Dottorato di Ricerca in Informatica

Università di Bologna e Padova

Whole Platform

Riccardo Solmi

March 2005

Coordinatore:	Tutore:
Prof. Özalp Babaoğlu	Prof. Andrea Asperti

To my mother and my grandmother

Abstract

The Whole Platform is a technology for engineering the production of software. We think that programming is an activity concerning the development of languages; so, we provide an environment for developing new languages and tools in a much easier way than now.

Most languages are designed as if their feature set will never change. Adding a simple feature to a mainstream language such as Java requires a big effort which becomes huge for adding a set of inter-dependent features. For example the support of Java 5 in the Eclipse platform, has already taken 1.5 years of development.

Implementing and extending a programming language is more difficult today than few years ago because a language is perceived by programmers together with the facilities provided by a dedicated development environment such as code refactoring and content assistance.

We think that these difficulties have encouraged non linguistic approaches to extend languages. For instance, one of the most common non linguistic way to extend a language is represented by libraries which work well with standard compilers and tools but have the drawback that you loose the specific compiler optimizations and the specific support provided by tools.

We advocate that facilitating the implementation and the extension of languages and language processing tools, new domain languages and new extensions of the existing languages will arise as well as it has happened for XML based languages.

The thesis focus on the design and the implementation of a software platform for writing new languages and tools using a model driven technology.

Acknowledgments

My sincere gratitude goes to Professor Andrea Asperti who has supported my efforts for five years. I want to thank also Ferruccio Guidi, Luca Padovani and Enrico Persiani for their helpful discussions about the contents of this thesis.
A special thanks goes to Professor Joost Visser for his precious suggestions on how to improve this document.
Finally, a thought to a friend, Tommaso Borromei, who has always believed in my research projects.

Contents

List of Figures

Chapter 1

Introduction

Programming languages, like human languages, are evolving. The last version of two mainstream languages: Java[36] and C#[41] includes new important features. Notice that usually a feature gets discovered and experimented on research languages before being added to a mainstream language (if ever).

The evolution of a programming language is a slow process controlled by an authority responsible for evolving the standard of the language. When a language stops to evolve, it becomes a *legacy language*; it is still used in production code and thus other languages need to interface to it for some time.

Language design and evolution is known to be difficult.

Adding a feature to a language is difficult largely because every feature tends to affect every other feature, and so the complexity of evolving a language grows exponentially with the number of features.

If you try to add even a non interfering simple feature to an existing mainstream language, you realize that you are facing engineering problems of existing languages and tools. Mainstream languages are difficult to evolve for lack of modularity in the design of the language and language tools. They are designed with a fixed language scenario in mind; so, according to Simonyi[63], we call also them *legacy languages* even while they are still evolving.

The problem is even worse because a language is perceived by programmers together with the tools for manipulating it (editor, compiler, runtime). So, adding a feature to a

mainstream language requires also adding tool support for that feature; otherwise you are reducing the language perceived and not extending it, because the programmer has to loose the facilities of the tools for using the new feature.

The tools commonly used by a programmer are collected in a single environment called Integrated Development Environment (IDE). IDEs are evolving much faster than the languages they support. Most popular IDEs such as Eclipse[2] and NetBeans[11] are extensible and language neutral. They provide extended functionalities for *supported* languages such as refactoring[31] and content assistance. Not surprisingly, the language support is highly coupled with the features of the supported language; in fact a new version of the language requires a new version of the IDE. The features contributed by an IDE are not considered part of the language specification today but they play an important role in determining the success of a language and the adoption of new language features.

The problem of adding features can be regarded from a more general perspective. Features can be packaged not only as languages (linguistic constructs) but also as libraries. A significant difference between these two forms of packaging is that language-based features can be fully supported by IDEs and processed by tools while library-based features can not have an analogous support because it is difficult to distinguish them from ordinary code.

So, adding a feature directly to a language is difficult but all the benefits of tools can be achieved, on the other hand trying to bypass the problem using libraries means to loose the specific support that tools can provide for linguistic features.

In this thesis we try to exploit all the expressive power of features facilitating their integration in languages and in tools.

1.1 Modern programming languages

The most popular programming languages are general purpose and they are used to write almost all types of applications. Today, knowing a programming language means not only to be able to program with its constructs (features) but also to know many standard libraries written for the language. These libraries, today, are considered part of the language while historically, they were considered part of the operating systems. The

Java[36] platform, for example, has native support for concurrency, memory management, networking, GUI and runtime, and contains also a rich set of standard libraries that facilitate the writing of different types of applications.

We can even say that the competition between languages is almost focalized on libraries and programming environments rather than on language constructs. This is forced by the fact that the feature set of these languages is fixed by the language designer and so the libraries are the principal way for extending the language.

1.2 Domain knowledge representation

Writing an application means to code part of the knowledge you have in the application domain. The domain knowledge can be coded with a *domain specific language*, i.e. a language including first class citizen for representing concepts and processes of that domain. But, this approach today is advocated only by a restricted number of researchers and few research projects [25][64][38][26][22] that are not mainstream. Nonetheless, in our opinion, many domain-specific languages are in use today hidden behind less straightforward representation forms: libraries, XML dialects, annotations and domain editors. Let us consider each of them in turn.

1.2.1 Libraries

Languages with a fixed feature set, as we have seen, can encapsulate domain knowledge only in libraries. A library can be written and used using the standard compiler, runtime and debugging tools available for the language. The constructs of a general purpose language are of course sufficient for expressing domain abstractions at the semantics level.

A library exposes an Application Programmer's Interface (API) that gives access to the domain abstractions hiding implementation details. Unfortunately, libraries are not expressed in terms of domain concepts but they are defined using lower level general-purpose abstractions such as classes and methods; so programmers are forced to use the generic notation of method invocation rather than a more specific domain notation.

The fixed nature of most popular language implementations is of course due to the higher complexity of designing and implementing an extensible language but it is mainly

a design choice. Having a fixed syntax is supposed to facilitate understanding of programs and learning of new libraries. Unfortunately the understanding you get for free is limited to the ability to interpret the program step by step much like a computer.

Learning libraries is not a simple task either if you are an expert in the domain because you have to learn the mapping between the library API and the modeled domain. Even after learning, using the library is verbose and the development environment can not facilitate or enforce a correct use.

1.2.2 XML dialects

Using a general purpose language and representing all domain knowledge with libraries is no longer sufficient for writing a modern enterprise application (see for example the J2EE[6] standard). Several domain specific languages mainly packaged as XML dialects have to be used. Whenever is not required the encapsulation (mixing) of domain knowledge in code written with a general purpose language, we can choose to use an XML dialect. The XML[75] notation, when used for representing a domain language, gives to programmers a more concrete syntax feeling than a library because it fixes only the structure of the syntax, permitting the definition of so called tags libraries.

XML shows us another important fact: if the mixing of languages is practical it is used and considered useful. Many XML dialects are designed to work together in a single document (see for instance XSLT[72] and JSF[8]).

1.2.3 Annotations

A third emerging approach consists in using user-defined annotations written on certain places of the source code to specify additional information. The annotations can be retrieved both at compile time and at run time through language reflection. Both C# and Java have just introduced support for annotations, calling them respectively *attributes* and *metadata*.

A language supporting annotations permits a more declarative programming style and gives an even more concrete domain syntax feeling with respect to libraries and XML. But annotations represent a constrained linguistic extension of an host language; so they are suitable only for writing a few types of domain specific languages.

1.2.4 Domain editors

The solutions seen so far can be improved introducing domain editors supporting a concrete syntax. A *domain editor* is an editor which complements the main editor when you have to write code belonging to its particular domain. Domain editors are usually based on graphical notations while the main one is textual. The domain editors improve only the presentation of the code, in fact they translate whatever you write in the language below.

For example the Graphical User Interface (GUI) construction tool VEP[14] allows to edit a user interface manipulating it graphically, this representation is translated in ordinary calls to the standard Java Swing library.

Developing domain editors is difficult because the domain representation has to be extracted from code interleaved with other unrelated code. One of the main challenge designing a GUI construction tool is synchronizing the graphical notation with the library calls in the source code; this task requires reverse engineering techniques and often imposes some restrictions to developers.

1.3 Goals

The state of the art depicted above shows that the need to extend languages is provided now by libraries, the use of XML dialects, source code annotations and domain editors; none of them is completely satisfactory.

In this thesis a tool - the Whole Platform (pronounced as wool) - has been designed and implemented that:

- supports rapid definition of families of integrated languages
- covers complete languages: concrete syntax (notation), abstract syntax (metamodel) and semantics (translation)
- has built in support for sharing, versioning and staging abstractions
- has a persistence layer that subsumes the source role
- is layered and extensible

The tool is able to generate language processing tools from the definition of languages expressed as metamodels. The implementation consists of a generative system (Whole Generative System), an Eclipse[2] based development environment (Whole IDE) and a meaningful library of languages (Whole Languages).

The tool is an improvement with respect to all the current practices depicted above. In fact it allows:

- to define new languages without the syntax structure of the XML dialects (i.e. the markup),
- to add new features to existing languages by extending the syntax in an unrestricted way without the constraints imposed by annotations,
- to introduce new rich domain editors for languages using multiple notations,
- to replace libraries with languages having a concrete syntax.

1.4 Outline

The thesis is divided into three parts.

In the first part, the architecture of the Whole Platform is defined, the design choices are motivated and a comparison is made with related architectures. The Whole Platform includes a visual programming environment, a generative system and a family of languages.

In the second part, the Whole Generative System is described. It consists of a set of frameworks for modeling the structure, the behavior, the persistence and the editing of a language. The frameworks support the implementation of a family of model-driven languages and of the tools to manipulate them; they are designed to support languages and operations obtained by composition.

In the third part, we introduce the family of languages implemented on top of the Whole Generative System. The Whole Languages includes popular languages like Java, XML, EBNF and plain Text together with some new metamodeling languages used to define and extend the Whole family of languages itself.

Part I

The Whole Platform Architecture

In this part the architecture of the Whole Platform is defined.

The Whole Platform is a software platform including a visual programming environment, a generative system and a family of languages. The programming environment (Whole IDE) consists of a set of plugins for the Eclipse[2] platform supporting rich graphical editing of model driven languages. The generative system provides the infrastructure for implementing model driven languages and for giving them a translational semantics using advanced model to model generative technologies. The family of languages includes popular languages and some new metamodeling languages used to define and extend the family itself.

In chapter 2 the design choices are outlined and motivated. In particular the main characteristics of language processing technologies and tools are considered. In the following chapter 3 the overall architecture is presented. Finally, in chapter 4 the Whole architecture is compared with some related architectures.

Chapter 2

Guidelines for Architecture and Design

In this chapter the main characteristics of language processing technologies and tools are considered. For each characteristic a design choice is outlined and motivated.

2.1 Abandoning the primacy of source code

Adaptiveness requires the ability to change, but only a running system can change itself. Documents with their computer counterparts - the files - contain information, they are useful to store knowledge and to transmit it between systems but they do not evolve in their own; a person or a running system is required to change them. Adaptiveness is a property of life, documents are lifeless.

In a traditional programming language the developer writes the source code of an application and uses the compiler to produce an executable running in some runtime environment. Sources are required to modify the program and developers continue to change them to fix problems and to introduce new functionalities. Source modification is a responsibility of developers.

All non trivial language processing tools have to build an internal representation of the sources to accomplish their job. However, they usually do not persist that representation; instead, they build it on the fly by extracting information from sources, use it, and then discard it when the job finish.

An example is the refactoring operations supported by modern IDEs. A *refactoring*[31] is a transformation from source to source that improves the code without changing its be-

havior. Current implementations of refactorings build the internal representation whenever the operation is requested and then discard it.

All the available programming languages known by the author share a common perspective: they are built around the source code. According to the idea introduced in [64], we think that the perspective should be reversed: the internal representation should be persisted and regarded as the program. Source code should be derived from it and all kinds of tools should operate directly on it.

In this way source views of the code limited and specific to the parts in which the programmers is interested in can be derived. Tool writers are also facilitated because the internal representation is given and they have to code only the specific behavior of the tool.

2.2 Model-driven development

A *model* is an abstract description of software that hides information about some aspects of the software to present a simpler description of other aspects. All internal representations built by applications and tools are models. We use models as a source code replacement.

We want to automate software development so we restrict ourselves to models that capture information in a form that can be interpreted by humans and processed by tools. This restriction avoids models that can be used only as documentation intended for human consumption.

A model describes a software system from a specific perspective; usually, it describes an aspectual part of the system. This means that multiple models are needed to completely describe a software system. We require a model to be a well-formed unit as defined by the language in which it is expressed.

2.2.1 Programming is modeling

There are no well-established definitions for the terms programming languages and modeling languages, the differences between them are mainly historical. Most programming languages have textual notations and are primarily imperative, while most modeling

languages (like UML[50] or GME[43]) have graphical notations and are primarily declarative. However, an increasing number of languages (like Emfatic[3] or Visula[15]) crosses these distinctions so we cannot take the notation or the style of specification as reliable basis for distinguishing between modeling and programming.

Recent trends [38][24] suggest that the distinction between programming and modeling languages may soon become entirely irrelevant.

We want to build a tool which does not force any distinction between features coming from programming or modeling languages. For instance we want to allow to mix graphical and textual notations; so you are free to compose features on languages and to select a notation for them.

We allow to attach a semantics to every fragment of code written in any supported language; so, depending on the attached semantics, a code fragment plays the role of a program or of a model. Given a code fragment, you get a program providing an executable semantics for it, while you get a model providing a translational semantics that uses the code as director (specification).

2.3 Anatomy of languages

To distinguish computer languages from human languages the formers are commonly called formal languages. A *formal language* includes a semantics, an abstract syntax, and one or more concrete or serialization syntaxes as depicted in Figure 2.1.

The semantics defines the meaning of the language.

The abstract syntax defines elements and elements composition rules.

The concrete syntax defines a human usable notation.

The serialization syntax defines persistent and interchangeable forms of the language.

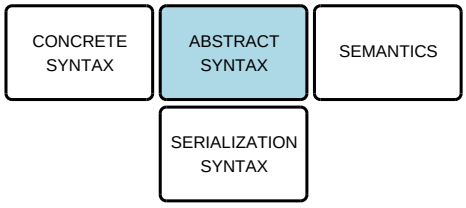

Figure 2.1: Formal language anatomy

In the following subsections we consider each part in turn.

2.3.1 Metamodels abstract syntax

The distinctive aspect of a language is its abstract syntax. There are different ways in which the abstract syntax can be defined; the most popular are context free grammars and metamodels. Context-free grammars are traditionally used to define the abstract syntax of textual languages. Since the introduction of UML [50] and other graphical modeling languages, the metamodels have been used as a valid alternative to define the abstract syntax.

Metamodels usually are object-oriented and characterize language elements as classes and relationships between them using attributes and associations.

Metamodels are at least as expressive as BNFs [38]. A BNF language specification can be translated into a corresponding metamodel specification. In general, non-terminals become classes; production rules become ownership associations and the multiplicity of an association corresponds directly to BNF quantifiers.

A metamodel represents an abstract syntax graph (ASG) because it has ownership and reference links. The ownership links, by themselves, form a tree; taking in account also the reference links you have a graph.

Metamodels have some advantages with respect to CFGs. A metamodel is more intentional than a CFG because the rule alternatives and the links are named. A metamodel specifies a family of object graphs instead of a family of trees specified by CFG.

2.3.2 Model to model translational semantics

We think that translation is usually the most effective way to give semantics to a language; so we consider here only the translational approach to defining the semantics of a language. The idea is that we can translate one language into another that has semantics. A language might be a directly executable language, such as a (virtual) machine instruction language, or, it might be a language that can be translated, with successive translation steps, into a directly executable language.

A traditional language processor (e.g. a compiler) performs two additional tasks: parsing and unparsing. *Parsing* is the process of constructing an abstract syntax representation (e.g. a tree) from a concrete syntax notation (e.g source code); *unparsing* is the reverse process: constructing a concrete notation starting from an abstract representation.

A traditional language processor performs a parsing of the source notation, then it performs a syntax-directed translation into a target language abstract representation, and then emits (unparses) it. The translation may or may not use an abstract representation of the target language; that is, the unparsing process is unnecessary for simple translations.

We are considering model-driven languages, so our abstract syntax representation is a model. We choose to perform always a model to model translation. Given that the abstract syntaxes of all languages are defined as metamodels, then we can also define the translation as a metamodel [17].

With respect to a traditional language processor this approach leads to more reusable translators because this way they are composeable and the parsing and unparsing processes are done only at the ends of the translation chain and only if the entire process has been requested for human consumption.

2.3.3 Specialized concrete and serialization syntaxes

The *concrete syntax* of a language defines a notation intended for human reading and editing. The *serialization syntax* defines a persistent and interchangeable form of the language intended for processing by tools.

A language may have several concrete or serialization syntaxes but, usually, most languages having a textual notation use it even for serialization purposes (e.g. Java[36]) while most languages having a graphical notation (e.g. UML[50]) have a different form for serialization (e.g. XMI[54]).

The two complementary processes of converting a concrete or a serialization syntax to and from an abstract syntax are called *parsing/unparsing* and *marshalling/unmarshalling* respectively. These processes are well known and there are tools for generating parsers and to add persistence to a language (with or without code generation). Unfortunately, the most popular tools [44][57] are not model-driven and they fail to satisfy the additional requirement of mixing multiple languages in an unanticipated way.

For instance, the most popular parser generator tools[44][57] are grammar-driven and are limited to a subclass of context free languages open for composition. The serialization syntax is much less constrained so we can define it to make straightforward the marshalling/unmarshalling processes even for multiple mixed languages.

We choose to always have two different specialized syntaxes for human interaction and for tools processing (persistence). So we can relegate the parsing/unparsing processes to a *legacy languages exchange facility*: the language processing system should be able to *import* existing programs written with legacy languages and to re-*export* them if requested. In general, we expect that visualisation and editing of a concrete syntax will be performed within a model-driven editor and that the persistence of the programs is a responsibility of the language processing system and is always performed using a serialization syntax.

The serialization syntax can be specialized for persisting models. We need a way to handle multiple models and to partition a model in smaller models retaining the ability to define references between them.

Mixing multiple language notations

Most programming languages have historically used textual notations and most modeling languages have used graphical notations. Neither one is a requirement. For this historical reason, editing tools for programming languages are usually based on textual widgets and cannot visualize graphical notations, while editing tools for modeling languages are usually based on graphical widgets and cannot visualize textual notations. The former limitation is inherent to textual widgets, the latter is only a choice motivated mainly by the difficulty of implementation.

All languages can be visualized using a wide variety of notations including textual, schematic (UML) and widget-based notations (tables, trees). Nevertheless, most languages have a default standard notation.

There are several interesting scenarios that require the embedding of a language within another language. For instance, in a metaprogramming scenario a meta language embeds a possibly different template language. The two languages may have a different preferred notation or may have the same kind of notation but different validation criteria and preferred presentation styles (i.e. syntax coloring). We are faced with the problem of visualizing them mixing multiple notations.

We want to write an editing tool that supports all kinds of notations and permits to mix them freely even within a single view. Such an editing tool will permit us to write

programs using different languages each visualized using its own preferred notation and interactive behavior.

2.4 Domain specificity

We prefer to distinguish languages focusing on specificity of the abstractions they include. According to [25] languages ranges from *domain specific languages* (DSL) to *general purpose languages* (GPL). A general purpose language encodes mainly generic abstractions; while a DSL defines abstractions that encode the vocabulary of a specific domain.

To solve a software problem we have to design a conceptual model of the solution first; then we choose one (or more) implementation languages for writing the solution. If we choose a general purpose language, the gap between the conceptual model and the implementation code is large enough to force us encoding many programmer intentions in a very indirect manner. Using a well designed DSL the mapping is much more straightforward because, by definition, the available abstractions are in the domain of the problem.

To become a systematic approach to programming we have to make the development of new languages, in particular DSL, in top of already defined languages and tools much easier than now.

2.5 Families of languages

The *one-off development* process is organized around an individual product. Today, near all programming languages are the result of one-off development.

The process of defining a language has been well engineered; there are tools to automate parts of the development process. Unfortunately, traditional language definition tools such as parser generators [44][57] and code generators [9][13] are oriented to the definition of a single standalone language.

Extending an existing language is a difficult task; integrating the extension in a existing state of the art development environment is even worse. Take as example one of the few success stories: the development of the AspectJ[40] programming language. It

is an aspect oriented[39] extension of Java and it has development tools (AJDT) well integrated with the Java development tools (JDT) of the Eclipse platform[2]. The AspectJ tools require continuous development to be kept in sync with the evolving Java tools.

The need for extending a language is not limited to the research community. The definitions of software products, already contain large amounts of code and metadata expressed in DSLs different from the main base language.

An alternative process consists on developing a software *product line*[25]. Product lines are not a new idea, they form the basis of modern industry, but applying this idea to produce software products is a very recent trend[38].

In this perspective, it is necessary to introduce a domain specific language for defining new languages and to regard a language definition as a component so that new languages can be assembled from existing language components.

We want to define an architecture for building families of languages.

Chapter 3

The Whole Architecture

Whole is a software platform that runs on top of other platforms.

The Whole Platform includes a graphical development environment (Whole IDE), a family of languages (Whole Languages) and a generative system (Whole Generative System).

The Whole architecture enforces a model-driven perspective built around the Whole Generative System. All software artifacts including program sources are persisted and managed by the Whole Generative System. The Whole Languages are mainly conceived for human-system communication; they are used by developers to write source programs but the code is not stored in developer managed artifacts. The Whole Generative System is responsible for maintaining all the programs knowledge and is able to make source representations suitable for reviewing and editing within the Whole IDE.

In this chapter we follow a top down presentation order reflecting a user perspective. The other parts of the thesis follow a bottom up order reflecting a developer perspective.

3.1 The Whole IDE

The Whole IDE is an interactive development environment for programming with the Whole family of languages. The Whole IDE consists of the Eclipse platform extended with some plug-ins as depicted in Figure 3.1.

Eclipse[2] is a popular, open source platform for building rich clients and development environment tools. The Eclipse platform is structured around the concept of plug-ins. A *plug-in* is a bundle of code and/or data that contribute libraries, extensions or

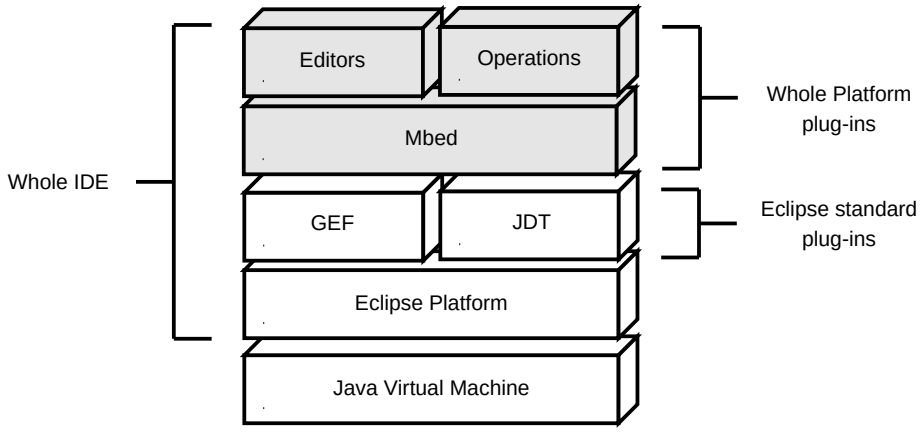

Figure 3.1: The Whole IDE architecture

documentation to the platform. The platform and the plug-ins have well-defined places called *extension points* where other plug-ins can add functionality.

The Java development tools (JDT) is the Eclipse standard implementation of a full featured Java development environment.

The Graphical Editor Framework (GEF)[4] is an Eclipse Tools project allowing developers to create a rich graphical editor. It follows the MVC architectural pattern: it takes an existing model and provides many facilities related to the controller and view parts.

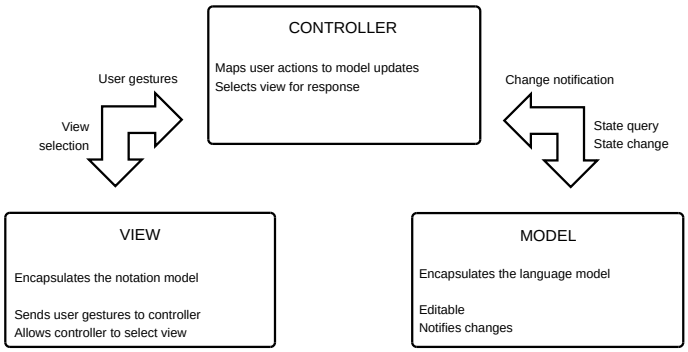

Figure 3.2: Model View Controller architectural pattern

The JDT exposes both the Java compiler API and the API of the abstract syntax tree (DOM/AST) used for representing Java compilation units. Exposing a Java AST, the front-end and the back-end parts of the JDT Java compiler can be used separately. We use the JDT for providing our Java model the ability to import Java source code and to

produce both Java source code and Java compiled code (bytecode).

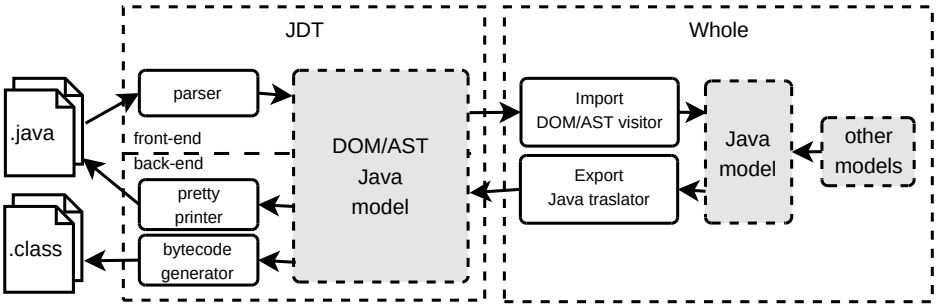

Figure 3.3: Java compilation interaction diagram

In Figure 3.3 you can see the interactions involved. Java source code is parsed by the JDT resulting in an AST that is translated into our Java model. In the reverse direction our Java model is translated to the JDT DOM/AST and from here using the back-end of the compiler we can pretty print the code or compile it.

3.1.1 Mbed Plug-in

The *Model Based Editor* (Mbed) is the implementation of the Whole Generative System packaged as a standard Eclipse plug-in. Mbed defines a few extension points for adding to it models, editors and operations.

For instance, the `org.whole.mbed.editors` extension point allows clients to contribute custom editors. Each editor is a complete language (model, notation, semantics) which will be added to the family of Whole Languages.

The following is an example of the editor extension point usage:

```
<extension point="org.whole.mbed.editors">
  <editor
    name="Mbed DSL Editor Kit"
    id="org.whole.lang.mbed.MbedEditorKit"
    class="org.whole.lang.mbed.MbedEditorKit"
    extensions="mbed">
  </editor>
</extension>
```

3.2 The Whole Languages

The Whole Languages is the extensible family of languages available to program the Whole Platform. The Figure 3.4 shows a Whole language with all its collaborations. Each language has an in-memory model representation encapsulated in a model context that disciplines its construction and manipulation.

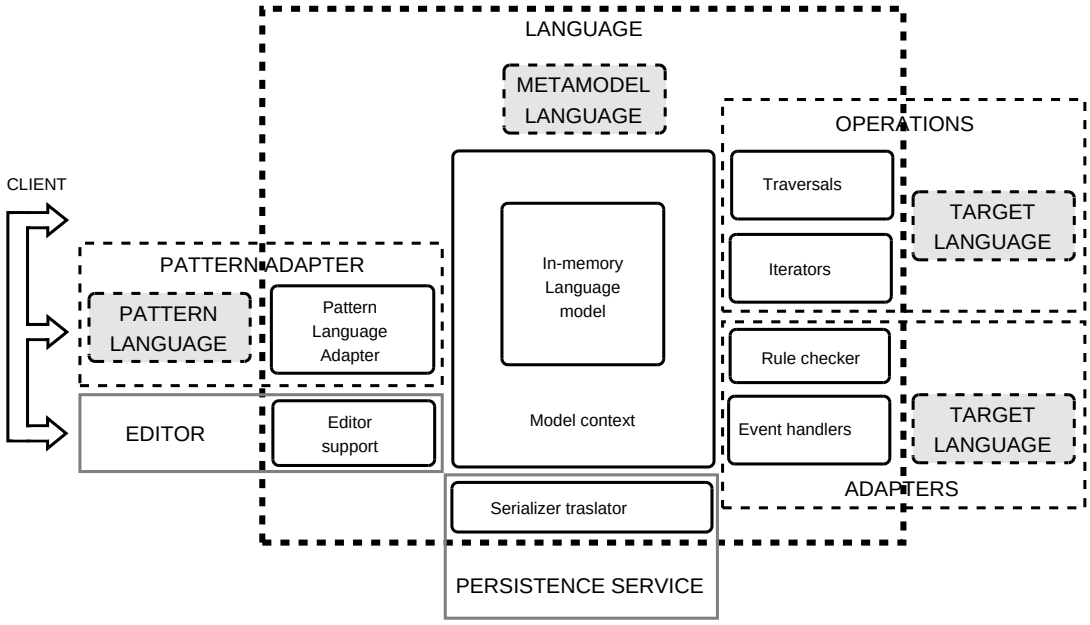

Figure 3.4: Whole Language collaboration diagram

The structure of a language is defined using a metamodel language; the language implementation is generated from its metamodel definition. Furthermore, at runtime, you can navigate the metamodel hierachy as long as the metamodel languages are deployed together with the base level language.

Programs written in a given language can be persisted. The persistence service is able to transform a model to(from) a stream of builder events and from there to(from) a serialization syntax. Notice that serialization syntaxes are pluggable.

The behavior of a language is defined using operations and adapters. An operation defines a batch behavior using traversals or iterators. An adapter defines a live (continuous) behavior using event handlers. With an operation you can define, for instance, a model to model transformation to a target language; while with an adapter you can define a live synchronization between two or more models where the other models act as

target languages or pattern languages. The name adapter suggests that the synchronized languages can be used as alternatives for programming the adapted language.

The ability to edit a language program from within an editor in the Eclipse platform is added using a mix of operations and adapters that define one or more concrete syntaxes and the desired interactive behavior.

Notice that the in-memory representation of the model of a language is constructed only if required by a model operation; furthermore, all given operations are defined to minimize this requirement. For instance, you can write a code generator for XML and Java that fires a stream of SAX events and builds a DOM/AST Java Model *without* constructing neither our XML model nor our Java model as intermediate steps.

3.3 The Whole Generative System

The Whole Generative System is a set of coordinated frameworks including (see Figure 3.5): Modeling, Traversal, Notification, Persistence, Java Model Generation and Editing.

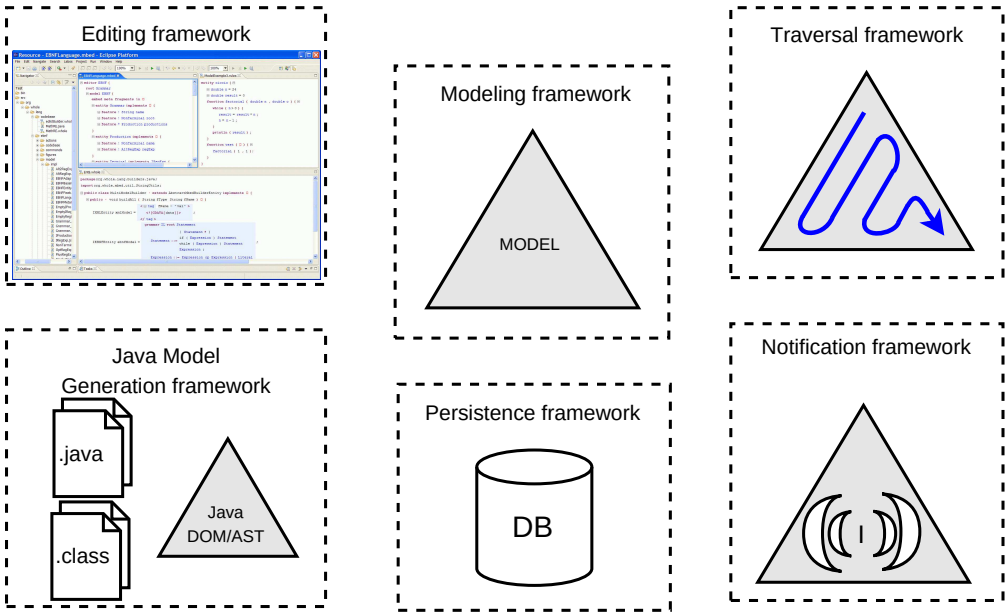

Figure 3.5: The Whole Generative System frameworks

The Modeling framework supports the definition and the manipulation of the struc-

ture of models. The Traversal framework supports batch model operations based on a traversal of the model. The Notification framework supports live model operations triggered by state changes of the model. The Persistence framework is a persistence and metaprogramming facility for generating and persisting models. The Editing framework supports the creation of rich graphical editors for model based languages. The Java Model Generation Framework is a bootstrap code generation facility for building models of Java conforming to the JDT DOM/AST API [7] standard in Eclipse.

The frameworks have been designed together as a system of frameworks so that they achieve a good level of coordination and optimization; in particular, each framework exposes an API that facilitates inter frameworks collaborations.

In the Whole Generative System, frameworks are designed using a generative approach in mind. Each framework (Figure 3.6) addresses an aspect of the system and exposes two kinds of API: a generative API and an infrastructure API. All commonalities are packaged in the Infrastructure API that is much like an ordinary library, while specific functionalities are generated on top of the Generative API.

The patterns used in the infrastructure are designed to be extended and partially overridden with generative design patterns [45]. Many functionalities of the platform are implemented both in a generic way and, using generative design patterns, in a model-specific way.

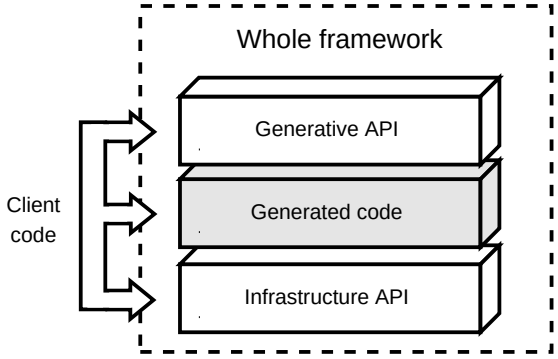

Figure 3.6: A Whole generative framework

The Whole Generative System is deployed without the Editing and the Java Model Generation frameworks (that are deployed together with the Whole IDE), this way it is independent from the Eclipse platform.

Chapter 4

Related Architectures

4.1 Conventional development environments

Conventional IDEs such as Eclipse[2] and NetBeans[11] have built-in support only for textual programming languages. They integrate tools for compiling, debugging, managing projects and, of course, editing code. Recent trends show major improvements in editing support and integration with external tools.

Editing support includes: navigation in the type hierarchy and in the def-use hierarchy, code creation based on wizards and templates, content assistance showing all possible completions, automatic management of import statements and code refactoring[31]. Unfortunately, all this sophisticated editing support is available only for a fixed number of languages built-in the development environments.

The integration with external tools has adopted an inversion of control pattern. It is no more the environment that calls external tools; all IDE services are packaged as reusable components and the environment can be programmatically used as a framework for building development environments. With the public programming interface API the tool writer is able to extend the environment with new languages and new features for existing languages.

This last innovation permits us to write our development environment as an extension of an existing widely used and supported IDE, the Eclipse platform.

4.2 Metamodeling frameworks

In this section we consider tools that use a metamodel for language definition and use it to implement the language tools. These tools are often called *meta-tools*.

4.2.1 Model Driven Architecture (MDA)

The Model Driven Architecture[49] is an open, vendor-neutral suite of standards adopted by the Object Management Group (OMG). MDA aims to enable: platform-independent specification of a software system, platform specification and transformation from platform independent models (PIM) to a platform specific models (PSM). It includes (between others) three standards for modeling (MOF), notation (UML), and persistence (XMI).

The support for model query, views and transformations has been submitted but is not adopted yet. The first implementations of the overall architecture are expected next year (2005).

Meta Object Facility (MOF 2)

The Meta Object Facility (MOF)[55] provides a metadata management framework, and a set of services to enable interchange and manipulation of metadata.

MOF can be used to define and integrate a family of metamodels using simple class modeling concepts.

Unified Modeling Language (UML 2)

The Unified Modeling Language (UML)[52][51][53] is a visual language for specifying, constructing and documenting the artifacts of a system.

XML Metadata Interchange (XMI 2)

The XML Metadata Interchange (XMI)[54] is an XML based serialization syntax that facilitates the standardized interchange of models and metadata.

4.2.2 Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework[1] is a modeling framework and code generation facility for building software systems based on a structured data model. Models can be specified in XMI or using annotated Java. Given a model specification, EMF is able to produce a set of Java classes for the model and related services (including a basic editor).

The code generator facility is a traditional template language (Jet) not a model to model transformation tool.

4.2.3 The ASF+SDF MetaEnvironment

The ASF+SDF MetaEnvironment[65] is an interactive development environment that can be used to describe the syntax and semantics of programming languages and to describe analysis and transformation of programs written in such programming languages.

The system is able to parse arbitrary context-free grammars.

4.3 Meta-programming with concrete syntax

The MetaBorg[10] system provides concrete syntax for domain abstractions allowing programmers to embed domain specific languages in a general purpose host language. The embedded domain code is assimilated into the surrounding host language translating the domain abstractions in terms of existing APIs leaving the host language undisturbed (see [22] for details).

4.4 Meta-modeling architectures

This section outlines forthcoming products that follow much of the architecture we propose. All four systems are developed by vendors and so only limited technical information is publicly available.

4.4.1 Software Factories

Software Factories[38] are a rapid and flexible way to build systems using domain-specific tools, methods and content. They capture knowledge of business domains, platform technologies and software architectures and permit to customize development tools.

Software Factories is a methodology developed at Microsoft that aims to significantly increase the level of automation in application development. Actual technology preview (December 2004 release) of a Software Factory development environment is at a very early stage of development. There is only a graphical editor for defining the structure of a model and a traditional text based code generator.

4.4.2 Meta-Programming System (MPS)

JetBrains calls this technology approach *Language Oriented Programming*[26]. There is already a prototype plugin for IntelliJ IDEA which *will* allow you to include MPS concept models in your project. The models *will* automatically be translated into Java source code in the background as you edit them. So, you *will* be able to write part of your Java applications using MPS, as much or as little as you want. This means that you get all the power of MPS, such as the ability to create and use specialized DSLs, to make whatever language extensions you want, as well as to use customizable editors with code completion, error highlighting, refactoring, etc. The plugin *will* be tightly integrated with IDEA, allowing you to embed Java code in your MPS models, navigate to embedded or generated Java code, and even perform concept-level debugging similar to the JSP debugging support already available in IDEA.

Actual technology preview (November 2004 release) is a standalone graphical text editing environment for defining the structure of a model; no code generators are available.

4.4.3 Intentional Programming

The Intentional Programming[25] is an extensible programming and metaprogramming environment. Any part of the system can be extended. Each language (called active source) provides its own editing, rendering, compiling, debugging and versioning behavior. A language defines a set of abstractions (called intentions) and can be composed

with others. The compilation framework defines protocols for coordinating the compilation of code using independently developed language extensions.

Our architecture appears similar to the vision behind this technology. Charles Simonyi started working on Intentional Programming in the 90s when he worked at Microsoft Research; in 2002 it has co-founded the Intentional Software company.

No products or technology previews are publicly available so it is unknown whether they have a working system, but we can speculate that very likely they have one.

4.4.4 HyperSenses

Delta Software technology is implementing the core ideas of Intentional Programming as a new technology: HyperSenses[5]. Source code is rendered using different representations both textual and graphicals. The persistence layer is model driven and bassed on open standards such as MOF[55] and XMI[54].

No products or technology previews are publicly available so it is unknown whether they have a working system.

Part II

The Whole Generative System

The Whole Generative System is a system of frameworks supporting the implementation of a family of model-driven languages and of language processing and editing tools. Languages are implemented with a model-driven approach, so all facilities provided by frameworks are built around the models.

The Whole Generative System includes the following frameworks:

Modeling For building and manipulating models

Traversal For adding traversal-driven behavior to models

Notification For adding event-driven behavior to models

Persistence For adding a persistence service to models

Editing For adding concrete syntax and graphical editing to models

The different frameworks have been designed together to form a system. Clients can use directly the System but the frameworks are designed to be the target infrastructure in a generative scenario. The third part of the thesis will introduce a set of languages built on top of the Whole Generative System.

The structure (model) and the semantics (translators) of the languages have a meta-circular definition. Building a system with circular dependencies has required us to pass through several development cycles. To simplify the bootstrapping of translators (code generators) we have implemented one last framework: the Java Model Generation framework.

This part of the thesis is organized one chapter for each framework.

Chapter 5

The Modeling framework

5.1 Introduction

The Modeling framework supports the definition and the manipulation of the structure of models. A model is a first-class citizen, that is, models and model fragments can be passed as arguments to functions, be assigned to variables and so on.

The Modeling framework defines the structure of the models and provides the following sets of APIs:

Creational API For building entities and fragments

Modeling API Model structure definition, editing and one step traversals

Metadata API For accessing the model type system

Model Context API For supporting higher level services

5.2 The model

The model structure is represented by a typed object hierarchy of entities. The design of the model is a compound pattern[61][68] of Composite, Visitor, Iterator, Observer and Command roughly equivalent for functionalities to the *Tooled Composite*[69] design pattern.

The modeling framework defines an interface: `IEntity` common to all modeled entities. The framework includes several abstract and concrete implementations as depicted in Figure 5.1.

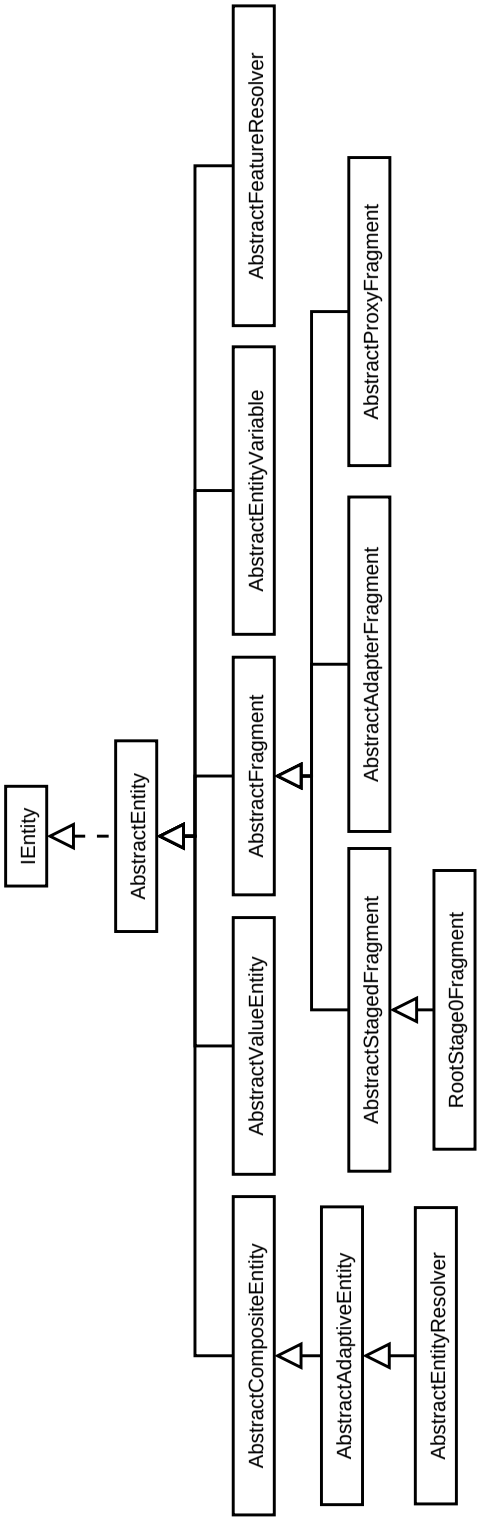

Figure 5.1: Class diagram of model entities

We have defined an abstract implementation for each kind of model entity: simple, composite, value, adaptive, resolver, and a few kinds of fragments. A simple *entity* has a fixed set of named structural features. A *composite entity* has a variable set of structural features. A *value entity* has a single value type feature. An *adaptive entity* has a variable set of named structural features. A *resolver entity* is an adaptive entity with the feature set constrained by a given model type. A *fragment entity* is used to assemble different fragments into a compound model. One abstract implementation is provided for controlling staging, another one is provided for assembling different models and a third one is provided for building a fragment proxy. More details on each kind of entity will be given on the following subsections.

5.2.1 Compound models support

We call *compound model* a model embedding other models with possibly different class hierarchies of entities. Each embedded model is called *model fragment*; for uniformity also the root model is referred to as a fragment.

Model embedding is required both for behavior assimilation and for staged execution. The model behavior can be defined both in a generic way and in a specific per model way. With *assimilation* we mean that we provide a mechanism for composing model specific behavior definitions. With *staging* we mean that each model fragment can be annotated with an execution stage. All operations are executed on each model fragment according to their annotated stage; for instance a code generator program uses the staging facility for separating the base level code from the template code.

To support compound models, a few design choices are possible including the use of generic adapters, specific adapters or no adapters at all. To embed a model within another model we use specific fragment adapter entities (see Figure 5.2). The default semantics associated to a fragment entity is a switch to the behavior specific to the embedded fragment. This way, a model specific behavior can be defined without taking in account other fragments, unless desired.

To facilitate the definition of behavior and state common to a fragment or the whole compound model, we provide additional objects; the compound model has an object representing it, and each model fragment also has a corresponding object.

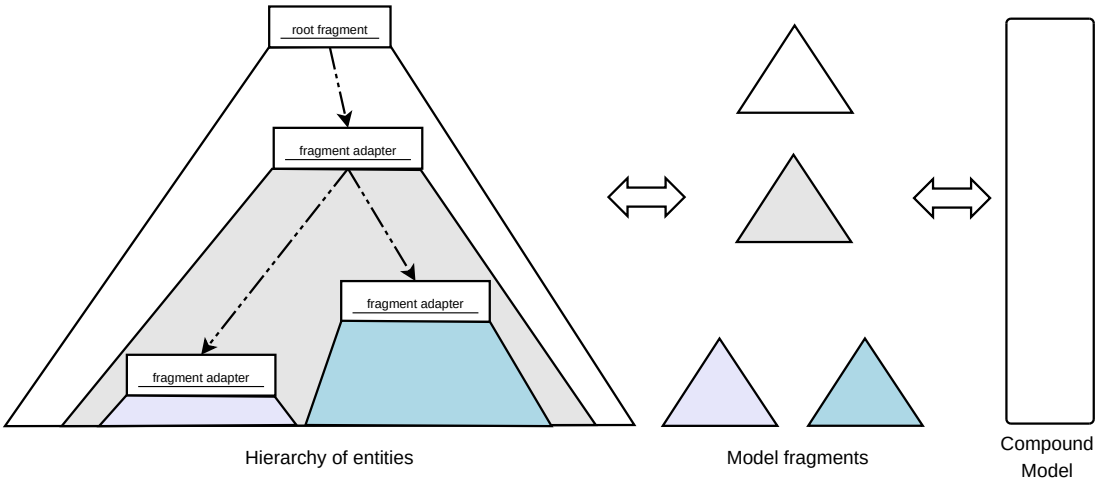

Figure 5.2: The Whole model structure

So, the architecture of a *Whole model* has three layers: a hierarchy of entities, a tree of model fragments and a compound model. Each layer is useful for modeling behavior and state common to the underlaying layers.

5.2.2 Entity resolver

We introduce the *Resolver Object* design pattern. It represents a long departure from the original *Null Object*[34] design pattern that is only a do nothing place holder.

Intent A Resolver Object provides a surrogate for another object that shares the same interface but has an adaptive behavior. A Resolver Object is a place holder for the set of implementation types of its interface. If the ambiguity of the implementation type is resolved at some time or by using setter methods or explicitly, the Resolver Object replaces itself with the right implementation type.

Motivation Sometimes, manipulating a hierarchy of objects, we do not have enough information to choose the type of an instance node while we know how to build parts of its descendants. The lack of information might be permanent like in a pattern having underspecified internal nodes.

A common but partial way to solve this problem would be to build separate object hierarchies until information necessary to merge them become available. Unfortu-

nately, this way operations called on the model do not consider separated hierarchies.

Applicability Use the *Resolver Object* pattern

- When you want clients to be able to ignore the difference between a complete model and a model in construction. This way, the client does not have to explicitly check for `null` values.
- When you want to build a model without following a rigid top down construction order. This way un(der)specified internal nodes are adaptively managed by the pattern and you have always a whole model not a forest with a main model and some fragments.

Structure and participants The structure of the *Resolver Object* pattern is shown in Figure 5.3.

- `Client` - Requires a collaborator with a specific interface.
- `IEntity` - Defines a type with the object hierarchy interface.
- `RealObject` - Defines a concrete implementation of `IEntity`.
- `ResolverObject` - Provides an interface that can be polymorphically substituted for a `RealObject`. Implements its interface to perform a consistency check whenever a setter is called and, replaces itself with a `RealObject` if the `resolve` method is called or if exists only one `RealObject` compatible with the set of properties setted.

Collaborations Clients use the `ResolverObject` like any other object. It accepts calls to setter methods as long as there exists at least one implementation type having all the setted properties; otherwise it throws an exception. If the set of compatible implementation types reduces to one the `ResolverObject` replaces itself with an instance of that implementation type initialized with all setted properties. Here, with compatible implementation types we mean the set of types having all the setter methods invoked so far. The ability to replace all references to a given object with another object is available for all model entities and is discussed elsewhere. Calling a getter method behave as expected if the corresponding property has been already setted otherwise it throws an exception.

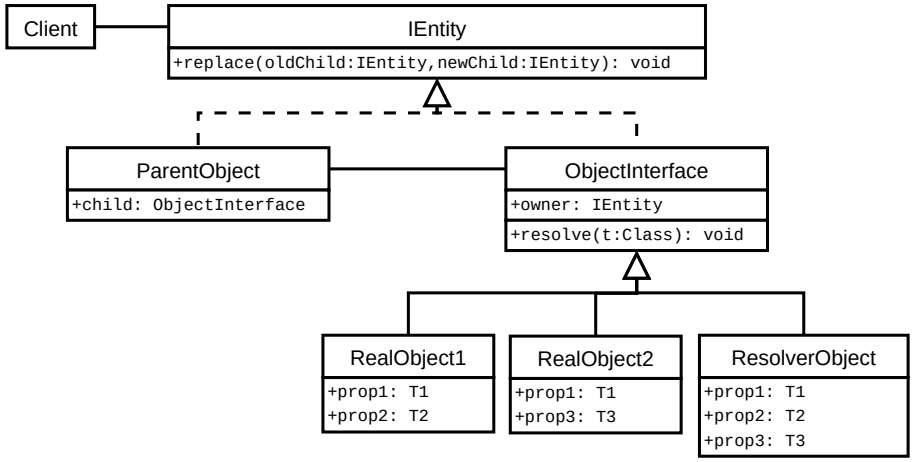

Figure 5.3: Structure of the Resolver Object pattern

5.3 Creational API

The framework defines three kinds of APIs for building model entities: `ModelFactory`, `PrototypeManager` and `TemplateManager`.

The `ModelFactory` is a low level, model specific API for creating instances of single model entities. It is a Singleton concrete Factory implementing the Abstract Factory pattern. Each factory method is responsible for creating an instance of a model entity given all its required features as arguments. This API is used only for instantiating and configuring the prototypes of the `PrototypeManager`; it is not visible outside the implementation of a language.

The `PrototypeManager` is a higher level generic API for building a dynamic set of prototypes. A prototype is a model fragment cloned for every instantiation request; The initial set of prototypes includes all the entities of the language. It is a registry for prototypes.

The `TemplateManager` is a higher level generic API for building a dynamic set of templates. A template is a model fragment built on demand for every instantiation request. It is a registry for templates.

5.4 Modeling API

The modeling API defines the common interface exposed by all model entities. It includes a set of abstract and concrete implementations.

The interface `IEntity` is the root type of all modeled entities; It provides support for behavior and features common to all modeled entities. All method names start with `w` to distinguish the modeling methods from the user defined methods.

Part of the Modeling API is available both with a model-specific interface and with a generic reflective interface.

Model-specific API

Each model defines an interface `I`*MyModel*`Entity` that extends `IEntity` with model-specific accessors to the features.

This model specific API duplicates the generic API for manipulating features by name; it is defined only for convenience. Using the reflective API is slightly less efficient than calling the model-specific methods directly. Actual performance penalty is as low as entering a switch case and forwarding to the model-specific method.

Model-generic reflective API

With the reflective API we can query and manipulate a model generically. Here, *generic* means using the common entity interface (`IEntity`) in place of a model specific type.

Using the reflective API, we can manipulate the model structure in four different ways: by name, by index, by child and by value. According to the Composite pattern, not all APIs are meaningful for each kind of entity: by name methods are of course available only for named features and by value methods only for non structural features. Notice the choice to define by index methods for *all* kinds of entities and not only for composites; this way it is more easy and efficient to define behavior based on iterators or traversals.

All the APIs described on the following subsections are part of the (modeling) reflective API and are declared on the `IEntity` interface.

5.4.1 Containment features methods

Containment features methods permit to navigate the containment axis of the model.

```
IEntity wGetRoot();
IEntity wGetParent();
```

`IEntity wGetRoot()`
Returns the root entity of this model fragment if any. Returns `null` if there is no root entity.

`IEntity wGetParent()`
Returns the parent container entity of this entity if any. Returns `null` if there is no parent entity.

5.4.2 Manipulating features by child

These methods operate on the feature having the given child. These methods are useful in direct manipulation editors.

```
int wIndexOf(IEntity child);
FeatureDescriptor wGetDescriptor(IEntity child);
void wRemove(IEntity child);
void wReplace(IEntity oldChild, IEntity newChild);
```

`int wIndexOf(IEntity child)`
Returns the index of the given `child`.

`FeatureDescriptor wGetDescriptor(IEntity child)`
Returns the feature descriptor corresponding to the given child if it is contained in a feature. Otherwise throws an `IllegalArgumentException`.

`void wRemove(IEntity child)`

If the given child is contained in an indexed feature this method behaves like `wRemove` on the child index. If the given child is contained in a named feature this method behaves like `wUnset` on the child feature. Otherwise throws an `IllegalArgumentException`.

`void wReplace(IEntity oldChild, IEntity newChild)`

Replaces the old child with the new child. If the old child is contained in a feature this method behaves like `wSet` on the child index or feature. Otherwise throws an `IllegalArgumentException`.

5.4.3 Manipulating features by index

By index methods use an index to select the feature to operate on. These methods are implemented by all abstract entity types, this way faster and easier to write model iterators and traversals can be defined.

The `AbstractCompositeEntity` uses this API for manipulating its children. The `AbstractEntity` and all other abstract entity types having only named features assign an index to every named feature and use this mapping for manipulating named features using an index.

```
boolean wIsEmpty();
int wSize();
FeatureDescriptor wGetDescriptor(int index);
void wAdd(int index, IEntity child);
void wRemove(int index);
IEntity wGet(int index);
void wSet(int index, IEntity child);
```

`boolean wIsEmpty()`

Returns `true` if the entity has no features or if it is an empty collection.

`int wSize()`

Returns the number of features of the entity or the number of children in the collection.

`FeatureDescriptor wGetDescriptor(int index)`
Returns the feature descriptor corresponding to the given index if this entity has a corresponding feature. Otherwise throws an `IllegalArgumentException`.

`void wAdd(int index, IEntity child)`
Adds `child` to the specified `index` in the sequence, shifting following entities. Throws `IllegalArgumentException` if this entity has a fixed size.

`void wRemove(int index)`
Removes the child at the specified `index` from the sequence, shifting following entities. Throws `IllegalArgumentException` if this entity has a fixed size.

`IEntity wGet(int index)`
Returns the entity at the given `index` in the sequence.

`void wSet(int index, IEntity child)`
Replaces the entity at the specified `index` with the given `child`.

5.4.4 Manipulating features by name

Named features methods use a `FeatureDescriptor` to select the feature to operate on. The methods are implemented in `AbstractEntity` and can be called only on simple entities.

```
int wIndexOf(FeatureDescriptor feature);
void wAdd(FeatureDescriptor feature);
void wRemove(FeatureDescriptor feature);
IEntity wGet(FeatureDescriptor feature);
void wSet(FeatureDescriptor feature, IEntity value);
void wUnset(FeatureDescriptor feature);
boolean wIsSet(FeatureDescriptor feature);
```

`int wIndexOf(FeatureDescriptor feature)`

Returns the index of the given `feature` in this entity if present, otherwise -1 is returned.

`void wAdd(FeatureDescriptor feature)`

Adds the given `feature` to this entity. Throws `IllegalArgumentException` if this entity is not adaptive.

`void wRemove(FeatureDescriptor feature)`

Removes the given `feature` to this entity. Throws `IllegalArgumentException` if this entity is not adaptive.

`IEntity wGet(FeatureDescriptor feature)`

Gets the value of the given `feature`. The result is always a single non null entity. If the feature is composite, the result is a composite entity containing the values of the feature. If the feature is primitive, the result is a value entity wrapping the primitive value of the feature. If the feature has no value, a resolver entity is returned. Throws `IllegalArgumentException` if the feature is not defined for this entity.

`void wSet(FeatureDescriptor feature, IEntity value)`

Sets the value of the given `feature` to the given `value`. The given `value` must be not null. There is no return value. The operation is performed atomically independently of the multiplicity of the feature.

`void wUnset(FeatureDescriptor feature)`

Sets the value of the feature to its default value. There is no return value. The operation is performed atomically independently of the multiplicity of the feature.

`boolean wIsSet(FeatureDescriptor feature)`

Returns true if the value of the feature is different than the default value of that feature.

5.4.5 Manipulating features by value

Value features methods are implemented in `AbstractValueEntity` and can be called only on value entities. The setter method is overloaded for all primitive types and `String`; the generic getter `wGetValue` is helped by additional type-specific methods.

```
boolean wBooleanValue();
byte wByteValue();
char wCharValue();
double wDoubleValue();
float wFloatValue();
int wIntValue();
long wLongValue();
short wShortValue();
String wStringValue();
Object wGetValue();
void wSetValue(boolean value);
void wSetValue(byte value);
void wSetValue(char value);
void wSetValue(double value);
void wSetValue(float value);
void wSetValue(int value);
void wSetValue(long value);
void wSetValue(short value);
void wSetValue(String value);
void wSetValue(Object value);
```

5.5 Metadata API

The Metadata API provides model-specific static constants and generic convenience methods for accessing the type system of a (language) model. All other metadata are available only in the metamodel of the model and can be accessed using the `ReflectionFactory`.

The metadata consists of two enumerations one for the model entities and the other for the model features together with one `EntityDescriptor` for each entity type and one `FeatureDescriptor` for each feature type.

5.5.1 Metadata of the model type system

Within the model implementation, you can access the two enumeration singletons directly using:
MyModel`EntityDescriptorEnum.instance`
MyModel`FeatureDescriptorEnum.instance`

More in general, you can use the `ReflectionFactory` and an entity object or a language name:
`ReflectionFactory.getModelEntities(`*anEntity*`)`
`ReflectionFactory.getModelFeatures(`*anEntity*`)` or
`ReflectionFactory.getLanguageKit(`*aLanguageId*`).getModelEntities()`
`ReflectionFactory.getLanguageKit(`*aLanguageId*`).getModelFeatures()`

Each enumeration exposes the methods for accessing the language that defines it, the set of names, the list of values and two methods for getting a value given its name or ordinal.

5.5.2 Metadata of the model types

In a static context, you can access the entities and features descriptors and ordinals directly using:
MyModel`EntityDescriptorEnum.`*EntityType*`_ord`
MyModel`EntityDescriptorEnum.`*EntityType*
MyModel`FeatureDescriptorEnum.`*FeatureType*`_ord`
MyModel`FeatureDescriptorEnum.`*FeatureType*

Given an entity object, you can access to its `EntityDescriptor` using the method `wGetEntityDescriptor`; while the `FeatureDescriptor` of a given feature is accessible using the `wGetDescriptor` methods already seen.

Each feature descriptor (`FeatureDescriptor`) exposes its name and ordinal and a method for accessing the language that defines it.

Each entity descriptor (`EntityDescriptor`) exposes also the methods for accessing the list of feature descriptors and the following methods related to type queries:

```
boolean has(FeatureDescriptor feature);
int indexOf(FeatureDescriptor feature);
FeatureDescriptor getDescriptor(int index);
boolean isAssignable(int index, EntityDescriptor entity);
boolean isAssignable(FeatureDescriptor feature,
        EntityDescriptor entity);
```

5.6 Model Context API

We introduce the *Model Context* design pattern.

Intent The intent of the Model Context design pattern is to encapsulate all model entities and references for their lifetime and to hide all details concerned with creation, assembling and traversing including the logic for choosing between traversing and building.

Structure and participants The structure of the Model Context pattern is shown in Figure 5.4.

- `Client` - A client of the model
- `IModelContext` - The model context interface defines a model-generic interface based on the Reflective API.
- `ISpecificModelContext` - The model-specific context interface defines a model-specific interface.
- `ModelContext` - A concrete model context implementation
- `IEntity` - The common interface of all model entities
- `Entity` - A concrete model entity

Collaborations Clients manipulate the model sending all requests to a `ModelContext`. The `ModelContext` forwards all calls to the current model entity. Clients knowl-

edge is limited to the `ModelContext` object; they perform all model operations on it and they are not allowed to gain references to the model entities.

Consequences The model context pattern has several important consequences.

1. The Model Context API is a *Facade* [35] for all the Modeling API and is also a *Builder* separating the construction of the model from its representation.
2. Encapsulation control. All getter methods return an `IModelContext` object that behaves like the corresponding model entity expected as a result. This way, aliasing control of model entities is straightforward.
3. Scope control. Getters and setters can be easily configured to operate in a scope that can range from the current model entity to a chain of model entities resulting from the model structure or a traversal. For instance, inherited features can be supported using a dynamic scope for model features.
4. Concurrency control. Because all accesses to the model entities are mediated by a model context, concurrency support can be encapsulated at the model context level.
5. Quantification support. The current model context may be a single entity but may also be a set of entities. In the latter case we can define a semantics for applying methods to a set of entities. Specific methods for manipulating the context quantification can be included in the model context interface.
6. Intensional model view. When used together with an history manager, the model context can provide a view of the model tied to a particular version. This way a consistent view of the model can be granted to time consuming operations without blocking model manipulation.
7. Virtual models. You are not required to explicitly represent the model with a hierarchy of objects. For instance the model context building operations can be used to trigger directly event-driven model transformations and code generation *without* constructing an actual model in memory (see SAX[12] and ASM[23] to get elegant examples). Same way, an intensional view can be backed by a virtual model that behaves as expected but is synthesized on demand.

8. Model reconciliation. Getters, setters and factory methods can be implemented with a reconciliation behavior. That is, the operations can check if the existing model is consistent with them and if not they can try to reconcile the differences switching from a traversing to a building behavior.

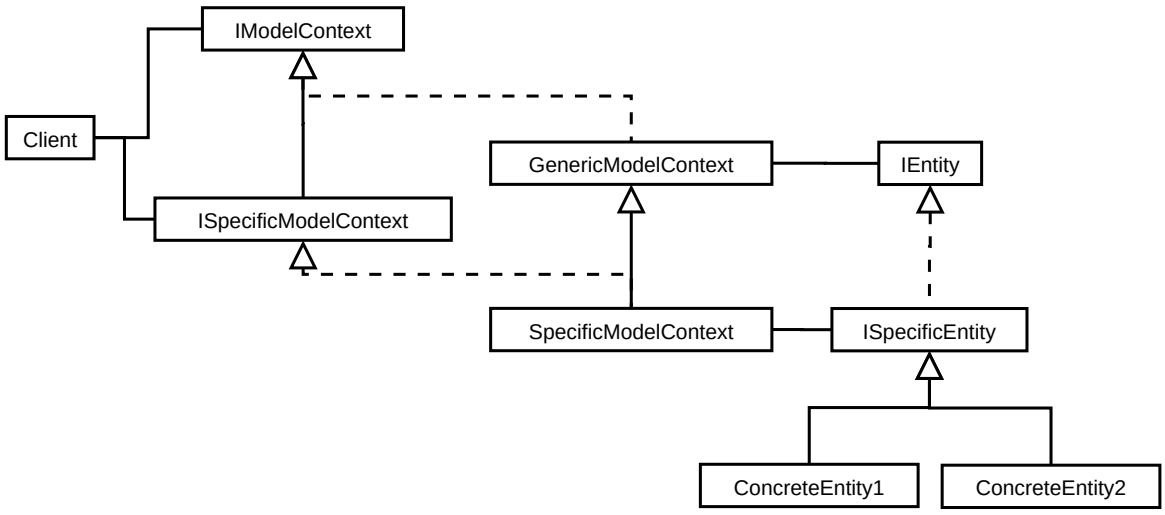

Figure 5.4: Structure of the Model Context pattern

5.7 Intensional versioning support

All (model) computations are made according to an actual context. Once we add the ability to *change* the computation context we begin to have intensional programming[28]. The behavior of an intensional program depends on the execution context and it has been modeled using the possible world semantics of intensional logic[16].

Each model operation that changes the structure of the model implicitly build a new version of the model itself. Without a specific programmer effort, an imperative language (for instance Java) discards all previous values of the model: i.e. there is only a current model version.

To support an intensional style of programming we need the ability to access to previous versions of a model and to select them as execution context for successive model operations. In particular, calling *history* a time related sequence of versions, we expect

the ability to select a version from the history of an entire model, of a given entity, or of a given entity feature.

A *dimension* defines the coordinates of a context in which we can navigate. Any number of dimensions can be defined by an intensional language such as Indexical Lucid[18]. We restrict ourselves to time related dimensions like the original Lucid[19][20] language because we are dealing only with intensional versioning of hierarchical structures. Furthermore, currently, we allow only changes to the current version so that no branching is possible.

All model related definitions are assumed to vary through the time dimension; we say that they are versioned. Any part of a model, from a simple entity feature to a complete model, may exist in different versions. Changes occur at the granularity of single entity features. So, the history of a feature is given straightforward by the successive values assumed by that feature. A particular version of an entity is formed by combining the most relevant existing versions of the features composing the entity. A particular version of an entire model is formed by combining the most relevant existing versions of the various entities of the model. We call it an*intensional view* of the model.

Of course, we do not require that every model component exist in every version. Instead, we consider the absence of a particular version as meaning that a previous version is adequate. We follow the *variant structure principle* introduced in [59] to select what is inherited: the general rule is that when constructing version V of a system (entity, model), we choose the version of each component which most closely approximates V according to the ordering on versions. We call it the *most relevant version*.

To manipulate the time dimension, we support the two well known[58] intensional operators: *intensional query* and *intensional navigation*. The first is used to query the version of an entity, the latter to select the one to use.

In our approach version labels are numbers assigned by the framework and they have a global significance. Human readable versions using string labels or common dot notation can be attached to our (system provided) versions by an ordinary model dedicated to user-level versioning.

We introduce a version algebra to define an ordering on versions with a refinement relation that can be used to automate the building of intensional views of models.

Definition 5.1 *The* version algebra *is specified by the pair* $(V, \sqsubseteq)$ *where:*

- V *is a countable set of versions*
- $\sqsubseteq$ *is a binary relation on* V *which we call the* refinement relation*:*

  reflexive $v \sqsubseteq v$

  transitive *if* $u \sqsubseteq v$ *and* $v \sqsubseteq w$*, then* $u \sqsubseteq w$

  total order $u \sqsubseteq v$ *or* $v \sqsubseteq u$

We define $u = v$ *as* $u \sqsubseteq v$ *and* $v \sqsubseteq u$*.*

We read $v \sqsubseteq w$*, as* v *is* refined *by* w*, or* v *is* relevant *to* w*, meaning that* w *is the result of further developing version* v*.*

The *most relevant version* of a component c and a version v is the component version w such that does not exist a component version w' with $w \sqsubseteq w' \sqsubseteq v$. In configuring the version v of a compound system (entities, models), we assemble the most relevant version of each of its components.

5.7.1 Implementation

In the implementation, V is instantiated with the set of natural numbers and $\sqsubseteq$ is instantiated with the $\leq$ relation.

Each model changing operation is encapsulated in a command object. To each command is assigned a unique progressive version number in execution order. The model and its components are only indirectly versioned through associations to commands.

We have chosen to partition versioning related data and behavior between four participants: the model, the commands, the history event handler and the model context. The model context exposes both the intensional operators and a behavioral representation of an intensional view of the model. Each model entity is associated to exactly two commands: the command used to bind the entity to a parent entity and the last command used to change the entity state (features, children). The links between commands maintain both the parent axis of the model structure and two subsequences of the model history. So we have one global history stored by the history event handler containing the sequence of all commands executed and two local histories storing the history of each entity and of each entity feature of the model.

Let us consider in greater details the implementation of each participant; use the class diagrams in Figure 5.5, Figure 5.6 and Figure 5.7 to follow the description.

A command represents a model change. Each command is labeled with a version which is a unique progressive number in execution order. A command exposes an intensional query method and two intensional navigation methods to access to a local history built on the previous command association. The command API also defines methods for accessing information about the entities involved on the operation.

```
public interface ICommand {
  int getKind();
  int getExecutionTime();
  ICommand getPrevCommand();
  ICommand getCommand(int contextTime);

  IEntity getSource();
  FeatureDescriptor getSourceFeatureDescriptor();
  int getSourceIndex();
  IEntity getOldValue();
  IEntity getNewValue();
}
```

The model support to versioning is given by the following two pairs of methods declared on the `IEntity` interface:

```
  ICommand wGetBindingCommand();
  void wSetBindingCommand(ICommand command);
  ICommand wGetLastCommand();
  ICommand wSetLastCommand(ICommand command);
```

`ICommand wGetBindingCommand();`

Returns the command used to attach the entity to a parent model feature.

`void wSetBindingCommand(ICommand command);`

Associates the entity to the command that established the connection with a parent model. The binding command represents a step in the history of changes of the parent feature. This method is supposed to be called only by the framework when a setter of the entity that will become the parent is called.

```
ICommand wGetLastCommand();
```

Returns the last command used to change the entity. For a simple entity it is the command that last changed a named feature; for a composite entity it is the last command that manipulated the entity children; and for a value entity it is the command that setted the current entity value.

```
ICommand wSetLastCommand(ICommand command);
```

Associates the entity to the last command executed on it and returns the previous command. This method is supposed to be called only by the framework when a setter method of the entity is called.

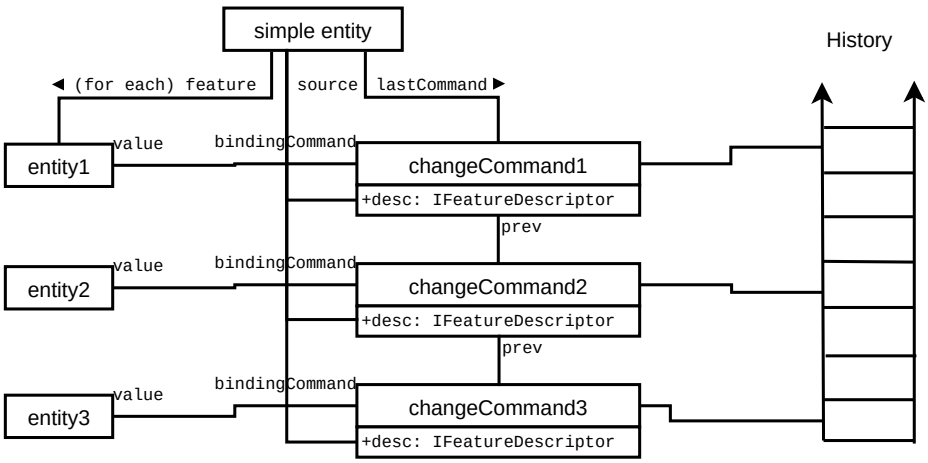

Figure 5.5: Simple entity versioning structure

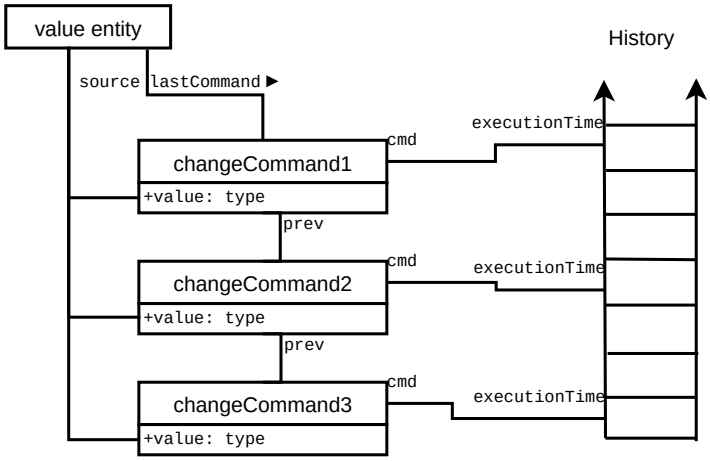

Figure 5.6: Value entity versioning structure

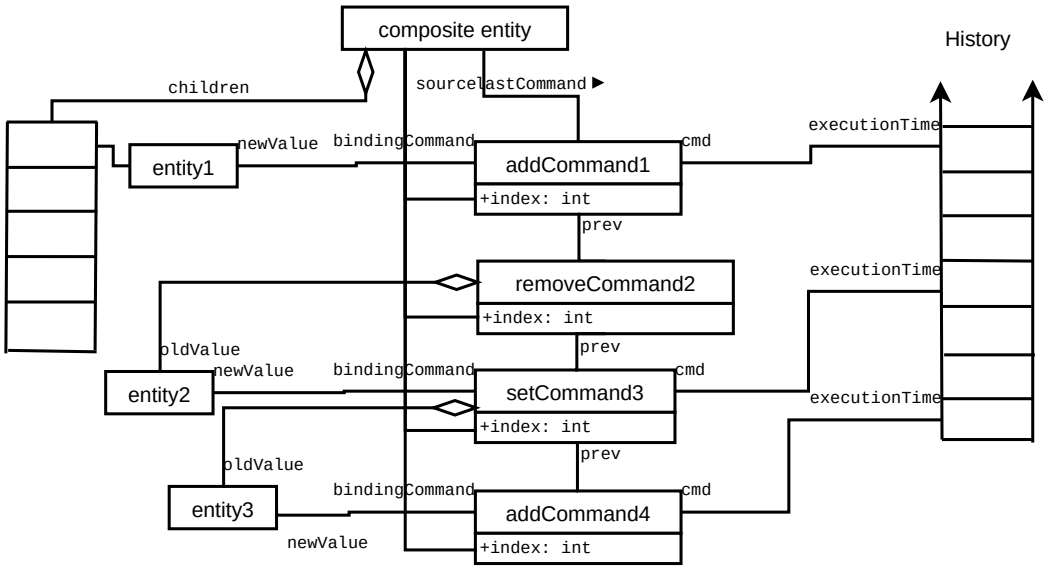

Figure 5.7: Composite entity versioning structure

Chapter 6

The Traversal framework

6.1 Introduction

The Traversal framework complements the Modeling framework facilitating the writing of model operations. By definition, a *model operation* traverses the model, in some way and to some extent, to perform its computation; in general the behavior of a model operation is polymorphic.

The traversal framework supports the definition of the traversal part of a model operation providing the following facilities:

Modular definition An operation can be implemented in a modular unit; this way, new operations can be added without modifying existing models (class hierarchy).

Traversal strategies Ability to compose and to control traversal strategies. A *traversal strategy* defines a sequential order for traversing the entities of a model. We call *compound traversal* a traversal defined by composition of two or more traversals. For traversal *control* we mean the choice of what parts of the model to skip and when to stop the traversal. The Traversal framework comes with a rich set of traversal strategies.

Polymorphic behavior The extension points of the traversal code can distinguish model-specific types so you can easily write polymorphic behavior. For each part of the operation you can choose between a model-specific or a model-generic behavior. The supported granularity includes: model-specific implementation types and abstract types, model-generic entity types and model types.

Compound models support Transparently supports operations on compound models. We call *compound model* a model embedding other models (with different class hierarchies). You can write a default model-generic traversal and behavior. Each model in turn can override the generic behavior with a model-specific traversal and/or behavior.

Staging support Transparently supports staged behavior for operations. Each model is annotated with an execution stage (i.e. base level or meta level). You can write an operation with a different behavior for each stage. This ability can be combined with compound models support: for each operation a model may define a different behavior for each stage, either model-specific or generic.

Variability time control At runtime, new traversal strategies can be defined by composition and new operation behavior can override predefined behavior. At generation time, a compound traversal can be generated inlining its traversal components; same way an entire operation can be inlined in a model class hierarchy. Even an inlined operation can be overridden at runtime with the granularity of a model.

To design this framework, we have taken these facilities as requirements and we have analysed existing design patterns and frameworks.

Two kind of patterns have been proposed to deal with traversals: visitors and iterators. The Visitor[35] pattern defines a *push* API that you have to implement in order to get called during traversal. The Iterator[35] pattern exposes a *pull* API that you call for getting each entity of the model sequentially in traversal order.

Both patterns allow the encapsulation of polymorphic behavior outside the class hierarchy of the model on which they operate.

The Visitor pattern requires the definition of a class implementing the visitor interface for encapsulating the behavior. With the Iterator pattern you have the control of the traversal so you can write the behavior code within a method of your choice. Having control is a pro, but you have not type-specific knowledge this way, so, for writing a polymorphic operation, you have to determine for each element of the traversal the polymorphic variant to apply. With the visitor pattern, you write the polymorphic variants of an operation in different methods each called with a type-specific element to work on;

that is, a traversal visitor encapsulates the choice of what polymorphic variants you want to write.

Different iterators can be defined for supporting different traversal strategies. Writing an iterator is straightforward if the model has a reflective API that supports all basic one step traversal operations. In general, combination is limited to specialization and traversal control is limited to choosing when to stop and subtrees to prune[32]. Iterators are the preferred choice for traversing collections but they are also used for traversing models. For the DOM model[74] two kind of iterators are provided as an optional feature[73]: node iterators hiding the hierarchical structure and tree walkers presenting a tree-oriented view; node filters are also supported for skipping nodes. The EMF framework[1] provides a tree traversal with a fixed top-down strategy and a prune method for skipping subtrees.

Unfortunately, also the original *GOF visitor*[35] pattern resists combination and allows little traversal control. Combination is limited to specialization and the traversal strategy is either hard-wired into the accept methods, or entangled in the code of visitors. Several variations have been proposed to remove these limitations [56][47][48] and to allow the use of the pattern in frameworks [70][67].

The GOF Visitor suffers of an inherent dependency cycle between the class hierarchy (the model) and the visitor interface that makes it inconvenient in scenarios where the class hierarchy changes. A full exposition of the problem and an elegant solution called *acyclic visitor* is proposed in [46]; unfortunately, this solution uses many dynamic casts.

The most important changes to the class hierarchy we have to support is the composition of class hierarchies. That is we want to write *generic visitors* working with a compound model embedding model types unknown at compile time. This problem is simpler and has been resolved by Vlissides with the *staggered visitor*[70] pattern; it supports generic visitors by introducing, on model-specific visitors, two forwarding methods from generic to model-specific visit methods and vice versa.

Visitor combination and traversal control can be obtained with the *Visitor Combinators* [66]; the JJTraveler framework [42] implements visitor combinators and make it possible to write libraries of generic algorithms. This solution satisfy near all our requirements. Compound traversals can be easily defined by composition of existing traversals but there is no a traversal factory allowing transparent substitution of model-generic

traversals with model-specific ones. The support for compound models is not efficient enough and lacks the ability to have an operation state shared by all visitors composing the operation. Compound visitors are always composed at runtime so more dispatching calls are necessary. The library provides a set of model-generic traversals; the generation of model-specific traversals is possible but considered out of the scope of the library.

We are working with models defined using the Modeling framework, so a compound model has an adapter entity separating each embedded model. Taking advantage of this knowledge we can mix model-specific visitors between them and with model-generic visitors with less performance penalties. Furthermore, the choice of the visitor to use for a given entity is done in constant time (performing a simple switch) and not in linear time on the number of models types (performing multiple catches of exceptions).

The Traversal framework uses a generative approach to add the ability to compose the traversal combinators at generation time, to generate a family of model-specific visitor combinators and to inline visitor behavior in the class hierarchy itself. An operation can be defined with a model-generic behavior and a shared context for state variables; each model in turn can choose to specialize the operation behavior either in a modular way or with code inlined in the class hierarchy.

6.1.1 Applicability

The simpler use case for a traversal is an operation that can be performed directly with simple or no auxiliary model. For example an operation performing an *analysis* requires a traversal that derives properties or values from the model. Examples: computation of metrics, interpretation.

The main application for traversal operations is in model transformations. A *model transformation* is a polymorphic operation executing on a model. According to a common taxonomy, we categorize the transformations over a model representation depending on the kind of participant models.

Traslation is a transformation where the source and target models are different. Examples: compilation, migration, reverse engineering, pretty-printing, type inference.

Rephrasing is a transformation where the source and target models are the same. Examples: refactoring, normalization, optimization, desugaring, renovation, cloning.

With this framework the implementation of a transformation requires a compound pattern: visitor + builder.

The visitor is a composition of a traversal and the director part of the builder. So the complexity of the visitor is similar to the simple use case even for complex model transformations. Note that the responsibility for building a consistent target model is not on the visitor side of the transformation, so, the composition of transformations is not a problem of coordination between visitors but a matter of following target builder rules.

6.2 Defining polymorphic operations

An `OperationFactory` API is used to discover all operations available. The factory defines methods for selecting operations using the operation name, the kind and the target model.

All polymorphic operations have to implement the `IVisitorOperation` interface; they can do that using or extending the `DefaultGenericOperation` implementation. The operation class has the responsibility of maintaining the set of all polymorphic variants of the operation specialized for execution stage and for model type.

```
public interface IVisitorOperation {
  boolean isStage0Fragment();
  void stagedVisit(IEntity entity);
  void stagedVisit(IEntity entity, int stage);
}
```

`boolean isStage0Fragment()`

Returns `true` if the current entity is executed at stage 0 (base level); otherwise we are at a template level stage.

`void stagedVisit(IEntity entity)`

Visits the given entity using a model-specific visitor for the current execution stage. If a model-specific visitor is not defined, use the model-generic visitor or if not available throw an exception.

`void stagedVisit(IEntity entity, int stage)`

This overloaded variant permits to change the execution stage during the staged visit.

Each visitor is instantiated the first time it is requested then it is reused for all the operation call, so it can accumulate model-specific and stage-specific state.

An operation is performed by a set of visitors implementing the `IVisitor` interface.

```
public interface IVisitor extends IVisitorOperation {
  IVisitorOperation wGetOperation();
  void visit(IEntity entity);
}
```

`IVisitorOperation wGetOperation()`

Returns the current operation. The operation object stores all the state of the operation not specific for a givn model.

`void visit(IEntity entity)`

This is the generic method used for visiting an entity. The implementation can perform the visit itself or more likely can forward the visit to a method in the model that in turn can choose to do the visit itself or to complete the double-dispatching returning in an entity specific visit method on the visitor.

The operation continues using the same model specific visitor or the inlined methods until an adapter entity separating an embedded model is encountered. At this point a staged visit is required.

The Traversal framework includes several abstract and concrete visitors as depicted in Figure 6.1. Two dotted lines are used to separate model-specific visitors from model-generic ones; the other line separates user defined visitors from visitors included in the framework or generated from it.

The `IVisitor` interface is the root of two hierarchies of visitors: model-generic visitors implementing the interface `IGenericVisitor` and model-specific visitors implementing a different model-specific interface for each model. All (provided) visitor implementations extends the `AbstractVisitor` class.

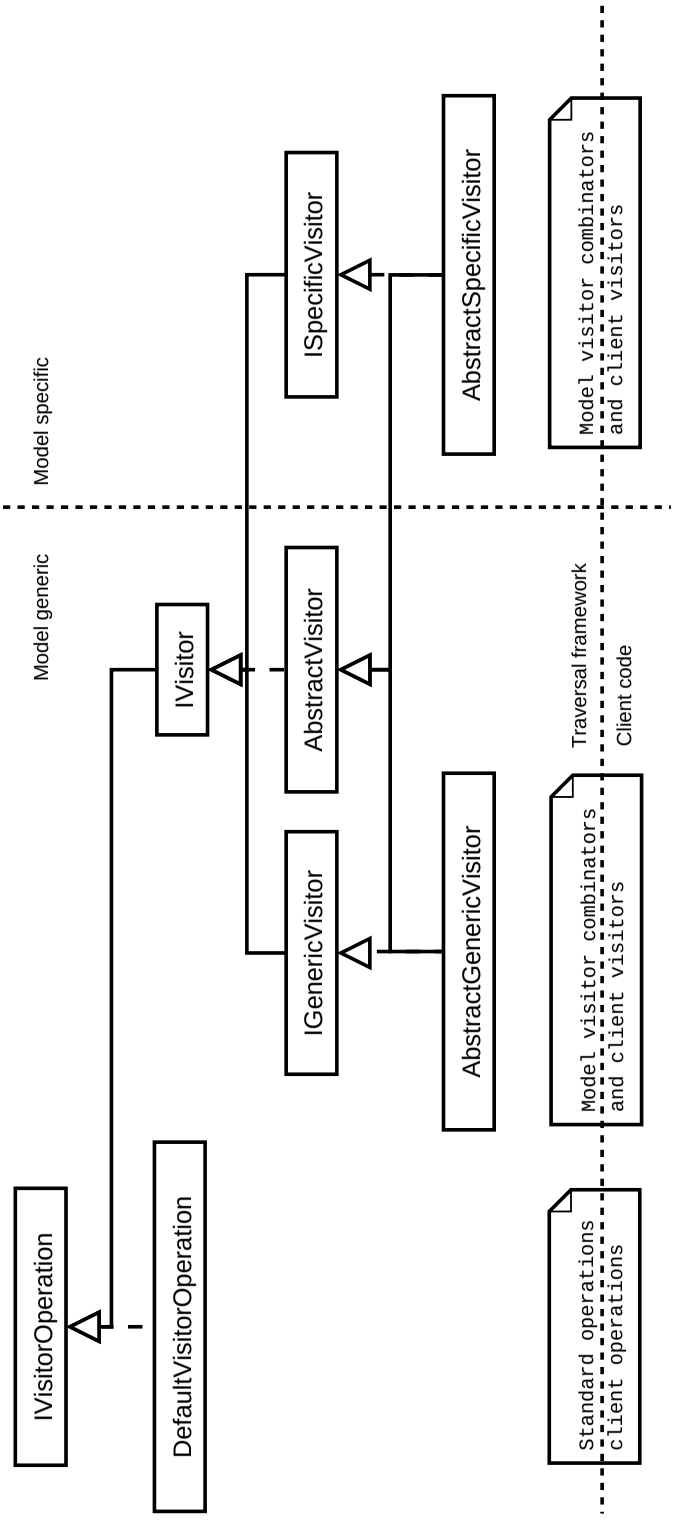

Figure 6.1: Class diagram of operation visitors

Generic visitors use the model reflective API to perform the expected operation; the `IGenericVisitor` interface provides an API for writing a polymorphic behavior based on the abstract entity types defined by the Modeling framework.

```
public interface IGenericVisitor extends IVisitor {
  void visit(AbstractBaseFragment entity);
  void visit(AbstractMetaFragment entity);
  void visit(AbstractAdapterFragment entity);
  void visit(AbstractProxyFragment entity);

  void visit(AbstractEntity entity);
  void visit(AbstractCompositeEntity entity);
  void visit(AbstractValueEntity entity);

  void visit(AbstractAdaptiveEntity entity);
  void visit(AbstractEntityResolver entity);
  void visit(AbstractFeatureResolver entity);
  void visit(AbstractEntityVariable entity);
}
```

The generic entity interface `IEntity` supports only generic visitors with the following accept method:

```
void wAccept(IGenericVisitor visitor)
```

Each model in turn may support model-specific visitors adding an appropriate accept method to the entity interface of the model.

6.3 Visitor combinators

The framework provides the implementation of an extensible family of traversal visitors that we call, according to Visser[67], *visitor combinators*. The framework defines a set of generic visitor combinators and is able to generate a corresponding set of model-specific

combinators for a given model. Model-specific variants are useful only for improving performance.

The set of visitor combinators provided is almost a subset of the one defined in the JJTraveler framework [66].

Traversal	Args	Description
identity		Do nothing with per Entity methods
identityWithDefault		Do nothing with per Entity and per Type methods
failure		Throws an exception (VisitFailException)
sequence	v1, v2	Apply v1 and v2 in sequence
ifThen	v1, v2	Apply v1 and if it succeeds apply v2
ifElse	v1, v2	Apply v1 and if it fails apply v2
ifThenElse	v1, v2, v3	Apply v1 and if it succeeds apply v2 otherwise v3
traverseAll	v	Apply v sequentially to all immediate children
traverseSome	v	Same as all but fails if all children fail
traverseOne	v	Same as all but stops the first child that succeeds
topDown	v	sequence(v, traverseAll(this))
topDownWhile	v	ifThen(v, traverseAll(this))
topDownUntil	v	ifElse(v, traverseAll(this))
bottomUp	v	sequence(traverseAll(this), v)
downUp	before, after	sequence(before, bottomUp(after))
downUpWhile	before, after	ifThen(before, bottomUp(after))

Table 6.1: Visitor traversal Factory (excerpt)

The first group includes primitive traversals, the second includes derived traversals. A *primitive traversal* is implemented in a corresponding visitor class. A *derived traversal* is defined by composition of other traversals.

For instantiating traversals, we have defined a `GenericVisitorFactory`. Model specific factories can be defined extending it. A model specific factory must redefine all primitive traversals and may redefine derived traversals.

6.4 Pattern matching visitors

A specialized visitor interface is provided for matching a model with a pattern. A *pattern* is a model with zero or more entity variables. Notice that patterns are implemented as first-class entities like models.

Actual implementation allows variables to occur more than once in a pattern (*non-linear* pattern matching); the pattern cannot be matched against another pattern (no *unification*).

Three generic concrete visitors are provided with the following semantics:

Exact match Returns true if the pattern is structurally identical to the model.

Contains pattern Returns true if the model contains the pattern.

Template learning The model is extended with parts taken from the template.

Pattern matching visitors can be mixed with other visitors.

6.5 Model interfaces and polymorphic behavior

A *model* is a hierarchy of objects linked by association relations. An *association* is implemented by an object field with a reference type pointing to another object of the model.

The type of an association determines the operations you can perform on the associated object without the need of a type cast.

The most used choice is to declare in an association type the behavior common to all the compatible implementation types. This way, traversing the model, you get an interface with the larger set of behavior you can perform on the object without knowing its concrete type. Unfortunately, this way you are also restricting future changes to the model. We say that your model is less adaptive because you are assuming actual commonalities will never change.

An alternative solution that poses no restrictions on future changes to the model hierarchy is the use of marker interfaces. The *marker interface*[37] pattern uses interfaces that declare no behavior to group classes in categories. This way, to define a polymorphic

variant of an operation you have to check the runtime type of an object for the presence of the marker interface. Examples of this pattern can be found on the Java API: `Cloneable`, `Serializable`, `EventListener`.

We introduce the *Visitable Marker Interface* design pattern that gives us near all the benefits of the two approaches without reducing the adaptiveness of the model and the performance of polymorphic operations in a significant way.

Intent To declare a reference type that maximize the behavior you can perform on the associated object and that minimize the constraints on the implementation of compatible concrete types.

Motivation When you define an association relation from an object to another you have to choose the association type. You have two competing goals: to have access to all the behavior of the associated object and to are free to define new compatible implementations not constrained by the association type.

Applicability Use the *Visitable Marker Interface* pattern

- when the association relation is part of the definition of an object hierarchy (i.e. a model) and
- when you want to define an adaptive model and
- when you define all polymorphic behavior in a modular way (i.e. using a variant of the Visitor pattern).

Structure and participants The structure of the *Visitable Marker Interface* pattern is shown in Figure 6.2.

- `Visitable` - The interface used to support the Visitor pattern.
- `VisitableMarker1` and `VisitableMarker2` - The visitable marker interfaces.
- `Entity1`, `Entity2`, `Entity3` - The model entities. Each entity implements at least one visitable marker and all association types are also visitable markers.

Collaborations Visitor clients use the `accept` method to perform a double dispatching that call the polymorphic variant of the operation with a concrete parameter type

allowing direct use of all public behavior and state of the implementation. Clients can also use a reflective API to access the object behavior and state.

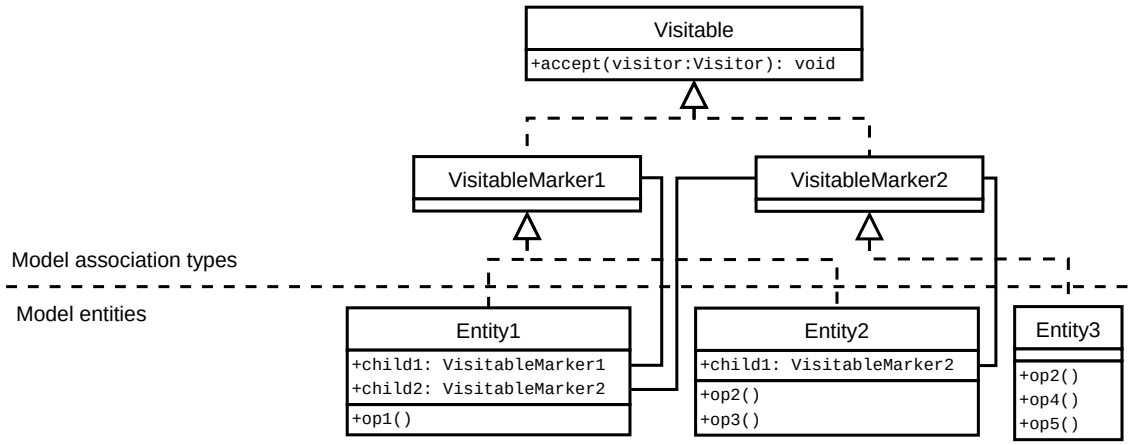

Figure 6.2: Structure of the Visitable Marker Interface pattern

Chapter 7

The Notification framework

7.1 Introduction

In a scenario of *continuous synchronization*, the model transformation cannot be achieved using the Traversal framework; the support of an Observer[35] like pattern is required.

The Notification framework is a dependency manager, it deals with maintaining state dependencies between different model features both within a model and across several models. State changes of one feature can be reflected in state changes of further dependent features.

The framework defines a Dependency Management pattern and a generator API.

7.2 Dependency Management pattern

Every model entity is a notifier, that is, it can send notification events whenever a feature is changed.

The model delegates the notification process and the management of dependencies to an *event handler*.

The Dependency Management pattern defines a general notification mechanism between a Model and an `EventHandler`. An `EventHandler` is a modular unit of dependency control; it is the counterpart of a visitor: a visitor is triggered by a traversal, an event handler is triggered by a state change.

The framework generates a family of combinable event handlers and a few handlers that implements specific model synchronisation services. The pattern lets a Client dynamically configure a Model with a composition of dependency managers.

In a typical use case, the event handler plays the role of a synchronizer. Both intra and inter model synchronization are supported.

Live refactoring A synchronization module can be defined to enforce all intra model dependencies corresponding to refactoring operations. Editing a model with a refactoring synchronizer produces the same behavior of applying a refactoring operation by hand.

Live generation In place of writing a traversal for generating from scratch a target model, you can define a synchronizer that updates the target model whenever the source model changes.

Two-ways synchronization Source and target models can be synchronized so that editing either model updates the other accordingly.

Abstracting generation Do suppose to have a design pattern level model synchronized with an implementation model in a legacy language say Java; then you can write builders targeting the design patterns model in place of the Java implementation model. Such builders do not have to deal with implementation details. In fact, they are also more reusable because you can change the target implementation model without affecting them.

Multiple generation Multiple target models can be generated at once.

Generation time control Defining both traversal generators and synchronizers, you can control the generation time (even at runtime) ranging from batch generation to live model synchronisation.

Notice that some existing tools (i.e. UML modeling) provide two-ways synchronization with code but they achieve this result in an ad-hoc manner; with the Whole Generative framework you get support for writing both intra and inter model synchronization modules.

7.2.1 Event handlers interface

All event handlers have to implements the following interface:

```
public interface IEventHandler extends Serializable {
  IEventHandler clone(IEventHandler parentEventHandler);

  boolean hasSharingEventHandler();
  SharingEventHandler getSharingEventHandler(IEntity entity);
  IEventHandler add(IEventHandler eventHandler);

  void notifyAdded(IEntity source, FeatureDescriptor feature,
          int index, IEntity newValue);
  void notifyRemoved(IEntity source, FeatureDescriptor feature,
          int index, IEntity oldValue);
  void notifyChanged(IEntity source, FeatureDescriptor feature,
          int index, IEntity oldValue, IEntity newValue);
  void notifyChanged(IEntity source, FeatureDescriptor feature,
          IEntity oldValue, IEntity newValue);

  void notifyChanged(IEntity source, FeatureDescriptor feature,
          boolean oldValue, boolean newValue);
  void notifyChanged(IEntity source, FeatureDescriptor feature,
          byte oldValue, byte newValue);
  void notifyChanged(IEntity source, FeatureDescriptor feature,
          char oldValue, char newValue);
  void notifyChanged(IEntity source, FeatureDescriptor feature,
          double oldValue, double newValue);
  void notifyChanged(IEntity source, FeatureDescriptor feature,
          float oldValue, float newValue);
  void notifyChanged(IEntity source, FeatureDescriptor feature,
          int oldValue, int newValue);
  void notifyChanged(IEntity source, FeatureDescriptor feature,
          long oldValue, long newValue);
  void notifyChanged(IEntity source, FeatureDescriptor feature,
          short oldValue, short newValue);
```

```
  void notifyChanged(IEntity source, FeatureDescriptor feature,
          String oldValue, String newValue);
  void notifyChanged(IEntity source, FeatureDescriptor feature,
          Object oldValue, Object newValue);
}
```

Notice that the values are not encapsulated in an event object and that the `notifyChanged` method is overloaded for each primitive Java type.

7.2.2 Entity notification code

Model change notification is triggered by feature setters calling an (overloaded) `notifyChanged` method. Take the following template code as example:

```
  public void setFeature(FeatureType value) {
    notifyChanged(ModelFeatureDescriptorEnum.feature,
        this.feature, this.feature = value);
  }
```

7.3 Predefined event handlers

There is a set of predefined event handlers that can be grouped in two categories: composable and specialized event handlers.

7.3.1 Composable event handlers

The Notification framework comes with a small set of event handlers to facilitate the writing of new event handlers.

The `DefaultEventHandler` is a do nothing implementation that can be extended with feature type specific behavior.

The `CompositeEventHandler` maintains a sequence of event handlers and forwards all events to each of them in order.

The `GenericEventHandler` forwards all interface notification methods to the one with `Object` type for values parameters.

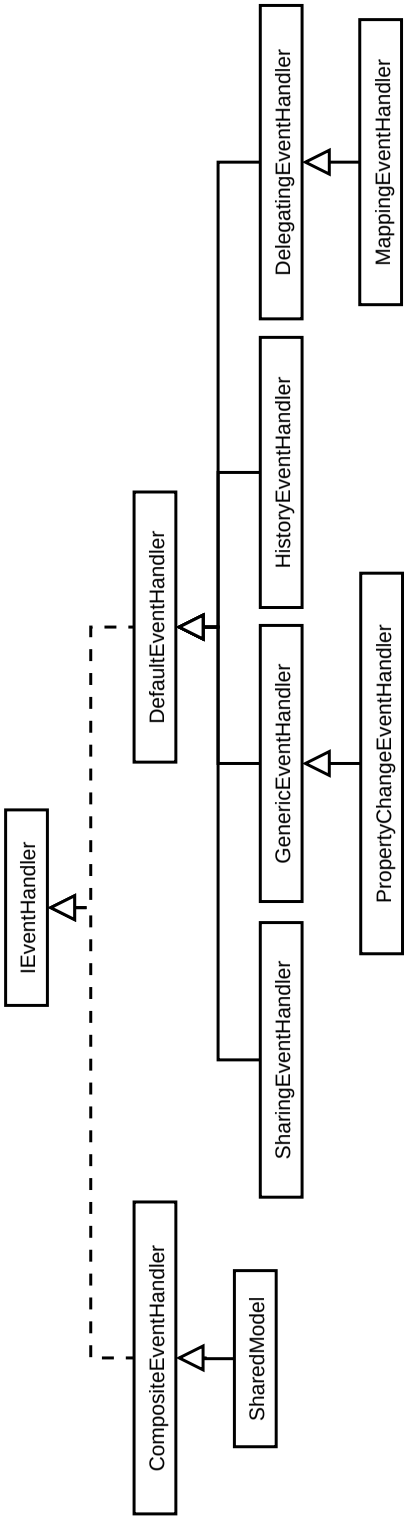

Figure 7.1: Class diagram of adapter event handlers

The `DelegatingEventHandler` is an abstract implementation that forwards all notification methods to an event handler selected by an abstract `getEventHandler` method.

The `MappingEventHandler` is an abstract specialization of the `DelegatingEventHandler` that uses a map to associate an event handler to an entity-feature pair. A few concrete implementations are defined.

7.3.2 PropertyChange event handler

The `PropertyChangeEventHandler` is a standard event handler installed on the shared model. Is is a Java compatible event handler that encapsulates events in `PropertyChangeEvent` objects and forwards them to all registered `PropertyChangeListeners`. It includes a set of methods to control the set of listeners:

```
boolean hasEventListeners();
Set getEventListeners();
void setEventListeners(Set eventListeners);
void addAllEventListeners(Collection eventListeners);
void addEventListener(PropertyChangeListener eventListener);
void removeEventListener(PropertyChangeListener eventListener);
```

The shared model duplicates the methods for registering listeners and lazily install this event handler.

7.3.3 History event handler

The *history event handler* is a standard event handler installed on the shared model. It provides three services to model clients:

Transparent undoable operations All operations performed over a model can be undone and redone.

Transactional behavior A sequence of changes can be encapsulated in a transaction that can be committed or rolled back.

Optimistic feature validation A feature setter updates its value and then notifies changes in a transaction context. Each event handler can reverse all changes consequents to the setter simply calling the `rollback` method.

The `HistoryEventHandler` stores every event it receives in a sequence of model changes each encapsulated in an object implementing the `ICommand` interface:

```
public interface ICommand {
  void undo();
  void redo();
}
```

The idea is to hide a logical unit of change behind a fixed interface. The history service uses this interface to undo/redo the change.

The concrete commands are model-generic. They store a reference to the entity changed, the feature affected and the old and new values. To undo/redo the change, they use the model reflective API.

The `HistoryEventHandler` provides the following API:

```
void undo();
void redo();
void begin();
void commit();
void rollback();
```

The `begin` operation initiates a transaction. Each transaction eventually terminates either with a `commit` or a `rollback`.

The `commit` operation encapsulates in a `CompoundCommand`[71] all commands executed after the `begin` operation so that the entire transaction can be undone and redone atomically.

The `rollback` operation reverses all changes happened after the call to `begin`. All commands executed after the `begin` operation are undone and discarded (they are not available for a successive `redo`).

Notice that there are a few differences between this pattern and the well known Command design pattern [35]. First, the commands encapsulate a change already applied to

the model and, second, the service is behind the model not in front of it so the creation of the history is transparent for clients of the model.

The `HistoryEventHandler` interface is also available from the model and the model context.

7.3.4 The Sharing event handler

The *sharing event handler* implements a sharing relation between model entities. Each entity is part of a sharing set and all model operations performed over the entity are applied to all entities in the sharing set. The sharing relation is a structural quantifier for entity operations.

7.4 Event handler deployment

Event handlers can be deployed at three levels: per entity, per model and per shared model.

An event handler deployed *per entity* receives notifications only by the entity. An event handler deployed *per model* receives notifications from all entities of the model. An event handler deployed *per shared model* receives notifications from all entities of a compound model.

To deploy an event handler you have to call the `addEventHandler` method in an entity, model or shared model. The `addEventHandler` method is responsible for returning a possibly different event handler that replaces itself. This signature allows for transparent composition and on demand deployment.

The default behavior returns a new `CompositeEventHandler` with the old and the added event handlers as children. The `DefaultEventHandler` is the default per entity event handler and cannot be explicitly deployed; adding an event handler to it has the effect of replacing it with the added handler. The `CompositeEventHandler` cannot be explicitly deployed; added event handlers are appended to the children list. The `SharingEventHandler` is deployed on demand the first time `getSharingEventHandler` is called.

7.5 Event handler clone behavior

The model `clone` operation interacts with event handlers deployed per entity. The `IEventHandler.clone` method receives the cloned parent event handler and is responsible for cloning itself and returning a new parent. Each event handler can define a clone behavior choosing between:

Clone the event handler is cloned and added to the parent event handler.

Share the event handler is shared

Discard the event handler is discarded; the parent event handler is returned.

Discard is the default behavior for event handlers deployed per entity.
The `DefaultEventHandler` discard itself returning the parent event handler. The `CompositeEventHandler` discard itself if, after cloning, it has less than two children.

Chapter 8

The Persistence framework

8.1 Introduction

The Persistence framework is a persistence and metaprogramming facility for generating and persisting models.

The main idea is taken from the Builder design pattern [35]. The persistent form of a model plays the role of the Director for the Builder that (re)constructs the model.

The framework consists of a set of generic builders and a few level 2 generators emitting the specific builders for a given model. The model persistence builder fires a stream of building events; other standard builders are chained to serialize and deserialize the stream using different formats.

The idea of persisting a model using building events is atypical but not entirely new to the field. The Long-Term Persistence API for JavaBeans [27] is standard in Java since version 1.4 and it is based on similar concepts. Unfortunately the JavaBeans API is tied to an XML representation so only small advantages of the approach can be seen.

The default stream serializer builder we provide is a Java code generator. Using it, the persistent form of a model is a Java class that when (compiled and) executed rebuild the original in memory representation of the model.

We have a few other requirements: we want to provide the ability to partition a software system across multiple models and to partition a model across multiple persistence files, and we need a way to define references between models. The problem of mapping language names to file system files has been solved elegantly by programming languages like C♯ and Java. For instance, in Java a class has a package declaration and a name and is

stored in a file with the same name within a hierarchy of directories reflecting the package name; to reference a class you can use its name possibly qualified with its package name and the mapping to the class file in the file system is straightforward. Using Java as a persistence language gives us this solution to the problem.

Another very attractive advantage of this approach is *automatic versioning* through refactoring. All refactoring operations performed over the model update also all persistent model instances.

The Persistence framework supports languages built using multiple models and model fragments.

The standard Java code representation is also a natural representation for meta programs.

8.2 The generic interface of Builders

All builders have to implements the `IBuilder` interface:

```
public interface IBuilder {
  void wSetBuilderContext(IBuilderContext context);
  void wSetEntityContext(IEntityContext context);

  public void wDefault();
  void wEntity();
  void wEntity_();
  void _wEntity();

  void wEntity(EntityDescriptor entity);
  void wEntity_(EntityDescriptor entity);
  void wEntity_(EntityDescriptor entity, int initialCapacity);
  void _wEntity(EntityDescriptor entity);

  void wEntity(EntityDescriptor entity, boolean value);
  void wEntity(EntityDescriptor entity, byte value);
  void wEntity(EntityDescriptor entity, char value);
```

```
  void wEntity(EntityDescriptor entity, double value);
  void wEntity(EntityDescriptor entity, float value);
  void wEntity(EntityDescriptor entity, int value);
  void wEntity(EntityDescriptor entity, long value);
  void wEntity(EntityDescriptor entity, short value);
  void wEntity(EntityDescriptor entity, String value);
  void wEntity(EntityDescriptor entity, Object value);

  void wFeature(int index);
  void wFeature(FeatureDescriptor feature);
  void wFeature(IDirector pattern);
}
```

The methods are designed to support a streaming behavior. We say that an API supports a model *streaming behavior* iff it fires (building) events in a top down traversal order without exposing any detail of the model not yet traversed. We allow the freedom of choosing the children order and of skipping one or more child forcing a default behavior.

Each build method is supposed to operate at the implicit location resulting from the method call order in the stream. No entity objects are passed as arguments of the build methods because they violate the streaming constraint. Only overloaded build methods with primitive values are defined.

We have introduced a naming convention - the underscore symbol - to mark the start or the end of an entity build scope depending on its relative position in a method name. So you can build an entity using either a single build method or a pair of build methods delimiting the start and the end of the stream used to define the children of the entity.

The entity build methods are defined in two flavors: with or without a parameter with an `EntityDescriptor`. If a descriptor is given, the corresponding entity type is used to build the entity, otherwise an entity resolver is used.

The `wFeature` methods are used to select the child to build. The behavior associated to these methods is a change to the implicit location to be used for the following build methods.

For example the following code build a model of a model based editor (Mbed language):

```
IBuilder b = new BuilderContext(new NewModelBuilder());
b.wEntity_(MbedEntityDescriptorEnum.ModelBasedEditor);
  b.wDefault();
  b.wDefault();
  b.wEntity_(MbedEntityDescriptorEnum.ModelComponent);
    b.wEntity(MbedEntityDescriptorEnum.Identifier, "TestModel");
    b.wEntity_(MbedEntityDescriptorEnum.ModelComponent_Body);
      b.wEntity_(MbedEntityDescriptorEnum.ModelEntity);
        b.wEntity(MbedEntityDescriptorEnum.Identifier, "E1");
      b._wEntity(MbedEntityDescriptorEnum.ModelEntity);
      b.wDefault();
      b.wDefault();
      b.wDefault();
      b.wEntity_(MbedEntityDescriptorEnum.ModelEntity);
        b.wEntity(MbedEntityDescriptorEnum.Identifier, "E2");
      b._wEntity(MbedEntityDescriptorEnum.ModelEntity);
    b._wEntity(MbedEntityDescriptorEnum.ModelComponent_Body);
  b._wEntity(MbedEntityDescriptorEnum.ModelComponent);
  b.wDefault();
  b.wEntity(MbedEntityDescriptorEnum.ControllerComponent);
b._wEntity(MbedEntityDescriptorEnum.ModelBasedEditor);
```

The code feels similar to an XML language without any distinction between attributes and children. Note that the behavior of the builder context is defined by the builder strategy argument; in the example a new model is built from scratch.

8.3 The language specific interfaces of Builders

For each language we define a model specific builder interface that defines for each entity a set of builder methods similar to the generic `wEntity` methods but with the entity

name; and it defines for each feature a set of child selection methods similar to the generic `wFeature` methods but with the feature name.

Using the model specific builder `IMbedBuilder` the above example becomes:

```
b.ModelBasedEditor_();
  b.wDefault();
  b.wDefault();
  b.ModelComponent_();
    b.Identifier("TestModel");
    b.ModelComponent_Body_();
      b.ModelEntity_();
        b.Identifier("E1");
      b._ModelEntity();
      b.wDefault();
      b.wDefault();
      b.wDefault();
      b.ModelEntity_();
        b.Identifier("E2");
      b._ModelEntity();
    b._ModelComponent_Body();
  b._ModelComponent();
  b.wDefault();
  b.ControllerComponent();
b._ModelBasedEditor();
```

The resulting code feels much more intentional than in the generic case; furthermore the code introduces only one type dependency with the specific builder of the language used.

8.4 The hierarchy of Builders

Is easy to observe that the `IBuilder` API subsumes the functionalities of a creation API based on the Abstract Factory pattern or based on the Prototype Manager pattern. We

have the added ability to support default parameters in every position, and arguments passed by name and by index in addition to by position.

But the more interesting advantage is the ability to choose the builder behavior with much more freedom given by the support to the streaming behavior.

The Persistence framework includes several abstract and concrete implementations of the `IBuilder` interface as depicted in Figure 8.1.

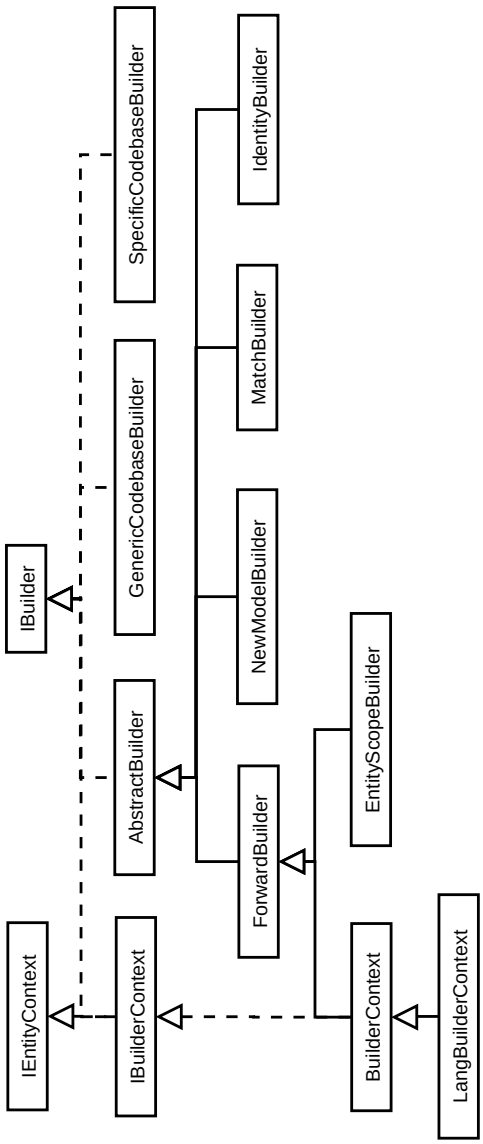

Figure 8.1: Class diagram of the builders hierarchy

There are two codebase builders that serialize a model to the Java director code for

rebuilding it; one of the codebase builder uses the generic `IBuilder` interface while the other uses only model specific builder interfaces. The `NewModelBuilder` build a new model from scratch using the `PrototypeManager` API. The `MatchBuilder` performs a comparison between a model and a stream of build events. The `ForwardBuilder` forwards all build calls to a user definable builder strategy. The `EntityScopeBuilder` forwards all the build calls used to build an entity.

Each builder is associated to an `IBuilderContext` and to one `IEntityContext`. The `IBuilderContext` forwards all the builder method calls to its builder strategy and all entity method calls the the entity context. The `IEntityContext` is an `IEntity` with additional methods for performing one step traversals; it represents the implicit location where building events take place.

Chapter 9

The Editing framework

The Model Based Editing framework (Mbed) allows developers to create a rich graphical editor for languages based on models written with the Modeling framework.

Model based means that the language presentation (user interface) is synchronized with an internal object structure (the model).

Mbed is built on top of the Graphical Editing Framework (GEF)[4] and resulting editors works within the open source development environment Eclipse [2]. A snapshot of the editor appearance is visible in Figure 9.1.

Mbed employs an MVC (model-view-controller) [33] architecture pattern. Specific API and generative support is provided for each architecture part.

The framework is tied to models written with the Modeling framework. To work with legacy models, you have to write adapters or, better, synchronizers (see [30]).

The controller supports all common interaction types with user through standard keyboard and mouse devices including: menu, toolbar, palette, syntax aware context menu, semantic selection and drag and drop.

The view part supports the creation of several kinds of presentations including:

- grammar layouts for rich text oriented languages
- math layouts for mathematical expressions
- tree and graph layouts for diagram oriented languages

The Mbed Editing framework extends the support of model composition provided by the Modeling framework to the presentation level. It is possible to edit in a single editor window a language resulting from the composition of several fragments and legacy

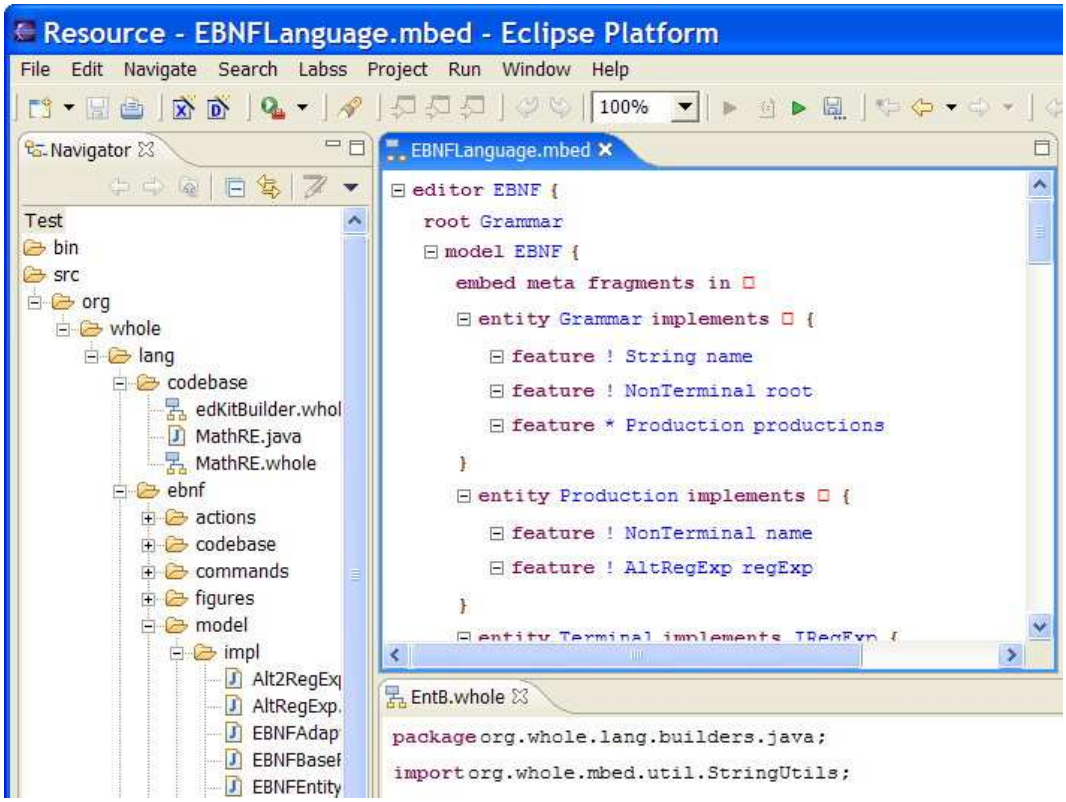

Figure 9.1: Eclipse with Mbed editors

languages mixing different kinds of presentations. For example in Figure 9.2 a base level Java code is mixed with template level XML, EBNF and Java code fragments; each language is represented with its own default concrete syntax and the template level is highlighted with a different background color.

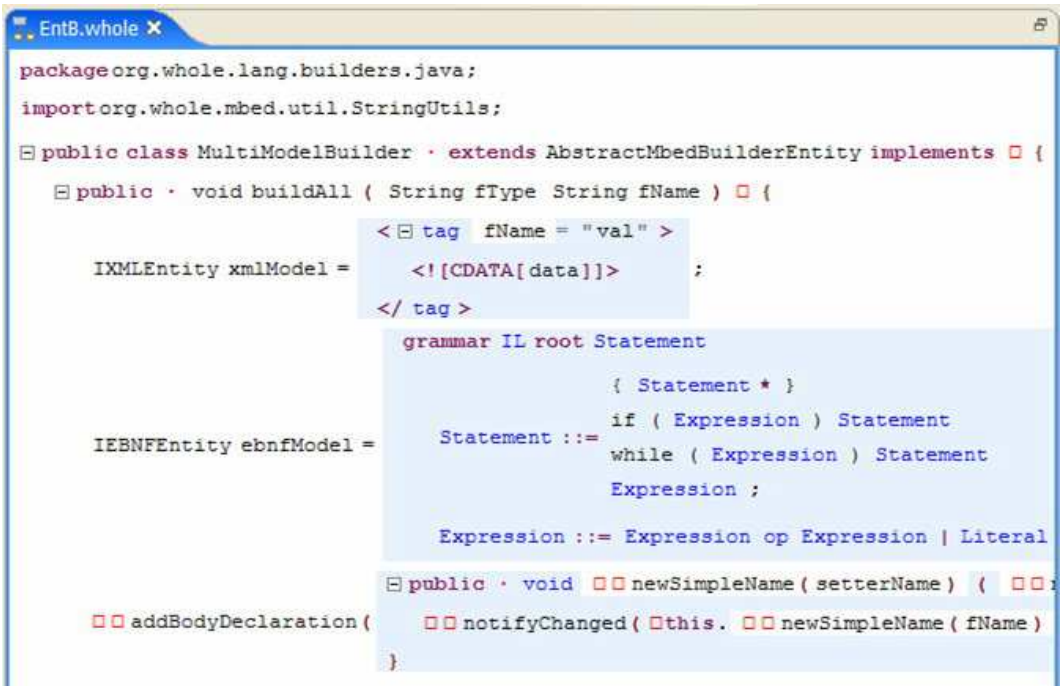

Figure 9.2: Mbed editor with templates

The Mbed Editing framework has required us mainly a big implementation effort; no innovations at the design level are introduced so we do not spend any more time to describe its implementation details here.

Chapter 10

The Java Model Generation framework

The Java Model Generation Framework is a code generation facility for building and manipulating models of Java compilation units.

This framework is built on top of the Eclipse platform JDT DOM/AST API [7]. The JDT API is a factory level no arguments API that provides a direct interface to the back end of the Java compiler of the Eclipse platform and has operations for unparsing, formatting and saving Java compilation units.

Our framework provides two additional sets of APIs and some facilities to speedup Java model building and manipulation. Code level API defines factory methods for creating single Java constructs passing common arguments. Pattern level API for building Java code using pattern abstractions such as: singletons, delegation, constructors, getters/setters, visitors, factories and factory methods. The framework has a facility for writing code fragments representing names and types starting from familiar textual representations. Tied with this facility we provide the ability to add import declarations with control of ambiguous types and automatic selection between simple and qualified names.

The framework has a model based on the Modeling framework API; it defines two model entities: `JavaModelGenerator` and `CompilationUnitBuilder`.

The CompilationUnitBuilder is decoupled from Java compilation units. One CompilationUnitBuilder can manage a set of compilation units but can even produce code delegating to the current compilation unit managed by another CompilationUnitBuilder. This feature is useful for writing specialized builders that add or modify behavior of code produced by others.

A CompilationUnitBuilder is also a collector for names and types declared on compilation units that it manages.

As an example of the Java Model Generation Framework API takes the following code for building the first part of an `EditorKit` class:

```
public class EditorKitBuilder extends CompilationUnitBuilder {
  public EditorKitBuilder(MbedEditorGenerator generator) {
    super(generator);

    addClassDeclaration(generator.editorKitName(),
        AbstractEditorKit.class.getName());

    FieldDeclaration fieldDec = newFieldDeclaration(
        "String", "ID", newLiteral(generator.editorKitName()));
    fieldDec.setModifiers(
        Modifier.PUBLIC | Modifier.STATIC | Modifier.FINAL);
    addBodyDeclaration(fieldDec);

    methodDec = newMethodDeclaration("String", "getId");
    addStatement(methodDec,
        newReturnStatement(newSimpleName("ID")));
    addBodyDeclaration(methodDec);

    methodDec = newMethodDeclaration("String", "getName");
    addStatement(methodDec,
        newReturnStatement(newLiteral(generator.EditorName)));
    addBodyDeclaration(methodDec);


    methodDec = newMethodDeclaration(
        EntityDescriptorEnum.class.getName(), "getModelEntities");
    addStatement(methodDec,
        newReturnStatement(newFieldAccess(
```

```
            generator.specificEntityDescriptorEnumName(), "instance")));
    addBodyDeclaration(methodDec);
```

The following Java code will be produced when the builder is applied to a model called *Mbed*:

```
public class MbedEditorKit extends AbstractEditorKit {
  public static final String ID = "org.whole.lang.mbed.MbedEditorKit";

  public String getId() {
    return ID;
  }
  public String getName() {
    return "Mbed";
  }

  public EntityDescriptorEnum getModelEntities() {
    return MbedEntityDescriptorEnum.instance;
  }
```

As you can see the API of a model based generator, like this framework, hide the output language syntax. We have preferred this kind of generator over the much more commonly used text based generators [9][13][72] because having a concrete syntax is much less important than having the full expressive power of a model generator. We are able to build a single compilation unit in several steps without any order constraint; for instance we can add interfaces, change a visibility modifier, add parameters to a method declaration and so on.

This framework is used to give an executable semantics to our implementation of the Java language and to all metamodeling languages. Now, that the bootstrapping phase of the Whole Platform is terminating, we plan to replace all use of this framework with the builders of our Java language. This way, we can have both a concrete syntax for the template code (when edited with our development environment) and a model of the generated code with all standard services provided by the Whole Platform.

Part III

The Whole Languages

The Whole Languages is the family of languages defined using the Whole Platform. They are the communication languages used to represent the programs knowledge to programmers for reading and editing.

Languages abstractions are packaged as languages. The set of language abstractions is extensible. A few language abstractions are built-in on every language: embedding, sharing and versioning.

Each language is regarded as a language component. New languages can be assembled from language components.

Actual Whole Languages include two sets of languages: metamodeling languages and legacy languages.

The legacy languages are provided because they are popular today and because they have an executable semantics. Metamodeling languages are introduced for defining other languages. The *Languages* DSL is assembled from all other metamodeling languages; each Whole language (including itself) is defined using it.

Chapter 11

Guidelines for Language Design

The Whole Languages have been architected with the following design principles in mind:

Modularity This principle is applied to group related constructs into languages and to separate loose coupled constructs in different languages.

Extensibility Each language is regarded as a component. A language can be extended in two ways: 1) a new language can be defined assembling language components; 2) a new language can be defined reusing (sharing) parts of existing languages.

Layering The layered metamodel architectural pattern is consistently applied to separate concerns across layers of abstraction. In particular each language is defined using a metamodeling language.

Meta-circular The Whole family of languages includes metamodeling languages for defining all the members of the family. In particular, metamodeling languages have a meta-circular definition.

11.1 Specification approach

The Whole family of languages is defined using a metamodeling approach. A metamodel is used to specify the models defining a language. The architecture supports an unlimited number of meta-layers. That is, a model instantiated from a metamodel can in turn be used as a metamodel of another model in a recursive manner. In fact our metamodeling

hierarchy uses only the following three-layers and we think that this number is generally enough:

M0 instances of models

M1 models

M2 metamodels

The models layer defines languages that describe semantic domains. A *model* is an instance of a metamodel meaning that every element of the model is an instance of an element in the metamodel. The Whole languages *Java*, *XML*, *EBNF* are examples of models.

The metamodels layer defines languages for specifying models (including metamodels). A *metamodel* is an instance of a metamodel acting as meta-metamodel. The Whole languages *Models*, *Operations*, *Editors* and *Languages* are examples of metamodels.

A meta-metamodels layer would include languages for specifying metamodels only. Because we have not a metamodel used only to define other metamodels, we prefer to say that we have metamodels *acting* as meta-metamodels without introducing an explicit meta-metalayer in the hierarchy.

For comparison in the OMG metamodeling hierarchy there is one meta-metamodel: the MOF model used to define itself and all other metamodels.

The semantincs of a language is given with a generative plus framework completion approach as depicted in Figure 11.1.

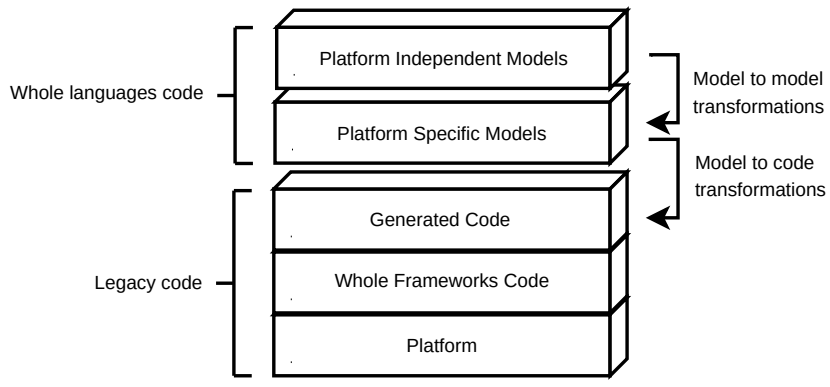

Figure 11.1: Whole Languages semantics architecture

The Whole family of languages include only languages having a model driven representation. Even legacy languages do have a model driven counterpart in the Whole family of languages. For each platform independent language we define one or more model transformations to platform specific languages. For each platform specific language we define a model transformation to code that completes the Whole frameworks for the given platform.

The Whole languages can be defined at a high level of abstraction because they leave to the model transformations and to the frameworks the responsibility to bridge the large gap between the languages and the implementation platform. The frameworks capture all commonalities of abstractions that appear in the models. The model to code transformations use the models to complete extension points in the frameworks.

Chapter 12

Models DSL

We are designing a language for defining the model of a language. Popular programming language design books[62][29][60] and mainstream language specification books[36] define languages by giving a grammar with a (sort of) BNF[21] notation. In fact, the only exceptions to this unwritten rule we know is the UML[50] language and other *modeling* languages having a graphical notation.

So, before considering other specific modeling languages we try to use BNF for modeling and we hilight what is wrong.

The BNF notation is conceived for defining language grammars so, it *over specifies* the model with information (i.e. tokens) tied to a given textual concrete-syntax. In a model driven approach, we want to separate the definition of the abstract-syntax of a language (i.e. the model) because we want to define any number (including none) concrete-syntaxes. Furthermore, we want to use different notation types (textual, graphical).

Even if we do not specify the terminals in a BNF grammar, we get an abstract syntax but not a true model because BNF *under specifies* the language not permitting to give a name to the alternate productions and to each part of a production. We can regard a BNF specification as a model definition with all types declared but without names for concrete types and fields. Having names makes it possible to define polymorphic behavior depending on them.

Also, BNF is not precise enough regarding the specification of multiplicity of nonterminals (sequences vs. sets, ordered or not).

The most popular modeling language is UML; it has a graphical notation and it resolves all points hilighted before. But, we think that a graphical notation alone is too

dispersive to use for defining a big model.

Very recently (November 2004) a language with a textual notation and an expressivity comparable to UML has been proposed: Emfatic[3]. The main advantage of Emfatic is that it represents an entire model in a single source file and it uses a familiar Java-like syntax.

We have chosen for our language a textual syntax that forces to define a name for each type and for each feature.

With respect to cited modeling languages, we have taken a different approach regarding the concept of multiplicity. The *multiplicity* of a feature constrains the lower and upper bounds allowed. The lower bound subsumes the concept of optional/mandatory. Representing the multiplicity in a unique concept hides the difference between simple types and composite types.

We have chosen to separate the two concepts introducing a modifier for optionality and specific composite types. Composite types are also forced to be declared with a name. With uniformity considerations in mind, we have also encapsulated legacy types such as primitive types.

12.1 Language metamodel

The `Model` is the root entity of the language. It is defined by a `name`, two list of types and `modelDeclarations`.

```
entity Model types IModelDeclaration
  ModelName name
  Types templateTypes
  Types noBaseTypes
  ModelDeclarations modelDeclarations

list<TypeName> Types
list<IModelDeclaration> ModelDeclarations
```

The two list of types constrain the metaprogramming ability of the model.

The `templateTypes` is the set of model types that can be staged to a template level and that can embed a template program. Usually a model allows metaprograms written

in any language but only at certain places. The `noBaseTypes` is the set of model types that, when used in a template, cannot be staged to the base level. Usually a model allows all entities (types) at template level to be down staged to the base level of possibly a different language.

A type name can be an entity name or a simple name optionally qualified with a model name.

```
value<String> ModelName
type TypeName
  value<String> EntityName
  value<String> SimpleTypeName
  entity QualifiedTypeName
    ModelName model
    SimpleTypeName type
```

The `Type` entity can be used to introduce a type together with a set of entities typed by it. Using a qualified type name you can extend the qualifier model type with new entities and new type relations.

```
entity Type types IModelDeclaration
  TypeName name
  Types types
  ModelDeclarations? typeDeclarations
```

An `Enum` entity introduces an enumeration type.

```
entity Enum types IModelDeclaration
  SimpleTypeName name
  EnumValues values
list<TypeName> EnumValues
```

Three types of entities are defined: simple, composite and value.

```
entity Entity types IModelDeclaration
  EntityName name
  Types types
  EntityDeclarations entityDeclarations
list<IEntityDeclaration> EntityDeclarations

entity Feature types IEntityDeclaration
  Modifiers modifiers
  TypeName type
  FeatureName name
  IExpression? exp
set<Modifier> Modifiers
enum Modifier
  shared, optional
value<String> FeatureName
```

A collection of entities is explicitely modeled with a `Composite` entity. A `Composite` entity represents a group of entities, known as its elements.

Three modifiers are used to constrain the content of the `Composite`. The *ordered* modifier enforces an order on the elements. The *sorted* modifier enforces the key order on the elements. The *unique* modifier prohibits duplicate elements. Common collection types can be defined combining these modifiers as follows:

collection	ordered	sorted	unique
bag (multiset)			
set			•
list	•		
ordered set	•		•
sorted list	•	•	
sorted set	•	•	•

```
entity Composite types IModelDeclaration
  CompositeModifiers modifiers
  EntityName name
  Types types
  TypeName componentType
  IEntity? keyPattern
set<CompositeModifier> CompositeModifiers
enum CompositeModifier
  final, ordered, sorted, unique
```

Primitive types and other legacy types are wrapped in a `Value` entity.

```
entity Value types IModelDeclaration
  EntityName name
  Types types
  ValueType valueType
value<String> ValueType
```

12.1.1 Type System

Definition 12.1 *The Models DSL* type system *is specified by a tuple* $(\mathsf{E}, \mathsf{T}, :, /)$ *where:*

- E *is the finite set of implementations (entities)*
- T *is the finite set of types*
- $:$ *is the* typing relation
- $/$ *is the* substituibility relation

Notice that the two sets of implementations and types are disjoint.

The *typing relation* is a relation mapping an implementation to a set of types with the following properties (axioms):

total $\forall e \in \mathsf{E}, \exists t \in \mathsf{T} \mid e : t$

compatibility $e : t, t/u \rightarrow e : u$ where $e \in \mathsf{E}$ and $t, u \in \mathsf{T}$

The *substituibility relation* is a relation with the following properties (axioms):

reflexive t/t for all types in T

transitive if t/u and u/v, then t/v where $t, u, v \in T$

We define the following *type equivalence relation* on T:

$$t \equiv u \text{ iff } t/u \text{ and } u/t \text{ where } t, u \in T$$

A circular definition of types is allowed and introduces an equivalence class of types. By definition the equivalence classes are disjoint and in general do not partition the set of types.

The possibility of defining an equivalence class of types is fundamental for allowing flexible language composition. A type system without equivalence classes can only relate two types in an asymmetric way (usual subtyping).

Type rules for Models constructs

A *type* is a marker for grouping types and implementations. Notice that a type does not define a common behavior.

All constructs for entity definition define an implementation and one or more types. A default type with the same name of the implementation is introduced. For instance:

entity e types $t_1, \ldots, t_n$ defines $e : t_e, t_e/t_1, \ldots, t_e/t_n$

type t types $t_1, \ldots, t_n$ defines $t/t_1, \ldots, t/t_n$

Chapter 13

Editors DSL

The Editors domain specific language has been designed for defining rich graphical editors for Whole languages.

13.1 Language metamodel

The definition of an editor strictly follows the *Model View Controller*[33] architectural pattern (MVC): a model, a view and a controller components have to be defined. An editor, of course, has also a name and a root entity.

```
entity Editor
  EditorName editorName
  EntityName rootEntity
  ModelComponent modelComponent
  ViewComponent viewComponent
  ControllerComponent controllerComponent
value<String> EditorName
```

In the following subsections we define each editor component.

13.1.1 The model component

The `Model` type of the Models DSL is used to define the model component type of an editor. Other two types: `EntityName` and `FeatureName` of the Models DSL are used to define two types declared in this language with the same name.

```
type Models.Model types ModelComponent
type Models.EntityName types EntityName
type Models.FeatureName types FeatureName
```

13.1.2 The view component

The view component of an editor defines a set of view declarations for building a presentation model. Notice that a view make no assumptions about the entities it displays.

```
entity ViewComponent types ViewComponent, IViewDeclaration
  ViewDeclarations body
set<IViewDeclaration> ViewDeclarations
```

A *TextualView* is a kind of view using a textual representation. It consists of a list of rows each defining a list of figures.

```
entity TextualView types IViewDeclaration
  ViewName name
  TextualRows rows
value<String> ViewName
list<ITextualRow> TextualRows
entity TextualRow types ITextualRow
  TextualFigures figures
list<ITextualFigure> TextualFigures
```

Three kind of textual figures are available: a place holder for children figures, an indentation figure and a token figure. All child places of a figure are filled with the figures of the children of the presentation model. The token figure permits to define a text literal to display and a category for selecting the rendering font, style and color.

```
type ITextualFigure
  entity ChildPlace
  entity Indent
  entity Token
    Category category
    TokenImage text
enum Category
  keyword, operator, delimiter, parenthesis, identifier, literal
value<String> TokenImage
```

Other kind of views can be defined for the editor but currently they have not a corresponding set of constructs in this domain specific language.

13.1.3 The controller component

The controller component of an editor has three responsibilities: it defines the mapping between the model and the view; it defines the set of prototypes and the set of actions.

The mapping is used to build a representation of the model. The prototypes can be displayed on the editor palette view and on contextual menus. The actions can be displayed on the editor toolbar and on contextual menus.

```
entity ControllerComponent
  MappingDeclarations mapping
  PrototypeDeclarations prototypes
  ActionDeclarations actions
```

The mapping consists of a list of `ControllerPart` declarations each defining a correspondence between a model entity and a view. The `ControllerPart` contains, in presentation order, the list of entity features to be displayed on the view. Recursively the view of a feature is constructed and displayed in the corresponding place holder of the entity view.

```
set<IMappingDeclaration> MappingDeclarations

entity ControllerPart types IMappingDeclaration
  PartName name
  EntityName entityName
  ViewName viewName
  ControllerFeatures features
value<String> PartName
list<ControllerFeature> ControllerFeatures
entity ControllerFeature
  FeatureName name
```

Prototypes define a set of model prototypes each with optional presentation name, icon and description. The prototype is an arbitrary model fragment ranging from a single entity to a full program.

```
set<IPrototypeDeclaration> PrototypeDeclarations

entity Prototype types IPrototypeDeclaration
  IEntity prototype
  TextName? textName
  IconName? iconName
  Description? description
```

Actions define a set of model operations each with optional presentation name, icon and description. The operation call defines the invocation code to use; some information taken from the execution context of the action such as current selection and the model are available for defining operation arguments.

```
set<IActionDeclaration> ActionDeclarations

entity Action types IActionDeclaration
  IOperationCall operationCall
  TextName? textName
  IconName? iconName
  Description? description

value<String> TextName
value<String> IconName
value<String> Description
```

Chapter 14

Languages DSL

14.1 Language metamodel

The *Languages* DSL is able to define a complete Whole language including the model, the behavior and some editors. The language extends three other domain specific languages: *Models*, *Operations*, and *Editors*.

```
language Languages extends Models, Operations, Editors

model
  type Models.Model types IModelDeclaration
  type Operations.Operation types IOperationDeclaration
  type Editors.Editor types IEditorDeclaration
```

A language has a name and can extends some other languages. The language model is defined using the *Models* DSL. The language operations are defined using the *Operations* DSL. The language editors are defined using the *Editors* DSL.

```
entity Language
  LanguageName name
  LanguageNames extendedLanguages
  IModelDeclaration model
  IOperationDeclarations operations
  IEditorDeclarations editors

value<String> LanguageName
set<LanguageName> LanguageNames
set<IOperationDeclaration> IOperationDeclarations
set<IEditorDeclaration> IEditorDeclarations
```

14.1.1 Language extension

Each language is defined in a separate namespace. No automatic merge of types is performed when extending a language. A language extending other languages is able to access to external types using qualified names.

Chapter 15

Legacy languages

We call *legacy languages* all existing languages defined without a model-driven approach. We have chosen four legacy languages for inclusion on the first distribution of the Whole platform: Java, XML, EBNF and Text.

For each language a metamodel has been defined using the Languages DSL so editing of these languages is fully supported in the Whole IDE. This chapter does not cover the metamodel definitions; here we outline the kind of transformations defined for these languages.

15.1 Java

A model of the Java language can be imported from and exported to its standard textual notation. A two-ways model to model transformation has been defined for interoperation with the Java abstract syntax tree representation used on the Eclipse platform Java tools[7]. The compilation of the Java model is supported both using a standard compiler on the textual representation and using the Eclipse Java compiler on the transformed model. The Java model is used both as meta language and as target language for model transformations defining the semantics of other languages.

15.2 XML

A model of the XML[75] language can be imported from and exported to a stream of SAX[12] events. The Extensible Markup Language (XML) is widely used as a model and

notation for several domain specific languages. Near all applications use some XML dialects, for middleware configuration, deployment automation, object-relational mapping specification and so on. We use the XML language for generating all artifacts of a given application written in XML and for importing existing DSL defined using such a notation.

15.3 EBNF

The Backus Naur Form (BNF)[21] is the most widely used formalism to specify grammars for languages. We have provided a model for this language both for importing existing grammars and for generating a grammar suitable for processing with parser generator tools[44][57].

15.4 Text

A model of a generic unstructured textual language can be imported from and exported to a textual file. The text language is used as target for model transformations when the target language is not supported with a model. The Text language is also useful as domain specific language embedded in other languages for supporting string literals or textual templates.

Chapter 16

Conclusions

A new technology for engineering the production of software through the development of languages has been designed, implemented and successfully applied to real programming languages.

The Whole Platform consists of a visual development environment, a generative system and a family of languages.

The platform supports some services unique or improved with respect to competing technologies appeared so far such as: a model history with the ability to execute operations intensionally, a streaming API for building and persisting models, a model driven metaprogramming support with concrete syntax, a pluggable model type system, dynamic scopes for operations and (inherited) properties, and composable families of traversals, iterators, adapters and pattern matchers.

It has been possible to achieve the advanced services outlined above introducing several innovations in the design of the platform architecture and frameworks. A contribute of the thesis is the introduction of design patterns such as Model Context, Resolver Object and Visitable Marker Interface and the improving of others like: Command, Builder, Staged Visitors and Visitor Combinators. Without these design innovations it would not be possible to realize a software platform extensible in a modular way both with new languages and new operations.

These innovations in the solution proposed make possible: to add features to supported languages, to extend the set of supported languages (including the languages used for generators and metamodels), the use of model driven generators, the support of

both batch and live synchronization of models, the mixing of different notations even on a single source code.

The Whole platform implementation has required near one year of development to one person. Statistical details about the implementation are given in Figure 16.1. Note that the Whole column includes all the frameworks code; the Mbed language column includes all metamodeling languages and only the two major legacy languages (Java and XML) are also included in the table.

	Whole	Mbed	Java	XML	tot
Java Lines of Code (LoC)	8356	11973	13060	2947	36336
Java LoC written by hand	8356	1592	601	102	10551
Metamodels LoC	0	78	490	118	686
Number of Packages	23	16	24	13	76
Number of Interfaces	36	14	30	9	89
Number of Classes	267	167	451	154	1039
Number of Methods	2286	1519	5152	1472	10429
Number of Attributes	402	132	333	123	990

Figure 16.1: Whole Platform implementation statistics

All the implementation code is written using the Whole frameworks API or the Whole metamodeling languages; i.e., the Whole Platform is bootstrapped.

Near the 75% of all the implementation code has been generated. The code written by hand includes the implementation of the frameworks and the implementation of Whole languages metamodels and behavior. Note that no metamodeling languages for describing model behavior are available so far.

Ranging from the 8% to 14% of the implementation of each language is written by hand. We plan to introduce metamodeling languages for describing the transformation and the synchronization behavior of a model.

References

[1] The eclipse modeling framework. http://www.eclipse.org/emf.

[2] The eclipse platform. http://www.eclipse.org.

[3] Emfatic language for emf development. http://www.alphaworks.ibm.com/tech/emfatic?Open&ca=daw-hp-pr.

[4] The graphical editing framework. http://www.eclipse.org/gef.

[5] Hypersenses. http://www.d-s-t-g.com/neu/pages/pageseng/et/common/techn_hypsen_frmset.htm.

[6] Java 2 platform, enterprise edition (j2ee). http://java.sun.com/j2ee/.

[7] The java development tools subproject. http://www.eclipse.org/jdt/index.html.

[8] Javaserver faces. http://java.sun.com/j2ee/javaserverfaces/index.jsp.

[9] Jet tutorial. http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html.

[10] Metaborg. http://www.stratego-language.org/Stratego/MetaBorg.

[11] The netbeans ide. http://www.netbeans.org/.

[12] Simple api for xml. http://www.saxproject.org/.

[13] Velocity. http://jakarta.apache.org/velocity/.

[14] Visual editor project. http://www.eclipse.org/vep.

[15] Visula. `http://visula.org/index.html`.

[16] Formal philosophy, selected papers of r. montague. Yale University Press, 1974.

[17] David H. Akehurst and Stuart J. H. Kent. A Relational Approach to Defining Transformations in a Metamodel. In Jean-Marc Jezequel and Heinrich Hussmann, editors, *UML2002 - The Unified Modeling Language: Model Engineeing, Concepts, and Tools*, volume 2460 of *Lecture notes in computer science*. Springer-Verlag, October 2002.

[18] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge. *Multidimensional Declarative Programming*. Oxford University Press, London, 1995.

[19] E. A. Ashcroft and W. W. Wadge. Lucid - a formal system for writing and proving programs. *SIAM Journal on Computing*.

[20] E. A. Ashcroft and W. W. Wadge. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.

[21] John Backus and Peter Naur. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–314, 1960.

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

[23] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: a code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, 2002. `http://asm.objectweb.org/current/asm-eng.pdf`.

[24] Tony Clark, Andy Evans, Paul Sammut, and James Willans. *Applied Metamodelling: A Foundation for Language-Driven Development*. 2004.

[25] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming*, chapter 11. Addison-Wesley, 2000. Intentional Programming.

[26] Sergey Dmitriev. Language oriented programming: The next programming paradigm. Technical report, JetBrains, 2004. `http://www.onboard.jetbrains.com/articles/04/10/lop/mps.pdf`.

[27] JSR-57 expert group. Jsr-000057 long-term persistence for javabeans specification, 2001. `http://www.jcp.org/aboutJava/communityprocess/review/jsr057/`.

[28] A. A. Faustini and W. W. Wadge. Intensional programming. In J.C. Bourdeaux, B.W.Hamill, and R.Jernigan, editors, *The Role of Languages in Problem Solving 2*, North-Holland, 1987. Elsevier Science Publishers.

[29] Raphael A. Finkel. *Advanced Programming Language Design*. Addison-Wesley Publishing Company, 1996.

[30] Nathan Foster, Michael Greenwald, Jonathan Moore, Benjamin Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *POPL 2005*, 2005.

[31] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Publishing Company, 2000.

[32] Frank Budinsky, *et al. The Eclipse Modeling Framework*. Addison-Wesley Publishing Company, 2003.

[33] Frank Buschmann, *et al. Pattern-Oriented Software Architecture: A System of Patterns*. Wiley Publishing Inc., Indianapolis, IN, 1996.

[34] Frank Buschmann, *et al. Pattern Languages of Program Design 3*. Addison-Wesley, 1998.

[35] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[36] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley Publishing Company, 2000.

[37] Mark Grand. *Patterns in Java Volume 1*. Wiley Publishing Inc., Indianapolis, IN, 1998.

[38] Jack Greenfield and Keith Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley Publishing Inc., Indianapolis, IN, 2004.

[39] Gregor Kiczales, *et al.* Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 1997.

[40] Gregor Kiczales, *et al.* An overview of aspectj. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2001.

[41] A. Hejlsberg and S. Wiltamuth. *The C# Programming Language*. Addison-Wesley Publishing Company, 2004.

[42] T. Kuipers and J. Visser. Object-oriented tree traversal with jjforester. *Electronic Notes in Theoretical Computer Science*, 44, 2001.

[43] Akos Ledeczi, Miklos Maroti, and Peter Volgyesi. The generic modeling environment. Technical report, Institute for Software Integrated Systems, Vanderbilt University, 2004.

[44] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, Inc., 1992.

[45] S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling, and K. Tan. Generative design patterns, 2002. `citeseer.nj.nec.com/593908.html`.

[46] Robert C. Martin. *Acyclic Visitor*, pages 93–104. Addison-Wesley, 1998.

[47] Robert C. Martin. *The Visitor Family of Design Patterns*. Prentice Hall, 2002.

[48] Fabian Buttner Oliver. Digging into the visitor pattern. `citeseer.ist.psu.edu/632686.html`.

[49] OMG. Mda guide version 1.0.1, 2003. `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.

[50] OMG. Omg unified modeling language specification, v1.5, 2003. `http://www.omg.org/cgi-bin/doc?formal/03-03-01`.

[51] OMG. Uml 2.0 diagram interchange specification, 2003. http://www.omg.org/cgi-bin/doc?ptc/03-09-01.

[52] OMG. Uml 2.0 infrastructure specification, 2003. http://www.omg.org/cgi-bin/doc?ptc/03-09-15.

[53] OMG. Uml 2.0 superstructure specification, 2003. http://www.omg.org/cgi-bin/doc?formal/03-08-02.

[54] OMG. Xml metadata interchange (xmi) specification, 2003. http://www.omg.org/cgi-bin/doc?formal/03-05-02.

[55] OMG. Meta object facility (mof) 2.0 core specification, 2004. http://www.omg.org/cgi-bin/doc?ptc/03-10-04.

[56] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 19–21 1998.

[57] Terence Parr. Antlr another tool for language recognition, 1989-2004. http://www.antlr.org.

[58] J. Plaice and Joey Paquet. Introduction to intensional programming. In *Intensional Programming I*, pages 1–14, Singapore, 1996.

[59] J. Plaice and W. W. Wadge. A new approach to version control. *IEEE Transactions on Software Engineering*, 19(3):268–276, 1993.

[60] Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages Design and Implementation*. Prentice Hall, 2000.

[61] Dirk Riehle. Composite design patterns. In *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, pages 218–228, 1997.

[62] Michael L. Scott. *Programming language pragmatics*. Morgan Kaufmann Publishers Inc., 2000.

[63] Charles Simonyi. The death of computer languages, the birth of intentional programming. Technical Report MSR-TR-95-52, Microsoft Research, 1995.

[64] Charles Simonyi. Intentional programming - innovation in the legacy age. In *IFIP WG 2.1 meeting*, 1996.

[65] Mark van den Brand, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter Olivier, Jeroen Scheerder, Jurgen Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of Compiler Construction 2001 (CC 2001)*, LNCS. Springer, 2001.

[66] Joost Visser. Visitor combination and traversal control. In *Proceedings of the 16th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'01)*, pages 270–282, November 2001.

[67] Joost Visser. *Generic Traversal over Typed Source Code Representations*. PhD thesis, University of Amsterdam, 2003.

[68] John Vlissides. Composite design patterns (they aren't what you think). *C++ Report*, 1998.

[69] John Vlissides. Tooled composite. *C++ Report*, 1999.

[70] John Vlissides. Visitor in frameworks. *C++ Report*, 11(10):40–46, 1999.

[71] John Vlissides and Richard Helm. Compounding command. *C++ Report*, pages 47–52, 1999.

[72] W3C. Xsl transformations (xslt) 1.0. Technical report, W3C, 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`.

[73] W3C. Document object model (dom) level 2 traversal and range specification, 2000. `http://www.w3.org/TR/2000/REC-DOM-Level-2-Traversal-Range-20001113/`.

[74] W3C. Document object model (dom) level 3 core specification, 2004. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`.

[75] W3C. Extensible markup language (xml) 1.1. Technical report, W3C, 2004. `http://www.w3.org/TR/2004/REC-xml11-20040204/`.