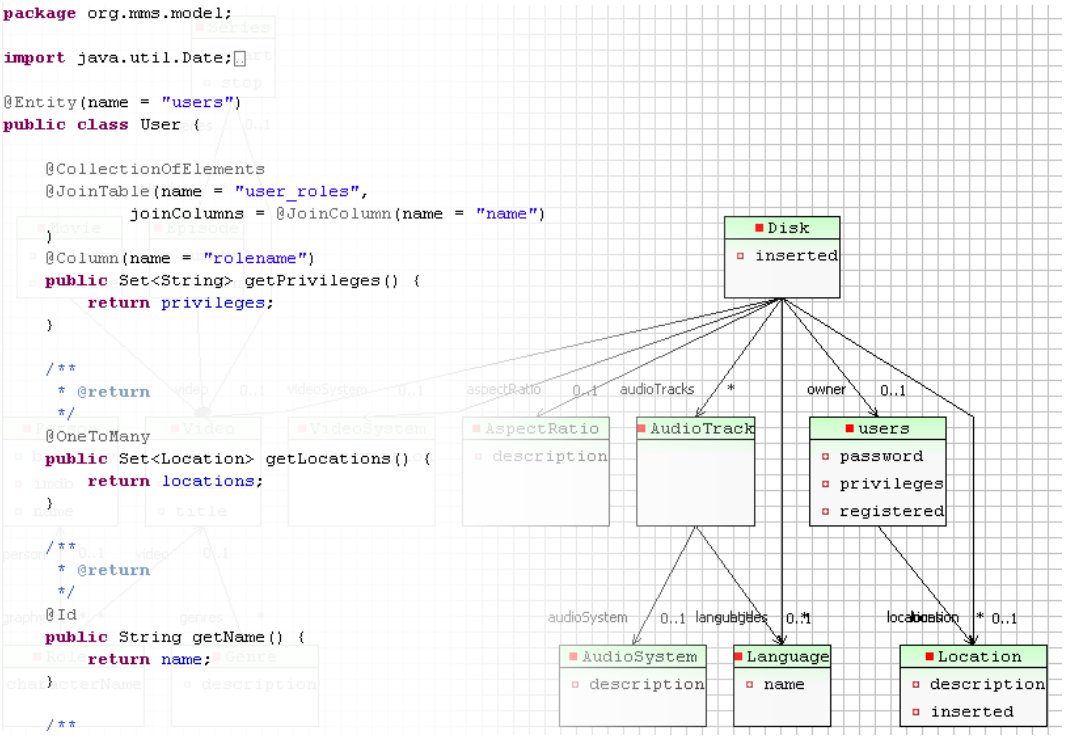
Framework Aware Domain Extraction and Refactoring of Java Applications

October 12, 2007



Rob Schellhorn

Framework Aware Domain Extraction and Refactoring of Java Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Rob Schellhorn
born in Schelluinen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Finalist IT Group B.V.
Stationsplein 45
Rotterdam, the Netherlands
www.finalist.com

Framework Aware Domain Extraction and Refactoring of Java Applications

Author: Rob Schellhorn
Student id: 1100408
Email: rschellhorn@xs4all.nl

Abstract

Model-driven engineering, where functioning system implementations are derived from design models automatically, promises to be the next leap in efficient software development. Still a number of obstacles block the realization of this vision however. In this thesis an intermediate solution is introduced to the reader. Instead of making the implementation subordinate to the design it does the opposite. Static properties of an architecture can be recovered by inspecting a system implementation. Frameworks expose such properties through their configuration data. Combining knowledge from multiple frameworks allows us to recover a complete domain model each time an implementation changes, without a significant delay for the user. A modelling perspective is added to the Eclipse IDE, which is populated with these derived models. The view presents models to the user much like UML class diagrams. Moreover, due to the tight coupling between implementation and model, manipulations can be translated back to the code. Co-evolution becomes implicit.

Thesis Committee:

Chair:	Prof. Dr. Arie van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Eelco Visser, Faculty EEMCS, TU Delft
Company supervisor:	Ing. Nico Klasens, Finalist IT Group B.V.
Committee Member:	Dr. Koen Langendoen, Faculty EEMCS, TU Delft

Preface

I would like to take this opportunity to thank the people who have supported me this last year. First of all I like to thank Eelco Visser, my supervisor from the TU Delft, for his ideas, comments and constructive criticism. The other person who owes my gratitude is Nico Klasens, my supervisor from Finalist IT Group. Without his input, this project would probably not have taken this form. Nico, I still have not figured out how you managed to always have some free time to discuss yet another implementation problem with me...

I thank all my Finalist colleagues for being so interested in my project and of course for the good fun we had Friday after work. Special thanks go to Jean-Luc van Hulst for granting me the time and place to carry out this project.

A personal thanks goes to my parents, who made it possible for me to study in the first place and supported me whenever possible.

Rob Schellhorn
Rotterdam, the Netherlands
October 12, 2007

Contents

List of Figures

Chapter 1

Introduction

Fall 2006 I came across a tool called the Java Application Generator[1]. This application has been created by Finalist IT Group[2] as a means to reduce the repetitive work involved in creating web-based Java applications. JAG reads a model (specified as an UML class diagram) and generates Java code around a number of preselected frameworks. Such a software development approach is known by the term Model-Driven Engineering [41, 1]. Researchers into the topic of MDE are motivated by the following research question:

> How can modelling techniques be used to tame the complexity of bridging the gap between the problem domain and the software implementation domain? [20]

The transformation of abstract software system models into a concrete machine implementation can only be realized, if the modelling language has enough expressive power to describe the envisioned system. On the other hand this language is preferably bound to the problem domain at hand to reduce complexity. So a modelling language should be both *powerful* and *small*. This observation illustrates only one of the challenges identified in [20], blocking the realization of the MDE vision:

1. Problems related to finding the appropriate abstraction level of domain modelling languages.
2. Utilizing different viewpoints, each using (possible different) languages, arises separation of concerns challenges.
3. Model manipulation and management challenges involve problems of defining model translations and managing system evolution.

This work investigates on whether models implicitly embedded in the implementation of arbitrary applications, along with tool support to visualize and edit these models, fit the

[1] `http://jag.sourceforge.net`
[2] `http://finalist.com`

MDE vision or not. The project was carried out at Finalist IT Group as a continuation of the JAG project.

1.1 Project Description

The search for higher level languages is easily justified by the observation that programmers tend to produce statements in the same rate whatever the language [9]. An increase in the expressiveness of a language is thus directly translatable into an increase in productivity. MDE seeks such a higher level language in system models. Promoting models to be the primary resource artefact during the development process has another, very attractive advantage. The modelling process is an excellent way of transferring knowledge from domain expert to software engineer. Certain model views, such as diagrams, are capable of capturing software requirements, while staying understandable for people other than software engineers more than code text can. So models can be an excellent tool for communication. Chapter 3 explains this claim in more detail.

The models created during this inquiring phase of a project are an excellent candidate to be reused. Unfortunately these models are abstracted so much from a software implementation, that they lack all sorts of crucial information required to be executed by a computer. Traditionally, such information is added in an adjacent step by programmers, who translate the model to (for instance) Java code. This translation is performed manually and is very time consuming. Also, like all manual labour, this task is subjected to human errors. Requirements are translated incorrectly or forgotten all together. To make things worse, verifying an implementation for correctness against its design is far from trivial [31, 35]. A machine driven translation, such as MDE promises, is expected to solution for such drawbacks. As mentioned above, there are challenges to overcome first. Besides these challenges other, organisatorial obstacles need to be considered:

1. Developers are grown accustomed to a certain development process. MDE requires a different mind set.

2. Adopting a new technology is subjected to a bootstrapping problem. Businesses will only adopt proven technology. However to proof itself at least some projects must utilize the new technology.

3. The professions of software engineer, requirements engineer and developer will more or less merge.

4. Replacing the current toolset (editors, compilers, frameworks) with a model oriented equivalent will take time and money.

The approach taken in this work seeks for an intermediate solution. Assumed is that an implementation still reflects its design in great detail. Then, the architecture of a system can be recovered by applying *static code analysis* techniques. Knowledge of how frameworks expose architectural properties will improve analysis greatly. The envisioned system is a

modelling viewpoint, which is populated with models found embedded in the source. Second, the impact on the source code, when such derived models are manipulated is looked into. The work is scoped down solely to the problem domain of modelling business logic, other components (e.g. the user interface) will not be part of the derived models. Nevertheless, since business logic is referenced throughout a system, the coupling with other components must be considered. The research questions that will be addressed are:

1. How can embedded domain models efficiently be derived from source code?

2. Can transformations applied to this deduced model consequently be carried through to the originating source text, and if so, how?

1.2 Outline

This thesis consist of two main parts. The first reports on my findings during the research assignment. The second part describes the creation of the Domain Editing Tools, a domain view for derived models. The chapters are outlined as follows.

Chapter 2 is meant as an introduction to the theory of language engineering. Terms like what a *language* or a *grammar* is are explained. Also, the different strategies to write domain specific statements are compared.

Chapter 3 further elaborates on the process of software modelling. The first goal is to establish why software modelling is useful in the first place. Closely related to this question is where, when and by whom software models are used during development. This Chapter concludes with the introduction of formal domain meta-model.

An overview of the research topics, tools and development processes related to the MDE vision are given in Chapter 4. Model Driven Engineering can be seen as the successor of Computer-aided Software Engineering, where models are preliminary used for documentation purposes. Another term often seen is roundtrip engineering, the activity of transforming artefacts in different views. Finally the different MDE initiatives are summarized and compared.

In Chapter 5 related fields of Computer Science are explored. Many aspects of these fields are important for creating a domain driven tool. It is therefore very worthwhile to look into these fields. Included are compilers, model visualization and code generation techniques.

The second part of this thesis reports on creating a prototype of the envisioned domain viewport for the Eclipse IDE. Building on the knowledge established in the first part, an implementation of the functions mapping code to model and the other way around is given in Chapter 6.

In Chapter 7 the popular Eclipse IDE is explored as a platform for the proposed system. The choice for Eclipse is justified by the enormous amount of components that can be reused in the extension.

The next Chapters are devoted to the realization of the implementation of the plug-in. Chapters 8, 9 and 10 successively describe the core, Hibernate builder and user interface.

Finally I reflect on some implementation related obstacles (Chapter 11), discuss what is achieved (Chapter 12 and draw some conclusions and give my suggestions how to continue from here (Chapter 13).

Chapter 2

Languages

As described in the introduction the envisioned system will break with existing modelling approaches, in the sense that model artefacts are not distinct, but rather part of source code. Models will be part of the actual implementation, which implies that the *modelling* language must be part of the *implementation* language. Models themselves capture a language as well, in this case how businesses work. This chapter establishes a basis for creating a modelling language within the semantic constructs of a programming language.

2.1 Formal Languages

A language definition based on a set of mathematical correct formulae is said to be *formal* [29, 37]. For instance:

Definition 1. A *language* is any set of sentences over an alphabet, where:

- An *alphabet* is any finite set of symbols.
- A *sentence* over an alphabet is any string of finite length composed of symbols from the alphabet.

In computer science we are particular interested in regular languages. A formal language is called *regular* if it can be recognized by a (non)deterministic finite state machine.

2.2 Context-free Grammars

A language L is said to be context-free (CFL) if there exists a context-free grammar G such that $L = L(G)$. All regular languages have this property. Context-free grammars are used in the construction of parsers (see Chapter 5). The Java Language Specification [24] gives a nice, accessible definition of context-free grammars, and the related terms I will use in this section.

> A *context-free grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.
>
> Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a *language*, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

Another notation often seen [29, 37, 42] is:

Definition 2. A context-free grammar G is a 4-tuple (V_N, V_T, P, S), where:

- V_N is a finite set of *variables*, or *nonterminals*
- V_T is a finite set of *terminals*
- P is a finite set of *productions*, or *substitution rules*
- $S \in V_N$ is the *start variable*

Then:

- the sets of terminals and of nonterminals have no elements in common; $V_N \cap V_T = \emptyset$
- each ordered pair $(\alpha, \beta) \in P$ is called a *rule* (or *production*) of the grammar and is usually denoted as $\alpha \rightarrow \beta$

2.3 General Purpose Languages

The most powerful kind of languages in respect to the number of problem domains that can be described are General Purpose Languages. GPLs are designed to capture a broad range of software problems. Usually a GPL's syntax is feature rich, allowing the programmer to choose from many different statements (for instance branching or loops). GPLs can be categorized according to their paradigm, such as functional and Object Oriented to name a few. Often the latter is used to model business logic, because many domain concepts map to the OO paradigm [17] quite naturally. These mappings will be discussed in the next chapter. Unfortunately not all domain concepts can be translated evenly well. When such a mapping is problematic, this is called the *impedance mismatch*. Code libraries and frameworks translate domain specific problems into OO solutions. These libraries, and programs utilizing them, are still bound to the syntactic boundaries of the GPL however. Virtually all programming languages used today (e.g. *C#* and *Java*) are examples of GPLs.

2.4 Domain Specific Languages

A Domain Specific Language (DSL) is, just like the name suggest a language specifically tailored for one particular problem domain. Words from the problem domain are used, and its syntax is designed to conveniently match how these domain concepts interact. DSL program text is usually not compiled directly into machine instructions, rather GPL code is used as an intermediate [19] step. Although such a transformation is easier to achieved than creating a binary, it still requires a complete toolset. The use of an intermediate language allows for exploiting an already available compiler and runtime environment. A study into the benefits and disadvantages of using DSLs is given in [14]. This study lists the following advantages:

- DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs.
- DSL programs are concise, self-documenting to a large extent, and can be reused for different purposes.
- DSLs enhance productivity, reliability, maintainability, and portability.
- DSLs embody domain knowledge, and thus enable the conservation and reuse of this knowledge.
- DSLs allow validation and optimization at the domain level.
- DSLs improve testability.

Known disadvantages of the use of a DSL are:

- The expected higher cost of designing, implementing and maintaining a DSL.
- The cost of education for DSL users.
- The limited availability of DSLs.
- The difficulty of finding the proper scope for a DSL.
- The difficulty of balancing between domain-specificity and general-purpose programming constructs.
- The potential loss of efficiency when compared with hand-coded software.

An application will likely cover a range of disciplines. Think for instance about modelling the user interfaces, business logic and database access. Each of these aspects deals with a different problem domain and consequently will use a different DSL. This leaves the problem how such languages can cooperate. Eventually the database needs to be shown on the screen somehow. So it must be possible to connect languages somehow, both at the syntax level and the semantic meaning. I will discuss four alternatives below.

Listing 2.1: Hibernate provides two DSLs for querying the database. HQL is build on runtime evaluated expressions, where the Criteria API is a Domain Specific Library

```
class Invoice {
        public int getRef() { return ref; }
}

// Querying for an invoice using HQL
Invoice invoice = (Invoice) session
       .createQuery("from Invoice a where a.ref=123;")
       .uniqueResult();

// Querying for the same invoice using the Criteria API
Invoice invoice = (Invoice) session
       .createCriteria(Invoice.class)
       .add( Restrictions.eq( "ref", 123 ) )
       .uniqueResult();
```

2.4.1 Runtime Evaluated Expressions

The simpelest solution for embedding a DSL inside another language, is by representing its program text through the host language's string facility. This way the host language is not aware of the embedded DSL at all. The DSLs program text is evaluated just like any other character string. Special components of the system, which are aware of the special meaning of these strings, can decode the instructions and act upon it. Many languages deal with database queries and regular expressions in this way.

A consequence of such an approach is that statements of the DSL are not evaluated until runtime. Therefore errors do not manifest at compile time, but rather at runtime, which can be considered a huge drawback. Also, refactoring an application with such an embedded DSL is dangerous. Consider how in fragment 2.1 for example renaming the `ref` property of an invoice would break the query.

2.4.2 Domain Specific Libraries

Embedding a DSL inside a GPL can be achieved by providing a library for the DSL [22]. Some GPLs are more suitable for this goal than others. Take for instance example code fragment 2.2, which demonstrates that operator overloading is usually preferred over library calls for mathematical problem domains.

Example: the Criteria API has the same purpose as HQL, but instead of runtime evaluated strings, it is implemented as a domain specific library. Also note that in the refactoring example the Criteria API doesn't solve anything (since *ref* is still a string). A domain specific library is usually more verbose than a real DSL.

Listing 2.2: Library call versus syntax support

```
// Matrix operations using a Java framework instance
Matrix result = matrix1.multiply(matrix2)
                       .add(matrix3);

// Or using operator overloading in C#
Matrix result = matrix1 * matrix2 + matrix3;
```

2.4.3 Create a Dialect

Another approach is to let some language A absorb the domain language B, by taking the union of the two. Because A is usually much bigger than B, one can say that the new language A' is a dialect of A. Either A is superceded by the new dialect A' or they both continue to exist side by side.

In the first case the tool support needs to be adapted for A'. Parsers need of course be aware of the new language features. This case is basically equivalent to evolving language A. The later case has to deal with all these problems as well, and on top of that is the language forked. Instead of one set of tools, now two sets have to be maintained.

The main advantage of creating a dialect is, that in the end only one language exists. Programmers only have to learn one grammar and one compiler will do.

2.4.4 Language-in-Languages

Frameworks suffer from the fact that they need to be built on top of the host language, making awkward library calls inevitable [8, 7]. The syntactic gap between frameworks and the problem domain they target can be resolved by utilizing an approach where one language is embedded inside the other. Where a framework implements a language within the semantic constructs of the host language, this approach does not. Compilation for such source files takes place in two steps. First the embedded language is translated into domain specific library calls (or this library is generated in-line) as described above. Then the resulting source is compiled as if it was a normal program. An example of this approach is the pre-processor instructions commonly seen in C. Another examples are found in [8].

2.5 Island Grammars

Sometimes source files contain sentences of different languages. For example HTML pages not only contain hypertext, but also layout (CSS) and scripts (Javascript). Parsing such files is not a simple task. Island Grammars [32, 39] allow for defining specific productions in great detail (*islands*) while the remainder is only vaguely defined (*water*). This property makes Island Grammars also useful for:

- parsing source files which contain syntax errors.

- partially parse files.
- parsing dialect source code.
- constructing a parser when no language definition is available.

2.6 Summary

When projects are set up in a constantly changing environment, falling back to a GPL is sometimes just more convenient. DSLs suffer under such conditions from the fact that not only the language definition has to be kept up-to-date, but also the generators and other tool support. A GPL however, although not tailored to a specific problem domain, can provide the means to model it. This holds especially for Object Oriented languages in combination with domain specific libraries. Such libraries can (partly) make up for the awkward syntax of a GPL in respect to the problem domains.

Chapter 3

Software Modeling

Writing a good software product is more complicated than most people realize. Not many people can grasp the requirements of the user and transform these in a well designed product. Domain modelling results in a better understanding of the problem domain. A satisfactory domain model resembles the current understanding of the problem domain at hand. This chapter investigates on the relation between domain model and software implementation. As a result a domain meta-model is created, which allows for creating models to be embedded in object oriented code.

3.1 Agile Development

Software modelling is often compared to the construction of buildings, because at first sight the two share a similar approach. Like construction, building a new software product involves two very distinct tasks, usually performed by different people. Besides the actual implementation of the system, first the specification what the final product should look like must be agreed on. Traditionally there is a *design* phase and a *implementation* phase. In software engineering such a sequential approach is referred to as the waterfall model.

The analogy with construction does not hold however. Design and implementation seem to be much more intertwined in software engineering than it is the case with construction. History has proven that it is impractical treat the activities as strictly separated. On the one hand problems arise in the inability to act on a changing environment. The time span between agreeing on the design and finishing the actual product may very well take several years. In the mean time the user's expectations might are likely to have changed. Also during implementation new insights in the domain may be revealed. Whether insights revealed during implementation are incorporated back into the design depends not on whether the design would improve, but rather on budget or time constraints. Third, most software projects do not have a strict end, but are merely the beginning for the next release. Design and implementation must co-evolve. This is best achieved by making small development iterations. Such a software development approach is known as agile development and is applied in formal processes like eXtreme Programming [4, 21]. The work in this thesis will

assume an agile environment.

3.2 Ambiguous Communication

Businesses structure their recurring work according to a set of processes. This way they profit from what economist refer to as *learning by doing*; getting better, and thus more efficient, in performing a task simply because you have done it before. However, when asked it appears that explaining such a process to a third party is hard. Moreover incorporating all these processes in one model is difficult, sometimes impossible. A major obstacle is that there are people with understanding of (parts of) the domain and *other* people capable of building software. The domain knowledge needs to be transferred from domain experts to the software architects through communication. Unfortunately communication between people is not trivial; we communicate in an *ambiguous* language. Ambiguity is the cause of misunderstandings and must be avoided. Domain experts and software architects need to come up with a common formal language. A domain model can be such a language.

3.3 A Meta-Model

Domain centric software design, but still directly modelled in an OO language, is called Domain-Driven Design [17]. In his book Evans describes a numberof approaches how domain elements can be modelled in object-oriented code. These constructs are comparable, or even similar to the design patterns described in [23]. A design pattern presents a template solution to solve some common modelling problem. I will briefly summarize the main concepts Evans uses and adopt them in the rest of this thesis.

3.3.1 Modules

A group of coherent objects or components. A rule of thumb for object-oriented programming is to “maximize cohesion” and “minimize coupling”. A person can only think about so many things at once. Grouping small parts into bigger chunks allows for reasoning about whole concepts at once. No real guidelines exist how components must be classified among modules. For instance modules can be scoped based on a particular activity, e.g. sending an invoice to the costumer. Another arrangement is to group components that perform similar responsibilities (all user interface components). What the best solution is, depends on the project and even the architects personal preferences. There is no single object oriented equivalent of a module. Packages, namespaces or projects are just a few suggestions.

3.3.2 Entities

Objects not defined by their property values, but rather by a *thread of continuity and identity*. An entity instance, can outlive the application that created it or can be shared by many different applications. Therefore entities must be uniquely identifiable. A person is usually

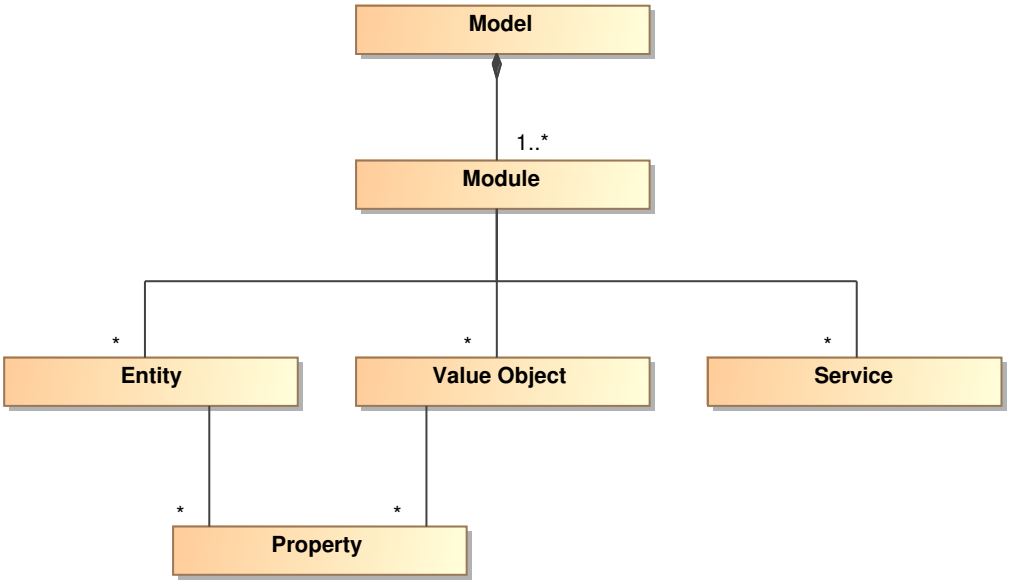


Figure 3.1: A model for domain modeling (simplefied)

modelled as an entity. Although we grow older (property), you still are the same person (well, depending on the context). Entities are encapsulated in types and can participate in a type hierarchy.

3.3.3 Value Objects

Objects that describe some characteristic of a thing. Value Objects should be *immutable*. Therefore other objects do not refer to a Value Object, but hold a copy. A name is usually modelled as a Value Object. People have one immutable name at a time. Renaming a person is equivalent to assigning a *new* name, not changing the old name *instance*. Most object-oriented platforms have some Value Object types build in. Think of all primitive types, character string representations and date representations. Also the programmer is free to compose his own Value Objects from these basic types.

3.3.4 Properties

Entities and Value Objects have properties. A property describes one attribute of the real life equivalent of the type. Some object-oriented languages (e.g. C#) support properties at the syntactic level. Java does not however, rather there is am official specification [38] composed how fields and methods can be combined into a single property. Properties have a type, which can either be a value object or a reference to an entity (only for properties defined in an entity). A property may refer to a single instance or a collection.

3.3.5 Services

Services do not model a thing, but rather *encapsulate some functionality* which cannot be assigned to a single type. Take for instance making a withdrawal at the bank. This process can be described as a function taking some parameters (account, identification, amount), a result (success or a failure) and a list of actions to take (check identification, check balance, subtract money from account, output money). Other examples of services are creating objects, or storing them for later use. A service is best implemented as a method.

3.3.6 The Context-free Grammar

In the previous chapter the definition for context-free grammars was given (see page 5). The modelling language is described by the following context-free grammar:

$$
\begin{array}{rcl}
< MODEL > & \rightarrow & < MODULES > \\
< MODULES > & \rightarrow & < ENTITY >< MODULES > \\
 & \rightarrow & < VALUE-OBJECT >< MODULES > \\
 & \rightarrow & < SERVICE >< MODULES > \\
 & \rightarrow & \varepsilon \\
< ENTITY > & \rightarrow & < NAME >< PROPERTIES > \\
< VALUE-OBJECT > & \rightarrow & < NAME >< PROPERTIES > \\
< SERVICE > & \rightarrow & < NAME > \\
< PROPERTIES > & \rightarrow & < PROPERTY >< PROPERTIES > \\
 & \rightarrow & \varepsilon \\
< PROPERTY > & \rightarrow & < NAME >< TYPE > \\
< NAME > & \rightarrow & IDENTIFIER \\
< TYPE > & \rightarrow & IDENTIFIER
\end{array}
$$

3.4 Refactoring

Repeatedly switching between design and implementation cause entropy to occur. The original architecture deteriorates over time and maintenance becomes problematic. Refactoring [13, 18] is the activity of systematically improving an implementation without changing the semantic meaning. The situation sketched is especially applicable to agile development, for which short development cycles are characteristic. Therefore, formal processes for applying agile development (for instance “eXtreme Programming” [4, 21]) have encapsulated this activity as a daily task.

A refactoring is an atomic modification in which one aspect is improved. Renaming a field declaration to match the meaning of the variable more closely is a refactoring often applied. The last few years IDEs have improved greatly in respect to automating the refactoring process.

The system proposed in this thesis allows for switching between a modelling and implementation perspective even more quickly. Moreover, the modelling perspective exposes implementation specific names directly to the user. The need for refactoring is likely to increase. Code manipulations through the modeling viewpoint can be connected with refactorings (see Chapter 6).

3.5 Summary

Modeling is an excellent activity to gain understanding of the problem domain at hand. A good implementation should reflect much of its original design, and thus embodies this knowledge. Domain related information can be assigned to special object-oriented constructs. This allowed us to define a domain modelling language within the semantic constructs of Java. By systematically improving a codebase, in the form of applying refactorings, a design and implementation can co-evolve.

Chapter 4

Related Research, Tools and Development Processes

Model Driven Engineering is not the only methodology developed in the search for bridging the gap between the problem domain of modelling business logic and software implementation domain. A number of different approaches has been devised, along with the tools to support them. The approaches share the common goal of trying to make models a more prominent part of the development process. They differ however in the way to achieve this. In this chapter an enumeration over such research projects, tools and development processes is given, comparing them to this work.

4.1 Generations of Programming Languages

Programming languages can be categorized according to their generation. One speaks of a new generation if the speed-up in productivity is significantly greater than the previous one. In the early days software was written in the language a computer "speaks": binary. The second generation introduced a more human readable language known as assembly. Instructions are still easily mapped onto to machine instructions, hence little translation was needed, but the notations allowed humans to reason about the program in a more natural way. The term third generation language covers basically all programming languages used to date, including Java and C#. Characteristic is how for each subsequent generation the implementation language abstracted more and more from machine specific details.

In the 1980's it was believed that tool support would bring the next speed-up in development. Therefore such tools now carry the label of being "fourth generation languages" (or 4GL) [40]. Examples are the drag-and-drop user interface builders seen in many IDEs or the editors allowing the user to visually create database schema. Though very useful in certain problem domains, such tools are of course no real languages. Todays vision on this matter has therefore been revised. Current developments indicate that DSLs are the prime candidate for becoming the next generation in programming languages. However whether DSLs are in the basics equivalent to 4GLs, or just different names for the same concept, or

one is a subset of the other is open for debate [14].

4.2 CASE

In about the same period the first 4GLs were introduced, it was established that the available tools were not adequate in supporting software development [34]. It was believed that projects failed in respect to not meeting time and budget constraints due to this lacking tool support. A solution was sought in automating certain aspects of the software development process. Computer-aided Software Engineering (CASE) is a set of software tools or an environment that supports software engineering methodologies and helps the automation of software development [36]. It can be argued that such tools are "fourth generation languages" aimed to supporting the software engineer in performing his job. CASE tools cover a wide variety of problem scopes[1]:

- Code generation tools
- Modelling tools (both data and system)
- Transformation Tools
- Configuration management tools

4.3 Reverse Engineering

Reverse engineering [6] is the process of analysing a subject system to a) identify the system's component's and their relations and b) create representations of the system in another form or at a higher level of abstraction [12]. Reverse engineering is closely related to the following activities:

Redocumentation The process of creating a semantic equivalent representation of the subject system. The same level of abstraction is maintained. One must think of tools like pretty printers, diagram generators or cross-reference listings [6] and views like data flow, data structure or control flow diagrams [10].

Restructuring Transforming a subject system from one representation to another, again the level of abstraction is maintained. Think of code rewriting. Reformulating expressions does not require information about the domain or the applications purpose what so ever. For example transforming multiple cascading if-then-else statements into a switch-statement is what is meant with the term restructuring.

Design Recovery Deducting a higher level of abstraction by combining system source code with tacit knowledge and other system artefacts. Note that fuzzy-logic is required to

[1] Adaptation of: `http://en.wikipedia.org/wiki/Computer-aided_software_engineering`

merge the knowledge. This process is usually driven by human input and cannot be automated (completely).

Forward Engineering The process of realizing a concrete implementation from a design. The design is usually defined at a higher level of abstraction than the concrete system. Forward engineering is a typical MDE activity.

Some popular modelling tools claim to support *roundtrip engineering*, an umbrella activity for reverse and forward engineering. Modelling and implementation produce distinct artefacts, which can be synchronized on request. Roundtrip engineering sounds promising in theory, but still fails in practice. Since the time span between two synchronization points can be arbitrary large, it is extremely difficult to create the synchronization functionality. This results in undesired behaviour. For example there are implementations that annotate source code solely with the purpose to later identify special parts. Such redundant information should be avoided.

This research can be classified as a redocumentation activity since it focuses on presenting domain knowledge that is already present in the source to the user in a more convenient format. Still no information is extrapolated and the same level of abstraction is maintained. So this project does not go into the field of design recovery. When it comes to domain manipulations forward engineering activities are applicable. Note that the proposed system is substantially different from roundtrip engineering, since manipulations trigger directly rewrite the code. This is possible since models are derived from code at all times.

4.4 Model Driven Engineering

MDE refers to the systematic use of models as primary engineering artefacts throughout the engineering lifecycle. MDE can be applied to software, system, and data engineering. Models are considered as first class entities. In the ideal case models can be executable[11] by a computer, like GPL code can. Human driven conversions are not "forbidden" however.

4.4.1 Model Driven Architecture

Where MDE is only a fuzzy set of best practices how to deal with model driven development, the Object Management Group[2] (OMG) has proposed a complete guideline under the name Model-Driven Architecture. This guideline describes how a machine independent model (PIM) can be created and transformed to a machine specific model (PSM). When a platform definition model (PDM) for the target platform is available the transformation can (partially) be automated. For fully automated transformations it is possible to set up a runtime environment for models. Note that MDA is strictly a forward engineering facility. Since the PSM is derived from the PIM, it should not supposed to be edited by the user.

Designing the modelling language for all these kind of models is the real challenge of MDA. For this purpose OMG has defined several standards. UML is the de factor standard

[2] `http://www.omg.org/`

when it comes to modelling. With the second version several different views are available, such as class diagrams, sequence diagrams and state diagrams. Where UML falls short is in defining constraints on items, the Object Constraint Language (OCL) can be used. OMG advocates storing UML models in XML Metadata Interchange (XMI) files, which is easily exchanged among different tools. Since UML has a fixed meta-model, it may fall short in modelling a particular system. For such challenges OMG has introduced the MetaObject Facility (MOF), which allows for adding any arbitrary meta-model. Part of the MOF standard is QVT, which stands for Query, Views, Transformations, and provides a syntax for these purposes.

Several tools have been created under the MDA label. One of the first, and still widely used, is AndroMDA[3]. This tool reads XMI and transforms the model using "cartridges". The transformation to a platform specific model is defined in cartridges. Cartridges can be combined, where each cartridge embodies a part of the PDM. JAG, the application I mentioned in Chapter 1, works in a very similar matter.

The Eclipse Modelling Project[4] is another example. Note that the Eclipse group did not restrict themselves to MOF standard, but rather searched for a practical implementation. The result of this search is eCore, a lightweight meta-modelling language. This technology is used in many tools based on the Eclipse platform including its UML editor.

4.4.2 Software Factories

For her .Net platform Microsoft has designed a model driven development approach as well, known as software factories[5]. A software factory is not so much a strategy to software engineering, but rather a tool, which targets a specific domain. Such tools are intended to integrate into Microsoft's development environment Visual Studio and add domain languages, editors and views for specific problem domains. Although different names are used, the ideas are just the same.

[3] `http://www.andromda.org/`

[4] `http://www.eclipse.org/modeling/`

[5] `http://msdn2.microsoft.com/en-us/architecture/aa699360.aspx`

Chapter 5

Other Related Fields

MDE can be connected to fields of Computer Science other than modelling. This chapter explores these related fields and summarizes the parts that are important for creating the proposed system.

5.1 Compilers

Computers cannot execute source text directly. First, a conversion to a list of machine instructions is needed. This conversion can either be performed before execution (*compilation*) or at runtime (*interpretation*), but the fundamentals are roughly the same. A compiler [2, 25] can roughly be divided into two parts, often referred to as the front-end and the back-end. The first is responsible for reading source text (lexical analysis), converting it to an internal representation (syntax analysis), and validate this tree against the semantic rules of the programming language (semantic analysis). The internal representation is called the *abstract syntax tree* (AST). The back-end takes this AST as input and performs the translation into the target language.

Among the objectives of this thesis is establishing a methodology for deriving a domain model from Java code, solely using *static code analysis* techniques. An AST is an excellent representation of code, in respect to analysis. The remainder of this section will describe the techniques used by a compiler in general and as a case study how these parts are implemented in the Eclipse, a popular Java IDE. The idea is to reuse this compiler's front-end to obtain the required ASTs.

5.1.1 Lexical Analysis

A compiler reads program text, encapsulated in files. The ultimate goal of the front-end is to convert such character sequences into an AST. The inputted text should comply to the rules of the programming language(s) used, but may very well not. It is the task of the lexical analyser, or lexer, to tokenize the input text. Each token is is represented by a terminal (see Chapter 2) and each terminal is described using a regular expression.

For arbitrary languages the lexer has to have a set of regular expressions describing the possible tokens for that language. Tools exist which are capable of deriving a working lexer, if regular expressions are available for every terminal in the grammar. The output is a sequence of tokens, which are fed to the next part of the compiler chain, the syntax analyser.

5.1.2 Syntax Analysis

The next step is to detect the structure in the token stream, produced by the lexer. This process is called *parsing*, and is performed by a *parser* during the syntax analysis. The token stream should be processed according the production rules of the grammar in such a way that it results in the start symbol. Two fundamentally different parsing strategies are used, top-down or bottom-up. A top-down parser chooses the correct production p for known non-terminal alternatives. Suppose the lexer provides input token t, then the parser needs to decide which production p needs to be applied to create a correct sub tree. On the other hand bottom-up parsers repeatedly find the first rule which has not been constructed yet, but all its children have. Such a rule is called a handle and creating a node in the tree for that handle is called reducing the handle to p. Each time a production rule fires a piece of the AST is constructed.

Different classes of context-free grammars (Chapter 2) exist. Top-down parsers are utilized when it comes to parsing *LL* grammars, where *LR* grammars are associated with bottom-up parsing. Both grammars read program text from left-to-right (hence the first 'L'). The first set of parsers choose the *L*eftmost derivation, where the latter chooses the *R*ightmost. Often a parser needs to look ahead a number of tokens to reach a decision. Such are denoted as $LL(k)$, or respectively $LR(k)$ parsers where k is the number of tokens read ahead.

Relatively new developments allow for dealing with ambiguities in a language definition in a declarative manner. This takes away the need to solely introduce non-terminals for this purpose. The Syntax Definition Formalism (SDF) language applies such techniques.

5.1.3 Semantic Analysis

At this point a complete syntax tree has been constructed. Still first some more checks need to be performed to decide whether or not to accept the input as valid. For example many programming languages require variables to be declared first before it is allowed to use them. Java byte code is guaranteed to be type safe. Semantic analysis is the process of verifying the input to satisfy such constraints.

The reason why a tree-like data structure is especially suitable for analysis lays in the fact that tree nodes represent language construct instances. Information required for verification is often available local or in close proximity of this node. The "visitor" design pattern allows for conveniently walking the tree and defining actions to be taken for every node type. Information derived during semantic analysis is often saved inside the tree node for reuse. As a consequence at the end of this phase one often speaks of an *annotated* AST.

Listing 5.1: Compilers often optimize code by performing partial evaluation

```
void compute()
{
    const int a = 4;
    const int b = 6;

    int c = a * (a + b);
    print( c ); // Will always yield 'print( 40 )'
}
```

5.1.4 Code Generation

In a much similar fashion the annotated AST is walked again in order to producing target language text. Often each node is directly translatable. However this approach is rather naive. Special care must be taken to get the control flow of the target code right. Also simple optimizations may result in a significant speed-up. Think of statements for which all information is constant (see also example 5.1). Evaluating such statements at compile time results of course in faster runtime code.

5.1.5 The Eclipse Java Compiler

Java compilation is available in Eclipse through a plug-in called the Java Development Tools (JDT). Many of the features expected from a modern IDE, require a tight coupling with the programming language. For instance views such as the program outline or features like code completion, expose the structure of programs *at all times*, even while the source file is being changed. This is not easily achieved using an external compiler. Therefore IDEs usually have a compiler build in, rather than relying on external build tools.

JDT ships with such a compiler, based on a parser generated by Jikes Parser Generator[1]. This parser generator is capable of creating parsers for $LALR(k)$ and $SLR(1)$ grammars and thus powerful enough for describing the Java language as specified in [24]. Other parts of the compiler were inherited from IBM's VisualAge[2] development environment. The primary unit of work is just like for most compilers one file, called a *compilation unit*, which can be processed in batch or incremental upon file change. A compiled compilation unit is called a class file. Eclipse can create ASTs not only from compilation units, but from class files as well. Parsing fragments of Java code is also supported, but will of course result in incomplete ASTs. Code fragment 5.2 demonstrates how ASTs can be obtained for files.

Most views in Eclipse do onyl expose a small portion of the AST. Since obtaining an AST is quite expensive, JDT also provides a lightweight data model. Packages, types and members are part of this model, but the contents of method bodies (expressions and statements) are omitted since they do not contribute to the programs structure. Elements in

[1] `http://jikes.sourceforge.net/`

[2] `http://en.wikipedia.org/wiki/VisualAge`

Listing 5.2: Building ASTs from source files

```
ASTRequestor requestor = new ASTRequestor() {
    public void acceptAST(ICompilationUnit source,
                          CompilationUnit ast) {

        // Traverse and/or manipulate the AST
    }
};

ASTParser parser = ASTParser.newParser(AST.JLS3);
parser.setProject(project);
parser.setResolveBindings(true);
parser.createASTs(sources, bindings, requestor, monitor);
```

this model are said to be handle objects, since such objects do not have to be backed with actual code. Many of the handle objects act as a façade when it comes to manipulating source code. Think of deleting, moving or renaming a compilation unit.

5.2 Model Transformations

Model transformations are a common activity in model-driven development. Compilation and transformation are much alike, or actually compilation is a special transformation. During compilation program text in some language *A* is transformed to a semantic equivalent version in language *B*. In general model transformations deal with deducing different model views. Although MDE strives for automating such transformations for a maximum gain in productivity, a human driven approach is not prohibited.

Most tools created under the MDE label derive (complete) system implementations from models. This process involves generating machine code, though often some higher language is used as an intermediate medium. Remember how the JAG application reads UML and writes Java code. Such transformations are best modelled by a template system. JAG utilizes the Velocity[3] engine for this purpose.

Templating are effective, because each transformation contains little context sensitive information. No information needs to be shared among templates. The domain element types defined in Chapter 3 are mapped to templates in the target language using a surjective function. So each domain element instance corresponds to processing one or more templates. The few context specific parameters that do exist are filled in with information derived from the instance.

Thus transformations generate new code. This is not sufficient for the proposed system though, since existing source files must be transformed. The new insights must be merged into the existing program state. A document object model is a tree-like data structure much

[3] `http://velocity.apache.org/`

like an AST, but allows for to be changed. Each AST produced by the Eclipse compiler is actually a DOM and therefore a good foundation to build the desired transformation on.

5.3 Model Visualisation

A completely different area, but nevertheless important is how domain models are presented to the user. Difficulties arise when derived models have to be layout in a diagram, since no information like coordinates is available. Several algorithms [5, 16] have been designed, which find a suboptimal layout based on a predefined list of criteria. Such constraints are usually simple goals like minimizing the number of edge crossings[4] or placing related objects in a close proximity [26]. Unfortunately, readability is not determined solely by such properties. An extensive list of aesthetic criteria which influence readability is presented in [15]. In this paper the author concludes that graph-based algorithms, which do not take the semantic meaning of a program into account, cannot create aesthetically appealing diagrams for arbitrary input.

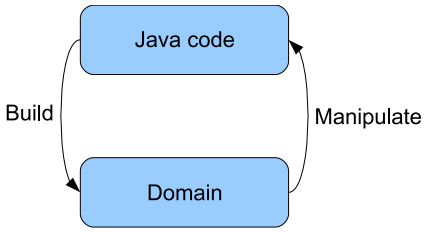
[4]Minimizing the number of edges is actually a NP-complete problem. A solution cannot be computed within polynomial time.

Chapter 6

Global Design

In the first part of this thesis the theory needed for designing a modelling language has been established (Chapter 2). Also such a language was composed within the semantics constructs available in typical Object Oriented programming languages (Chapter 3).

Unfortunately, this definition is not powerful enough to map an arbitrary model to code and vice versa. For example entities are mapped to types, but not all types will be an entity. Additional information is required to distinct the parts of a program that do contribute from the parts that do not. This work seeks this information in hints provided by frameworks. Many frameworks operate within a domain that intersects with the modelling domain.



This chapter describes what frameworks can be used to complement Java as a modelling language. Also two projections are defined, one that maps code to domain model and one that maps domain manipulations back to code manipulations. The remainder of the thesis is used to describe how tool support for such a language can bring design and implementation closer together.

6.1 Domain Extraction

The language $L(I)$ in which a programmer can express himself, the implementation language of an arbitrary project, is the aggregation of the Java language, the standard runtime library and frameworks used within that project. Remember how a framework is a means to embed domain specific knowledge within the semantic constructs of a host language. It can be said that each framework x adds a language $L(x)$ for a certain problem domain to $L(I)$. The domain modelling language $L(M)$ is found in (parts of) these components and thus is $L(M)$ a subset of the implementation language $L(I)$.

Important is that no two frameworks should recognize the same model elements. If this were the case it becomes impossible to guarantee consistency throughout the model. Con-

Listing 6.1: Contradicting names can be declared. Here the name specified in the annotation overrules the type name

```
@Entity(name = "Foo")
class Bar
{
    // members
}
```

sider what will happen if the entity in the what seems trivial code fragment 6.1 is processed by two rules, one picks the name specified by the annotation (`Foo`) and the other the type name (`Bar`), since that second framework does not know about the annotation. There is simply no way to merge such information.

Normally you do not want two frameworks with more or less the same functionality within the same project. The implementation is made unnecessary complex. There are cases in which such situations may occur however. When a project is being migrated from one framework to another for example, there will likely be a period of time the two frameworks live side by side. This work has no answer for such cases and considers the frameworks used to be fixed during the project's lifespan.

Definition 3. $L(M) = L'(x_0) \cup L'(x_1) \cup \ldots \cup L'(x_n)$, where

- $L(M)$ is generated by the context-free grammar defined in Section 3.3
- $L'(x_i) \subseteq L(x_i)$
- $L(x_0)$ is the Java language
- $L(x_i)$ for $1 \leq i \leq n$ is a language extension contributed by framework x_i
- $L'(x_i) \cap L'(x_j) = \emptyset$ for all $0 \leq i, j \leq n, i \neq j$

Therefore the problem is reduced to pick a suitable combination of frameworks. I found in Hibernate[1] and Guice[2] a combination, which together provide enough information to construct $L(M)$.

6.1.1 Hibernate

Many applications work on vast amounts of data. Because such a great number of applications require accessing persistent data, this functionality is available as a separate, but highly specialized component, the database. Data is best modelled as tuples (tables), which can hold related data. This principle is implemented in what is called a Relational Database

[1] `http://hibernate.org`
[2] `http://code.google.com/p/google-guice/`

Listing 6.2: Entities, value objects and properties defined by Hibernate

```
@Entity
public class MyEntity {
    private int a;

    public int getA() { return a; }
    public void setA(int a) { this.a = a; }
}

@Embeddable
public class MyValueObject {

}
```

Management System (RDBMS). The gap between the OO and relational paradigm is often referred to as the O/R mismatch. Mapping inheritance is the most prominent manifestation of this mismatch. A RDBMS can model associations and aggregations ("has-a" relations), but lack support for inheritance ("is-a" relations)[3]. Solving this mismatch is the domain of O/R mappers, such as Hibernate.

Hibernate requires from the programmer to identify the persistent part of the domain model. Entities and value objects as well as their properties are usually persistent. Two notations are supported by Hibernate to mark such element. Hibernate Core [28] utilizes XML mapping files, where Hibernate Annotations [27] uses the metadata facility available since Java 5[24]. Thus the persistent part of an application's domain is modelled by the Hibernate configuration.

Contributions are straight forward and one-on-one. Entities are juest regular types annotated as such. Value objects are types annotated as embeddable types. Properties are encoded following the Java Bean guidelines. I will explain how to recknize these types in more detail later.

6.1.2 Guice

Coupling between software components is considered bad. It reduces the ability to switch between implementations and makes code harder to test. Coupling occurs when one component implicitly refers to implementation details of another component. Better is to hide implementation details by exposing a component's functionality through a contract. OO languages provide interfaces for this purpose. Using interfaces solves only part of the problem though. The concrete objects must be instantiated and obtained where needed somehow. Factories and service locators only shift the problem as explained[3] by Bob Lee, one of the creators of Guice. The design pattern known as "Dependency Injection" allows for binding a concrete implementation to each contract at *startup time*. With this specified a compo-

[3]Google webcast: `http://video.google.com/videoplay?docid=6068447410873108038&q=guice`

Listing 6.3: Modules and services defined in the Guice configuration

```
public class MyModule extends AbstractModule {
    protected void configure() {
        bind(MyService.class).to(MyServiceImpl.class);
    }
}
```

nent called the "Inversion of Control container", then takes care of injecting the correct implementation for each contract where needed.

Guice is a framework implementation of this design pattern. It provides *typesafe* dependency injection in contradiction[4] to Spring, the best known dependency injection framework for the Java platform. Because of this I could restrict my domain compiler to the Java model and not create the need to parse the XML configuration files used by Spring.

6.1.3 Other Frameworks

Of course, other frameworks can do the job as well. Many frameworks exist with more or less the same possibilities, and more important, way to be configured. Guice can be replaced by Spring, if the compiler is also capable of recognizing XML meta-data. In the same way other O/R mappers can replace Hibernate. Since the system is designed to be modular, with all framework specific knowledge in a separate plug-in only minimal effort should be required.

Only very few projects will use this particular combination of Hibernate and Guice. It is thus important that extensions for other frameworks are created in the future.

6.2 Manipulating the Domain Model

When a model has been derived, it would be interesting to see how changing this model would affect the program text it was derived from. Therefore each model manipulation has to be translated into a source manipulation. If such a manipulation exists and can be applied successfully, model and code are synchronized again. Some manipulations are easier to realize than others however. For instance, introducing new elements to the domain is trivial, since it only requires generating new code. Difficulties are to be expected when existing elements are altered or deleted. Other code referencing these elements will break, if not taken care of. This section explains the implications on the code of each manipulation and presents ideas how to deal with conflicts. In chapter 11 the actual implementation of these manipulations will be discussed.

[4] Since Spring 2 (released November 2006) Java 5 annotations can also be used for configuration.

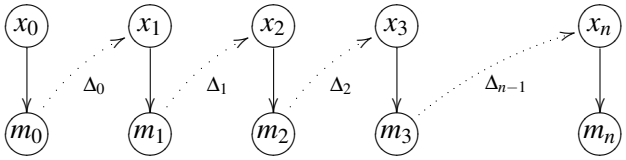


Figure 6.1: Manipulating a model m_i triggers a state transition on the code from revision x_i to x_{i+1}. Such a manipulation is described by delta Δ_i.

6.2.1 Introducing new Domain Elements

When new elements are added to the domain model the Java code must be updated accordingly. For domain element based on a Java type (either a Module, Entity, Value Object or Service) this action boils down to generating new types. Properties are a bit more complicated, since they have to be added to an existing type. Code fragments 6.4 to 6.8 define templates for each domain element type. Context sensitive information is controlled through parameters. Every Java type has a qualified name, where the qualifier is represented by the package name and the simple name is a unique name within that package. Entities are a bit more complicated, since they can contribute in an inheritance hierarchy. Therefore an optional supertype is also controlled by a parameter.

6.2.2 Changing and Deleting Domain Elements

As said changing domain elements is complicated in the sense that the code representing the element may be referenced throughout the system. For instance consider how deleting the `Invoice` entity would break even the simple service listed in code fragment 6.9. The manipulation may only be carried out when it is found to be safe to rewrite the code, hence no references to the element exist. In other cases actions must be taken. Automated measurements are preferred over human judgment, but not always available as illustrated in the `Invoice` example. If the change computed by the system will result in compile errors, the result is first presented to the user. It is up to the user to review the change and decide whether or not to commit it.

Listing 6.4: Template for generating a new Module

```
package <Qualifier>;

import com.google.inject.AbstractModule;

public class <Name> extends AbstractModule {
    protected void configure() {

    }
}
```

Listing 6.5: Template for generating a new Entity

```
package <Qualifier>;

import org.hibernate.Entity;
import org.hibernate.Id;

@Entity
public class <Name> (extends <Type>)? {
    private Long id;

    @Id
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
}
```

Listing 6.6: Template for generating a new Value Object

```
package <Qualifier>;

import org.hibernate.Embeddable;

@Embeddable
public class <Name> {

}
```

Listing 6.7: Template for generating a new Service

```
package <Qualifier>;

public class <Name> {

}
```

Listing 6.8: Template for generating a new Property

```
package <Qualifier>;

(import <Qualifier>;)*

public class <Name> {
    private <Type> <Name>;

    (Modifier Annotations)*
    public <Type> get<Name>() { return <Name>; }
    public void set<Name>(<Type> <Name>) { this.<Name> = <Name>; }
}
```

Listing 6.9: Deleting the domain element “Invoice” is prohibited, since removing the type will break this fragment.

```
/**
 * Sends a reminder for all invoices that are overdue.
 */
public void sendReminders() {
    for( Invoice invoice : repository.retrieveAll() ) {
        if( isOverdue( invoice ) ) {
            // Logic for sending the actual reminder
        }
    }
}
```

Chapter 7

Eclipse

This chapter reviews how the Eclipse IDE can be used as a host platform for the proposed system. The Eclipse community has grown steadily since the first release in fall 2001. Many industry partners, including big internationals like IBM, Nokia, Oracle and others, have contributed. The latest release shipped over 17 million lines of code, which makes Eclipse the biggest open source project around. Much of the functionality required by the proposed system is part of the platform. For instance the Java compiler, which was already described in Chapter 5, provides a framework for compilation related operations. It is believed that many of these components can be reused or build upon. This chapter describes these components and presents ideas how we can let Eclipse work for us, followed by a description of the concrete implementation in the next chapters.

7.1 The Platform

The Eclipse project started out as an open source Java IDE. Over time, all language specific components were removed from the main architecture and made available through plug-ins. This resulted in a small, but highly generic and highly flexible workbench. Nowadays a rich set of both official[1] and third party[2] plug-ins is available. To name just a few:

1. Development tools for Java (JDT), C/C++ (CDT) or Web languages (WTP).
2. Database connectivity (DTP).
3. Functional and performance testing (TPTP).
4. Modeling (EMF, GMF and of course the proposed plug-in).

Actually in a strict sense Eclipse no longer is a Java IDE, but rather a self hosting platform to evolve itself as a language workbench, and as a framework for building rich clients upon.

[1] `http://www.eclipse.org/projects/`
[2] `http://www.eclipseplugincentral.com/`

7.1.1 The Workbench

The user interface component of the Eclipse platform is called the workbench. In figure 7.1 it's common concepts are highlighted. It is build on the Standard Widget Toolkit[3], so the look and feel is much like a native application. SWT wraps native controls in Java classes or emulates them in the case a certain control is not available on a platform. As a result resources must explicitly be disposed of; a responsibility SWT leaves to the user. Typical UI components (labels, images and text input) are called widgets, hence the name of the toolkit. More advanced concepts such as wizards and monitors for tracking the progress of long running operations also exist.

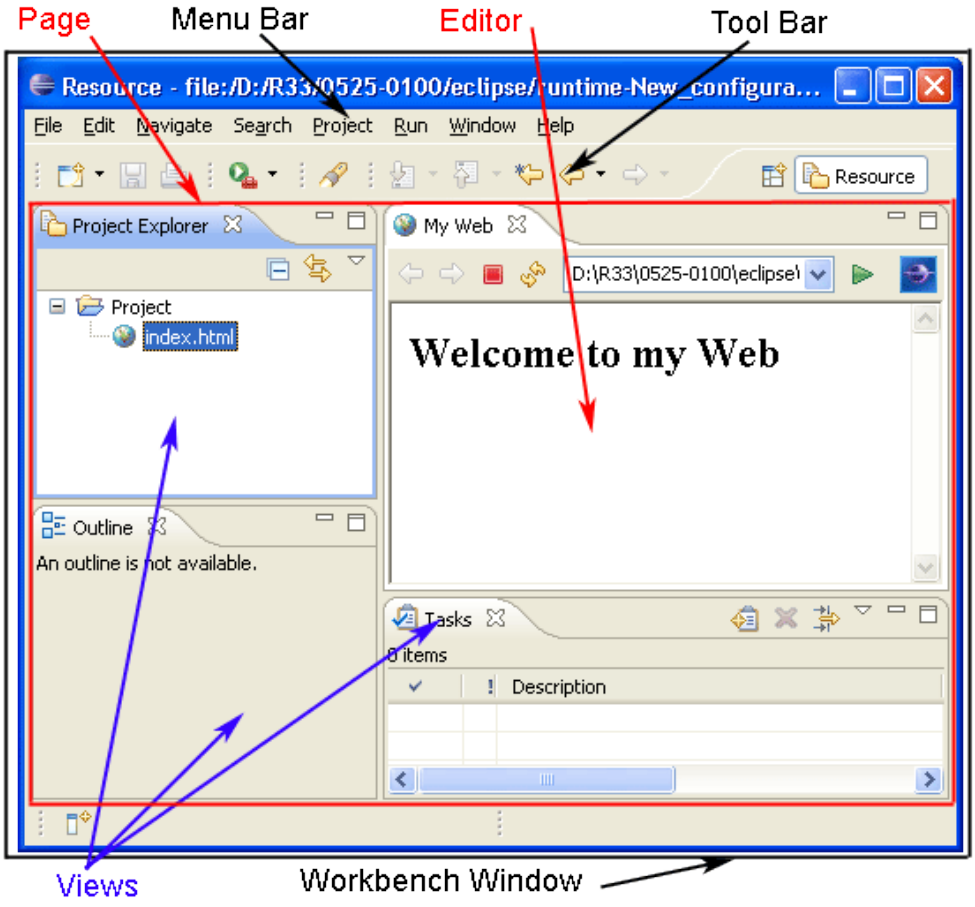


Figure 7.1: Common concepts of the Eclipse Workbench. *Source: Eclipse manual*

[3] http://www.eclipse.org/swt/

7.1.2 Resources

As an IDE Eclipse often needs to create, read and modify files. Considered it's platform independent nature, such operations are not trivial. An abstraction layer (see figure 7.2) was introduced to deal with shortcomings of the functionality provided by the standard Java library. The work of a user is organized in *projects*. All projects share a common parent, namely the *workspace*. The user is completely free in creating *files* and *folders* inside a project.

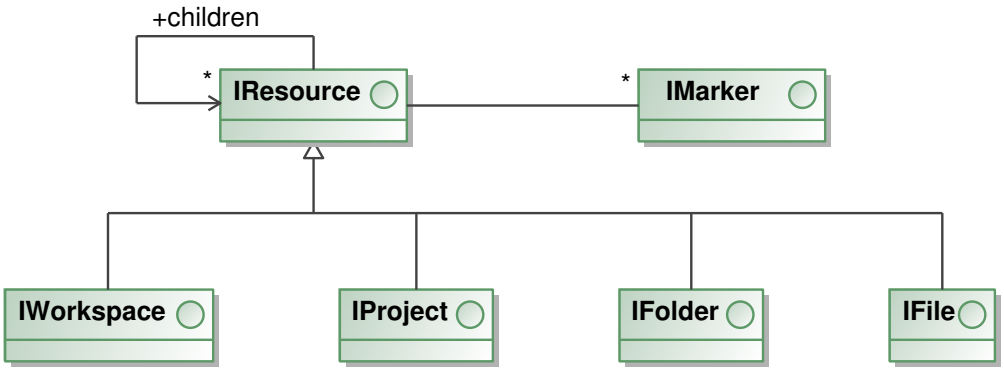


Figure 7.2: The Eclipse resource model.

Moreover Eclipse is capable of annotating any resource in the workspace. Each resource manages a set of *resource markers*, which may hold any kind of (textual) meta-data. Within Eclipse user tasks are persisted this way, but also compilation warning and error messages end up as a resource marker. Code fragment 7.1 illustrates the life-span of a marker.

Listing 7.1: Resource marker operations

```
// Create new markers
IMarker marker = resource.createMarker(type);

// Set attributes
marker.setAttribute(key, value);

// Find markers on a resource
IMarker[] markers = resource.findMarkers(type,
                                         includeSubtypes,
                                         depth);

// Delete a marker
marker.delete();
```

7.1.3 Equinox

The Eclipse platform itself is quite minimalistic in the number of features available. As described additional functionality can be plugged in the workbench. To make this possible Eclipse has a sophisticated plug-in mechanism, known as Equinox, a runtime environment for controlling the lifecycle of plug-ins. Equinox is an implementation of the OSGi[4] specification, which is a standard for this purpose. In OSGi's termology a plug-in is referred to as a *bundle*. The lifecycle of a bundle includes dynamically installing, starting, stopping, updating and uninstalling bundles. Other parts of the system can locate bundles through the service registry.

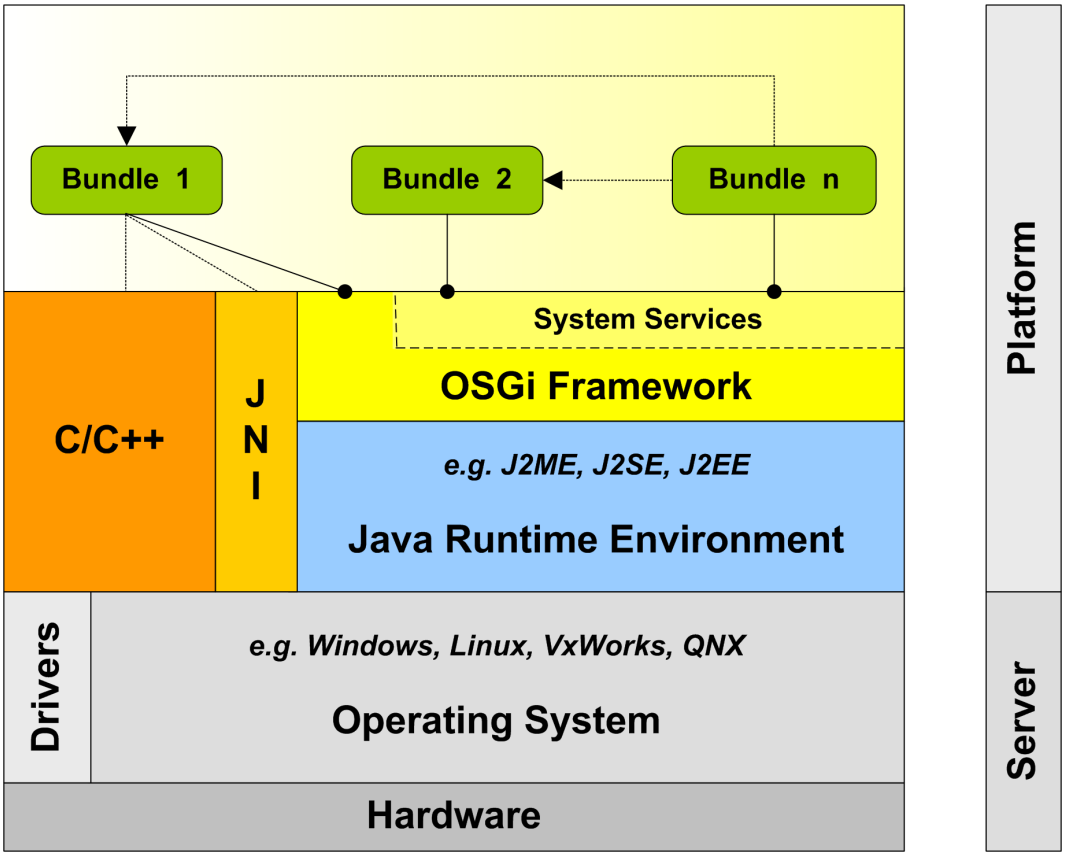


Figure 7.3: The OSGi framework provides the means to control the lifecycle of bundles. Source: [43].

[4] http://www.osgi.org/

7.2 Java Development Tools

Although no longer part of the workbench, Java development is still the foremost feature of Eclipse. Therefore JDT is part of most Eclipse distributions. JDT consists of[5]:

- An incremental Java compiler. Implemented as an Eclipse builder, it is based on technology evolved from VisualAge for Java compiler. In particular, it allows to run and debug code which still contains unresolved errors.
- A Java Model that provides API for navigating the Java element tree. The Java element tree defines a Java centric view of a project. It surfaces elements like package fragments, compilation units, binary classes, types, methods, fields. See also figure 7.4.
- A Java Document Model providing API for manipulating a structured Java source document.

7.2.1 The Java Compiler

The Eclipse Java compiler was already described in Chapter 5 as an example case. In summary it is a framework façade for compiling, analyzing and manipulating Java code. Especially the means to construct ASTs from source files are of interest for creating a domain compiler.

7.2.2 A Java Model

JDT extends the resource model with a thin, Java aware layer, the Java model. In figure 7.4 the different element types are enumerated. The model represents a simplified version of the source code, and is used to provide feedback to the user. For instance the different views depend on the Java model, but also auto-complete retrieves it's information from it. Eclipse reads an element in memory the first it is requested. A most recently cache is used to decide which elements can be removed from memory.

7.2.3 A Java Document Model

Domain manipulations trigger a change in the source code. Either new code must be generated or existing code rewritten. The latter can be achieved with help from the Java Document Model, a structured representation of Java documents, which usually allows to be modified. JDT allows for recording such modifications on ASTs created from entire compilation units (see example 7.2). Modifications in the DOM can be written back to the source code. Note that the DOM is not verified for correctness before saved. Only when the source document is saved a compilation is triggered.

[5]Adapted list from: `http://www.eclipse.org/jdt/core/index.php`

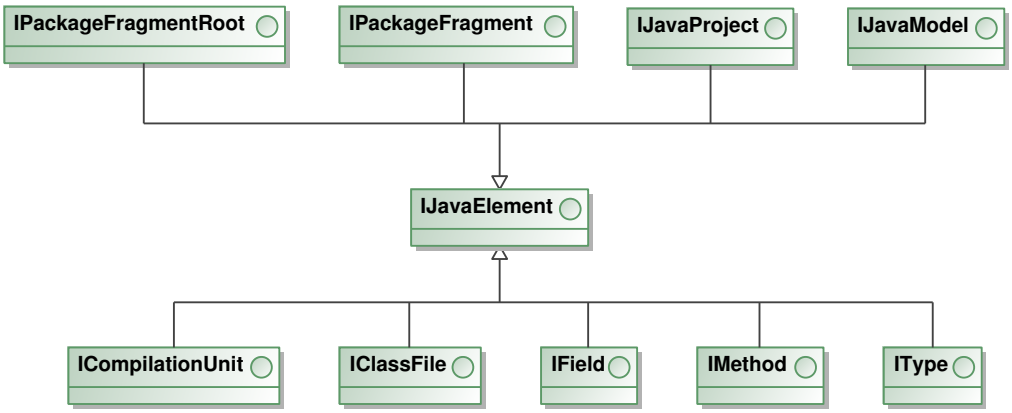


Figure 7.4: The Eclipse Java model.

Listing 7.2: Example of how Java source can be manipulated through the DOM tree.

```
protected UndoEdit modifyAst(CompilationUnit cu,
                             IDocument document) {

    // Record any changes in the DOM tree
    cu.recordModifications();

    // Add a new class "MySource" to the compilation unit
    TypeDeclaration type = cu.getAst().newTypeDeclaration();
    type.setName( cu.getAst().newSimpleName("MySource") );
    type.modifiers().add( ModifierKeyword.PUBLIC_KEYWORD );
    cu.types().add( type );

    // Apply changes to the source document
    TextEdit edit = cu.rewrite(document, options);
    UndoEdit undo = edit.apply(document);

    return undo;
}
```

7.3 The Language Toolkit

Eclipse belongs to the latest generation of IDEs[6]. The criteria Fowler uses is whether an IDE provides proper refactoring support. Present day Eclipse is the front runner in this area. A refactoring modifies program code with the goal of improving one or more structural aspects. The Language Toolkit is an framework for modelling such transformations in

[6]Martin Fowler: `http://martinfowler.com/bliki/PostIntelliJ.html`

Eclipse. In the LTK programmer's documentation we can read:

> Refactorings are used to perform behaviour preserving work space transformations.

A refactoring is an *atomic* and *undoable* modification to the workspace. This makes LTK a suitable framework for recording the state transitions as described in Section 6.2. All refactorings follow the following life cycle:

1. Some action triggers a refactoring, for instance the programmer indicates that some type must be renamed.
2. Optionally a user dialog is presented in which the user can input paremeters, in this case the new name. In the same time the system checks whether the entered name is suitable and provides feedback in the case it is not.
3. When the value is submitted the *change* is computed in memory without modifying the actual resources. Optionally, the change is presented to the programmer in a two-way compare dialog (see figure 7.5), for reviewing purposes.
4. If the user accepts the change, the actual source is modified. The reverse of the change can be computed automatically and will be applied in the case of an undo.

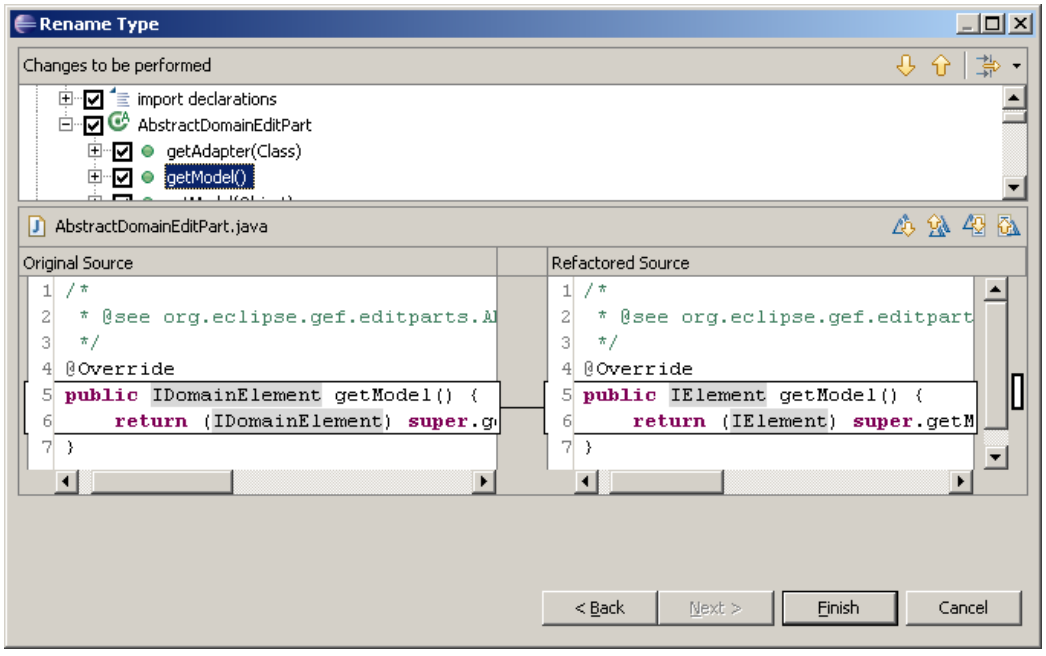


Figure 7.5: Two-way compare dialog of the Language Toolkit.

7.4 Graphical Editing Framework

Graphical oriented editors can easily be created using the Graphical Editing Framework [33]. For the developer contributions are twofold. In the first place it provides a library for drawing two dimensional figures on a canvas: Draw2d. Next it connects model and view by providing an abstract framework implementation of the model-view-control pattern. GEF is marked as "Tools" project, meaning that by itself no functionality as added to Eclipse. The Visual Editor[7] and Graphical Modelling Framework[8] are just two examples of plug-in build on GEF technology.

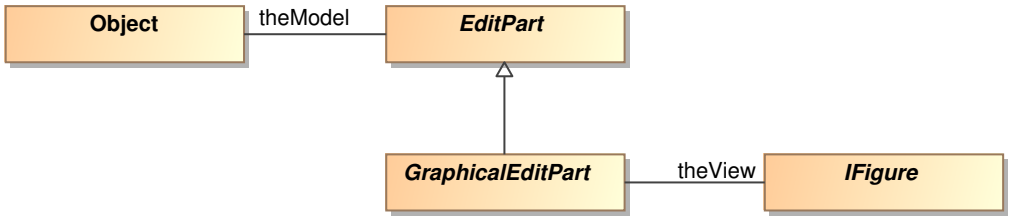


Figure 7.6: GEF's Model-view-controller framework.

[7] http://www.eclipse.org/vep/WebContent/main.php
[8] http://www.eclipse.org/modeling/gmf/

Chapter 8

Core Functionality

A prototype of the proposed system has been implemented as an Eclipse plug-in under the working name "Domain Editing Tools". One design goal was to make the plug-in independent of the frameworks used to complement Java as a modelling language. Therefore framework specific code was extracted in a separate, replaceable component. The architecture is as follows:

The generic component which defines the data model and provides facilities to interact with Java code.

A framework specific component which provides the domain builders for the specific frameworks (see Chapter 9).

The user interface which contains the means to visualize and edit the domain model (see Chapter 10).

The framework specific component and the user interface depend on the generic component. No other dependencies exist. It is therefore possible to replace the framework component. This chapter highlights the three responsibilities of the generic component.

8.1 The Data Model

In Chapter 3 a meta-model for domain modelling in an Object Oriented language has been established. The main responsibility of the generic component is to define a data model based on this meta-model. The data model is implemented in a similar fashion as the Eclipse resource model (Figure 7.2) and Java model (Figure 7.4). One important difference exists however. Elements from the resource and Java model are said to be "handles". Such handles capture the state of an element at a given point in time, but are not updated when the underlying model changes. A handle is stateless. If a workbench part needs to react upon model changes, it must listen for them itself. The domain editor will be based on the GEF framework. This framework defines controller hooks based on the MVC design

pattern. Using handles would be extremely impractical. Therefore the delta processing is part of the data model itself.

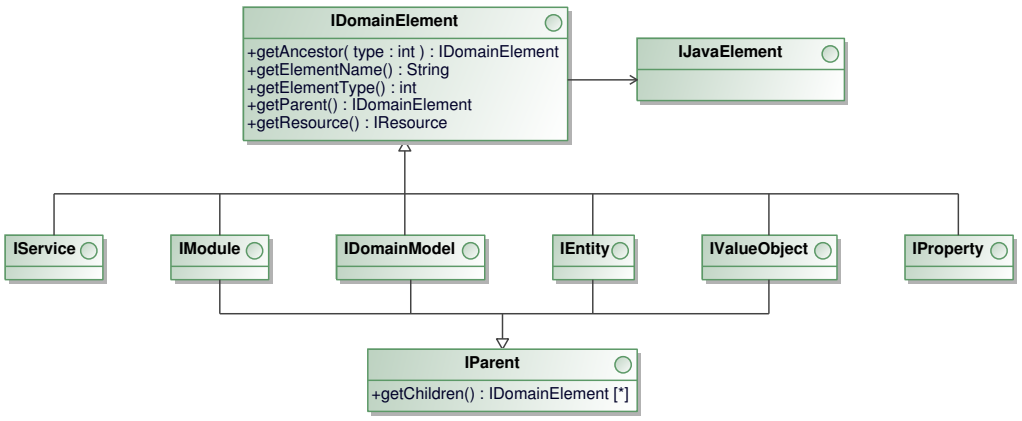


Figure 8.1: Class diagram of the meta-model (simplefied)

8.2 The Domain Compiler

In the same fashion class files are generated as a result of compiling Java code, a domain model can be derived by a domain compiler component. Each change to the Java model may cause the domain model to be changed, so the domain compiler should run just after the Java compiler.

8.2.1 A Java Compiler Participant

Compilation is expensive in respect to the amount of processing power it takes. Even on modern hardware, compiling a large project may take minutes or even hours. Still Eclipse provides real-time feedback to the user through views, populated with information which can only be derived during compilation, even while the text is being edited. This is achieved through incremental compilation. An incremental compiler only processes those portions of a project that have been altered. The domain compiler must take a similar approach, because static analysis can even be more costly than compilation. The domain model can only change when the Java code has been altered. And, to perform static analysis, an updated AST is needed. Therefore the domain compiler should be invoked after the Java compiler has run. JDT allows compiler *participants* to hook into the Java compiler. These participants are informed on the events of starting and finishing a compilation. In the case of such an event, each participant is called, providing the context as an argument. The domain compiler is implemented as such a participant.

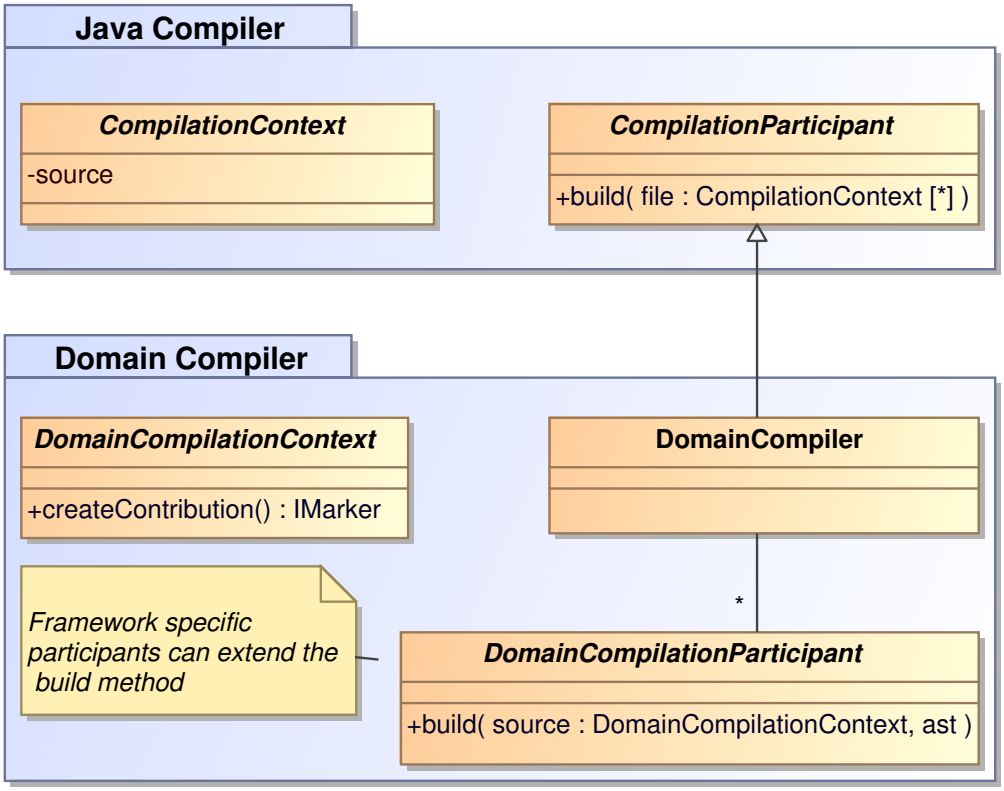


Figure 8.2: The Domain compiler is a participant of the Java compiler.

8.2.2 Contributions

Domain contributions are derived data. Such data should not manifest in the users workspace. To remember information across workbench sessions, the contributions must be persisted however. Therefore the plug-in will store contributions as resource markers, the Eclipse resource meta-data mechanism. For each domain element type a marker is defined, which holds the attributes of an instance of that element type.

8.2.3 Delta processing

The domain compiler creates a new set of contributions for every compilation unit that is processed. This information must be merged with the information already present in the in-memory model. Therefore a listener to the Eclipse resource model is created. This listener is informed each time the workspace has changed. Eclipse fires a message containing a resource delta tree, a data structure that resembles the change. This tree clearly specifies which markers have been created, removed or altered. And since each marker contains the information of a particular domain element the tree can be merged with the model, simply

by processing all domain resource markers.

8.3 Manipulations - The Contract

Like JDT adds a layer on top of the resource model, the domain model is yet another layer on top of the Java model. Since domain models are derivatives of the underlying Java code, manipulations to the domain have to be translated to Java changes (remember figure 6.1). Then the domain compiler can pick up the changes and update the model accordingly. The core part of the domain plug-in provides an interface for the user interface that abstracts from the underlying Java manipulations. This abstraction is implemented using LTK refactorings. The following domain manipulations can be applied to domain elements:

Introduce

New elements can be introduced to the model. It is however dependend on the type in what context an element can be created.

- `Module` elements can only be added to the `DomainModel` element.
- `Entity`, `Service` and `ValueObject` elements must have a `Module` element as parent.
- `Property` elements belong to `Entity` or `ValueObject` instances.

Within a scope each element must be uniquely identifiable by its name. To ensure this the name must be given prior to creating the actual element, so this can be ensured. For `Property` elements also the type must be specified, since this is a required attribute. The user can choose whether or not to automatically generate a default identifier property for an `Entity`, because one is required anyway.

Move

Modelling is called as such, because it is a process of trial and error. Sometimes a mistake is made. For instance a `Property` is first thought to be a child of `Entity` X, but later it is realized that the `Property` should belong to Y. In such cases it must be possible to move the element. In Java this means the field, getter and setter are removed from the type represented by X and recreated in Y. This refactoring is far from trivial.

The contents of the getter and setter method *should* only reference the field, but this cannot be guaranteed. Pre-, post and class invariant conditions can be checked. Also a getter is sometimes used to place a lazy loading mechanism for the property value. Such code should also be relocated if possible.

It is of course also to be expected the property is referenced in other parts of the software. These references will no longer point to the correct code. The referencing code does not have to know about Y. In such cases no correct change can be computed automatically. There are cases where a better solution can be achieved however. When a property is moved

within a type hierarchy (also known as *pull up* or *push down* depending on the direction) it is sometimes possible to rewrite references as well, because to *instance* stays the same, only its new *type* is more general or specilized.

Rename

Like moving domain elements, something simple as giving a domain element a more appropriate name can improve the clearity of the model much. Therefore all domain element types can be renamed. Note again that a name must be unique within the scope of its parent. Also the name must meet the Java restrictions for identifiers.

Eclipse has excellent refactoring capabilities when it comes to rename refactorings in the Java model. This support is reused, since each domain element type derives its name from the Java model. Renaming the underlying Java code is exactly whould should happen. A property rename is covered by renaming the Java field, since getter and setter method are automatically processed as well when specified in the refactoring.

Delete

To remove elements from the domain, no additional information is required, besides the element to remove. Note that a delete action is cascaded to all the element's children. Removing an `Entity` will also cause all it's properties to be deleted. The same problems occur in respect to references to the deleted element as in moving domain elements.

Changing a Property's Type

Properties exhibit the coupling between domain model and Java model to the user. A property's type is something that manifests both in the domain and Java model. It must be possible to change this type. Two sorts of types can be distinguished, primitive types and object types. The later present some challenges.

First, a type can either be a single object or a collection. A single object can either be nullable or not. Hibernate models this as a property of the `@Column` annotations. Collections are modelled using one of the Java collection interfaces. Note that the actual property type is in such cases not the Java type, which is `Collection<Entity>`, but the type parameter. Object types that are also domain elements model a relationship between the properties parent an the type.

Also Hibernate requires more meta-data than necessary to detect these relations. Collections must be marked with *@OneToMany* where single object properties are annotated with *@ManyToOne*. Although this information is redundant and can be derived by inspecting whether the object type is a persistable type or not, Hibernate requires this information. These annotations must be dealt with correctly.

Changing an Entity's Supertype

Entities may participate in a type hierarchy from a domain modelling perspective. It should thus be possible to set the supertype for an Entity. An Entity inherits all the properties of its parent. This causes problems when a supertype is altered. Three distinct cases can be identified.

The first is where subtype *A* has no supertype and a generalization is created to supertype *B*. Now *A* inherits all properties from *B*. Problems occur when properties with the same name, but different (not assignment compatible) type exist in *A* and *B*, since this is not allowed in Java. When the properties are compatible one can argue that the property can safely be removed from the subtype. It is also possible to just leave the code in the subtype as is. Java 5 then will require an `@Override` marker annotation on both the getter and setter method on the subclass however.

The second case is when subtype *A* inherits from *B* and the generalization is removed. References to *A* may invoke inherited methods, which will no longer be valid. In such cases one can first perform a *pull up* refactoring for all those properties.

The last case is when *A* inherits from *B* and is assigned a new supertype *C*. This refactoring basically covers first to remove the existing generalization and then create a new one.

Chapter 9

Framework Builders

The part that distincts this work from other round-trip tools is twofold. First, the round-tripping is restricted to code with a special purpose, namely modelling a domain. Second, because framework specific knowledge is used in the mappings, the synchronization can be much faster and more accurate than when arbitrary code is inspected. In this thesis I looked into how Hibernate and Guice together provide enough hits to decide what domain code is and what is not.

9.1 The Hibernate Builder

Hibernate can create mappings from an object oriented data model to a relational database and vice versa. Hibernate is actually an implementation of the EJB3 persistence API, defined in the Java language extension `javax.persistence.*`. To do this, the programmer must specify those elements that must be persisted. Since Java 5 annotations are the preferred way to add such meta-data. Annotations are part of the Java model and thus available at compilation. Three different domain element types are deduced from the configuration:

- Entities
- Embeddable objects
- Properties

Embeddable objects serve the same purpose as Value Objects do, model an immutable object without a lifespan of its own. Hibernate refers to such objects as embeddable since they occur only as children of entity types in a composite tree. The basic idea of the Hibernate builder is as follows. For every compilation unit all types marked as Entity are recorded as such. Similar Embeddable types are recorded as Value Objects. Also all members of these types are inspected whether they meet the EJB3 property requirements.

Listing 9.1: The EJB3 persistence API allows an Entity name to be specified as string literal

```
@Entity("AnotherName")
public class MyEntity {
    // ...
}
```

9.1.1 Entities

The description of the builder is somewhat oversimplified. In respect to Entities three exceptions must be considered.

The first is the location of annotations. An EJB3 property is a composite of a field, a getter method and optionally a setter method (which may be omitted for read-only properties). The standard allows annotations on both the fields and getters, but mixing both strategies within a particular type is not allowed. Either all meta-data is defined on fields or everything is defined on the getter methods. Hibernate determines which strategy is used, based on the location of the `@Id` annotation:

> The access type is guessed from the position of `@Id` or `@EmbeddedId` in the entity hierarchy. Sub-entities, embedded objects and mapped superclass inherit the access type from the root entity.

Also EJB3 allows Entities take part in a inheritance strategy. From a database perspective three mapping strategies can be defined:

- Table per Class
- Single Table per Class Hierarchy
- Joined Subclass

Hibernate only considers supertypes annotected as `@Entity` or as `@MappedSuperclass`. The latter types are not part of the domain model. What is meant with such a construction is that all properties of the supertype should be "copied" in the subclass.

The third and last obstacle is that the name of an Entity can be specified in two different locations (see fragment 9.1). This possibility was created so legecy database schema could be mapped to Java code without the need to adopt an existing table name. New elements should only specify the name as type identifier.

9.1.2 Value Objects

With EJB3 embeddable types can be declared in two ways, as a top level type or inline inside an Entity. The first is marked as `Embeddable` and the second `Embedded`. From a domain perspective both are the same however and model a Value Object. These element types are processed in the same way Entities are.

9.1.3 Properties

As said property meta-data is either accessed through the field or getter. Important attributes of a property are the name and type. When field access is used, the field name corresponds to the property otherwise the getter name is used, without the "get" part and the first character is in lower case. For properties with a boolean type "get" is replaced with "is".

When Entity and Value Object types are processed, all non-static, non-transient members are considered to be a Property. A combination of field, getter and setter with the same name, but different type occur, a warning should be given, since the type is not compatible with the Java Bean specification.

9.2 The Guice Builder

Where Hibernate more or less produces domain contributions without room for interpretation, Guice does not. Both Module and Service elements are not modelled by any framework in such a manner that the meta-data directly results a a suitable domain element description. Still, much useful information can be deduced.

9.2.1 Modules

Guice modules all implement the *Module* interface. Therefore detecting the modules is rather trivial. Eclipse can search for implementations of that interface within the scope of the project. Unfortunately not every module has to be a domain Module and there is not way to separate one from the other. The system may contain, next to the Domain, modules for bundling all different kinds of functionality together. Think of a Authorization module or a Replication module.

9.2.2 Services

A service is something you typically would like to be injected into the system and not depend on a specific implementation. Therefore all objects tht are bound to a Guice module are considered to be a service.

A service does not have to be the embodiment of domain logic, but can also perform some complicated functionality used in the rest of the system. Here I refer to creating (Factory) or locating (Repository) complicated data structures. Both may produce a domain element, but do not belong to the model themselves. Without some heuristic there is no way to distinguish one from the other. A heuristic could be based on the name (My*Factory*) or, better, on a framework that specifically annotates services as such.

9.3 The Location of Refactoring Logic

In the previous chapter a list of available manipulations has been given. It was also explained why the contract for such manipulations must be part of the data model. The frame-

work specific logic however should not be part of this model, since that would imply a coupling between the domain model and frameworks. It would not be possible to simply replace one framework with another. The actual refactoring logic should be contributed separately like the builder is. In the current implementation this is not the case.

Chapter 10

User Interface

An important aspect of the plug-in is how models are presented to the user. The way diagrams are visualised can improve insight in the system greatly. An Eclipse editor, which paints diagrams much like UML2 class diagrams has been created for this purpose. This chapter presents the design decisions of this editor, and all other user interaction.

10.1 The Editor

The domain model editor is based on the Graphical Editing Framework, which allows users to create an editor based on an existing data model. The user can hook into existing implementations of the model-view-controller design pattern for connecting model and view. Manipulations are encapsulated in commands, atomic undoable mutations of the model.

10.1.1 Layout

Visualising derived models is not trivial. No information about the placement of elements (*vertices*) and their relations (*edges*) is stored. Diagram must be created automatically. Many algorithms have been devised[5, 16] for this purpose. Such algorithms optimize a layout according to several aesthetic criteria (e.g. the number of overlapping elements or diagram size). None the less readability also depends on a user's preferences [30]. Normally the inputted model must meet certain constraints. For instance most algorithms do not allow cycles in the input. GEF contains several implementations of such algorithms, called routers:

- The bend point router layouts connections through a list of bend points.
- The fan router layouts connections in a fan like fashion upon collision.
- The Manhattan router creates paths with an orthogonal route between the source and target.
- The shortest path router bends a collection of paths around rectangular shaped figures.

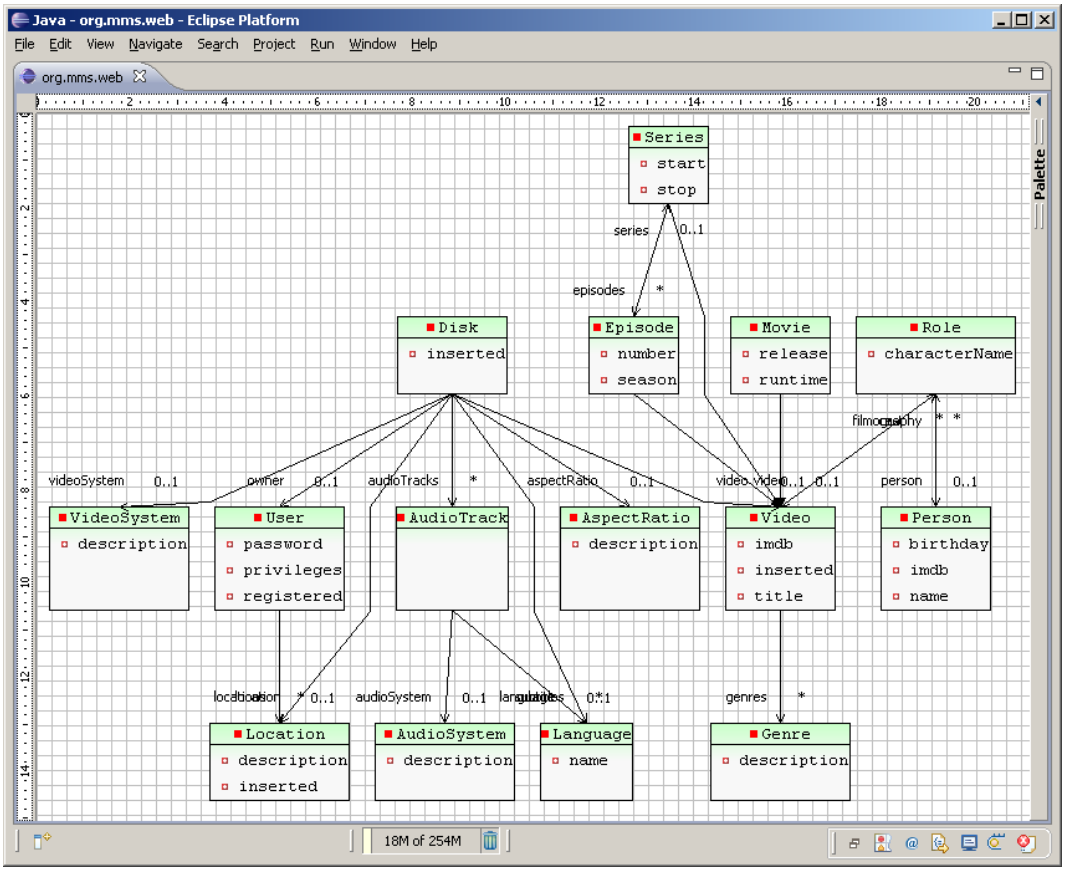


Figure 10.1: Screenshot of the domain editor

These algorithms have in common that they require a directed acyclic graph representation of the input. Every element based on a type (Entity, Value Object or Service) is represented by a vertex in the DAG, while edges are defined by the references and generalizations in the model. References and generalizations have a sense of direction, so all edges are naturally directed. Cycles may exist however and must be removed in the graph. Another constraint is that the DAG must be connected, which is usually not the case for every arbitrary models. If this constraint is not met vertices can be connected by adding artificial connections.

Creating an automatic layout algorithm is not part of the scope of this project. Since developing such an algorithm is rather difficult and the layouts created by the shortest path router where quite satisfactory no additional effort was put into improving this algorithm. The user can select one of the other algorithms mentioned above as well.

10.1.2 Navigate a Project

Derived domain diagrams are used as a very intuitive frontend for navigating source code. Most editors do not compute relations between compilation units. Eclipse allow the user to explicitly query for one relation, but only on demand (e.g. a reference search, or type hierarchy). The domain editor can display these references at all times. Since a tight coupling between domain elements models and the source code exists, the domain editor can be used as a means to navigate source code as well. This enhances the way models are used as a means to provide more insight in the structure of a project even more.

Double clicking opens a Java text editor for the code that represents the clicked element. Selecting an element will populate the appropriate views, for instance Javadoc and properties. A feature that is not implemented yet, but would also increase insight is focussing the domain editor on the element that is currently being edited in the Java editor.

10.1.3 The Palette

New elements are created by dragging a *palette* entry onto the diagram. Each entry represents a specific element factory, which embodies the logic for creating new instances of that domain type. Elements can only be created within a specific context. For instance properties must be added to entities or value objects. Therefore a GEF controller (GraphicalEditPart) may review whether its model is a suitable drop point for the given factory. This is achieved by installing a policy, which creates the actual create command if the given type is acceptable.

10.1.4 Interaction in the Diagram

Other manipulations are carried out on the figures in the diagram directly. Trivial and like most graphical editors, selected elements can be deleted from the domain model by pressing the delete button. Renaming elements is achieved by clicking an element's name, which opens a direct text editor. Relations are edited in a very intuitive way. The following three manipulations are defined:

- By dragging the target of a reference to a new type, the property which declares the reference is assigned a new type.
- Dragging the source of a reference reallocates the property to the a new type.
- Changing the supertype can be achieved by dragging the target of a a generalization to a new type.

The goal was to mimic behaviour seen in most graphical (UML2) editors and connect the manipulations.

Listing 10.1: Create, configure and run a refactoring to create a new Property

```
public void execute() {
  try {
    IntroduceProperty ref = entity.createProperty();
    RefactoringWizard wizard = new CreatePropertyWizard(ref);
    RefactoringWizardOpenOperation operation =
        new RefactoringWizardOpenOperation(wizard);

    operation.run(shell, Messages.introduceProperty_title);
  } catch (InterruptedException e) {
    // Interrupted by the user
  }
}
```

10.2 Manipulations - The User Interface

Some manipulations require additional user input. For instance creating a new property can only be performed if it's parent, name and type are given. The first is derived from the location the user drags the property factory onto in the diagram. The other two fields however cannot be computed somehow and must be given by the user. The domain refactorings each specify an API for setting the required fields. For input LTK provided abstract wizards to conveniently configure refactorings in a user interface. A wizard can directly configure the refactoring based on user input. The refactoring is responsible for providing feedback whether this input is valid or not. Domain refactoring logic and user interface logic is thus strictly separated.

10.3 Preferences

The editor has some visual aspects that some people like more than others, therefore the user can modify some of this behaviour. The domain editor adds a number of pages to the Eclipse preference system, in which the user can set:

1. The algorithm to use to layout the diagram.
2. Whether or not to animate model transitions.
3. The severity of problems in domain code.

The possibility to specify these preferences was considered "nice to have", but not a crucial part of this thesis.

Chapter 11

Implementation Related Considerations

The actual implementation proved to be quite challenging for two reasons. One, although many frameworks could be utilized, the amount of new code involved is quite large. Two, some mechanisms are quite complex. Especially translating Java changes back to the model caused me lots of problems. In this Chapter I will explain the implementation related choices I made.

11.1 Scope

The first thing that became clear to me when I started implementing the plug-in was I set my self a rather ambitious goal. Too ambitious if I was planning to implement the complete plug-in within the time constraints I set myself. Therefore it was decided to create a solid basis and then add as much functionality needed to illustrate the strengths and weaknesses of the modelling approach taken in this work.

Also, various techniques were new to me, include GEF, LTK and Guice. I had already played around with the Eclipse platform and Hibernate a bit, but not so much that I could rely solely on my existing knowledge. Getting familiar with such techniques takes time. This caused me more than once to take a wrong decision in how things should be implemented. For instance most manipulations I have created twice. The first time implemented as a workspace operation, the second based on LTK refactorings.

11.2 Working with an Incomplete Implementation

As said some parts of the implementation are left out. Much effort has gone into the core, user interface and Hibernate builder. This left little time for Guice. As a result Modules and Services are recognized only partially. Without Modules it is not possible to introduce Entity and Value Objects. Therefore I created a user interface which allows the user to specify a location in the Java model to create the element in. As a result input through a

dialog is required, instead of dragging the factory onto the parent `Module`. When Module support is finished, this behaviour should be adapted.

11.3 About Manipulations

This thesis was driven by two research questions. The first was whether it is possible to derive enough information from code, using the knowledge of particular frameworks, to display an accurate domain model. I think I have proven that, for static models, this is possible. The insight in what domain elements are and how they are related is something that was not part of the Eclipse IDE before.

The second question, whether manipulating the derived model could directly be translated back to the underlying Java code, cannot be answered with a determined *yes* or *no* at this point. Some domain modelling manipulations can be translated to uniform code changes, but not all. I decided that in cases where I have no good solution, it is up to the user to first prepare the code for the refactoring. The plug-in provids feedback in the form of the compare dialog, so the user knows how the code will be changed. In my opinion most refactorings can be improved in such a way, that at least a reasonable, but not necessarily the desired, suggestion is presented to the user.

Chapter 12

Discussion

This work looked for a means to improve productivity in an agile environment by introducing a domain modelling viewpoint into the IDE. The goal was set to construct the means to identify, and interact with, domain models embedded in a project's source code. Such a viewport may act as a link in the chain of evolving the object oriented development process to a model driven approach. In this chapter I will discuss whether these goals were achieved with the implemented Eclipse plug-in, based on the following six criteria.

The single most important aspect of the plug-in is that it *does not break existing code*, without the user's approval. All actions that alter, and thus may break program text, should first allow the user to review the edit. This is implemented in the form of a LTK two-way compare dialog, but only for a small amount of manipulations however. Also some changes can touch many files, which makes the process of reviewing time consuming. A more sophisticated (e.g. filtered) approach is preferred.

Of course the domain state must be *correct* and *complete* in respect to the represented elements in code. This is not always the case for references defined by property mappings. Many-to-many relations can be defined using the *map*-semantics. Such constructs are not recognized by the domain compiler. Another aspect of the domain compiler that can be improved is the cardinality of references. The current implementation makes a distinction between either "0..1" (single types) or "*" (collections). Hibernate allows simple types to be annotated as *not null*, which corresponds to a cardinality of "1".

Third, the *availability* of domain models is important. Models are derived as part of the compilation process. Creating an AST and performing static code analysis each time the Java model changes is rather time consuming. Incremental compilation improves responsiveness greatly, since only a small number of files is processed at a time. Furthermore whether a file is worth processing can be decided fairly quickly, since contributions only exist in compilation units containing special top level types.

Additionally, the viewport must give a *gain in insight* of the structural part of the project.

This is what the whole project is about. For small projects the viewport gives much more insight than the package explorer can. However diagrams can become huge, which reduces the explanatory power. The problem lies in balancing the amount of information that is presented to the user at a time. Diagrams containing up to about 40 elements work fine. Diagrams containing more elements will become messy. The lack of a module implementation, which groups elements into sub diagrams is missed here. Adding sub diagrams is believed to improve on this situation. Also, when modules contain so many elements the project has to be considered anyway.

Fifth, the *effort* it takes to carry out domain manipulations using the plug-in should be significantly lower than editing the code directly. This holds for some manipulations more than for others. For example editing an entity's name is achieved through direct edit of the label. This is not significantly less effort than renaming the type using a code refactoring. A counter example is changing a property's type, which is simply achieved by dragging a reference to the new type in the diagram. It should not require knowledge of writing Java (much).

Finally, the plug-in should not be *intrusive*. Users may choose to install and use the plug-in or not. Whether one project member uses the plug-in should not matter for the other members. Also resources should not be altered solely for the purpose of making the plug-in work. Since markers are stored at a separate location in the workspace this goal is achieved.

Chapter 13

Conclusions and Further Work

It is believed that in the long run system models will replace conventional source text written in for instance the Java language, as the primary resource during development. Applying models as such is expected to give software development a boost in efficiency. Unfortunately MDE has not matured enough to be put up to the task of modelling arbitrary design challenges seen in projects. Both language development and organisatorial (in software and business) challenges have to be overcome first.

In this thesis I introduced an intermediate development approach based on conventional Java development, but with domain driven aspects mixed in. This approach is particular suitable in an agile environment, where a design is subjected to change even when implementation has started already. As test case an Eclipse plug-in, called the Domain Editing Tools, has been created. When the Eclipse compiler is invoked, a domain model is derived from source using solely static code analysis. This model is presented to the user much like an UML class diagram. Manipulating the diagram triggers the source code to be rewritten accordingly.

I found that this approach significantly improves on making the static structure of small projects visible. It immediately becomes clear what the domain elements are and how they relate. Larger projects however contain so many elements that the amount of information becomes overwhelming. Splitting up the diagram in several sub-diagrams will improve on this, but has not been implemented yet. A sub-diagram can for instance be created for every module. Another means to focus the attention of a user is by grouping corresponding files in a working set. When a working set is active, all information not in the working set is hidden from the user.

From a practical point of view the supported number of frameworks must be extended. Although many new applications are build on the Hibernate framework, only very few utilize Guice. It is likely that legacy systems not even use Hibernate. Therefore the number

of supported frameworks must be extended. As an alternative for Guice I suggest a Spring[1] builder. Spring is used in many projects and can provide the same information Guice can. It must be possible for a user to indicate which frameworks are used in a project (or better, automatically detected). This also brings new challenges on how arbitrary frameworks builders must work together to derive domain information.

Since the domain compiler is implemented as a participant of the Java compiler, only Java files are processed. Therefore many references to the domain cannot be touched. Inline languages (e.g. regular expressions or HQL queries), although part of the Java source, are not compiled as such. Resources other than Java source files may refer to domain information as well. Think of XML configuration files, used by many frameworks or build scripts. Processing such reference would require the domain compiler to be a separate Eclipse builder, a major change in the design.

From a modelling point of view two more distinct domain element types exist, which have been neglected from the domain meta-model. Factories create domain element instances and repositories retrieve domain elements. Such types are injected just like services are. Therefore a way of distinguishing the three types must be come up with.

It has been said that all framework specific logic was abstracted from the core component. This is true for logic that derives domain information. On the other hand the templates that generate or rewrite existing code are part of the core component and contain framework specific knowledge. This dependency should be removed.

In the Domain Editing Tools I found an extension to the Eclipse programming environment, especially suitable in an agile context. The plug-in makes it possible to quickly switch between a modeling and code centric perspective, allowing the programmer to choose the appropriate level of abstraction. Although much is achieved in this initial implementation, first the issues mentioned above must be overcome to enable domain modeling in the Java language harmoniously.

[1] `http://www.springframework.org/`

Bibliography

[1] A. Agrawal, G. Karsai, and A. Ledeczi. An end-to-end domain-driven software development framework. *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications OOPSLA '03*, October 2003.

[2] A. V. Aho and J. D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., 1972. (ISBN 0-13-914556-7).

[3] C. Bauer and G. King. *Hibernate in Action*. Manning, 2005. (ISBN 1932394-15-X).

[4] K. Beck. *eXtreme Programming Explained*. Addison-Wesley, 2000. (ISBN 201 61641-6).

[5] K.-F. Böhringer and F. N. Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people CHI '90*. ACM Press, March 1990.

[6] M. G. J. v. d. Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovationan annotated bibliography. *ACM SIGSOFT Software Engineering Notes*, 22(1):57–68, January 1997.

[7] M. Bravenboer, R. de Groot, and E. Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In *Participants Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE05)*, Braga, Portugal, 2005.

[8] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, Vancouver, Canada, October 2004.

[9] F. P. Brooks. *The Mythical Man-Month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[10] E. Buss and J. Henshaw. Design technologies: A software reverse engineering experience. In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research CASCON '91*, pages 55–73. IBM Press, 1991.

[11] A. Cavarra, E. Riccobene, and P. Scandurra. A framework to simulate UML models: Moving from a semiformal to a formal environment. In *Proceedings of the 2004 ACM symposium on Applied computing SAC '04*, pages 1519–1523, March 2004.

[12] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

[13] S. Counsell, Y. Hassoun, G. Loizou, and R. Najjar. Architecture and refactoring: Common refactorings, a dependency graph and some code smells: an empirical study of java oss. In *Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering ISESE '06*, pages 288–296. ACM Press, 2006.

[14] A. v. Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 2000.

[15] H. Eichelberger. All things UML: Nice class diagrams admit good design? In *Proceedings of the 2003 ACM symposium on Software visualization SoftVis '03*, pages 159–167. ACM Press, June 2003.

[16] M. Eiglsperger, M. Kaufmann, and M. Siebenhaller. All things UML: A topology-shape-metrics approach for the automatic layout of uml class diagrams. In *Proceedings of the 2003 ACM symposium on Software visualization SoftVis '03*. ACM Press, June 2003.

[17] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004. (ISBN 0 3211252 1 5).

[18] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. (ISBN 0201485672).

[19] M. Fowler. Generating code for DSLs. `www.martinfowler.com/articles/codeGenDsl.html`, 2005.

[20] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. *2007 Future of Software Engineering (FOSE '07)*, May 2007.

[21] S. Fraser, K. Beck, W. Cunningham, R. Crocker, M. Fowler, L. Rising, and L. Williams. Hacker or hero? - eXtreme programming today (panel session). *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum) OOPSLA '00*, pages 5–7, 2000.

[22] S. Freeman and N. Pryce. OOPSLA practitioner reports chair's welcome: Evolving an embedded domain-specific language in Java. *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications OOPSLA '06*, pages 855–865, 2006.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. (ISBN 0 2016336 1 2).

[24] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification*, third edition.

[25] D. Grune, H. E. Bal, C. J. Jacobs, , and K. G. Langendoen. *Modern Compiler Design*. John Wiley & Sons, Ltd., 2000. (ISBN 0 471 97697 0).

[26] T. R. Henry and S. E. Hudson. Interactive graph layout. In *Proceedings of the 4th annual ACM symposium on User interface software and technology UIST '91*, pages 55–64. ACM Press, October 1991.

[27] Hibernate.org, `http://www.hibernate.org/hib_docs/annotations/reference/en/pdf/hibernate_annotations.pdf`. *Hibernate Annotations Reference Guide*.

[28] Hibernate.org, `http://www.hibernate.org/hib_docs/v3/reference/en/pdf/hibernate_reference.pdf`. *Hibernate Reference Documentation*.

[29] J. E. Hopcroft and J. D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc, January 1969.

[30] W. Huang, S.-H. Hong, and P. Eades. How people read sociograms: a questionnaire study. In *Proceedings of the Asia Pacific symposium on Information visualisation - Volume 60 APVIS '06*, pages 199–206. Australian Computer Society, Inc., November 2006.

[31] M. M. Jaghoori, A. Movaghar, and M. Sirjani. Software verification (SV): Modere: the model-checking engine of Rebeca. In *Proceedings of the 2006 ACM symposium on Applied computing SAC '06*, pages 1810–1815. ACM Press, 2006.

[32] L. Moonen. Generating robust parsers using island grammars. In *Eighth Working Conference on Reverse Engineering (WCRE'01)*, pages 13–22. IEEE Computer Society, 2001.

[33] B. Moore, D. Dean, A. Gerber, G. Wagenknecht, and P. Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbook, February 2004.

[34] D. Nash and H. Willman. Software engineering applied to computer-aided design (CAD) software development. In *Proceedings of the 18th conference on Design automation DAC '81*, pages 530–539. IEEE Press, June 1981.

[35] R. F. Paige, P. J. Brooke, and J. S. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Trans. Softw. Eng. Methodol.*, 16(3), 2007.

[36] G. Premkumar and M. Potter. Adoption of computer aided software engineering (CASE) technology: an innovation adoption perspective. In *ACM SIGMIS Database*, volume 26. ACM Press, May 1995.

[37] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[38] Sun Microsystems. *JavaBeans™*, 1997.

[39] N. Synytskyy, J. R. Cordy, and T. R. Dean. Robust multilingual parsing using island grammars. *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research CASCON '03*, 2003.

[40] A. L. Tharp. The impact of fourth generation programming languages. *ACM SIGCSE Bulletin*, 16(2), June 1984.

[41] D. Thomas and B. M. Barry. Model driven development: the case for domain oriented programming. *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications OOPSLA '03*, pages 2–7, 2003.

[42] Wikipedia. Context-free grammar. `http://en.wikipedia.org/wiki/Context-free_grammar`.

[43] Wikipedia. Osgi. `http://en.wikipedia.org/wiki/OSGi`.

Appendix A

Working with the Domain Editing Tools

A.1 Prerequisites and Installation

Before you can install the Domain Editing Tools, please make sure you have the following prerequisites installed on your system:

1. Eclipse 3.3 (Or better)
2. Graphical Editing Framework, matching your Eclipse version.

Then the plug-in can be installed from the following update site:

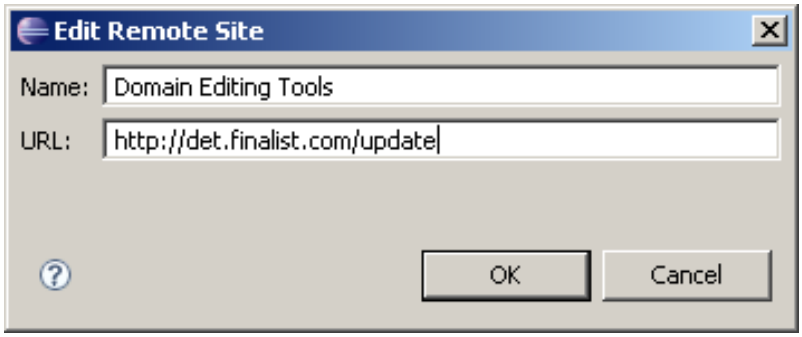


Figure A.1: Create a new update site.

A.2 Additional Information

When the plug-in is installed a manual is added to the Eclipse help system. Here the user can find information on how to open the model view and access the other functionality. Also

a developer manual is available explaining how to extend the system and in particular how to add builders and refactorings for new frameworks.