSLs.

In the realm of *component* software, a similar observation has been made by Szyperski et al. in 'Component Software: Beyond Object Oriented Programming' [?] (p. 57). Components need to interact through interfaces as well, and a formal and informal part of such a *contract* is distinguished. This

[2] Of course this equally applies to interface files of GPLs.

corresponds with what call the explicit and implicit part of the interface.

In the case of *DomainModel* the implicit part of the interface is dominated by the implementation (and our usage pattern) of the Java Persistence Architecture framework we target. An unfortunate side effect is that alternative back-end implementations for the *DomainModel* language must also conform to this implicit interface. If not, the composability of the implementation with the other existing languages is lost, even though the explicit interface format is unchanged.

The case of *BusinessRules* is a bit different. Essentially, the tables are turned: the implicit interface is specified by us, and any DSL implementation can choose to implement this interface in its code generation. That is, the DSL designer ascribes the implicit interface rather than the specific implementation framework. It is likely that such an interface is more suitable for supporting multiple implementations, rather than an implicit interface that stems from a specific implementation with all its quirks. The fact that the *BusinessRules* interface specification is very small and general helps as well in this regard.

Of course, the DSL designer can create such an independent interface even if a target framework is chosen, and wrap it around the actual implementation. In the case of the interface between *DomainModel* and *WebLayer* however, our implementing framework (JPA) already had a vast and above all standardized interface. Redoing the interface would constitute a very large task. Moreover, the implementing framework of *WebLayer* (Seam and JSF) is in itself already tailored to working with the JPA interface. Introducing a custom wrapper would only lead to overhead in the intended setting.

We conclude that, besides the normal steps of domain analysis when devising a DSL, attention should be given to the design of the implicit interface. Preferably, this part of the interface is defined independently of an actual implementation of the DSL compiler. We postulate that it is neither feasible, nor desirable to devise an explicit representation of every aspect of the interaction between DSLs.

5.1.5 Issues

In the previous section, we concluded that it is not sensible to try to encode every detail of the implementation of the generated code in the published interface of a DSL. In part, this is because this information is almost entirely static. It is not clear how we can encode conventions such as *'every property is backed by a get/set method'*, or *'entities must be registered with an `EntityManager` object before persisting them'*. While the first convention can be encoded by resorting to massive duplication of this information (e.g. list `getX()/setX()` for every concept member `X`), the second convention is already more semantically loaded, and pertains to behavioral constraints. We want to avoid a slippery slope of over-specification, where practical benefits are mostly lacking. Therefore, we have chosen to only include a qualified identifier for the model elements, leaving their usage and semantics implicit.

A more practical issue regarding the interaction of generated artifacts, has to do with the fact that we do not only generate implementation code from our DSLs, but also configuration files for the target frameworks. A compiled *DomainModel* requires a `persistence.xml` configuration file to bootstrap the JPA framework. However, when we actually want to use this *DomainModel* code in the Seam framework, different settings in `persistence.xml` are necessary. Still, we cannot directly generate these correct settings when compiling the *DomainModel*. This would entail that the *DomainModel* compiler cannot be used to create a stand-alone persistent domain model. Clearly, we would lose the ability of gradual introduction of this DSL into a project, one of the proposed advantages of having modular technical DSLs (Section 5.1.1). Our solution is to have the *WebLayer* DSL recreate the configuration file, adapting some of the settings to the new environment. Effectively this could be seen as a sort of overriding mechanism. Fortunately, the *DomainModel* interface file contains enough information to create the new version of `persistence.xml`. It does, however, mean that some JPA specific configuration knowledge has leaked (and is duplicated) into the *WebLayer* compiler, which is somewhat unsatisfactory. Apart from that, the same issue exists for libraries that are necessary to compile the generated code. In practice, the DSLs generate code into a 'skeleton' directory containing the appropriate libraries for the combination of target frameworks.

With this thesis, we are trying to prototype an environment of composable, interacting DSLs based on realworld application frameworks. In our opinion, we succeeded to create a workable (though admittedly small) system of interacting DSLs. However, it is also necessary to critically reflect upon how open

and modular the approach we have taken really is. First of all, the concrete format of our interface files, the ATerm format, is chosen since it is the native data format of our implementation tool. Fortunately, many libraries for reading and manipulating ATerms exist in different languages, meaning that possible DSL compiler implementations in other languages are not unduly disadvantaged.

More important is to assess how modular the languages themselves are. One observation is that the 'type system' as introduced by the *DomainModel* language, influences both of the other DSL designs. The *WebLayer* language knows and uses the details of the structure of concepts in its language definition. Also, one could almost state that the type checker of the *WebLayer* language (and *BusinessRules*, if we would have made an actual implementation) is parameterized by the format and structure of concepts as defined in the *DomainModel* language. This shared understanding of types and data structures is necessary to allow for any meaningful interaction, but it might also be a restriction to future DSLs that want to fit into this system. In the following section, we reflect upon our choice to work with separate compilers for the DSLs.

5.1.6 Dependencies

The introductory part of this chapter already established the hierarchy that exists between our DSLs. This hierarchy also enforces a compilation order between the models. It is trivial to automate this order using traditional software engineering tools (e.g. `Make`, `Ant`). In fact, our implementation provides a generic build script that incrementally compiles the DSL sources following this hierarchy.

However, if we were to have mutual references between models in different DSLs, this approach fails. A GPL compiler (e.g. `javac`) often does allow cyclic dependencies between modules. In that case, all files in the cycle are presented to the same compiler in the same compilation pass (assuming none of the modules have been separately compiled yet). Semantic checking can be performed using all the information of the members of the cycle at once. Note that the compiler still chooses an order for the compilation, and therefore requires special facilities to compile the first module in a cycle (i.e. it has to use assumptions for the still uncompiled members).

In our situation, with separate compilers for each DSL, this would mean the compilers would have to invoke each other, providing the necessary information for compiling the other model. This is certainly possible, though the additional complexity does not warrant the few use-cases that might benefit from the construction, especially in our domains. So far, we have not encountered compelling use-cases for having such cyclic dependencies between DSLs. If such a dependency does arise in the design of DSLs, most probably a better design would be to put abstractions that must refer to each other in the same DSL. Disallowing cyclic dependencies between modules is not unprecedented, Component Pascal[3] [?], for example, only allows uni-directional dependencies.

Alternative configurations were considered during the design of interaction between our DSLs. A possibility is to separate the front-end and back-end of our compilers into separate (in our case Stratego) libraries. This would allow the *WebLayer* compiler to invoke the *DomainModel* front-end on the imported domain model, obviating the need for the interface files. However, such a configuration limits openness of a system of interacting DSLs. If we wanted to implement another DSL that uses *DomainModel* definitions in a different language than Stratego, the complete front-end of *DomainModel* would have to be duplicated in that language. Having an interface file in a generic format (be it ATerm or XML) reduces the coupling between the DSL compilers to the signature of the interface. Parsing the interface file can be done using standard libraries available in virtually any programming language.

5.2 Interaction between DSL and user-written code

In this section we look at two mechanisms that integrate user-written Java code with generated code from the *DomainModel* compiler. Before looking at these mechanisms, we explore the motivation for allowing host language code alongside DSL code. Then, we will explain the rationale and implementation of data validation for members of *DomainModel* concepts, which is our first mechanism to add custom code. In this implementation, we believe to have found an interesting way of unifying user-written code and DSL abstractions. Furthermore, we will briefly reflect upon the consequences of the fact that

[3] A descendant of the Oberon languages rather than Pascal.

Java annotations can be inlined in *DomainModel* concept definitions, which is the second mechanism to integrate host language code and the DSL.

5.2.1 Motivation

Why should a model-driven (DSL based) software development method allow users to add custom written host language code? At first sight, such a possibility detracts many of the advantages of the DSL approach. However, DSLs are by definition restricted in the amount of variability they offer. Furthermore, it can be argued that certain tasks (i.e. strongly algorithmic problems) can be described better, or just as succinctly, in GPL code as in a DSL. In that case, forcing the user to use a DSL is actually detrimental.

Whenever the limits of variability in a DSL are encountered, several scenarios arise. First, the DSL designer can argue that the limitation is intentional, and that a different language should be used in case the current one is too restrictive. It is the easiest and most unsatisfactory response. Second, the DSL can be extended to accommodate for the missing feature. However, this most probably means the language must be changed, if possible at all, with all the associated problems. The last possible scenario is to recognize that a DSL, implemented by translation to GPL code, can allow access to (parts of) its generated implementation. In our opinion this is a valuable option if and only if this access is a natural (preferably controlled) extension to the DSL, and voids the advantages of the model-driven approach as little as possible. Furthermore, writing custom code should require as little knowledge as possible of the translation scheme of the DSL compiler. In Section 5.2.3 we compare various existing mechanisms and our approach in the light of these criteria.

5.2.2 Extended types

In an early version of the *DomainModel* DSL, we introduced *special purpose* types besides the native types such as String and Integer. The motivation for these types was that native types only delineate sets of values familiar to programmers. When modeling a data domain, one often needs to restrict the allowable set of values with respect to the semantics of the actual domain. For example, an age need not only be numeric but should also be non-negative, and bounded within reason (where the limits of reason vary from application to application). However, that last observation already hints at the fact that our initial approach of providing such types with extended semantics (i.e. an URL type) as built-in DSL constructs is too restrictive. While useful, it will never be complete. Therefore, a provision must be made to allow the DSL user to define additional semantics for value types of concept members. In the remainder of this section, we assume these extended semantics to be validation logic. However, the mechanism presented can be used for other extensions as well. Two quite different solutions can be distinguished for implementing additional semantics:

1. Introduce DSL abstractions to model validation logic.
2. Let the user write validation code, and link it to the model expressed in the DSL.

From a conceptual point of view, the first solution is the most attractive. It would be nice to have a declarative means of specifying additional properties on concept members within the DSL. Of course the question then arises what kind of properties we want to describe, hence what abstractions need to be introduced into the language. A prime candidate for embedding validation logic in the *DomainModel* language was regular expressions. This well known formalism allows for a concise and familiar notation of a validation constraint. Conceivably, a concept member for an email address could be expressed like this:

```
email :: String[^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$]  (required, unique)
```

However, several problems with such a regular expression embedding can be identified:

- Only strings can be validated, whereas we also want to validate, for example, Date and Integer values.
- Validation is binary: it either passes or fails, without possibility for precise feedback.

- No contextual information can be used in the validation (e.g. the current date).
- What if other semantics besides validation need to be described?

In our opinion it is unlikely that abstractions can be found that introduce the desired functionality in a flexible manner, while still being more succinct than implementing the validation directly in Java code. This realisation is what prompted the addition of *extended types* in our prototype. An extended type is a definition that refines native types with validation semantics (and possibly more, as discussed towards the end of this section). This definition is external to the *DomainModel* definition in that it is described in a different source file. An example, implementing a date type that may only contain a date in the future, is shown in Listing 5.1.

```
import java.util.Date;

public class FutureDate extends Date {

  public String validate(Date date) {

    if (date.before(new Date()))
      return "Dates in the past are not allowed";
    else
      return null;

  }

}
```

Listing 5.1: Extended type FutureDate

This example shows that writing an extended type amounts to writing a Java class. The *DomainModel* compiler parses these extended type definitions and integrates them in the generated code for a domain model. Consequently, syntax errors in the definition are detected at *DomainModel* compile time, whereas semantic errors are only detected when compiling the generated Java code. Since the transformation does not change the statements provided by the user, these errors will be fairly easy to map to the original source.

Continuing with Listing 5.1, we dissect this class definition in order to introduce the imposed format. The class name indicates the name of the extended type, while the mandatory `extends` clause indicates from which native type this extended type derives. Interestingly, this class does not actually derive from Date (which is a final class, derivation is prohibited by the Java compiler). We use the `extends` solely to let the user indicate the mapping for this extended type in a natural manner. Next, an implementation for the `validate` method must provided. It must conform to the following signature:

```
String validate(Type t)
```

where `Type` must be substituted with the actual native type the user is building an extended type upon. The existence of a method with the correct signature is checked by the *DomainModel* compiler. The access modifiers are ignored (replaced) in the translation scheme. In the body of this method, the user must adhere to one convention: return `null` if validation succeeds, or return a descriptive string if validation fails.[4] Furthermore, we guarantee that the method will only be called with non-`null` values. There are no restrictions on which constructs may be used in the body of the `validate` method. Libraries can be called to aid the validation. The user is responsible for adding the correct import headers to support these libraries, and to make these libraries available at the compilation of the resulting Java sources if they fall outside of the Java standard libraries. An example of the usage of Java regular expression libraries to validate a phone number is presented in Listing 5.2.
The example also shows we can give arbitrarily precise feedback as to why the validation fails.

Now, using an extended type within the *DomainModel* DSL is straightforward:

[4] Returning null as special value is a typical Java approach to mimic a data type like `Maybe` in Haskell.

```
import java.util.regex.*;

public class PhoneNumber extends String {

  public String validate(String s) {

    if(s.indexOf('+') != -1) return "No international prefixes allowed";

    Pattern p = Pattern.compile("[\\d+\\s-]");
    Matcher m = p.matcher(s);

    if(!m.matches())
      return "Phone number may only contain digits, space and hyphenation";

    String[] tokens = s.split("[\\s-]");
    if(tokens.length < 2) return "Phone number must consist of two parts";

    return null;

  }

}
```

Listing 5.2: Extended type PhoneNumber

```
number :: PhoneNumber
```

The name of the extended type can be used instead of the underlying native type (String in this case). Result of using this extended type is that a call to the `validate` method is made for this member when the enclosing concept is being persisted. A special JPA hook (using the `@PrePersist` and `@PreUpdate` annotation, is leveraged to make sure only valid data is stored. Note that the invocation of the corresponding `validate` methods must be guarded with a `try/catch` block to catch any runtime errors, since we cannot assume their absence in the custom written code.

Currently, the *DomainModel* compiler scans the directory, in which the *DomainModel* source file being compiled is located, for files with the extension `.dmtype`. By putting the code of Listing 5.2 in the file `PhoneNumber.dmtype`, the compiler will recognize this type. Thus, extended types are orthogonal to *DomainModel* definitions and can be reused across different models. It is also possible to let the compiler scan a fixed directory at each compilation pass, allowing for a library-like pool of extended types which can be leveraged in different projects. This is not currently implemented. In principle, it would have been possible to inline extended type definitions in *DomainModel* source file. However, this would constitute mixing code of varying abstraction levels in one file, which is undesirable. Using an extended type just by referring to its name, without implementation details, matches the declarative nature of the *DomainModel* language. Furthermore, having the definitions in separate files allows the user to leverage an existing `Java` editor. If the `Java` code would have been mixed with DSL code, this would be problematic.

The translation (or rather, transformation) of extended types consists of creating a new class for each type definition, containing a static method `validate`, containing the body of the method with the same name from the input file. This class is placed in a validation sub-package in the hierarchy of the generated code. Furthermore, the `import` statements are preserved. It also would have been an option to let the user write this translated class directly. However, there are some small advantages to our approach. First, the compiler now controls the namespace in which the translated class is put, allowing the fully qualified name of the `validate` method to be emitted in the interface file without having the user to make this information explicit. Consequently, the extended type definition is more decoupled from a specific *DomainModel* definition when it lacks a concrete package statement, hence it is better suited for reuse. Second, the compiler inserts the correct access modifiers (`public` and `static`) that are necessary to use the validation in the rest of the generated code. Thus, the user need not be aware of this convention. Furthermore, the mapping between the extended type and the native type is made explicit by extending a native type in the input file to the compiler, and this mapping is checked for validity. If the class were to be written directly by the user, and inserted into the target source tree,

another means of specifying this mapping should be available. Lastly, we believe that this approach is better equipped for future extensibility. When additional code needs to be generated for extended type implementations, we can do so freely in the current scheme. Also, additional methods in the input file can be introduced for more user-defined behavior (e.g. a parsing/conversion function or other type specific functionality). Both are harder to achieve when we let the user write the implementing class directly.

Of course, we can also identify downsides to the approach taken. These are mostly related to the restrictions on the format of the class that are imposed by this approach. First of all, the user must centralize all code in the `validate` method, since additional methods are not copied into the implementing class. It is, however, possible to delegate work to imported classes and/or libraries. Moreover, the functionality is expected to rarely surpasses the limits of reason with respect to the size of a method body. Second, no inheritance is possible since the `extends` clause is used to steer the mapping of the type. Again, we postulate that this is not a problem in practice. Since the user will never instantiate the class (or reference an instance for that matter) being written, the polymorphism offered by inheritance is not very useful. If the user wants to take advantage of implementation inheritance, however, this is not possible either. Rather, the same can be achieved through delegation rather than direct inheritance.

5.2.3 Comparison

In the previous section we described our approach of enabling the user to write *custom* code in a principled manner. Various other methods of integrating custom written GPL code with generated code exist. In this section, we compare our approach to different approaches found in production systems.

The MS Software Factory (described in more detail in Section 7.2) implementation mainly targets C# when generating code from its models. The prevailing line of thought is that 80% of the target code should be derived from the model, whereas the remainder may be hand-written code. Therefore, an additional feature was devised for C#: *partial classes*. This feature allows a single class to be defined in multiple source files. Logically, the C# compiler views these separate sources as one monolithic class. Thus, generated code is placed in one file of a partial class, leaving any functionality that must be custom coded undefined. Then, the user can write a complementary partial class without touching the generated source. Still, some awareness of what is happening in the generated partial class is required. Conflicts may arise in the case of overlapping or inconsistent definitions.

Java does not have a partial class mechanism. Model driven approaches in this field typically follow one of three other patterns, regarding the integration of custom code:

1. Extend a generated class through inheritance.
2. Modify/extend generated skeleton.
3. Modify a generated class, within *guarded regions*.

The first alternative is somewhat similar to C#'s partial classes. Only, with inheritance we create an additional class, instead of adding functionality to a generated class. This is an important distinction, since the surrounding generated code must be aware of the fact that a subclass of a generated class exists, containing user-written extensions. Naming conventions for the subclasses can help in this regard, or forcing the user to only override methods declared in the generated superclass. Often, design patterns are employed to ease the process for the user writing custom code in such a setting.

The second alternative can be discarded quickly. Generating a class only once and then leaving it to the user for modification puts the burden of synchronising the model and the modified code on the user. This is something that should be avoided at all costs.

The third alternative is also based on one-time generation of classes. Only, the user may solely change or fill pre-determined parts of these classes. Parts that may be changed are flagged (typically these are delineated by indicative comments) and guaranteed to be left alone by the DSL compiler in subsequent compilation passes. Since the user is writing (or even changing) code within generated code, great care must be taken with regard to dependencies on the surrounding code. It should be made clear by the DSL documentation what a user may assume and what may be referenced in the custom code. Also, the programmer must resist the urge to change any code outside of the guarded regions. Finally,

managing guarded sections is the responsibility of the DSL compiler, and in practice this is not always easily accomplished. An example of such a system is OptimalJ, which is discussed in Appendix B.2.

One caveat of the abovementioned generate-and-extend mechanisms is that it entails implicit mixing of source (model) and target code. Modified classes and additional hand-written classes now must be considered as source for a project as well, even though they reside in the generated target tree and most likely no explicit reference exists in the model to these artifacts. In practice, this means that the generated code (including the modifications and additions) is also checked in into version control systems, which a is rather unsatisfactory form of *source scattering*. On the upside, if the users makes a mistake in the custom code, the compiler errors are given in terms of code the user has written directly. Our approach does preserve the code on the statement level, but it is transferred to a new class by the DSL compiler.

The largest difference between the above mentioned existing approaches and our approach is that the custom code is considered *input* to the compiler, rather than a patch or extension on generated code. Thus, the source scattering mention earlier is minimized. Also, there is a clear and explicit link between the custom code and the usage in the *DomainModel* code. While the difference might not be earth-shattering, we believe our approach has subtle advantages. The mechanisms described in this section suffer from a very tight link of custom code to a specific model instance (i.e. code generated for a specific model). The extended type mechanism offers more orthogonality for the user-written code. Furthermore it can contain information (such as the type mapping) that need not end up in the target code. Since the DSL compiler has full access to the AST of the custom code, additional analysis could be performed to ensure the custom code adheres to the desired contracts. For extended types, this is implemented only partially: we do check for the existence of a method with the correct signature, but do not check for conformance to the 'return null on success' rule. Still, the possibilities at DSL compile time are evident. However, there are costs attached. A mechanism must be crafted for each extension point we want in our DSL, whereas the other approaches all more or less 'come for free', as long as the generated code offers the appropriate extension hooks.

5.2.4 Inlined Java annotations

This section introduces the second link between DSL code and host language code. We recall that concept members of a *DomainModel* concept cannot only have domain specific annotations on them, but can also have Java annotations:

```
member :: String (@Column(name="column1"), unique)
```

This is another instance of mixing host language code with DSL code, albeit in a very restricted setting. In this case, the Java syntax for annotations is embedded in the DSL syntax. In the previous sections we established that this mixing is not generally a good thing. However, there are several reasons why this approach was chosen. Prime reason is that the usage of Java annotations is exception rather than rule. Only when the defaults of the generated annotations (which are based on type and association kind of the member) do not suffice, users should add Java annotations. Rather than modifying generated code, with all of the pitfalls discussed in the previous section, we allow this to be done next to the domain specific annotations. Rather than trying to wrap these annotations for exceptional cases in DSL abstractions, we chose to open up the way for users to get specific. Furthermore, this embedding fits in naturally with the domain specific annotations, and may be ignored without ill effect in the case of different back-ends. However, by introducing this hook we also introduce the possibility of generating classes that do not compile. Fortunately, the syntax embedding guarantees us that if parsing of the *DomainModel* source succeeds, the Java annotations contained therein are syntactically valid (i.e. it is not treated as an unstructured string). While the user is responsible for writing only semantically correct annotations, not all semantically correct annotations that can be written will lead to a compilable class. This can be attributed to the fact that the *DomainModel* compiler emits annotations as well, depending on the type, association and *DomainModel* annotations of the member. In Java only one instance of an annotation may be provided on the same syntactical token. Therefore, *DomainModel* generated and custom supplied Java annotations might clash. The example from the beginning of this section shows the problem. Since the *DomainModel* annotation `unique` leads to the Java annotation `@Column(unique=true)`, the user supplied Java annotation (`@Column(name="column1")`) clashes. Since

compilation errors of the generated Java code due to such issues are confusing for the DSL user, we want to minimize the possibility of these errors occuring. To this end, an additional transformation is applied, merging the bodies of conflicting annotations and removing duplicate annotations. For our example, this means the following annotation is emitted:

```
@Column(unique=true, name="column1")
```

This transformation can be safely applied, since it is meaning preserving. Note that we cannot guarantee that every conflict is resolved. For example, the bodies of two annotations with the same name might also clash (e.g. if the user supplies `@Column(unique=false)` in the example). The fact that we are not able to resolve them, does not mean that the problem can only be discovered at the compilation of the generated code. The DSL compiler can detect the detect the clashing bodies. However, a large class of problems can be automatically resolved by this simple reduction strategy. If we would really want guarantee that all conflicts are resolved, the user-supplied annotation could override any conflicting generated annotation (or vice versa). However, the semantics of the DSL code do not become much clearer by introducing such a defaulting rule.

5.3 Concluding remarks

In this section, we elaborated on mechanisms for interaction between DSLs and DSLs with host language code. The interaction between *DomainModel* and *WebLayer* models the interaction between the two corresponding libraries. Modularity within a single DSL, however, is not implemented. Since our DSLs are stand-alone, such a mechanism must be explicitly crafted for each DSL. In this case, a natural embedding would definitely be advantageous, where the module system of the host language carries over to the DSL. However, the foundation for a module system in the DSLs has been by means of the interface files. These can also be used to exchange information between modules of the same language, though the extent of the information necessary might be different.

Interaction between DSL code and host language code ideally only happens to customize a certain project, created with a DSL, that needs a bit of customization. In practice, however, MDSD solutions revert to letting the user write GPL code whenever a DSL falls short. Either creating a DSL for the desired functionality is unfeasible, or unpractical (i.e. doing it in GPL code is just as much or less work). Consequently, adding custom code to code generated from the DSL should be as natural as possible. In this section have shown two mechanisms (extended types and Java annotations in *DomainModel* code) that improve upon current practice.

Chapter 6

BusinessRules DSL

Chapters 3 and 4 introduced two DSLs, and the previous chapter introduced the interface file mechanism that supports their interaction. This system is aimed at creating persistent domain models, and web-applications to present and modify such domain models. However, the action language introduced in *WebLayer* only provides a constrained set of actions. These actions all relate to *raw* data processing, e.g. assignments, list manipulation, save actions, and delete actions. Sometimes this is enough, but often more advanced actions are necessary. We will call such advanced processing of data *business rules*, since typically these actions encode business specific policies or rules. Examples of business rules range from mortgage calculations to checking the consistency of data (e.g. fraud checking of insurance claims). These two examples already show that the technical domain of business rules (often called the service layer in the three layer web-application model) covers a vast range of business domains. Creating a single unified language for business rules in our opinion is not possible, and would amount to creating a general purpose language.

Still, we want to incorporate such functionality into our DSL based web-applications. Therefore, the goal of the *BusinessRules* definition presented in this chapter, is to not let the action language of *WebLayer* deteriorate into a GPL, while still allowing actions beyond mere data processing to be defined and used within *WebLayer* (and possibly in other DSLs as well). In this chapter, we want to show how the system of DSLs can grow beyond its current state.

6.1 Language Description

The following description of *BusinessRules* is different from the two descriptions of the previous DSLs. Recognizing that no single DSL will fulfill all the needs for business rules, we have not actually implemented a business rules DSL for a certain domain. Rather, we look at a mechanism that tries to capture the technical commonalities that can exist between business rule DSLs. The premise is that business rules can be viewed as isolated actions that act on domain data in ways beyond the rudimentary actions that can be expressed in the *WebLayer* action language.

Figure 6.1 presents an example of how business rules can be invoked from within actions. In the example, we invoke two business rules from a (non-existent) banking DSL. It comprises a page with two page parameters. Besides showing an account and edit form for a customer, there is an action element that consists of four statements. The first statement is a standard assignment as we have already seen in Chapter 4. Also, the last `save` call is a built-in statement of the action language. In between these two statements we see two new statements that invoke business rules. A business rule invocation looks like a `Java` method invocation. Concepts that are necessary for the checks or computations are passed as parameter. The call to `createAccountNr` also returns a concept, which is assigned to the `number` member of the `Customer` concept.

While the analogy to a method call is correct, there is more to invoking a business rule than just passing a parameter and getting a return value. In particular, the application of a business rule can fail. In that case, we want to present the reason for failure to the user of the application (see Figure 6.2 for a concrete example), or even suspend any subsequent actions. Furthermore, all information passed to and from business rules is in terms of *DomainModel* concepts. Also, the concrete interaction of code generated by *BusinessRules* DSLs and *WebLayer* may take on different forms. In the *BusinessRules* definition, we want to take care of these issues.

```
page EditCustomer(Customer c, Account a) {

  form(
    show(a)
    input(c)
    action("Add account",
            a.customer = c
          ; verifyAccount(a)
          ; c.number = createAccountNr(a)
          ; a.save()
    )
  )
}
```

Figure 6.1: *WebLayer* page invoking business rules

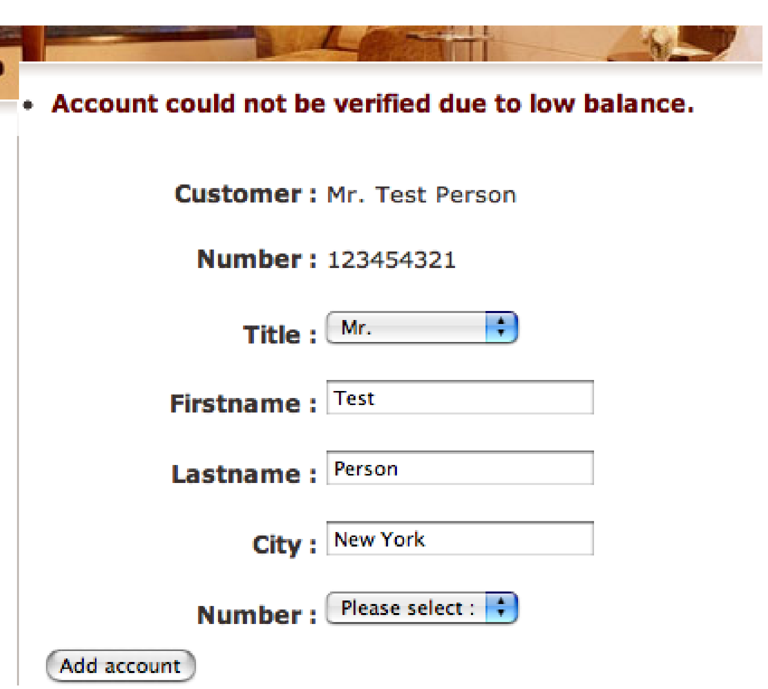


Figure 6.2: Feedback when the application of a business rule fails

6.2 Interface

To use business rules inside *WebLayer*, they must be made available to the *WebLayer* compiler. We achieve this by using the same interface file mechanism that was introduced in the previous chapter. Therefore, an implementation of a *BusinessRules* compiler must emit both the implementing code, and an interface file. The specific format of the *BusinessRules* interface file can be seen in Figure 6.3.

Following the explanation of Section 5.1.4, we will investigate the explicit as well as the implicit part of the interface. A *BusinessRules* interface consists of a term containing the name of the compiled module (`"banking"` in this case), and a list of `Rule` terms. Each `Rule` contains the following elements:

Name	to reference the rule within the *WebLayer* actions.
Parameters	a list of parameter types.
Return type	type of return value (may be `Void()`).
Modifiers	indicating additional semantics of the rule.
Access hooks	publishing the means through which the rule can be invoked.

```
BusinessRules(
   "banking"
 , [ Rule( "verifyAccount", [Concept("Account")], Void()
        , [ HaltOnFailure() ]
        , [ ("method", "com.corporate.verify")
          , ("webservice", "http://corporate.com/accounts/verificationWS")
          ]
        )
   , Rule( "createAccountNr", [Concept("Account")], Concept("AccountNumber")
        , []
        , [ ("method", "com.corporate.createNewAccount") ]
        )
   ] )
```

Figure 6.3: Interface file for *BusinessRules* implementations

Most of these elements are fairly self-explanatory. The parameter and return type information can be used to type check rule invocations. The last two, *modifiers* and *access hooks*, pertain to the issues raised in the previous section. The latter is necessary for the *WebLayer* compiler to generate code that links with the rule. As can be seen in Figure 5.3, it comprises a list of key-value pairs. This allows for multiple hooks into the *BusinessRules* generated code. The most obvious way, by creating a static method for the application of a rule, is tagged with `method`, mapping to the the fully qualified name of the method. However, alternative manners to invoke a business rule are conceivable, and should not be excluded beforehand by our interface specification. To illustrate this point, we added a `webservice` entry to the list, referring to a webservice *end-point* URL where the business rule can be invoked. Note that it is the responsibility of the actual *BusinessRules* compiler to generate and publish such a webservice, if such a hook is put in the interface. There are no restrictions on how many hooks are published in the interface, and what kind of hooks they are. Naturally, the compiler reading the interface must know how to use at least one of the published hooks.

The example in Figure 6.3 shows the `HaltOnFailure()` term as modifier on the `verifyAccount` rule. Its interpretation moves us into the realm of the implicit interface. With this modifier, we indicate that should the application of this rule fail, no further action is to be taken. If omitted, failure of the rule will not prevent the rest of the actions from taking place. Currently, we do not have any other modifiers in mind, but this part of the interface is easily extendable.

The `HaltOnFailure()` modifier does raise the question how success or failure of a business rule is communicated. Again, we define this is as part of the implicit interface. When considering a business rule, it accepts concepts as parameter and possibly returns a concept. However, we specify that the return value must be wrapped in a `Result` helper class (Listing 6.1). This class is parameterized by

```
public class Result<R> {

  public enum STATUS { SUCCESS, FAILURE }

  private STATUS _status;

  private String _message;

  private R _result;


  public Result(String message){
    _message = message;
    _status = STATUS.FAILURE;
  }

  public Result(R result) {
    _status = STATUS.SUCCESS;
    _result = result;
  }

  public STATUS getStatus() { return _status; }

  public void setStatus(STATUS status) { _status = status; }

  public String getMessage() { return _message; }

  public void setMessage(String message) { _message = message; }

  public R getResult() { return _result; }

  public void setResult(R result) { _result = result; }

}
```

Listing 6.1: Helper class for *BusinessRules* result values

the type of the returned concept, to allow for type safe processing on the receiving end. Furthermore, it contains a status flag indicating success or failure, and a message field that can contain the reason for failure of the application. So, invoking the `createAccountNr` rule yields a `Result<AccountNumber>` object, containing all information necessary to handle the result in *WebLayer*.

6.3 Concluding remarks

In this chapter we looked at a generic interface for business specific DSLs. The result is an operation-centric interface, with additional semantics added to the invocation of an operation. This implicit part is governed only by the specification of the interface given in this chapter, contrary to the interface of *DomainModel* which is governed by the existing framework we target. This shows that DSL interaction need not be constrained by third party library specifics in all cases, even though a *BusinessRules* implementation might very well use such libraries.

Another result of allowing business rules inside the *WebLayer* action language, is that possibilities for control flow constructs arise. For example, branching on success or failure of a rule application allows different actions to take place, and redirects to different pages may occur depending on the outcome.

Without an actual implementation of a DSL compiler that uses the *BusinessRules* interface, this effort can be seen as a foreign function interface from *WebLayer* actions to `Java` methods. We note that since the interface is quite generic, any specific knowledge of a rule is encapsulated in the rule implementation itself, and not published in the interface. Consequently, this allows rules from arbitrary *BusinessRules* implementations to be plugged into *WebLayer* effortlessly. Specific directives or properties for a rule can still be encoded as a modifier, which must be interpreted by the DSL that reads the interface. Alternatively, one could devise a more specific interface for each *BusinessRules* implementation, only then it is not possible to plug in such an implementation into *WebLayer* without adapting the *WebLayer* compiler.

Chapter 7

Related work

The goal of this chapter is to place our work in a broader perspective. To achieve this, we look at both theoretic aspects as well as concrete systems, giving descriptions and comparisons to our system. In Section 7.1, we look at theoretical work regarding the composition of models. In Section 7.2, the Ordina Software Factory is introduced and compared, a system with many similarities to our DSLs. Furthermore, we look at an alternative implementation environments for DSLs (Section 7.3), meanwhile discussing the role of IDEs in model driven development. We also discuss some existing DSLs for web-application development (Section 7.4). We conclude this survey of related work by looking at the problem from a rather different angle, namely an idea called *active libraries* (Section 7.5).

7.1 Model composition

Linking separate DSLs can also be seen from a model-based point of view. Kurtev et al. introduce special operators in a position paper [?] to compose distinct UML models. This is important, since UML itself does not provided primitives with respect to modularity. However, their composition does not take code generation scenarios into account. If code were to be generated, this must happen from the composed model, voiding our notion of true separation between generators. Furthermore, this composition only takes into account the general semantics of UML, and no semantics pertaining to the actual model expressed in UML. Consequently, the operator definitions work for arbitrary models in UML, but domain specific safety notions cannot be enforced. Our interface definitions are particular to the actual DSL, which allows us to be more precise and safe, at the cost of being less generic.

Closer to our notion of interacting DSLs is the work of Stirewalt et al. [?] on what he designates as the 'model composition problem'. Stirewalt describes three different declarative DSLs for user-interface generation, an approach which he calls Mastermind. Each of these DSLs accounts for a different aspect of the user-interface: presentation, dialogue, and interaction with the underlying application. The code generators for each of these DSLs are completely independent. Furthermore, the correctness of the composition of the results of these independent generators can be determined statically.

The key idea is to formalize the DSLs as concurrent agents that synchronize on common events. Moreover, a run-time environment supporting this view on the DSLs is developed. The run-time environment is the central communication hub between the generated modules of each DSL. Generated modules communicate solely through a notification mechanism (message passing) of the run-time environment, and this is guaranteed to compose correctly. The agent formalism used to provide this guarantee is called Lotos. In this formalism, sequencing constraints on messages between agents (in this case models in a DSL) can be expressed. Lotos specifications for DSL programs are automatically derived from these programs and used for the correctness guarantee. Hence, the DSL programmer does not need to know anything about Lotos, or even the run-time environment for that matter. However, the derivation process must be aided by the DSL programmer, in the sense that actions that are of interest for other models must be explicitly indicated in the DSL source. An overview of the system, as presented in Stirewalt's paper [?], is shown in Figure 7.1. The run-time environment is called the 'synchronization module'. Stirewalt shows with this solution how formal methods (i.e. Lotos) can steer the actual implementation of independent code generation.

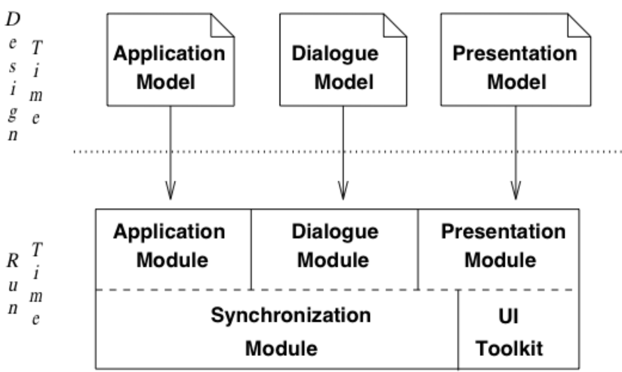


Figure 7.1: Multiple independent code generators

The code generators not only emit modules in the target language, but also an additional file containing the events and messages that can be emitted from this model. These files together are used by an *integrator* which sets up the run-time environment to cope with these messages. It is interesting to note that this integrator works globally on all compiled modules their additional files. It is not clear whether this integrator works for arbitrary combinations of DSLs conforming to Stirewalt's model composition formalism, or whether it is specific to this exact combination of DSLs.
Furthermore, their application model essentially creates abstract classes that need to be subclassed by the user. These classes, in the target language, are necessary to implement the desired application logic behind the UI. In effect, this is an instance of DSL interaction with host language code. This interaction is ignored as far as their formalism is considered. The paper states 'As long as the details of these extensions [subclasses] do not trigger behavior in dialog or presentation models, this application behavior may be ignored when defining model composition'. The aforementioned guarantee of correct composition therefore depends on the compliance of the user in this practical aspect.

An interesting parallel is that the three DSLs that are created, are based on the Presentation-Abstraction-Control (PAC) architecture (a lesser known variant of the Model-View-Controller architecture). As with our approach, the DSLs represent technical components of an existing architecture. However, instead of having hierarchical models, they all are independent. Still, at run-time communication must occur between the generated modules. Unfortunately, the Mastermind papers do not show concrete DSL specifications to make clear how exactly this is achieved in such an independent manner. Moreover, another paper on the system [?] states that '.. [the dialogue model] acts as the glue between the presentation and application models'. It is unclear how this statement relates to the assertion that Mastermind models are independent definitions, which are independently transformed to target code.

Because PAC, and with it most other GUI architectures, is based on events, notifications, and callbacks, there is a natural fit to the run-time solution chosen. However, it is unlikely that Stirewalt's approach, having a central run-time hub with explicit message passing, can be employed for arbitrary software architectures. In particular, when existing technologies are targeted that are not event-based, as is the case with our technical domains.

7.2 Ordina Software Factory

The Ordina Software Factory (Ordina SWF) is a modeling environment created by the Dutch software development company Ordina NV. It is targeted at web-application development in the .Net environment. As such, it was an important inspiritation for our prototype in this particular domain. In this section we explore the differences and similarities between our approach and the Ordina SWF. Our knowledge of the Ordina SWF is limited to two meetings with the lead-developer, one internal document and the (mostly marketing related) information published by Ordina. In this section we try to provide a picture as accurate as possible, based on casual examples we encountered. However, many details of the languages that comprise the Ordina SWF are protected as trade-secret.

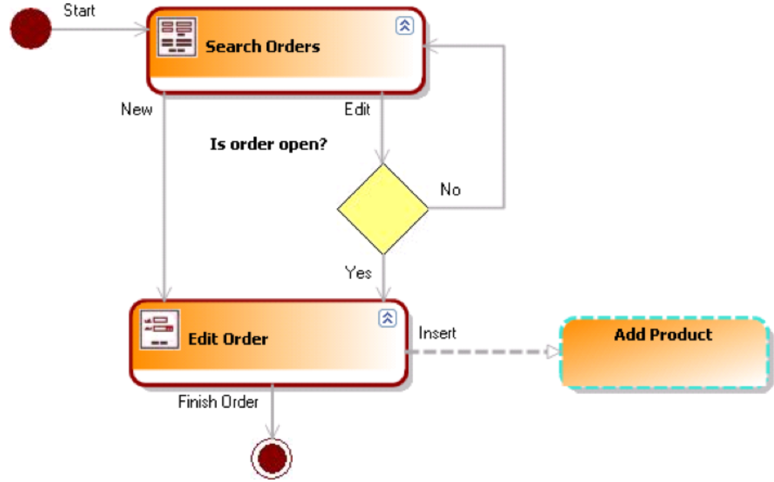


Figure 7.2: Model in Web Scenario DSL, taken with permission from an internal Ordina document.

The Ordina SWF consists of four DSLs created using the Microsoft DSL Tools (also known as the Microsoft Software Factory suite). MS DSL Tools is a relatively new endeavor to create modeling languages. It does not adhere to the MDA standard (as introduced in Section 2.5), but shares many characteristics with such environments. The DSLs that can be created are visual, and can only be used within the Visual Studio 2005 IDE. A very basic (text, not AST-based) template language is employed to generate code (predominantly C# code) from a model.

An example of an actual 'DSL program' can be seen in Figure 7.2. With these DSLs, web-applications can be developed according to the standard .Net architecture. As in our approach, different existing libraries are targeted from the languages. Nevertheless, the DSLs themselves are as technology-agnostic as possible. The goal of the Ordina SWF is to generate around 80% of the necessary code for a project, and to manually code the remaining part. Especially business logic is something that is left to be manually coded. The four DSLs comprising the Ordina SWF are:

1. Business Class
2. Data Contract
3. Web Scenario
4. Service

In the Business Class DSL, all domain (entity) classes are modeled. This DSL is very similar to our *DomainModel* language, only in a visual format. It does not allow for integrating validation of members (as is possible in *DomainModel* using extended types), but it does allow for the addition of *business rules*. Adding a business rule (which is just a name) in the model results in the generation of a corresponding empty method, to be filled by the user of the DSL.

The Service DSL mediates between the pages defined in Web Scenario and the Business Class data classes. Services define operations on data. Again, most of these services are left to be implemented in host language code. All wrappers (i.e. for *WebServices* calls) are generated.

The Data Contract DSL provides a decoupling of the Web Scenario from Business Class entities. Each Business Class entity also has a corresponding Data Transfer Object definition in the Data Contract DSL. Furthermore, aggregations of different Business Class entities, and special views on them can be defined in the Data Contract DSL. Especially this Data Contract DSL is interesting, since many models (or model elements) of this DSL are directly derived from Business Class definitions. Still, its viability is sustained by the need for different views on Business Class definitions and the fact that the reference architecture prescribes the use of DTOs between the different layers instead of passing business classes directly. We believe that we have obviated the need for such intermediate objects, since we allow projections or aggregations of concepts to be expressed in the *WebLayer* language. Thus, we observe that in the Ordina SWF the decomposition into different DSLs follows a prescribed .Net

architecture, whereas it is debatable whether every part of such an architecture should be retained in a DSL setting.

The Web Scenario DSL encodes what pages are part of the web-application, and what the flow is between these pages. This corresponds to the *WebLayer* DSL, which has similar goals. However, Figure 7.2, which shows a Web Scenario instance, already reveals some differences in the granularity of modeling capabilities. In *WebLayer*, a single page is composed of different page elements. In the Web Scenario DSL, on the other hand, a complete page is either a list, edit, or a view only page of a certain element defined in the Data Contract DSL. The example in Figure 7.2 shows two pages, a search page for multiple Orders objects and an edit page for a single Order object. Page flow is modeled by creating edges between page nodes. Generated artifacts from such a page definition are embedded in a 'master page', which is similar to our template.

It is not entirely clear how data-flow is modeled. The available examples all exhibit a master-detail relation, i.e. an edge going from a list (or search) page to a view (or edit) page. We believe that our *WebLayer* language in this regard offers more flexibility, making data-flow explicit by passing parameters to pages or by instantiating session variables.

So far, we discussed the different visual languages available in the Ordina SWF in relative isolation. It is clear that references between the models in different DSLs must exist. For instance, the example of Figure 7.2 refers to an Order object, which is in turn defined in the Data Contract DSL and the Business Class DSL. However, MS DSL tools does not allow for references between models, not even in the same DSL. Therefore, Ordina implemented a facility to allow this [**?**], on top of what MS DSL tools offers. This addition to MS DSL tools is called the *partial model* capability by the developers. Models in the Ordina SWF models publish details of their elements to a central repository, which is maintained by the Visual Studio IDE. The repository is named 'Non-persistent DSL Information Provider' (NDIP). This central repository can in turn be queried by the code generators of the DSLs. Therefore, only a notion of *provides* interfaces exist, which are all aggregated into the NDIP. Models can reference other partial models in the same DSL as well as partial models in other DSLs. All references between models are made by name, and additional information that is necessary for code generation is stored as key-value pairs. There is no formal description of what this additional information entails, and what is stored depends on the needs of the generators. A similar mechanism, called a reuse-repository is proposed by Nussbaumer et al. in the workshop paper 'Towards DSL-Based Web Engineering' [**?**]. Ordina implemented NDIP by hooking into the save event handler of MS DSL tools/Visual Studio. Whenever a model is stored, its information is published to the repository. Also, code generation is then performed.

There seems to be a many-to-many relationship between the DSLs. They all use the same repository, and can access each others information. However, the DSL designer can choose which constructs of a model are exported to the repository, so some encapsulation is possible. While there is an explicit notion of exporting information, importing information from different models is done implicitly by means of the code generators querying the NDIP. Hence, there is a notion of *provides* but not of *requires* interfaces, which makes the system fragile, since the code generation only has implicit dependencies.

One of the advantages of the run-time repository is that cyclic dependencies between models from different DSLs become a non-issue. References are made by name, and are allowed to be 'dangling' (i.e. reference a non-existent artifact), though this obviously breaks code generation. This is enough to be able to support the construction of mutual references through the NDIP.
The NDIP solution for interaction between models is a completely run-time solution. In this case, run-time actually means the time that the development environment is used. Furthermore, it is explicitly tied to and tailored for the Visual Studio 2005 IDE. A nice side-effect of this strong coupling is that the repository information can be used for refactorings and other IDE assisted tasks.

Additional code that needs to be written is separated from the generated code mostly by using *partial classes*. Partial classes are a feature of C#, which allows a single class to be defined in separate physical files, as discussed in Section 5.2.3. Generated code can be merged with user-written code through this mechanism. However, the programmer still has to know the internals of the generated code in order to write the correct extensions. Many methods in the Service DSL pertain to application logic, which cannot be modeled in the Ordina SWF DSLs. Consequently, only method signatures are generated by the DSL and the programmer has to provide the implementation for these stubs.

Visual and textual languages

MS DSL Tools (and therefore the Ordina SWF) only work with visual languages. Programs are created by drawing diagrams, and setting properties of diagram elements. However, our approach solely consists of text-based DSLs. We briefly compare both paradigms.

Visual languages are quite popular in current MDSD solutions. Quite possibly, this is due to the air of complexity that surrounds the creation of textual languages (i.e. creating a grammar, parsing). Visual languages, on the other hand, can reuse existing diagram editors and notation. In fact, we can equate the grammar of a textual language to the *meta-model* of a visual language, which dictates what graphical elements may be used in the language. The actual visual appearance of these elements is similar to the concrete syntax of a textual language.

One important difference between visual and textual languages is the way relations are encoded. In textual programming languages, references to abstractions (e.g., variables, classes, et cetera) are made symbolic, by name. The compiler resolves these implicit links by means of a symbol table (or some similar construction), and unresolvable names are reported as error. In a visual language, links between abstractions are generally created by drawing a connector between the elements. A crucial difference is that only valid links can be created by the user, avoiding a common source of compiler errors, or unintended behavior, while creating a program.

If no existing editor fits the design of a visual language, one must be created. Then, the effort necessary to create such an editor is substantial, compared to creating a grammar for a textual language. Also, the user of a visual language is tied to the environment of the language creator, whereas textual languages can be edited in any text editor, or in more advanced IDEs. An interesting problem for visual languages is version management. Version control systems (VCS) traditionally have good support for merging textual sources. Merging different versions of a visual program is a more involved problem, without a generic solution. Again, this calls for special purpose supporting tools.

A related issue is that of modularity of visual programs. Since generally links between abstractions are created explicitly, the most natural representation of a visual program consists of a complete diagram. However, for sufficiently large programs this representation tends to become incomprehensible. Again, excellent editor support is needed to traverse and manipulate such large diagrams. For example, by hiding details, navigational support, scaling, and so on. Or, diagrams can be split up into several smaller parts, as with the Ordina SWF (but note this was a custom add-on to MS DSL tools).

Textual programming is still the most widespread paradigm. Creating a visual language just for the sake of abandoning textual forms is therefore not a very good idea. That is not to say there are compelling cases in which visual languages excel. Especially when the concepts being programmed themselves are mainly visual (e.g., GUIs) or an accepted visual representation exists (e.g. state machine diagrams), visual languages can add value. On the other hand, lower level code is not easily (nor intuitively) captured in a visual language.

7.3 openArchitectureWare

OpenArchitectureWare (oAW) is a range of tools and components to modularly create generators from models to code. As such, it constitutes an alternative environment in which we could have implemented our DSLs. It is built on top of Eclipse. The core of oAW contains a workflow engine allowing the definition of transformation workflows, integrating different transformations.

A model can be either textual (using xText, a parser generator from BNF declarations that also emits a corresponding syntax highlighting editor), visual (UML and other languages), or come from any other front-end plug-in. Furthermore, checks on models can be performed using a constraint language. Checked models can be transformed to code using a proprietary transformation and templating language. This language does not provide the same guarantees on syntactical validity as using concrete syntax in Stratego does. The combination of the aforementioned components is commonly called a *cartridge* and equals a single DSL compiler like we created. Interaction between cartridges is possible, but not only in an explicit manner. If knowledge from two models is necessary in the compilation process, both need to be presented to the cartridge as input. This is made easy by means of the central workflow definition, but does not allow for true separation and interaction between DSLs.

In oAW, thought has been given to the interaction with custom code as well. Cartridges that need to do so, emit code that leaves hooks by means of appropriate design patterns (e.g., the proxy pattern or adapter pattern), or by letting the user extend generated classes. The user of the DSL then needs to know the conventions and naming of these hooks, or generated classes, in order to write the custom code in the host language. Interestingly, the developers of oAW recognized this is a very brittle process. Therefore, they introduce so-called *recipe frameworks*. A recipe framework is nothing more than a principled way of adding dead code (!) to the output of the DSL cartridge. This dead code contains references to custom code elements (classes, methods) that are expected to be written by the DSL user. If the user fails to provide the expected code, the dead code triggers errors when compiling the generated code. This hardly is a satisfactory approach, since care must be taken that the dead code is not optimized away by the compiler (this can be trickier than it seems), and the error messages provided by the host language compiler are not domain specific.

Role of the IDE

OpenArchitectureWare, and many other MDSD solutions award an increasingly important role to the Integrated Development Environment (IDE). This is completely understandable, since having a completely controlled development environment offers many possibilities for the vendors of MDSD products. Coupled with the fact that many products utilize visual languages, a lock-in with respect to the development environment is not uncommon. In our approach, we have shown that we can eliminate the dependency on a development environment when working with multiple interacting (textual) DSLs. Of course, an IDE could still enhance the user experience of the DSLs, for example, by offering syntax highlighting or code completion, and so on. We believe it is easier to create a pluggable IDE that handles multiple syntaxes and syntax-oriented concerns as mentioned before, invoking an external DSL compiler, than it is to create a pluggable IDE that supports semantic-oriented concerns for arbitrary DSLs. With semantic-oriented concerns we mean the semantic (type) checking and transformations that are performed in our DSL compilers.

Many GPL languages also have strong support for more productive coding through IDE assistance, i.e. Visual Studio for C#, and Eclipse or IntelliJ for Java. They all have in common that the IDE can generate code for you, ranging from simple templates, to code generation based on annotations or other meta-data. A popular notion for generating code is through *guidance* (also known as 'wizards'), in which high-level questions are answered by the programmer, and code is generated accordingly. Combining these IDE mechanisms, we could approximate some of the functionality of our DSL compilers. However, an important observation is that all the IDE assisted code generation lacks a source language, and is only concerned with the output of the generation process. Once a template is instantiated, or a wizard has been completed, there is only the generated code that acts as source. Hence, refactoring support is also in high demand. The choices and variability expressed by template selection or answers to questions are not recorded, whereas in our DSLs, the user always works at a higher level. The generated code is merely a (very useful) derivation of the DSL code.

Furthermore, it is debatable whether an IDE is the right place to integrate more and more domain specific knowledge, for example, by supporting specific libraries and frameworks in a generative manner. Should library designers create such support, and if so, for which of the many IDEs? By externalizing such support into a separate DSL compiler, these issues are avoided. Integration of DSL compilers such as ours is possible when the IDE allows external tasks to be called for files in the editor (which means virtually all IDEs).

7.4 DSLs for the web

Many domain specific languages for web-development exist [?, ?, ?]. In this section, we will review four interesting variants. The first two are a stand-alone DSLs, whereas the other two are embedded DSLs in Haskell and Java, respectively. These efforts do not correlate directly to our approach of modeling distinct technical concerns in different DSLs. However, each of the languages constitutes an effort to raise the abstraction level of web-application development. As such, we believe it is interesting to view them as alternative approaches when looking at just this domain.

7.4.1 Links

Initiated by Wadler, Links [?] is one of the most recent research endeavors concerning web-application development. Wadler et al. recognize that this development generally involves multiple tiers:

- Presentation in the web browser.
- Application logic running on the web server.
- Database back-end providing data.

Each of these tiers can be programmed in their own languages, such as HTML and JavaScript (in the web browser), PHP, Java or equivalent (on the web server), and SQL or XQuery (to query the database). Creating applications in this mixture of general purpose languages is cumbersome and error-prone. Their proposed solution is to provide a *tierless* approach, by creating a single language that links (hence the name of the language) all of the concepts that span the traditional tiers.

Links is a strongly typed, strict functional language, providing a compiler that produces HTML and JavaScript for clients, proprietary Links code to run on their own server, and SQL to query a database. The language's main properties are:

- Session state is preserved on the client side.
- Transfer of computation between client and server is supported in continuation-passing style.
- Typed message passing between concurrent processes is the primary means of communication on the server and on the client, and between these two.
- Database query optimization is provided by the compiler.

Especially the second property is one of Links' strong points. Continuation passing (suspending and resuming functions in different environments) provides a natural fit to supporting AJAX[1] like interaction. Also, it allows the 'back-button' problem to be solved: the state can be reconstructed and resumed wherever it was left off. The last property shows the strength of a domain specific approach: database interaction takes place using list comprehensions, filters, and other functions familiar to functional programmers. The Links compiler has the ability to compile highly optimized queries out of these constructs.

In effect, Links constitutes a new general purpose language. It has functions and other general abstraction mechanisms, all supported by type inference. On the other hand, it is compiled to code in a very particular setting (web-server and client), and contains many useful domain specific abstractions as first-class citizen. For example, `server` and `client` are keywords in the language, list-comprehensions can directly read from a database, there are special database and table expressions, and XML literals are in the language. Furthermore, Erlang-style concurrency (shared-nothing processes that communicate by message passing) is implemented for event-handling. Wadler et al. have chosen to create a new language, containing advanced GPL constructs (for example, continuations) but only if they support the domain concepts in a novel way.

This results in a language that is very expressive, albeit still a bit low-level. They concede this last point in their paper, stating that, for example, form construction should ideally be abstracted away from. Currently, it is not possible to reuse (parts of) forms, much less to generate them by induction over a type (as is done in *WebLayer*). No interaction with other languages is currently possible, resulting in the situation that Links must provide its own implementations for everything. Clearly, this is a downside of the GPL with extensions approach taken. A foreign function interface is mentioned as future work, though this is certainly not trivial given the mostly functional nature of Links. In accordance with the research nature of this project, this issue does not seem to have a high priority.

[1] Asynchronous JavaScript And XML: a method to overcome limitations of the per-page request/response cycle of HTTP.

7.4.2 bigwig/JWIG

JWIG [?] is a continuation of the somewhat obsolete <bigwig> [?] project. It consists of a port of the <bigwig> project to Java, meanwhile tailoring it in such a way that sophisticated static program analysis is possible for JWIG applications. The main features of <bigwig>, and therefore also JWIG are:

- Dynamic construction of HTML, while interspersing generated content.
- Analyses that guarantee correctness of dynamic pages.
- A type system to statically check client and server interaction.

A large objection to <bigwig> is that this stand-alone DSL was not powerful enough. It also did not provide enough libraries, or facilities to integrate existing libraries of other languages, mainly because its analyses depend on a closed-world assumption. Therefore, the authors decided to generalize the <bigwig> core language to Java, thereby allowing the use of arbitrary Java libraries. In the process, other parts were generalized as well. For example, safe construction of XML documents (instead of only HTML) is supported. Syntactic constructs are added to Java that provide concrete syntax for constructing XML. The type system and program analyses are also extended to the more expressive Java language. Program analyses are the key of JWIG, and they are defined as instances of a monotone framework. The main contribution is that for every possible output generated to the client, the validity of this output can be established with respect to a document type definition (DTD). Because arbitrary library functions can be invoked, outside of the JWIG framework, this analysis is not as precise as it was in <bigwig>.

7.4.3 WASH/WebFunctions

The Web Authoring System for Haskell (WASH [?]) project provides an embedded DSL to create dynamic webpages from within Haskell. It provides four sub-languages: the document language, widget language, session language, and persistence language. WASH communicates with a web-server through CGI. Well-formedness of HTML is guaranteed by the document language, which provides functions to build a Haskell datatype that represents an HTML page. A pre-processor is available to translate HTML in concrete syntax to the Haskell equivalent. Widgets (input fields) are automatically linked to the site where their input is used, WASH abstracts away from form and input field name resolution. State management is offered by WASH as well, using the session sub-language.

WASH offers a lot of power to experienced Haskell programmers. However, creating a typical web-application using WASH results in code that mixes a lot of concerns. It is not clear whether this is due to the fact that each of the sub-languages is embedded, or the design of WASH as a whole.

WebFunctions is a master's project by Robert van Herk [?] that tries to alleviate the issues of WASH, by building higher level abstractions upon it. WebFunctions also is an embedded DSL for Haskell, and is based on a Java web-language called WebObjects (which is discussed in the subsequent section). It abstracts over session and application state, as well as database interaction. The Haskell type system guarantees correct HTML and SQL to be emitted from WebFunctions. It constitutes an integration effort building upon HaskellDB [?] and WASH. Interaction between the several aspects (sub-languages in WASH) of web-development is implicit, because everything is embedded in Haskell.

7.4.4 WebObjects

WebObjects is a commercial (although some versions are available for free) web-application framework from Apple Computer. It is a Java framework, in many aspects similar to the frameworks we target with our DSLs. However, WebObjects offers a complete (proprietary) solution for all concerns of web-applications, from the data layer to the presentation layer, and even an application server.

In principle, it is an embedded language in Java, aspiring to address the issues of all layers of web-applications. Unfortunately, Java is not particularly suited to embeddings, resulting in a very verbose language (Apple's naming conventions only add insult to injury in this case). It also seems that WebObjects, as our target frameworks, makes heavy use of reflection and run-time interpretation

of bindings. In fact, this was one of the main motivations for the previously mentioned WebFunctions project. However, a supporting IDE environment, containing many generative aspects, and even some visual modeling tools, is available. With this, WebObjects has become a rapid application development environment, as long as the tools support the abstractions the developer needs. An interesting question then is, what constitutes the *WebObjects language*? Is it the libraries and the programmatic interfaces contained therein, or the capabilities offered by the tools, or a combination? The latter is probably the best answer, since developing a complete WebObjects application also involves supplementing and adapting generated code with custom code.

In general, an increasingly larger role is awarded to integrated development environments (or plug-ins for these) in the light of specific libraries. Ultimately, we believe this is not a satisfactory answer to the challenge of exposing domain specific capabilities to programmers. As a library or framework designer, having to support many IDEs through plug-ins (if this is even possible to a satisfactory degree), or creating a dedicated IDE is simply too much work, as was discussed towards the end of Section 7.2.

7.5 Active Libraries

We have already established (Section 2.1) that libraries currently are the primary means of expressing and reusing domain specific constructs. Veldhuizen [**?**] proposes to alleviate the associated problems of libraries by giving them an *active* role in the compilation process. The key idea is, that libraries should not only provide the desired functionality, but also the syntax, optimizations and safety checks (with corresponding, friendly error messages) for a problem domain. A pre-requisite for such an active library approach is that there exists a compiler supporting these types of additions. Furthermore, a unified way of expressing the aforementioned library responsibilities must be crafted. Veldhuizen envisions a staged compiler (and corresponding language), in which compile-time computations and transformations can be specified by libraries. If we apply this line of thinking to our work, it would entail mixing the meta-code code describing our transformations, and possibly the syntax definition of our DSLs, directly with the implementing libraries. Clearly, this is an enticing possibility for library creators. However, it is also evident that rigorously applying the ideas of active libraries demands a redefinition of traditional interactions between compilers, libraries and applications. As with Intentional Programming (Section 2.4.1), it is unlikely that this happens outside a laboratory in the near feature.

Chapter 8

Conclusion

8.1 Reflection

In this thesis project we have developed a prototype environment in which multiple DSLs interact to create a single, coherent application. The *DomainModel* language and compiler (Chapter 3) allows for concise definitions of persistent data models, whereas *WebLayer* (Chapter 4) is geared towards creating views for web-applications. Interaction between these two languages has been researched and implemented (Chapter 5) . Together with the *BusinessRules* interface definition (Chapter 6), we have shown that by using multiple interacting DSLs we can model applications with a layered architecture of technical domains, thereby providing the advantages of a high-level, domain specific specific approach (described in Chapter 1 and Chapter 2). We succesfully extracted high level abstractions from the target frameworks, even though this has proven harder than originally anticipated in the case of the *WebLayer* language. The DSLs provide much more clarity with respect to the domain concepts than the corresponding Java code. Also, we have shown that a vast improvement in productivity and quality is possible through our approach. As expected, we have also experienced the disadvantages of developing stand-alone DSLs. In particular, having to write every semantic check from scratch is not a very appealing aspect of our approach, even though it offers (domain specific) possibilities as well.

During the development of the two central domain specific languages, *DomainModel* and *WebLayer*, we have gained considerable insight in developing DSLs for current Java frameworks. In this conclusion, we will reflect on these insights before formulating answers to our research questions.

The most important observation is that even though Java's type system becomes more expressive with each new version, many frameworks layer dynamically checked, or interpreted mechanisms on top of (type safe) Java primitives. Particularly, the abundant (ab)use of Java 5 annotations for such mechanisms is notable. Possibly, this trend reflects the longing for more flexibility, akin to scripting languages, or a compulsion to avoid more complex language constructions at the expense of risking more runtime failures. While developing the DSLs, one of our goals was to avoid these runtime failures, by making sure the design of our DSL contains enough leverage to guarantee that the aforementioned dynamic checks will succeed as much as possible.

Furthermore, programmers are required to repeat the same (or subtly different) information in several places, when using libraries or instantiating a framework. This holds even more when combining frameworks, or in other words, when crossing the boundaries of layers in the architecture. Thus, another important observation is that such *usage patterns* of frameworks form a good indicator of variability that belongs in a DSL design. Gathering, combining, and analyzing occurences of this form of redundancy almost automatically leads to a design for the (typeful) representation of a higher level domain abstraction in the DSL. Our implementation of generic constructs in *WebLayer* takes this sharing of information even further, offering more powerful abilities than the underlying frameworks have, by sharing information between DSLs.

We now proceed with reflecting upon the research questions posed in Chapter 1. During the discussion of interaction between the DSLs, we have argued that multiple interacting DSLs have many advantages over traditional, monolithic DSL approaches, and are an excellent match to layered architectures in general (Section 5.1.1) This also forms a partial answer to our first research question, '*What interaction patterns can we identify?*'. Layered architectures are such a pattern. Consequently,

a guiding principle that we believe to hold emerges from this pattern. This principle states that dependencies between DSLs in our environment must form an acyclic graph (Section 5.1.6), and that if a cycle emerges when designing interacting DSLs, abstractions probably are distributed over the DSLs in a sub-optimal manner.

Another interaction pattern that is worth mentioning is that interaction should be as strongly typed (and as specific) as possible. The interaction between *DomainModel* concept definitions and *WebLayer* (generic) constructs forms a good example of such interaction. Though being specific in the interface precludes us from defining a general interaction mechanism, the benefits in terms of usability are clear. It is possible to define a slightly more generic interface, as we have shown with *BusinessRules* (Chapter 6). However, DSLs conforming to this *BusinessRules* interface (though currently none is implemented) cannot communicate much more information than that certain rules are available, and how they can be invoked.

Our second research question is more practical of nature: '*How can we implement DSL interaction?*'. Through the development of our prototype we introduced interface files for the DSLs. One of the merits of this approach is that it precludes a (too) tight coupling between the DSLs, while still allowing for effective, compile time linking of generated code from different languages. In particular, we have been cautious not to create a whole-program compiler in disguise. Having this form of separate compilation enables alternative compositions of languages, although we only have one concrete implementation, and example (*DomainModel*) of a language that is ready to be integrated into other DSLs. The *BusinessRules* design builds upon this concept, albeit without a concrete implementation. Furthermore, we have shown that integrating abstractions from different DSLs can be implemented in an intuitive fashion. No knowledge whatsoever of the underlying implementation is required from the DSL users. We have found that the interfaces that are necessary for the implementation of DSL interaction are quite specific to the language. A general solution to the interaction problem could not be formulated.

The last research question posed with regard to interaction between DSLs is: '*How does interaction affect the design of a DSL?*'. The most obvious answer to this question is that the DSL designer needs to consider whether integrating another DSL is necessary or beneficial. If so, a natural way of integrating abstractions from another DSL must found. We have provided two examples of this scenario: injecting *DomainModel* definitions as an underlying type system into *WebLayer* and allowing *BusinessRules* definitions to be called from *WebLayer* actions.

Furthermore, an (explicit) interface must be designed alongside the language definition itself, when reuse of the language in different settings is desired. Less obvious is the fact that a DSL also has an associated implicit interface. This implicit interface, a resultant of the underlying implementation libraries or frameworks, introduces additional dependencies on DSL interaction. We have tried two strategies to contain this problem. First strategy is choosing a standardized library as target for the DSL (*DomainModel* uses JPA). The second is by creating the implicit interface as well as the explicit interface during the design of a DSL (*BusinessRules*), rather than letting the libraries you work with dictate the implicit interface. Neither of these strategies is perfect, though at least they show the impact of interaction on DSL design, and the associated trade-offs.

The final research question given in the introduction of this thesis is: '*When and how should interaction between DSL and GPL code be implemented?*'. In second part of Chapter 5 we have looked at the issue of integrating GPL code with DSL abstractions. This form of interaction is necessary to still be able to use a DSL even when in itself the DSL is not sufficiently expressive. Rather than writing GPL code that links directly with generated code by a DSL (an approach taken by most of the related systems we surveyed), we propose a slightly different solution. In this solution, a DSL compiler also controls the GPL code. In our example of extended types, this allows for GPL code to be reused across many DSL projects, rather than being linked to a single project. Furthermore, the connection between GPL code and DSL abstractions must be clear from the perspective of the DSL source as well, which is an issue that is often overlooked.

Summarizing, we believe this thesis project has shown that creating an environment of multiple interacting DSLs for technical domains is feasible. Our case study has confirmed that the development of applications with layered architectures can significantly benefit from such an environment. Some DSLs may be more amenable to reuse in such a setting than others, but we are confident that many

program families exist that can be readily decomposed into technical DSLs. Though requiring significant effort and expertise, we believe that many software companies can benefit from the application of model driven software development through interacting DSLs.

8.2 Future work

Bibliography

[1] David L. Atkins, Thomas Ball, Glenn Bruns, and Kenneth Cox. Mawl: A domain-specific language for form-based services. *IEEE Transactions on Software Engineering*, 25(3):334–346, 1999.

[2] A.F. Blackwell. Cognitive dimensions of tangible programming techniques. In *Proc. 1st Joint Conf. Empirical Assessment of Software Eng. and the Psychology of Programming Interest Group (EASE & PPIG 03)*, pages 391–405, 2003.

[3] Claus Brabrand, Anders Moller, and Michael I. Schwartzbach. The <bigwig> project. *ACM Trans. Inter. Tech.*, 2(2):79–114, 2002.

[4] Martin Bravenboer, René de Groot, and Eelco Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using stratego/xt. In *GTTSE*, pages 297–311, 2006.

[5] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[6] Luca Cardelli. Program fragments, linking, and modularization. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 266–277, New York, NY, USA, 1997. ACM Press.

[7] Stefano Ceri, Piero Fraternali, and Stefano Paraboschi. Design principles for data-intensive web sites. *SIGMOD Rec.*, 28(1):84–89, 1999.

[8] Aske Simon Christensen, Anders Moller, and Michael I. Schwartzbach. Extending Java for high-level web service construction. *ACM Trans. Program. Lang. Syst.*, 25(6):814–875, 2003.

[9] Thomas Cleenewerck. Component-based DSL development. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 245–264, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[10] Charles Consel. From a program family to a domain-specific language. In *Domain-Specific Program Generation*, pages 19–29, 2004.

[11] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links : Web programming without tiers. Technical report, 2005.

[12] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[13] René de Groot. Design and implementation of embedded domain-specific languages. Master's thesis, Universiteit Utrecht, Utrecht, The Netherlands, September 2005. INF/SCR-05-10.

[14] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972. Turing Award lecture.

[15] Sergey Dmitriev. Language Oriented Programming: The next programming paradigm. `http://www.onboard.jetbrains.com/articles/04/10/lop/`, 2004.

[16] Sophia Drossopoulou and Susan Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In *Formal Syntax and Semantics of Java*, pages 41–82. Springer-Verlag, 1998.

[17] Martin Fowler. Language workbenches: the killer-app for domain specific languages? `http://www.martinfowler.com/articles/languageWorkbench.html`, 2005.

[18] Piero Fraternali. Tools and approaches for developing data-intensive web applications: a survey. *ACM Comput. Surv.*, 31(3):227–263, 1999.

[19] Paul Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the Web with high-level programming languages. *Lecture Notes in Computer Science*, 2028:122–??, 2001.

[20] Rick Hightower. The JSF lifecycle. `http://www.ibm.com/developerworks/java/library/j-jsf2/`, 2005.

[21] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28:196, 1996.

[22] Paul Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[23] I. Kurtev and L. Didonet Del Fabro. A dsl for definition of model composition operators. In *Second Workshop on Models and Aspects (ECOOP 2006)*, 2006.

[24] Avraham Leff and James T. Rayfield. WebRB: evaluating a visual domain-specific language for building relational web-applications. *SIGPLAN Not.*, 42(10):281–300, 2007.

[25] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 109–122, New York, NY, USA, 1999. ACM Press.

[26] F. Merillon, L. Reveillere, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming, 2000.

[27] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages.

[28] Oberon Microsystems. Component Pascal language report. Technical report, 2006.

[29] Martin Nussbaumer, Patrick Freudenstein, and Martin Gaedke. Towards dsl-based web engineering. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 893–894, New York, NY, USA, 2006. ACM Press.

[30] Rinus Plasmeijer and Peter Achten. The implementation of iData. In *IFL*, pages 106–123, 2005.

[31] Joost Rommes. Syntax macros: attribute redefinitions. Master's thesis, Universiteit Utrecht, Utrecht, The Netherlands, July 2003. INF/SCR-03-31.

[32] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 975–985, New York, NY, USA, 2006. ACM Press.

[33] Charles Simonyi. The death of computer languages, the birth of intentional programming, 1995.

[34] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. In *OOPSLA*, pages 451–464, 2006.

[35] Jeroen Snijders. Functional design patterns. Master's thesis, Universiteit Utrecht, Utrecht, The Netherlands, 2004.

[36] Daniel E. Stevenson and Andrew T. Phillips. Implementing object equivalence in Java using the template method design pattern. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 278–282, New York, NY, USA, 2003. ACM Press.

[37] K. Stirewalt and S. Rugaber. Automating ui generation by model composition. In *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*, page 177, Washington, DC, USA, 1998. IEEE Computer Society.

[38] R. E. K. Stirewalt and S. Rugaber. The model-composition problem in user-interface generation. *Automated Software Engineering*, 7:101–124, 2000.

[39] Clemens Szyperski. *Component Software.* Addison-Wesley Professional, November 2002.

[40] Alan L. Tharp. The impact of fourth generation programming languages. *SIGCSE Bull.*, 16(2):37–44, 1984.

[41] The GHC Team. GHC user's guide - Section 4.6.2. `http://www.haskell.org/ghc/docs/latest/html/users_guide/separate-compilation.html`.

[42] Peter Thiemann. An embedded domain-specific language for type-safe server-side web scripting. *ACM Trans. Inter. Tech.*, 5(1):1–46, 2005.

[43] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[44] Arie van Deursen and Paul Klint. Little languages: little maintenance. *Journal of Software Maintenance*, 10(2):75–92, 1998.

[45] Jeffrey van Helden and Niels Reyngoud. Functional design patterns. Master's thesis, Universiteit Utrecht, Utrecht, The Netherlands, 2005.

[46] Robert van Herk. WebFunctions. Master's thesis, Universiteit Utrecht, Utrecht, The Netherlands, 2005.

[47] Todd L. Veldhuizen. *Active Libraries and Universal Languages.* PhD thesis, Indiana University Computer Science, May 2004.

[48] Eelco Visser. *Syntax Definition for Language Prototyping (PhD thesis).* PhD thesis, Amsterdam, 1997.

[49] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[50] Jos Warmer and Anneke Kleppe. Building a flexible software factory using partial domain specific models. In *OOPSLA Int. Workshop on Domain-specific modeling*, 2006.

Appendix A

Implementation details

A.1 Syntax definitions

DomainModel

```
module DomainModel
imports languages/java-15/interfaces/Annotations

hiddens
  context-free start-symbols
    Domain

exports
  sorts FreeText QuotedText DMId UpperDMId AllUpper QualifiedDMId

  syntax
    “\”” <FreeText-LEX> “\”” → <QuotedText-CF> {<cons(“QuotedText”)>}

  lexical syntax
    ~[\”]*                  → FreeText
    [a-z] [a-zA-Z0-9\_]*    → DMId
    [A-Z] [a-zA-Z0-9\_]*    → UpperDMId
    [A-Z] [A-Z0-9\_]*       → AllUpper
    [\  \t \n ]             → LAYOUT
    “//” ~[\n]* [\n]        → LAYOUT

  lexical restrictions
    DMId-/- [a-zA-Z0-9\_]
    UpperDMId-/- [a-zA-Z0-9\_]

exports
  sorts Domain ConceptDecl EnumMember Concept ConceptMember DMAnno EnumType
        ConceptOrNativeType ListType DMType Assoc

  context-free syntax
    “domainmodel” DMId ConceptDecl*                → Domain               {<cons(“Domain”)>}
    “concept” UpperDMId “{” ConceptMember+ “}”     → ConceptDecl          {<cons(“ConceptDecl”)>}
    DMId Assoc DMType “(” {DMAnno “,”}* “)”        → ConceptMember        {<cons(“ConceptMember”)>}
    DMId Assoc DMType                              → ConceptMember        {<cons(“ConceptMember”)>}
    “::”                                           → Assoc                {<cons(“Native”)>}
    “<>”                                           → Assoc                {<cons(“Composite”)>}
    “->”                                           → Assoc                {<cons(“Reference”)>}
    “{” {EnumMember “,”}* “}”                      → EnumType             {<cons(“Enum”)>}
    QuotedText “:” AllUpper                        → EnumMember           {<cons(“EnumMember”)>}
    UpperDMId                                      → ConceptOrNativeType  {<cons(“BuiltIn”)>}
    “[” DMType “]”                                 → ListType             {<cons(“Collection”)>}
    EnumType                                       → DMType
    ListType                                       → DMType
```

```
ConceptOrNativeType                       → DMType
Anno                                      → DMAnno        {<cons("JavaAnnotation")>}
DMId                                      → DMAnno        {<cons("SimpleAnnotation")>}
```

WebLayer DSL

```
module WebLayer
hiddens
  context-free start-symbols
    WebLayer

exports
  sorts Id SimpleId UId QualifiedId FreeText QuotedText

  syntax
    "\"" <FreeText-LEX> "\"" → <QuotedText-CF> {<cons("QuotedText")>}

  lexical syntax
    [a-z] [a-zA-Z0-9\_]*   → Id
    [A-Z] [a-zA-Z0-9\_]*   → UId
    ~[\"]*                 → FreeText
    [\  \t \n ]            → LAYOUT
    "//" ~[\n]* [\n]       → LAYOUT

  lexical restrictions
    Id-/- [a-zA-Z0-9\_]
    UId-/- [a-zA-Z0-9\_]

  context-free restrictions
    LAYOUT?-/- [\  \t \12 \n \r ]

  context-free syntax
    Id                           → SimpleId    {<cons("SimpleId")>}
    SimpleId                     → QualifiedId
    SimpleId "." QualifiedId     → QualifiedId {<cons("QualifiedId")>}

exports
  sorts WebLayer Import BodyDecl SessionVarDecl PageDecl InitPage Param PageElem KeyVal
        Action ActionBinding NavBinding TextExpr TextExprPart

  context-free syntax
    "weblayer" Id Import* BodyDecl*                            → WebLayer        {<cons("WebLayer")>}
    "using" Id Id                                              → Import          {<cons("Import")>}
    PageDecl                                                   → BodyDecl
    SessionVarDecl                                             → BodyDecl
    "var" UId SimpleId                                         → SessionVarDecl {<cons("SessionVarDecl")>}
    InitPage? "page" UId "(" {Param ","}* ")" "{" PageElem+ "}" → PageDecl        {<cons("PageDecl")>}
    "initial"                                                  → InitPage        {<cons("InitialKeyword")>}
    UId SimpleId                                               → Param           {<cons("ParamBinding")>}
    "table" Param "in" QualifiedId "{" KeyVal* "}"             → PageElem        {<cons("Iterator")>}
    "table" Param "{" KeyVal* "}"                              → PageElem        {<cons("IteratorAll")>}
    "for" Param "in" QualifiedId "{" PageElem* "}"             → PageElem        {<cons("Repeat")>}
    "for" Param "{" PageElem* "}"                              → PageElem        {<cons("RepeatAll")>}
    "header(" TextExpr ")"                                     → PageElem        {<cons("Header")>}
    "text" "(" TextExpr ")"                                    → PageElem        {<cons("TextExpr")>}
    TextExpr                                                   → PageElem        {<cons("TextExpr")>}
    "navigate" "(" TextExpr "," NavBinding ")"                 → PageElem        {<cons("NavExpr")>}
    "action" "(" TextExpr "," {Action ";"}* ")"                → PageElem        {<cons("ActionBlock")>}
    "show" "(" QualifiedId ")"                                 → PageElem        {<cons("Show")>}
    "show" "(" KeyVal+ ")"                                     → PageElem        {<cons("Show")>}
    "input" "(" QualifiedId ")"                                → PageElem        {<cons("Input")>}
    "input" "(" KeyVal+ ")"                                    → PageElem        {<cons("Input")>}
```

“edit” “(” QualifiedId “)”	→ PageElem	{<cons(“Edit”)>}
“var” UId SimpleId	→ PageElem	{<cons(“VarDecl”)>}
“form” “(” PageElem* “)”	→ PageElem	{<cons(“Form”)>}
QuotedText “->” PageElem	→ KeyVal	{<cons(“KeyVal”)>}
UId “(” {QualifiedId “,”}* “)”	→ NavBinding	{<cons(“Navigation”)>}
“redirect(” NavBinding “)”	→ Action	{<cons(“Redirect”)>}
QualifiedId “=” QualifiedId	→ Action	{<cons(“Assign”)>}
QualifiedId “(” {QualifiedId “,”}* “)”	→ Action	{<cons(“Call”)>}
{TextExprPart “+”}+	→ TextExpr	{<cons(“CompositeTextExpr”)>
QuotedText	→ TextExprPart	{<cons(“LitTextExpr”)>}
QualifiedId	→ TextExprPart	{<cons(“RefTextExpr”)>}

A.2 Generated code

A.2.1 DomainModel

To illustrate the implementation of the DomainModel DSL, we present a verbatim translation of the `BlogEntry` concept. This concept was introduced in Figure 3.3, the running example of Chapter 3. Note that parts of the generated code are omitted because of space constraints. Comments show which parts have been left out.

```
package blog.domainclasses;

import java.util.*;
import javax.persistence.*;

@Entity public class BlogEntry
{
  public BlogEntry ()
  { }

  public BlogEntry (String _title, String _abstract, String _contents
                   , Date _date, List<Tag> _tags, List<Reply> _replies)
  {
    this._title = _title;
    this._abstract = _abstract;
    this._contents = _contents;
    this._date = _date;
    this._tags = _tags;
    this._replies = _replies;
  }

  private Long id;

  @Id @GeneratedValue public Long getId()
  {
    return id;
  }

  public void setId(Long id)
  {
    this.id = id;
  }

  int versionNum;

  @Version @Column(name = "OPTLOCK") protected int getVersionNum()
  {
    return versionNum;
  }

  protected void setVersionNum(int versionNum) { .. }
```

Listing A.1: Translation of `BlogEntry` to Java

```
private String _title;

@Basic @Column(nullable = false) public String getTitle()
{
  return _title;
}

public void setTitle(String _title) { .. }

private String _abstract;

@Basic public String getAbstract()
{
  return _abstract;
}

public void setAbstract(String _abstract) { .. }

// Code for "contents" and "date" members omitted.

private List<Tag> _tags;

@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
public List<Tag> getTags()
{
  return _tags;
}

public void setTags(List<Tag> _tags) { this._tags = _tags; }

public void addToTags(Tag b_0)
{
  if(this._tags == null)
  {
    this._tags = new ArrayList<Tag>();
  }
  this._tags.add(b_0);
}

public void removeFromTags(Tag c_0)
{
  if(this._tags != null)
  {
    this._tags.remove(c_0);
  }
}

public enum Enum_category
{
  TECH("Technical"), NONTECH("Other");

  private String label;

  Enum_category (String label) { this.label = label; }

  public String getLabel() { return label; }

  @Override public String toString() { return label; }

  public Enum_category[] getValues() { return Enum_category.values(); }
}

private Enum_category _category;

@Basic @Enumerated(EnumType.ORDINAL) public Enum_category getCategory()
{
  return _category;
}

public void setCategory(Enum_category _category) { this._category = _category; }
```

Listing A.2: Translation of `BlogEntry` (continued)

```
  // Code for "replies-> [Reply]" member omitted.

  public @Override boolean equals(Object o)
  {
    if(o != null && o instanceof BlogEntry)
    {
      Long own_id = this.id;
      Long other_id = ((BlogEntry)o).getId();
      if(own_id != null)
        return own_id.equals(other_id);
      else
        return this == o;
    }
    else
      return false;
  }

  @Override public int hashCode() {
    return getId() != null ? getId().intValue() : super.hashCode();
  }

  @Transient public String getToString()
  {
    return toString();
  }

  @Override public String toString()
  {
    StringBuilder name = new StringBuilder();
    if(_title != null)
    {
      name.append(_title.toString());
    }
    name.append(" ");
    if(_date != null)
    {
      name.append(_date.toString());
    }
    return name.toString();
  }

}
```

Listing A.3: Translation of `BlogEntry` (continued)

A.2.2 WebLayer

```
package org.blog.weblayer;
// Import statements omitted for the sake of brevity

@Name("ViewBlogComponent") @Scope(ScopeType.CONVERSATION)
public class ViewBlogComponent implements Serializable
{
  @In private EntityManager em;

  @In private FacesMessages facesMessages;

  private org.blog.domainmodel.BlogEntry blogEntry_be;

  @RequestParameter("blogEntry_beId") private Long blogEntry_beId;

  private Long cached_blogEntry_beId;

  public Long getBlogEntry_beId()
  {
    if(blogEntry_beId == null)
      return cached_blogEntry_beId;
    else
      return blogEntry_beId;
  }

  public org.blog.domainmodel.BlogEntry getBlogEntry_be()
  {
    return blogEntry_be;
  }

  // Code for the second parameter (User u) omitted

  public @Begin(join = true) void initialize()
  {
    try
    {
      System.out.println("Initialize is called, initialized = " + initialized);
      if(blogEntry_beId != null)
      {
        if(blogEntry_beId != cached_blogEntry_beId)
          this.blogEntry_be = em.find( org.blog.domainmodel.BlogEntry.class
                                      , getBlogEntry_beId() );
        cached_blogEntry_beId = blogEntry_beId;
      }
      if(user_uId != null)
      {
        if(user_uId != cached_user_uId)
          this.user_u = em.find(org.blog.domainmodel.User.class, getUser_uId());
        cached_user_uId = user_uId;
      }
      if(!initialized)
        this.reply_r = new org.blog.domainmodel.Reply();
      iterator3 = blogEntry_be.getTags();
      repeat1 = blogEntry_be.getReplies();
      initialized = true;
    }
    catch(NullPointerException npe)
    {
      facesMessages.add("One or more entities could not be retrieved");
      npe.printStackTrace(System.err);
      initialized = false;
    }
    catch(Exception exc)
    {
      facesMessages.add("Unhandled exception occurred: " + exc.toString());
      exc.printStackTrace(System.err);
```

Listing A.4: Translation of `ViewBlog` page to Seam component

```
      initialized = false;
    }
  }

  public boolean initialized = false;

  public boolean getInitialized()
  {
    return initialized;
  }

  private org.blog.domainmodel.Reply reply_r;

  public org.blog.domainmodel.Reply getReply_r()
  {
    return this.reply_r;
  }

  List<org.blog.domainmodel.Tag> iterator3;

  @DataModel(scope = ScopeType.PAGE)
  public List<org.blog.domainmodel.Tag> getIterator3()
  {
    return iterator3;
  }

  List<org.blog.domainmodel.Reply> repeat1;

  public List<org.blog.domainmodel.Reply> getRepeat1()
  {
    return repeat1;
  }

  public String action3()
  {
    System.out.println("action " + "action3" + "  called");
    try
    {
      getReply_r().setUser(getUser_u());
      getBlogEntry_be().addToReplies(getReply_r());
      em.persist(getBlogEntry_be());
      em.flush();
      em.refresh(getBlogEntry_be());
    }
    catch(javax.persistence.OptimisticLockException ole)
    {
      facesMessages.add("There has been a concurrent update to the
                        edited data, please review new data");
    }
    catch(IllegalArgumentException iae)
    {
      facesMessages.add("Validation failed because a field was empty: "
                         + iae.getMessage());
      iae.printStackTrace(System.err);
      initialized = false;
    }
    catch(Exception exc)
    {
      facesMessages.add("An error occured: " + exc.toString());
      exc.printStackTrace(System.err);
    }
    initialized = false;
    return "/ViewBlog.xhtml?" + ("blogEntry_beId" + "="
                              + cached_blogEntry_beId + "&"
                              + ("user_uId" + "=" + cached_user_uId + "&"));
  }
}
```

Listing A.5: Translation of `ViewBlog` page to Seam component (continued)

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                             "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" >
<ui:composition xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://java.sun.com/jsf/facelets"
                xmlns:h="http://java.sun.com/jsf/html" xmlns:f="http://java.sun.com/jsf/core"
xmlns:s="http://jboss.com/products/seam/taglib"
xmlns:a="https://ajax4jsf.dev.java.net/ajax" template="template.xhtml">
<!-- global error and validation messages for page  -->
 <ui:define name="globalMessages">
  <h:messages globalOnly="true" styleClass="errors"/>
 </ui:define>
<!-- content -->
 <ui:define name="content">
  <div class="section">
   <s:div rendered="#{!ViewBlogComponent.initialized}">
    <p> The page could not be rendered due to missing or
        invalid information in the request parameters. </p>
   </s:div>
   <s:div rendered="#{ViewBlogComponent.initialized}">
    <br/>
    <h1>
     <h:outputText
      value="#{ViewBlogComponent.blogEntry_be.title} (written on #{ViewBlogComponent.blogEntry_be.date})"
      escaped="true"/>
    </h1>
    <br/>
    <h:outputText value="#{ViewBlogComponent.blogEntry_be.contents}" escaped="true"/>
    <br/>
    <h:dataTable value="#{ViewBlogComponent.iterator3}" var="t">
     <h:column>
      <f:facet name="header">Assigned tags</f:facet>
      <h:outputText value="#{t.tagName}" escaped="true"/>
     </h:column>
    </h:dataTable>
    <br/>
    <h:outputText value="Reply to this post:" escaped="true"/>
    <br/>
    <h:form>
     <s:validateAll>
      <f:facet name="afterInvalidField">
       <s:div styleClass="errors">
        <s:span>
                                 Error: <s:message/>
        </s:span>
       </s:div>
      </f:facet>
      <f:facet name="aroundInvalidField">
       <s:span styleClass="errors"/>
      </f:facet>
      <div class="input">
       <div class="entry">
        <div class="label">
         <h:outputText value="Contents :" escaped="true"/>
        </div>
        <div class="input">
         <s:decorate>
          <h:inputText id="contents" required="false"
               value="#{ViewBlogComponent.reply_r.contents}">
          </h:inputText>
         </s:decorate>
        </div>
       </div>
       <div class="entry">
        <div class="label">
         <h:outputText value="Date :" escaped="true"/>
        </div>
        <div class="input">
         <s:decorate>
          <h:inputText id="date" value="#{ViewBlogComponent.reply_r.date}" required="false">
           <s:convertDateTime pattern="dd/MM/yyyy"/>
          </h:inputText>
```


Figure A.1: Translation of `ViewBlog` page to Facelets markup

```
          <s:selectDate for="date" dateFormat="dd/MM/yyyy">
           <h:graphicImage url="img/dtpick.gif" style="margin-left:5px;cursor:pointer"/>
          </s:selectDate>
         </s:decorate>
        </div>
       </div>
       <div class="entry">
        <div class="label">
         <h:outputText value="User :" escaped="true"/>
        </div>
        <div class="input">
         <s:decorate>
          <h:selectOneMenu id="user" value="#{ViewBlogComponent.reply_r.user}" required="true">
           <s:selectItems var="field" label="#{field.toString}"
                  value="#{AllUser}" noSelectionLabel="Please select : "/>
           <s:convertEntity/>
          </h:selectOneMenu>
         </s:decorate>
        </div>
       </div>
       <div class="entry">
        <div class="label">
         <h:outputText value="Level :" escaped="true"/>
        </div>
        <div class="input">
         <s:decorate>
          <h:selectOneMenu id="level" value="#{ViewBlogComponent.reply_r.level}">
           <s:selectItems value="#{StaticAccessor['org.blog.domainmodel.Reply.Enum_level']}"
                  var="enum" label="#{enum.label}"
                  noSelectionLabel="Please select : "/>
           <s:convertEnum/>
          </h:selectOneMenu>
         </s:decorate>
        </div>
       </div>
       <br/>
      </div>
     </s:validateAll>
     <h:commandButton value="Add reply" action="#{ViewBlogComponent.action1()}"/>
    </h:form>
    <br/>
    <h:outputText value="Replies for post #{ViewBlogComponent.blogEntry_be.title} :" escaped="true"/>
    <br/>
    <ui:repeat value="#{ViewBlogComponent.repeat1}" var="r">
     <div>
      <s:div rendered="#{r != null}">
       <div class="output">
        <div class="entry">
         <div class="label">
          <h:outputText value="Contents :" escaped="true"/>
         </div>
         <div class="output">
          <h:outputText value="#{r.contents}" escaped="true"/>
         </div>
        </div>
.. remaining fields omitted for the sake of brevity ..
      </s:div>
      <s:div rendered="#{r == null}">
       <i>empty</i>
      </s:div>
     </div>
    </ui:repeat>
    <br/>
    <s:link value="Home" view="/Blog.xhtml">
     <f:param name="user_uId" value="#{ViewBlogComponent.user_uId}"/>
    </s:link>
   </s:div>
  </div>
 </ui:define>
</ui:composition>
```

Figure A.2: Translation of `ViewBlog` page to Facelets markup (continued)

```
<!DOCTYPE pages PUBLIC "-//JBoss/Seam Pages Configuration DTD 1.2//EN"
                       "http://jboss.com/products/seam/pages-1.2.dtd" >

<page view-id="/ViewBlog.xhtml" action="#{ViewBlogComponent.initialize}"/>
```

Figure A.3: Generated configuration for `ViewBlog` page

```
<a:region>
  <a:outputPanel id="tagsList">
  <a:repeat var="listvar" value="#{NewBlogEntryComponent.blogEntry_be.tags}">
    <h:outputText value="#{listvar} "/>
    <a:commandButton ajaxSingle="true" value="Delete" reRender="tagsList"
                     action="#{Utils.ajaxRemove(NewBlogEntryComponent.blogEntry_be.tags, listvar)}"/>
    <br/>
  </a:repeat>
  </a:outputPanel>
  <br/>
  <s:decorate>
  <h:selectOneMenu id="tags" value="#{dev.nulll}"
                   valueChangeListener="#{NewBlogEntryComponent.tagsListListener}">
    <s:selectItems var="field" label="#{field.toString}" value="#{AllTag}"
                   noSelectionLabel="Please select : "/>
    <s:convertEntity/>
    <a:support event="onchange" reRender="tagsList" bypassUpdates="true"/>
  </h:selectOneMenu>
  </s:decorate>
</a:region>
```

Figure A.4: Generated viewcode for `edit` element with list argument

```java
package org.blog.weblayer.validation;

import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import javax.faces.component.UIComponent;
import org.jboss.seam.annotations.Name;

@Name("URLValidator")
@org.jboss.seam.annotations.jsf.Validator(id = "URLValidator")
public class URLValidator  implements Validator, java.io.Serializable
{
  public void validate(FacesContext context, UIComponent component, Object value)
      throws ValidatorException
  {
    String typed_value = (String)value;
    String message = org.blog.domainmodel.validation.URL.validate(typed_value);
    if(message != null)
    {
      FacesMessage fmessage = new FacesMessage();
      fmessage.setDetail(message);
      fmessage.setSummary(message);
      fmessage.setSeverity(FacesMessage.SEVERITY_ERROR);
      throw new ValidatorException(fmessage);
    }
  }
}
```

Listing A.6: Validator component for extended type `URL`

Appendix B

Model driven development environments

In this appendix we look at two MDSD approaches in somewhat more detail than in our related work section (Chapter 7). The first system, JetBrains Meta Programming System is rather experimental, whereas the second, OptimalJ, is a commercially used toolset.

B.1 JetBrains Meta Programming System

The JetBrains Meta Programming System (MPS) advocates itself as an instance of a language workbench [?]. MPS is a corporately funded research project and is not yet commercially available (an early access version is available though). In MPS, a programmer can define custom languages, and use these languages to create programs (or *solutions* in MPS terminology). Arbitrary languages can be mixed in this programming environment to create a solution. Languages are defined as a triple of three components:

1. Structure
2. Editor
3. Semantics

An obvious missing item in this list is syntax. Instead, MPS works as a *structured editor* directly on the abstract representation of languages. The reason for this approach is to avoid having to deal with parsing problems that arise when mixing arbitrary languages. Instead of defining a syntax and parser for a language, the programmer can now define an AST instead. The structure is expressed in terms of *concepts* and relations between concepts. An editor must be defined as well. The editor definition instructs MPS what the well-formedness rules are for the language described by the structure. In the editor definition the programmer also specifies the visualization of concepts (from the corresponding structure) and their mutual relations. MPS uses this definition to automatically generate a structured editor for the language. Note that this editor is not a generic editor for arbitrary MPS languages, but one specifically tailored to the underlying structure definition. According to JetBrains, this greatly enhances the usability, since specialized facilities, for example, error-handling can be provided. MPS editors are not diagram editors, as in some of the model driven approaches that feature visual languages. The editor resembles a standard text editor, but changes are only allowed in pre-defined *slots*, according to pre-defined rules in the structure and editor definition. An example can be seen in Figure B.1. The borders indicate editable *cells*.

The last part that has to be defined are the semantics of the language. In MPS, these semantics are defined by creating a mapping to another language. These mappings are called *generators*. The target can be any language, ranging from assembler to JavaȦ pre-requisite is that this language is represented in MPS as well. This means that you will have to re-create the target language (or just a part of it) within MPS, in such a way that a straightforward 1-1 mapping exists from the concepts of the language definition to the actual language. A base language that represents most of Java is already available. Generators can be expressed using three idioms:

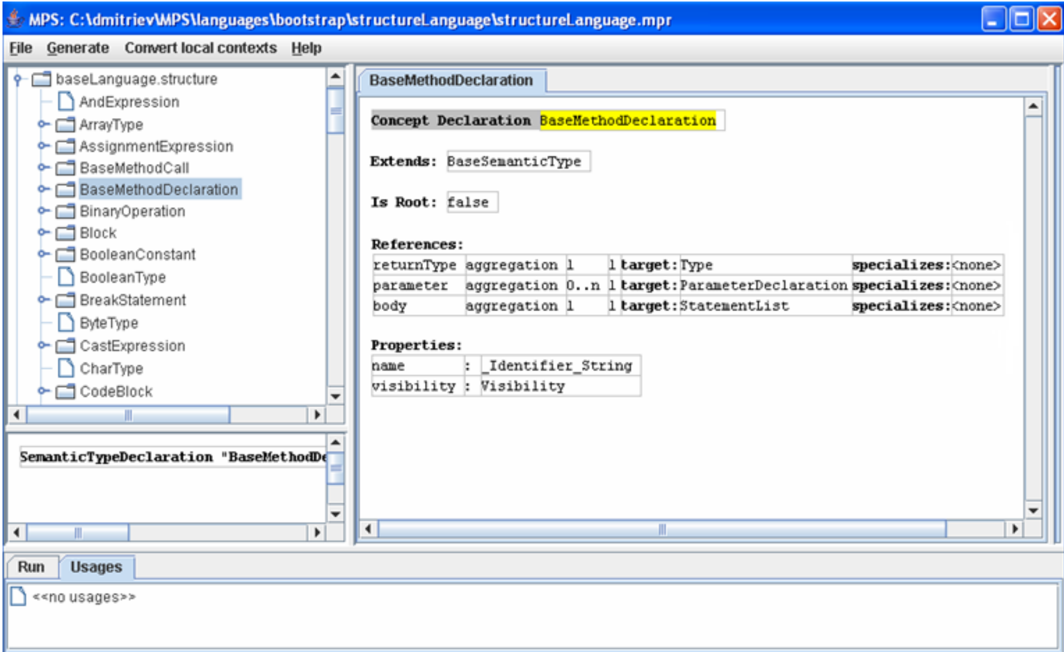


Figure B.1: Structured editor in MPS

1. Templates

2. Query-and-generate

3. Traverse-and-generate

The first idiom is a target-oriented one: a template is defined in terms of the target language, containing *holes* that are filled in at generation time. The three idioms can be mixed arbitrarily, so the hole in a template can be filled by the result of a query-and-generate generator. Such a generator is formulated as a query on the source structure. Specific information can be extracted and used to create a construct in the target language. The last remaining generator idiom is traverse-and-generate. The source structure is traversed, and each construct can trigger the generation of a target construct, much like the way a traditional compiler functions.

Each of the components of the language triple are defined using languages that themselves are first-class citizens in MPS. In other words, the meta-model of MPS is encoded in a bootstrapping manner. As a result, the languages defined using this meta-model are interoperable at least on the structure level. However, there are no guidelines as to how languages interact on the semantic level. It is still up to the programmer to delve into the intricacies of the generation part of a language, to see how it interacts with other languages.

Being the creators of the highly praised IntelliJ development environment, it is hardly surprising that an important role is awarded to the editor in the Meta Programming System. Features such as code completion and code navigation come for free in MPS user-defined languages. These features are very nice, although not very important from a conceptual point of view. More useful features such as facilitating domain specific analyses on languages are not considered. It is, for example, not clear if and how one could impose a type system on languages defined in MPS. A conceptually strong point is that language definitions can inherit from each other. However, a compelling example of this feature is not yet available. Finally, after using it for a while, it is clear that MPS is far from being a stable development environment. This holds for both the actual environment (the IDE) itself, as well as for its ideas and their implementation.

B.2 OptimalJ

OptimalJ is a model-driven development environment targeted at Java enterprise web-applications. It is based on MDA standards, however, the environment is only targeted at web-applications. Consequently, a lot of domain specific knowledge is encoded in the IDE, which is built on the Eclipse platform. Only the *domain class view* is similar to the Business Class DSL of Ordina SWF (Section 7.2). At the start of a new project, an architecture can be chosen. Also, a choice can be made amongst many different implementing libraries for specific parts of the chosen architecture. The remaining guidance offered to the programmer consists of architecture-aware wizards, in which the programmers fills in the variabilities present in the chosen architecture. Only the domain model is created by means of a true domain specific (visual) language.

Code is generated from the model, but again, not all of the code for an application is generated. Since Java does not provide partial classes, OptimalJ introduces the notion of *safe code* blocks. These blocks can be freely edited by the programmer after they are generated for the first time and are indicated within the IDE. Consequently, OptimalJ guarantees these safe code blocks to be safe from modification by the code generators in subsequent generation steps. However, changes in the code are not reflected in the model, i.e., roundtrip engineering is not possible. The code generators all work together to create the web-application, there is no inherent independence between different elements. It is possible to adjust code generators or create new ones using OptimalJ Architect edition. This is an interesting opportunity, since it allows the generators to be customized to the specifics of a certain type of applications. OptimalJ recognizes that no single DSL definition will suit all needs. However, it is undocumented what the extension possibilities in this regard exactly are.