

Modeling Object-Oriented Software for Reverse Engineering and Refactoring

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Sander Tichelaar

von den Niederlanden

Leiter der Arbeit: Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Modeling Object-Oriented Software for Reverse Engineering and Refactoring

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Sander Tichelaar
von den Niederlanden

Leiter der Arbeit: Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 14. Dezember 2001

Der Dekan:
Prof. Dr. P. Bochsler

Abstract

The increased popularity of the object-oriented paradigm has also increased the interest in object-oriented reengineering. First of all, object-oriented software systems suffer from similar maintainability problems as traditional procedural systems, displaying the need for reengineering techniques tailored to deal with object-oriented code. Secondly, the increased importance of iterative development processes make reengineering techniques valuable in forward engineering, and thus for all paradigms that software is developed in.

Reengineering requires tool support to deal with the large amounts of information and the wide variety of tasks to be performed. An important consideration in building tool environments for reengineering is what information must be provided and how this information is modelled. Design choices have a considerable impact not only on the ability to support reengineering tasks, but also on issues such as scalability and tool interoperability. Several metamodels exist that model software for the purposes of reengineering. However, they generally lack a discussion of the relevance of information for reengineering and the trade-offs of modeling alternatives.

This thesis presents FAMIX, a language-independent metamodel for modelling object-oriented software for reengineering purposes. We discuss the exact contents of the metamodel, including its relevance for reengineering and how the metamodel supports the different object-oriented languages through its language-independent core. We also discuss the infrastructural design decisions of FAMIX by placing it into a design space for infrastructural aspects of reengineering repositories and metamodels. The design space presents multiple interdependent aspects, their design alternatives and how these impact issues such as scalability, extensibility and information exchange.

We validate the ability of FAMIX to support reengineering on a language-independent level in two ways. First, we present Moose, a reengineering tool environment with a repository based on FAMIX. Moose serves as a foundation for multiple reengineering tools and has been applied to reverse engineer several large industrial case studies. Secondly, we define a set of fifteen low-level refactorings in terms of the information available in FAMIX. Refactoring requires sufficient, complete and 100% correct information as well as a clear interpretation of the supported languages in the language-independent core of the metamodel, in order to correctly perform transformations on the language-specific code level. As such the refactorings provide an in-depth validation of the language independence of FAMIX.

Acknowledgements

First of all, I would like to thank Prof. Oscar Nierstrasz for giving me the opportunity to work in the SCG and for the support I have received through the years. Then I am much indebted to Serge Demeyer and Stéphane Ducasse for their everlasting support and encouragement, as well as the work that we have done together on FAMIX, Moose and the reengineering metamodel design space. Without them this thesis would have been impossible.

I also thank the other members of the SCG, for the good and fruitful time on the work floor, for the careful reading of drafts of this thesis, and also for the concerts, dinners, and the late night whiskies in town: Franz Achermann, Gabriela Arévalo, Juan Carlos Cruz, Isabelle Huber, Michele Lanza (another major Moose contributor), Markus Lumpe, Robb Nebbe (for his work on the Ada extension to FAMIX), Matthias Rieger, Therese Schmid, Jean-Guy Schneider and Roel Wuyts.

I would also like to mention some other people I was fortunate to work with in the recent years. I thank all the FAMOOS people, in particular Holger Bär for his work on the C++ extension to FAMIX and Claudio Riva for his work on FAMIX-based CDIF and XMI. I also thank the students who chose to work with us, in particular Andreas Schlapbach and Michael Freidig for their work on XMI and Lukas Steiger and Pietro Malorgio for their unceasing comments on Moose and FAMIX.

Furthermore I am grateful to Prof. Theo D'Hondt for being on my PhD committee.

Finally, I want to thank my parents, for always supporting me and inspiring me to bring out the best in myself, in education and everything else. My family and (other) friends, and most of all Claudia, I thank for their love and support. Many thanks.

Sander Tichelaar

November 2001

Table of Contents

CHAPTER 1 Introduction.....	1
1.1 Modelling software to support reengineering tools	2
1.2 Contributions	3
1.3 Roadmap.....	4
CHAPTER 2 State-of-the-Art in Reengineering Metamodels and Tools	5
2.1 Definitions in Reengineering	5
2.2 Object-Oriented Reengineering	7
2.3 Reengineering Tools and Environments.....	8
2.3.1 Actual Reengineering Environments	9
2.3.2 Metamodels for Reengineering.....	10
2.4 Refactoring and Code Reorganisation	11
2.5 Discussion.....	12
2.6 Conclusion	14
CHAPTER 3 A Design Space for Reengineering Tool Infrastructures	15
3.1 Introducing the design space.....	16
3.1.1 Scenario.....	16
3.1.2 Infrastructural issues summarised.....	16
3.1.6 Design Space in a Nutshell	17
3.2 Language/Paradigm Axis	20
3.3 Level of Detail Axis	21
3.4 Multiple Models Axis.....	22
3.5 Grouping Axis	22
3.6 Extensibility Axis	23
3.6.1 Adding new entities to a metamodel.....	23
3.6.2 Adding attributes to existing entities.....	23
3.6.3 Annotating entities	24
3.6.4 Metametamodel extensibility limits	25
3.7 Incremental Loading Axis	25
3.8 Storage Medium Axis.....	26
3.9 Exchange Format Structure Axis	27
3.9.1 Nested, chunk and flat formats	28
3.9.2 Discussion	29
3.10 Entity Reference Axis.....	30

3.10.1 Unique identifiers	30
3.10.2 Unique naming scheme	31
3.10.3 Analysis	32
3.11 Metametamodeling Axis	33
3.12 Conclusion	34
CHAPTER 4 FAMIX, a Language-Independent Metamodel for Modeling Object-Oriented Software	35
4.1 Requirements	36
4.2 Overview of the FAMIX core	36
4.3 Extensibility	39
4.4 Multiple language support	40
4.4.1 General multi-language design decisions	40
4.4.2 Language mappings and extensions	41
4.5 Reference Schema	43
4.6 Support for information Exchange	44
4.7 Metametamodeling	46
4.8 Why not UML?	46
4.9 Conclusion	48
CHAPTER 5 The Moose Reengineering Environment	51
5.1 Requirements for a Reengineering Environment	51
5.2 Architecture	52
5.3 Querying and Navigation	54
5.3.1 Programming Queries	54
5.3.2 Querying and navigating using the Moose Explorer	56
5.4 Metrics and other analysis support	57
5.5 Grouping	57
5.6 Moose Refactoring Engine	58
5.7 Information Exchange and Tool Integration	58
5.7.1 Information Exchange with CDIF and XMI	58
5.7.2 Tool Integration Framework and Tools	59
5.8 Industrial Case Studies	60
5.9 Discussion	61
5.9.1 Observations	62
5.9.2 The requirements revisited	63
5.10 Conclusion	64
CHAPTER 6 Language-Independent Refactoring	65
6.1 Language subsets and mappings	66
6.1.1 Language subsets	66
6.1.2 Language mappings	67
6.2 The Refactoring Template	68
6.3 The refactorings in detail	68
Add Class (classname, package, superclasses, subclasses)	69
Remove Class (class)	71
Rename Class (class, new name)	74
Add Method (name, class)	76

Remove Method (method)	78
Rename Method (method, new name)	80
Pull Up Method (method, superclass)	83
Push Down Method (method)	91
Add Parameter (name, method)	97
Remove Parameter (parameter)	101
Add Attribute (name, class)	103
Remove Attribute (attribute)	105
Rename Attribute (attribute, new name)	107
Pull Up Attribute (attribute, superclass)	109
Push Down Attribute (attribute)	113
6.4 Validation	115
6.5 Discussion	115
CHAPTER 7 The Moose Refactoring Engine	119
7.1 Architecture	119
7.2 Validation	122
7.2.1 A non-trivial refactoring sequence on a toy banking system	122
7.2.2 Experiments on Moose and JUnit	124
7.3 Discussion	125
CHAPTER 8 Conclusion and Future Work	127
APPENDIX A Table of Refactorings	131
APPENDIX B The FAMIX 2.1 specification	137
2.1 Overview	137
2.1.1 Basic Data Types	137
2.1.2 Unique Naming Conventions	139
2.1.3 Level of Extraction	139
2.2 Definition of FAMIX	140
2.2.1 The abstract part: Object, Entity and Association	140
2.2.2 Model	143
2.2.3 Package	144
2.2.4 Class	145
2.2.5 BehaviouralEntity Hierarchy	146
2.2.6 BehaviouralEntity	146
2.2.7 Method	147
2.2.8 Function	148
2.2.9 StructuralEntity Hierarchy	149
2.2.10 StructuralEntity	150
2.2.11 Attribute	150
2.2.12 GlobalVariable	151
2.2.13 ImplicitVariable	152
2.2.14 LocalVariable	153
2.2.15 FormalParameter	154
2.2.16 InheritanceDefinition	154
2.2.17 Access	155

2.2.18 Invocation	156
2.2.19 Argument Hierarchy	158
2.2.20 ExpressionArgument	158
2.2.21 AccessArgument	158
2.3 Miscellaneous	160
2.3.1 CDIF Multi-valued String Attributes	160
APPENDIX C Smalltalk Extension to FAMIX	161
3.1 Extending FAMIX	161
3.2 Modified classes	161
3.2.1 Model (interpreted)	161
3.2.2 Package (interpreted)	162
3.2.3 Class (interpreted and extended)	162
3.2.4 BehaviouralEntity (interpreted and extended)	163
3.2.5 Method (interpreted and extended)	163
3.2.6 StructuralEntity (interpreted and Extended)	164
3.2.7 Attribute (interpreted)	164
3.2.8 GlobalVariable (interpreted)	165
3.2.9 ImplicitVariable (interpreted)	165
3.2.10 LocalVariable (interpreted)	165
3.2.11 FormalParameter (interpreted)	165
3.2.12 InheritanceDefinition (interpreted)	166
3.2.13 Invocation (interpreted)	166
3.3 Miscellaneous	166
3.4 Pending Issues	167
APPENDIX D Java Extension to FAMIX	169
4.1 Extending FAMIX	169
4.2 Modified classes	169
4.2.1 Model (interpreted)	169
4.2.2 Package (interpreted)	170
4.2.3 Class (interpreted and extended)	170
4.2.4 BehaviouralEntity (interpreted and extended)	171
4.2.5 Method (interpreted and extended)	171
4.2.6 StructuralEntity (interpreted and Extended)	172
4.2.7 Attribute (interpreted)	173
4.2.8 ImplicitVariable (interpreted)	173
4.2.9 LocalVariable (interpreted)	174
4.2.10 FormalParameter (interpreted)	174
4.2.11 InheritanceDefinition (interpreted)	174
4.2.12 Invocation (interpreted)	175
4.3 New classes	175
4.3.1 TypeCast	175
4.4 Miscellaneous	176
4.5 Pending Issues	176
Bibliography	177

CHAPTER 1

Introduction

The ability to reengineer legacy systems has become a vital matter in today's software industry. Systems easily get hard to maintain and adapt. Requirements change, platforms change and if a system is not properly maintained, its usefulness decays over time [LB85]. The law of *software entropy* dictates that even when a system starts off in a well-designed state, requirements evolve and customers demand new functionality, frequently in ways the original design did not anticipate. Additionally, new technology makes this system progressively less valuable. It needs to support new platforms, embrace emerging standards and leverage better understood technological advancements. A complete redesign may not be practical, and a system is bound to gradually lose its original clean structure and deform into a Big Ball of Mud [FY00] [BMMM98]. Typically such a system represents some value to its owner, for instance, in the task it performs or in the knowledge it represents. However, bringing the system back into shape is very costly due to the poor state it is in.

This is where reengineering — the examination and the alteration of a subject system to reconstitute it in a new form [CC90] — comes in. Reverse engineering techniques (the examination) help clarifying the structure by extracting information and providing high-level views on the subject system, while refactoring (the alteration) modifies software to improve its simplicity, understandability, flexibility or robustness [FBB⁺99] [Bec99]. Once the software is better understood and in a better shape, it is ready to fulfil its new requirements.

Reengineering is not restricted to legacy systems. In recent years the globalisation of markets and the resulting increasing competition make that business environments — and thus the requirements to the software that supports these environments — change ever faster. To deal with these rapid changes, iterative development paradigms have emerged that support constant adaptation of the software, rather than a single waterfall development cycle [Boe88] [Bec99]. The examination and alteration of the software happens much earlier in the life cycle and is repeated for every iteration. Reengineering essentially becomes part of forward engineering.

Originally most of the reengineering efforts were focused on systems written in traditional procedural programming languages such as COBOL, Fortran and C. But following the increased popularity of object-oriented programming a growing demand for reengineering object-based systems has emerged in recent years [Cas98] [WH92] [DD99]. Although sometimes thought of as a silver bullet to software development,

the mere application of object-oriented techniques is not sufficient to deliver flexible and adaptable systems. Applying the technology correctly requires knowledge and experience which development teams not always possess. On top of that, hybrid programming languages such as C++ and Ada often prevent programmers from making the necessary paradigm shift away from procedural programming towards object-oriented programming. Last but not least, object-oriented software systems suffer from the same entropy effects as any other software system.

As a consequence reengineering has become vital technology also in the area of object-oriented software development. The technical details of and solutions to the problems may differ from other paradigms, the source and symptoms are the same.

1.1 Modelling software to support reengineering tools

Reengineering large industrial software systems is impossible without appropriate tool support. First of all, there is the scalability issue (millions of lines of code are the norm rather than the exception) but there is also the extra complexity of supporting and combining multiple tools with a wide variety of tasks (standard forward engineering techniques must be combined with reverse- and reengineering facilities). The need for tool support in reengineering is reflected by the numerous tools and tool prototypes available in the reengineering research community [AT98] [SS00].

To be able to reason about software systems, tools need a common information base, a repository, that provides them with the information required for reengineering tasks. The properties of the repository, and thus of the complete environment, are highly influenced by the metamodel that describes what and in which way information is modelled. The metamodel not only determines if the right information is available to perform the intended reengineering tasks, but also influences issues such as scalability, extensibility and information exchange.

There are a number of existing metamodels for representing software. Several of those are aimed at object-oriented analysis and design (OOAD), the most notable example being the Unified Modeling Language (UML) [OMG99]. However, these metamodels represent software at the design level. Reengineering requires information about software *at the source code level*. The starting point is the software itself demanding for a precise mapping of the software to a model rather than a design model that might have been implemented in lots of different ways.

In the reengineering research community several metamodels exist that model the software itself. They are aimed at procedural languages (Bauhaus [CEK⁺00]), object-oriented/procedural hybrid languages (TA++ [Let98], Datrix [LLB⁺98]) and systems with multiple paradigms [LS99]. Most metamodels support multiple languages, either implicitly or explicitly. What is generally lacking in these metamodels is a discussion of the relevance of the represented information for reengineering and a discussion about the trade-offs of modeling alternatives. There are some exceptions [Kos00] [LLB⁺98], but especially in the area of reengineering object-oriented software no comprehensive list of design choices and clear semantics of the metamodel exist. This brings us to following research question:

*how can we model object-oriented software to adequately
support reengineering tools*

This thesis answers this question by specifying a language-independent metamodel for object-oriented software, which is called FAMIX. It includes an in-depth discussion of the ability of this metamodel to support reverse engineering and refactoring on a language-independent level, as well as the design decisions

that influence its scalability and tool integration properties. The main advantage of the language independence is that tools that are based on the metamodel, can be applied without adaptation on systems in all supported implementation languages.

The thesis starts with a design space for the infrastructural aspects of building software metamodels and repositories. Infrastructural aspects are the design aspects that deal with how the information is organized and stored rather than the exact contents of a metamodel. For every aspect the design space lists the implementation options with trade-offs, as well as the interdependencies with other aspects. Although the thesis focuses on the problems and solutions we have encountered in reengineering object-oriented systems, the design space is applicable to reengineering environments for any programming paradigm. Following the design space we present our instance of such a metamodel, namely FAMIX. Our solution to the specific requirements for this metamodel — support for mid-size to large industrial software systems in multiple object-oriented implementation languages¹ — are discussed in detail.

We validate the ability of FAMIX to support reengineering on a language-independent level in two ways. First we have developed Moose, a tool environment for reengineering object-oriented systems. It has a repository based on FAMIX and has been used as a foundation for multiple reengineering tools. Moose together with these tools has been used to successfully reverse engineer several large industrial software systems. The second validation consists of the definition of a set of fifteen refactorings in terms of FAMIX and their implementation as part of the Moose reengineering environment. Refactorings are behaviour-preserving code transformations [FBB⁺99]. Because they change the original software system rather than merely analyse it, they require sufficient, complete and 100% correct information. The result of a refactoring may never introduce any errors, while reverse engineering techniques typically are not affected by slightly incomplete or incorrect information [MNGL98] [Bis92]. Furthermore, refactoring needs a clear interpretation of the language-independent model information to perform the correct transformations on the (language-specific) source code level. Our analysis shows that it is possible to abstract the refactoring definitions for the greater part from the underlying implementation languages. As such the refactoring analysis and its implementation provide an in-depth validation of the language independence of FAMIX and especially how it maps the specific implementation languages to its language-independent core.

1.2 Contributions

The contributions of this thesis can be summarized as follows:

- A design space for infrastructural aspects of metamodels and repositories for reengineering. It makes explicit what the relevant aspects are, how they interrelate, what implementation options can be chosen and what the trade-offs of these options are.
- FAMIX, a language-independent metamodel for modelling object-oriented software. It makes explicit what information about object-oriented software is relevant for reengineering and how multiple languages are modelled in a common way to enable the reuse of analysis and tools.
- An analysis of fifteen low-level refactorings for Java and Smalltalk in the context of language independence. The analysis is based on the FAMIX metamodel and shows to which extent the refactorings can be abstracted from the implementation languages.

1. These requirements have been mostly determined by the FAMOOS project [DD99] under which the major part of this work has been realised. FAMOOS was a European Esprit research project (no. 21975) aimed at the transformation of object-oriented legacy systems into framework-based applications.

1.3 Roadmap

The rest of this thesis is organised as follows. It starts with an overview of the state-of-the-art in reengineering metamodels and tools (chapter 2). Then it presents the infrastructure design space for reengineering metamodels and repositories (chapter 3). Following it presents FAMIX (chapter 4) and its implementation in Moose (chapter 5). Afterwards we present the refactoring analysis (chapter 6) as well as its realisation in the refactoring engine of Moose (chapter 7). We end with a conclusion.

CHAPTER 2

State-of-the-Art in Reengineering Metamodels and Tools

This thesis is about modeling object-oriented software in a language-independent way for the purpose of reengineering. The problem is not new. Many reengineering tools exist and they all need to work with models of the software they act upon. This chapter presents the state-of-the-art in reengineering metamodels and tools. It starts with a set of definitions to set the vocabulary (section 2.1). Subsequently it discusses object-oriented reengineering in particular (section 2.2) and gives an overview of tool environments for reengineering including the metamodels they are built upon (section 2.3). Afterwards the chapter presents the state-of-the-art in the particular case of refactoring (section 2.4). Finally, we discuss the presented tools and metamodels and point to open problems in the area of modeling software for reengineering.

2.1 Definitions in Reengineering

This section presents definitions for reengineering and related terms. It is largely based on the taxonomy by Chikofsky and Cross [CC90]. We start with the definition of reengineering:

“Reengineering is the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.”
[CC90]

As stated by the definition, reengineering consists of two main activities, namely the *examination* and the *alteration* of a subject system. More formal terms for these activities are reverse engineering and forward engineering:

“Reverse engineering is the process of analysing a subject system to (i) identify the system’s components and their relationships and (ii) create representations of the system in another form or at a higher level of abstraction.” [CC90]

“Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.” [CC90]

The adjective ‘forward’ in ‘forward engineering’ is mainly used to distinguish traditional software engineering from reverse and reengineering. Figure 2.1 illustrates the three notions and their relationships [Cas98]. Reverse engineering is used to create models, i.e., higher level views, of an existing software system. Goals are to understand a system, document it or detect problems. Conversely, forward engineering is about moving from high-level views of requirements and models towards concrete implementations. Reengineering is a combination of the two, namely transforming concrete implementations to other concrete implementations. As in forward engineering, reengineering is driven by requirements (‘New Requirements’ in Figure 2.1 as opposed to the original requirements for the system). The requirements focus the reengineering effort on the relevant parts of the targeted legacy system.

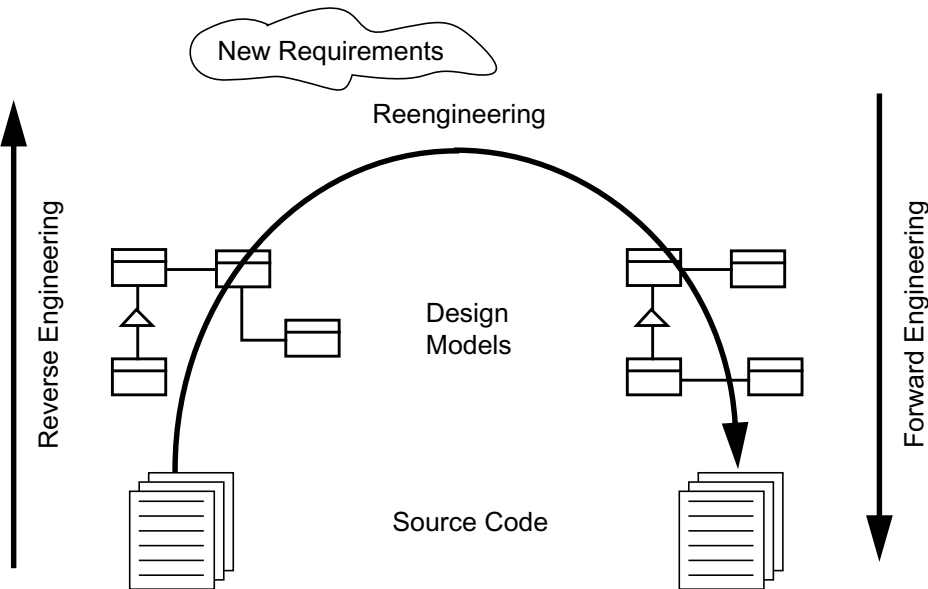


Figure 2.1 The reengineering lifecycle

The process to get from a legacy system to a reengineered system is described by Casais [Cas98] in a five-step reengineering life-cycle that can be mapped to Figure 2.1:

1. Model capture (documenting and understanding the design of the legacy system),
2. Problem detection (identifying violations of flexibility and quality criteria),
3. Problem analysis (selecting a software structure that solves a design defect),
4. Reorganization (selecting and applying the optimal transformation of the legacy system) and
5. Change propagation (ensuring the transition between different software versions).

The main difference between forward engineering and reengineering is that reengineering starts from an existing implementation. Consequently, for every change to a system the reengineer must evaluate whether (parts of) the system need to be *restructured* (or *refactored*) or if they should be implemented anew from scratch. According to Chikofsky and Cross, restructuring generally refers to source code translation (such as the conversion from unstructured spaghetti code to structured, or goto-less code), but it may also entail transformations at the design level. This is their definition:

“*Restructuring* is the transformation from one representation form to another at the same relative abstraction level, while preserving the system’s external behavior.” [CC90]

Refactoring is merely a special kind of restructuring, namely within an object-oriented context and focused on the level of code. In his catalog of refactorings Martin Fowler defines it as follows:

“*Refactoring* is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.” [FBB⁺99]

Typical goals of refactoring are to improve the simplicity, understandability, flexibility or performance [Bec99]. Section 2.4 describes refactoring and the state-of-the-art in tools and research in more detail.

Reengineering versus Software Maintenance

It may be hard to tell the difference between software *reengineering* and *software maintenance*. The ANSI/IEEE standard 729-1983 defines software maintenance as

“*Software maintenance* is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment”

This definition focuses on changes *after delivery* of a product. It does not cover changes that implement new functionality. However, often *any* change after delivery, i.e., also implementing new functionality, is considered maintenance. Indeed, Sommerville categorises maintenance in three kinds [Som96], namely

- *corrective* maintenance, i.e, fixing reported errors,
- *adaptive* maintenance, i.e, adapting a system to a new environment (e.g., platform or operating system),
- *perfective* maintenance, i.e, implementing new functional or non-functional requirements.

However, as already argued by Turski in 1981, adding new features does not conceptually fit the term maintenance (adding a new wing to a building is not considered maintenance either) [Tur81]. Consequently, the term ‘perfective maintenance’ is a *contradictio in terminis*. It is also not covered by the above ANSI/IEEE definition.

Another problem is that the use of ‘after delivery’ in the definition is outdated. It is a backlog of the waterfall model that decomposes development into a single directed flow of activities, the last one being maintenance. Such a sequential decomposition of activities prohibits the necessary interaction and feedback required by software development and addressed by more modern development models such as the spiral model [Boe88] and eXtreme Programming [Bec99].

Hence, reengineering goes beyond software maintenance. Reengineering techniques can be used to perform maintenance tasks such a bug fixing or any other adaptation within its original feature set. However, reengineering can also be applied to change systems in a more considerable way, i.e., to add new functionality.

2.2 Object-Oriented Reengineering

Although the term ‘legacy system’ is often associated with systems in assembler or procedural languages such as Fortran and Cobol, object-oriented systems suffer from similar problems. The Laws of Lehman

[LB85] [Leh96] tend to be true for systems in any language. This is supported by facts: object-oriented legacy applications exist even in relatively young languages such as Java [DD99]. Furthermore, reengineering techniques are starting to become part of modern software development processes. Hence, also in that context reengineering techniques are relevant for systems implemented in languages other than the traditional COBOL, Fortran or C.

Apart from common legacy problems such as duplicated functionality and insufficient and outdated documentation, reengineering object-oriented languages presents its own set of problems [WH92]. We list here some of the most preeminent:

- Polymorphism and late binding make traditional tool analysers like program slicers inadequate. Data-flow analysers are more complex to build especially in presence of dynamically typed languages.
- Incremental class definition, together with the dynamic semantics of `self` or `this`, make applications more difficult to understand.
- Dynamically typed languages such as Smalltalk, on the one hand, make the analysis of applications harder because types of variables are implicit and tool support is needed to infer them. On the other hand, statically typed languages such as C++ and Java force the programmer to explicitly cast objects, which leads to applications that are less maintainable and require more effort to be changed.

Apart from the above list, common code-level problems occurring in object-oriented legacy systems are often due to misuse or overuse of object-oriented features, such as the misuse of inheritance or the violation of encapsulation.

2.3 Reengineering Tools and Environments

This section discusses existing reengineering tools and tool environments. Too many tools exist to describe them all. Several surveys have been compiled [AT98] [SS00], but these do not provide exhaustive lists either. We focus on the tools that are of interest in the context of modeling software for reengineering.

All tool sets have basically the structure depicted in Figure 2.2. There is a repository to store data about software system. There are parsers to extract information from source code and model importers to read in models stored using an exchange format. The tools themselves, browsers, visualisers, etc., use the repository as their information base.

Some tools focus on providing an infrastructure that enables multiple tools to perform their reengineering tasks as well as interoperate with other tools. These are what we call the tool environments, or tool platforms. Other tools focus on one special task, for instance, visualisation of architectures. These tools still need a repository and import/export facilities, but the role of the repository is less central than in full-blown tool environments. Section 2.3.1 discusses existing tools and tool environments.

The properties of the repository, and thus of the complete environment, are highly influenced by the metamodel that describes what information the repository contains. The metamodel not only determines if the right information is available to perform the intended reengineering tasks, but also influences issues such as scalability, extensibility and information exchange. Section 2.3.2 introduces some of the metamodels in existing tools.

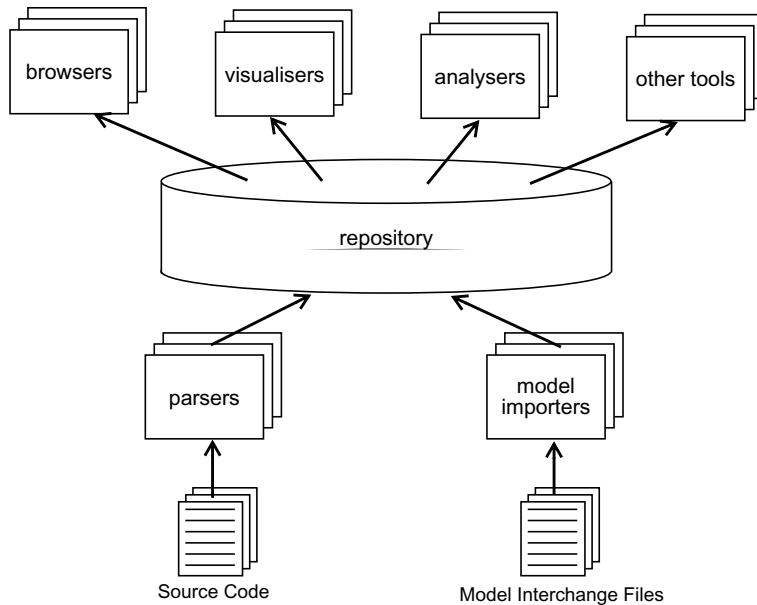


Figure 2.2 Standard structure for reengineering tool sets

2.3.1 Actual Reengineering Environments

Several groups of tools exist. There are the general visualisers, not necessarily aimed at software reengineering. Then there are tools that are highly specialised in a certain programming language or even a one vendor-specific dialect. Furthermore, there are the tool environments, which are explicitly aimed at supporting multiple, possibly cooperating tools, and generic metadata repositories. For all groups we present a few examples.

The first group of tools we discuss are the *generic visualisers*. They are typically based on simple generic metamodels to be able to easily handle many different kinds of information. Rigi [Mul86] supports reverse engineering by providing a scriptable tool with grouping and graph layout support. It is based on a graph metamodel, enabling it to easily visualise any entity-relationship model. The actual metamodel can be constructed by the user, leveraging his/her domain knowledge. For storing and exchanging models Rigi provides its own format, the Rigi Standard Format (RSF). Similar to Rigi is Shrimp [MADSM01], a tool to visually browse and explore complex graph-based information spaces. Exploring large software programs is only one of many possible applications.

Other tools are *focused on specific languages*. Consequently their metamodels are highly language dependent. The Mansart tool [HYR96] queries abstract syntax tree (AST), and uses ‘recognizers’ to detect language-specific clichés associated with specific architectural styles. Multiple views of a same system can be generated and combined to create new ones [YHC97]. Similarly, Datrix — a source code analysis tool developed at Bell Canada — stores ASTs with additional semantic information [LLB⁺98]. Acacia is a tool that supports reachability analysis and dead code detection for C++ applications [CGK98]. (Q)SOUL [Wuy01] is a program analysis system based on a logic programming language which is integrated in a

Smalltalk environment. A declarative framework based on logic rules allows one to reason about Smalltalk code.

A third group of tools are the *tool environments*. They are explicitly aimed at supporting multiple, possibly cooperating tools. The SPOOL environment is an object-oriented reengineering environment that supports program understanding (e.g., hotspot- and design pattern identification) [KSRP99] [RSK00]. It has UML as its metamodel with some proprietary extensions. The Generic Software Exploration Environment (G^{see}) [Fav01] is a tool framework targeted at very large software systems. It consists of a repository based on a generic entity-relationship metamodel, wrappers to all kinds of storage facilities, a visualisation framework and a tool builder to interactively and dynamically build exploration tools from the provided components. GOOSE [Ciu99] [DD99] is a tool set for analysing the design of object-oriented software systems. It extracts top-level object-oriented entities such as classes, methods and attributes and their relationships such as method invocations. It supports visualisation on higher levels of abstraction and the automatic detection of potential design flaws. It has its own line based relational storage and exchange format called the Simple Relational Format.

Instead of purpose-built reengineering environments, *generic metadata repositories* can be used. Well-known examples are the Unisys UREP [URE] and the Microsoft Repository [BBC⁺99]. These repositories attempt to address the general problem of sharing models between a large variety of different software tools. The advantages of these kinds of tools is that they are open to any information model and offer industry standard integration paths to existing systems, for instance, using XMI [OMG98]. All functionality for storing, querying and exchanging information is available. The disadvantage is that they need considerable tailoring for specific uses and can be an overkill for the task at hand.

2.3.2 Metamodels for Reengineering

There are several, mainly research groups that have created metamodels to represent software. Only a few of the metamodels, however, have explicit descriptions of what information is represented. We list some of those metamodels here. Note that the list does not contain metamodels aimed at object-oriented analysis and design (OOAD), the most notable example being the Unified Modeling Language (UML) [OMG99]. These models represent software at the design level. Reengineering requires information about software *at the source code level*. The starting point is the software itself, demanding for a precise mapping of the software to a model rather than a design model that might have been implemented in lots of different ways.

Bauhaus Resource Graph

The Bauhaus Resource Graph models source code by providing information such as *call*, *type* and *use* relations [CEK⁺00] [Kos00]. It is aimed at modeling constructs of procedural programming languages that have a bearing on architecture recovery. Next to the metamodel, it defines a simple, compact graph-based exchange format. The format is human-readable, non-nested and has built-in compression through string reference sharing.

TA and TA++

TA (Tuple-Attribute Language) [Hol98] is a language to record information about certain types of graphs. It defines a simple format to describe graphs and a basic schema for describing program items such as procedures and variables and relationships such as call and reference relationships. TA++ [Let98] is an extension to TA that describes a schema for program-entity level information. It is aimed at providing a representation of high-level architectural information about very large software systems. Target languages

range from Java and C++ to C, Pascal, COBOL, FORTRAN and even assembler. The language mappings to the described model, however, are not explicitly defined.

Datrix

Datrix is a source code analysis tool developed at Bell Canada [LLB⁺98]. The model used to describe software is the Abstract Semantics Graph (ASG) [BC00]. An ASG represents an abstract syntax tree (AST) with additional semantic information such as identifiers' scope, variables' type, etc. The goal of the Datrix ASG model is completeness — any kind of reverse engineering analysis should be doable on an ASG without having to return to the source code — and language independence — the model should be the same for all common concepts of C++, Java and similar languages. This language independence is, however, restricted to C++-like languages.

Acacia

Acacia implements a C++ metamodel, which is explicitly aimed at reachability analysis and dead code detection [CGK98]. The metamodel models software at the program entity level. The metamodel is C++ specific (although it has been used to analyse Java as well), hence it contains C++ specifics such as friend relationships, class and function templates, macros and C++ specific primitive types. It also deals consistently with nested classes and their references.

Acacia's database size is 1.5 to 2.5 times source code size. [CGK98] mentions to know about software vendors that create databases that are up to 150 times the source code size.

2.4 Refactoring and Code Reorganisation

Refactorings — behaviour preserving code transformations — are more and more discussed in the context of reengineering object-oriented applications [SGMZ98] [TB99a] [FBB⁺99] and as part of new development process models such as eXtreme Programming [Bec99].

Research on refactorings originates from the seminal work of Opdyke [Opd92] in which he defined some refactorings for C++ [JO93], [OJ93]. Similarly, Tokuda and Batory evaluate the impact of a refactoring engine for C++ [TB99a] [TB99b] and also [FR98] reports a reengineering experience where C++ was refactored and dedicated tools were developed. Werner analyses refactorings for Java [Wer99]. Roberts [Rob99] specifies the Smalltalk refactorings available in the Refactoring Browser and focuses on the possibility to combine refactorings by analysing postconditions and preconditions of the combined refactorings. Schulz et al. [SGMZ98] and Ó Cinnéide [OCN99] use refactorings to introduce design patterns.

Besides refactorings, research has addressed the reorganization of class hierarchies. Casais proposes algorithms for automatically reorganizing class hierarchies in Eiffel [Cas91] [Cas92]. These algorithms not only help in handling modifications to libraries of software components, but they also provide guidance for detecting and correcting improper class modelling. [DDHL96] proposes an algorithm to insert classes in a class hierarchy that takes overridden and overloaded methods into account. [Moo96] proposes to decompose Self methods into anonymous methods and then reorganize class hierarchies by sharing as much as possible of the created methods. Note that this work, while interesting from a scientific point of view, could only be used to shrink applications for deployment and not for increased understandability, as the symbolic meaning of method names is lost in the reorganisation process.

Integration of refactoring tools in development environments is getting more and more common. Examples are the Refactoring Browser [RBJ97] for several Smalltalk dialects and Java tools such as jFactor for

VisualAge for Java [jFa] and JRefractory for several other Java IDEs [jRe]. Integration in reengineering environments is to our knowledge not yet widespread. An example is Compost [Uni96], a Java analysis tool that supports some refactorings.

2.5 Discussion

This thesis is about modeling object-oriented software in a language-independent way for the purpose of reengineering. Looking at the state-of-the-art from this perspective we can make the following observations:

No metamodel descriptions. The repository and its underlying metamodel are a crucial part of a reengineering environment. However, although many tools and tool environments exist, for most of them *the metamodel is not explicitly documented*. Especially, the *relevance* of the available information for the reengineering tasks at hand is not made explicit. Consequently, every time a tool is developed the same analysis needs to be performed anew. Furthermore, multiple implicit metamodels hinder the interoperability between tools, because required and provided information hardly ever matches.

As section 2.3.2 shows, there are a few exceptions. There are explicit metamodels that focus on a particular task: Rainer Koschke's thesis discusses the relevance for modelled information of the Bauhaus Resource Graph model for reengineering procedural programs, in particular C, with the purpose of remodularisation [Kos00]. Acacia also models C++ with the distinctive purpose of performing reachability analysis and dead code detection. Other approaches focus on general reengineering support: Datrix has a clear description of what is modelled, namely abstract syntax trees of C++ programs with added semantic information. Basically complete programs are modelled in all possible detail. The relevance question is not discussed, as the goal is to model everything and make the availability of the original source code irrelevant for all possible analysis tasks. Only TA++ is a general metamodel for reverse engineering, targeted at many languages. It models high-level constructs, has a well-defined exchange format and also discusses issues such as storage and extensibility. However, its definition both lacks a discussion of the design choices and a clear mapping of the different supported languages to the common concepts.

No multiple-language support. Legacy systems exist in many languages. On top of that, many reengineering tasks are similar for multiple languages, especially within a single paradigm. Consequently, there is a vast potential for reuse over multiple similar languages. To be able to deal with multiple languages effectively it needs to be clearly defined how different languages are represented in a common way. Only in this way tools will be able to base common analysis on the metamodel and be sure that it provides the expected results for all supported languages.

However, not many metamodels have elaborate multi-language support. Tools like the generic visualisers use simple metamodels that cover common concepts of several languages. Their metamodels are not very detailed and often ad-hoc, providing simple multi-language support, which is sufficient for visualisation tasks, such as high-level browsing and grouping. From the metamodels described in section 2.3.2, only TA++ aims at generally supporting multiple languages for reengineering large software systems. Target languages range from object-oriented languages to assembler. It does not define, however, how these different languages are mapped to the TA++ metamodel. In any case, the potential for a common definition lies in the languages with a common paradigm, rather than in the whole scope of targeted languages.

No explicit infrastructural design choices. Apart from the exact contents of a metamodel and its relevance for reengineering, there are other properties that determine how successful a metamodel supports reengineering. Most notably these are:

- *Scalability.* It is not uncommon that legacy systems contain several millions lines of code. A tool environment must scale up to deal with the vast amounts of information involved. Because a metamodel determines what information a repository can contain, it has a direct influence on how much information is generated.
- *Extensibility.* Not all information needs are known in advance. A metamodel must be able to deal with information that was not anticipated in its original design. Furthermore, models must be able to store annotations to capture analyse results and knowledge gained through a reengineering process.
- *Tool integration.* The vast and heterogeneous set of possible reengineering tasks typically results in multiple specialized tools that need to work together to provide a full understanding of a system [DDT99].

Other aspects are the ability of a metamodel to support *grouping* — for the creation of higher-level abstractions or classifying model elements — or *multiple models* — for evolution analysis.

We call these *infrastructural* aspects: design aspects that deal with how the information is organized and stored. They are well-known and often stated as requirements (for instance, in the TA++ definition in [Let98] and for exchange formats specifically in [SDSK00]). However, similar to the metamodel contents, hardly any existing tool or metamodel makes explicit the underlying design choices affecting these aspects. Moreover, there is also no general description of what the relevant infrastructural aspects are, how they impact the metamodel design and how choices for one aspect influence the properties of other aspects.

No refactoring support in multi-language reengineering environments. Not many reengineering environments support refactoring. Most presented tools and environments can be considered reverse engineering rather than reengineering tools. The refactoring tools mentioned in section 2.4, are either stand-alone or part of a forward development environment. Only Compost [Uni96] can be considered a reengineering tool with refactoring support.

In the context of reengineering, refactoring is clearly interesting, providing the ability to quickly and safely transform software. Beyond this straightforward application, however, integrating refactorings in reengineering environments opens a whole new class of possibilities currently not yet explored. Not only can tools analyse software or apply standalone refactorings, both capabilities can be combined so that a tool can detect problems and propose solutions to resolve such problems and perform the required transformations.

From the perspective of metamodels for reengineering, refactoring poses additional constraints on their design. It demands *sufficient, complete and 100% correct information*, because the result of a code transformation should not result in a faulty software system. This is a stricter requirement than required for a typical reverse engineering task such as visualisation, which is normally not strongly affected if information is slightly incomplete or incorrect [MNGL98] [Bis92].

Support for multiple programming languages poses an additional challenge. While there is sufficient proof that a refactoring tool can be built for almost any object-oriented language, it is yet unknown whether it is feasible to build a language-independent refactoring engine. Only Ó Cinnéide mentions support for

multiple languages in a refactoring tool [OCN99]. He presents a layered architecture which shields language specifics as much as possible, but so far his tool prototype only supports one language, namely Java. Likewise, the Refactoring Browser [RBJ97] defines its refactorings in terms of a model of Smalltalk in order to easily deal with differences between Smalltalk dialects. In this case the language differences are quite small.

Multi-language refactoring requires an analysis to which extent refactorings can be abstracted from their underlying languages. Separating the analysis for refactorings in a language-independent and a language-dependent part has basically the advantage that complex analysis can be reused for many languages. This eases the integration in multi-language environments, such as reengineering environments or other kinds of CASE tools.

2.6 Conclusion

Wrapping up we can say that a lot of knowledge about how software needs to be modelled for reengineering purposes is implicit. In this thesis we make parts of this information explicit. First, we discuss in general infrastructural aspects of reengineering environments and their underlying metamodels. We present the different choices that a tool developer can make and how the choices for the different aspects interrelate (chapter 3). After that we look at the specific problem of modeling multiple object-oriented languages in a common way. We present one metamodel and discuss the trade-offs in its design. This discussion includes the specific choices for multi-language support and the infrastructural aspects that are discussed in general before (chapter 4). In the following chapter we discuss an implementation of the presented metamodel and to which extent it is successful in supporting multiple reengineering tools in practice (chapter 5). We then use our metamodel to provide an in-depth analysis of multi-language refactoring. Based on our language-independent metamodel we have analysed fifteen low-level refactorings. The analysis shows to which extent it is possible to abstract from the underlying languages, in our case Smalltalk and Java (chapter 6). The refactoring analysis has been validated with an implementation and case studies (chapter 7).

CHAPTER 3

A Design Space for Reengineering Tool Infrastructures

Figure 2.2 in the previous chapter shows the general structure of a reengineering environment. It illustrates that the repository is the central part that lets tools work on a common information base. The properties of the repository, and thus of the complete environment, are highly influenced by the metamodel that describes what and in which way information is modelled. The metamodel not only determines if the right information is available to perform the intended reengineering tasks, but also influences issues such as scalability, extensibility and information exchange. We call the latter *infrastructural* aspects: design aspects that deal with how the information is organized and stored.

This chapter makes explicit these infrastructural aspects. It presents a conceptual space that identifies the aspects necessary for a reengineering environment developer to consider. For every aspect the space covers the design options including implementation solutions and a discussion of the trade-offs. Furthermore, discuss the dependencies between the axes. The goal of this chapter is not to come up with a design for a *specific* reengineering platform such as PBS [FHK⁺97] or Rigi [Mul86] and their underlying metamodels. Hence we do not cover the ability of an environment to support specific reengineering tasks and the exact contents of a specific metamodel.

In the context of this thesis, the design space sets the infrastructural context for one particular metamodel, namely FAMIX. FAMIX is a concrete metamodel that models object-oriented software in a language-independent way. It is presented in CHAPTER 4. The design space puts the infrastructural options we have chosen for FAMIX in perspective. From the other side, FAMIX, presented as an instance within the design space, validates the usefulness and accurateness of the space.

The rest of the chapter starts with an introduction to the design space before we describe the different aspects in detail.

3.1 Introducing the design space

We start our introduction to the design space with a small scenario that shows how a typical reengineering environment is used. We use the scenario to illustrate several infrastructural aspects of such an environment. Afterwards we give an overview of the complete set of aspects that our design space covers in the form of a list of questions. After that, we show the design space with all the aspects and their interrelationships.

3.1.1 Scenario

We paint here a typical usage scenario of a reengineering tool environment. The reengineer, let's call her Claudia, is confronted with a legacy C++ system of about a million lines of code. Her intention is to extract the architecture and possible problems in changing it.

First she extracts information from the C++ system into a repository. She decides to only extract program entity information instead of a full abstract syntax tree. She applies a tool to visualise the structure of the application and uses grouping techniques to collapse classes into a higher level module view.

Claudia has a third-party metrics tool that can help her to understand the system. Claudia exports the information from her environment into a standard exchange format and imports it into her metrics tool. She computes some metrics and combines the obtained results to enhance the views she gets with size information. She detects a big class in module X on which many classes in other modules depend. Looking at the source code she finds that some of the functionality of this class can be distributed over the different modules.

Analysis

The scenario illustrates the following infrastructural aspects of a reengineering tool environment: the extraction *level of detail* (Claudia chose the program entity level), different *kinds of entities* (she produces architectural entities), the use of *grouping* (she groups entities to produce a modular view), the *tool integration*, *information exchange* and *incremental loading* (she used an integrated visualiser, an external metrics tool, an interchange format and merged metric results), and the *annotation* of entities (the metrics are associated with the entities they relate to).

The scenario only covers a subset of reengineering activities. For instance, Claudia might want to apply similar analysis on other implementation languages, compare multiple versions of the same system, etc.

3.1.2 Infrastructural issues summarised

We summarise the infrastructural aspects as a list of questions that a developer of a reengineering environment typically needs to answer.

3.1.3 Questions concerning the information to be modelled

Language/Paradigm support. How many implementation language(s) must be supported? Do all these languages belong to the same paradigm (e.g., procedural, object-oriented)?

Level of detail. How detailed should the extracted information be? Should the information suffice to regenerate the source code it represents? Should the information support the creation of higher-level views?

Scalability. How large are the programs you deal with? How many versions have been released? How many models do you need to extract?

Multiple Models. Do you need to represent the software system at several levels of abstraction (code, design, analysis)? Will you analyse several releases of the same software system?

3.1.4 Questions concerning the tasks to be performed

Grouping. Will you create higher level abstractions by grouping model elements? Do you need to group these groups? Do you need to group elements of multiple models? Do you need to group relationships?

Tool Integration. Must the tool environment exchange information with other tools? How do you merge information coming from different tools? Must the repository support parallel access from multiple tools?

Extensibility. Must the metamodel be able to accommodate new kinds of information? Is it needed to annotate model elements? Must the environment adapt itself to new kinds of information?

3.1.5 Questions concerning underlying implementation

In addition to the issues directly brought forward by the user requirements, some key under-the-hood implementation aspects need attention as well, because they can make or break the ability to fulfil a requirement. These implementation aspects are:

Storage medium. How is the information of a model stored? Does the medium fit your scalability needs? Does it fit your information exchange needs?

Entity reference. How are model elements identified? Can the reference schema handle references over multiple files? Can it handle groupings of any model element? Can it handle multiple models?

Incremental Loading. Should the tool be able to work with incomplete models? Must all information be 100% correct? Should the tool be able to merge information from different sources?

Exchange Format. Does the format need to be easily machine? How precise should it reflect the internal datastructure? Must an industrial standard be supported?

Metamodelling. How do you store information *about* a model (e.g., name of the creating tool, the level of detail of the extracted information)? Do you work with an explicit metametamodel? Which kinds of extensibility does the metametamodel support?

3.1.6 Design Space in a Nutshell

The aspects brought up in the previous subsection cannot be considered in isolation. A choice made for one aspect often influences the choices for another one. An example is the level of detail. When the toolset supports abstract syntax tree level of information rather than the program entity level, it is much harder to support multiple languages, because it is easier to map multiple languages to a more abstract higher-level representation than to a very detailed one.

We capture these dependencies in a so-called ‘design space’. Each aspect represents an axis into this multi-dimensional non-orthogonal space. Within the space a single reengineering platform is determined by the values chosen for each axis.

Before going into the details of the separate axes, we present a roadmap to the design space. Furthermore, we present the template we use to describe the axes.

A roadmap to the design space

Figure 3.1 provides a roadmap of the design space. It shows the design space with the axes, their main options and how they interrelate. For every axis it also indicates the number of the section that describes the axis in detail. We distinguish the following kinds of axes:

- *Requirement axes.* The requirement axes cover decisions that are depending on the user requirements that the environment and its underlying metamodel must fulfil. These cover most of the questions from sections 3.1.3 and 3.1.4. In Figure 3.1 the requirement axes are represented by ellipses.
- *High-level requirement axes.* High-level requirements represent key concerns of a reengineering environment. They do not have implementation options themselves, which is why we do not describe them in detail in a separate section. They are, however, affected by many of the other axes. The high-level requirement axes are two remaining aspects from sections 3.1.3 and 3.1.4, namely the Scalability Axis and the Tool Integration Axis. In Figure 3.1 they are represented by clouds.
- *Implementation axes.* The implementation axes cover issues that are not directly reflected in the user requirements. These are the questions brought up in section 3.1.5. In Figure 3.1 the implementation axes are represented by boxes.

Axis template

The rest of the chapter discusses the requirement and implementation axes in detail. We describe them according to the following template:

- **Name:** the name of the axis.
- **Description:** an overview of the axis including the design choices.
- **Dependencies:** a list of the dependencies with other axes.
- **Implementation issues:** a discussion of implementation solutions and their trade-offs.

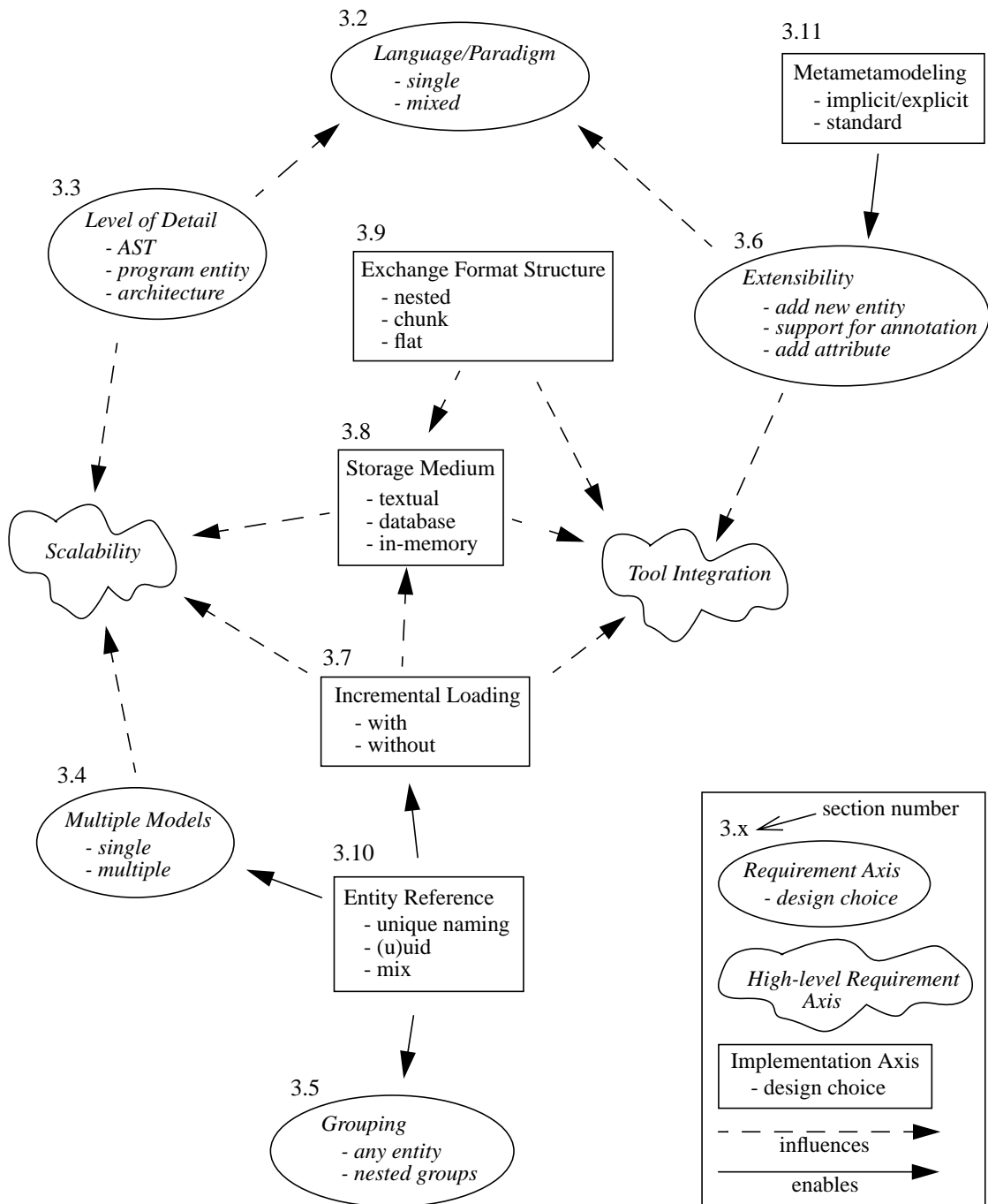


Figure 3.1 Roadmap of the design space. It shows the different axes and their relationships. The numbers correspond to the sections that describe the axes in detail.

3.2 Language/Paradigm Axis

The choice which language(s) or paradigm(s) to support is largely driven by the requirements of the end-user. We categorise the possibilities as follows:

- only one language (such as Smalltalk, C++, Fortran, Lisp)
- only one paradigm (such as object-oriented, procedural, functional, logic programs)
- multiple languages with one paradigm
- one language with multiple paradigms (such as hybrid C++ object-oriented/procedural programs)
- multiple languages with multiple paradigms.

The main issue is if language or paradigms are mixed. The implementation issues below discuss the consequences.

Dependencies

Level of detail. The higher the level of detail, the more reasoning power you get, but the harder it is to support multiple languages in a common way.

Extensibility. If the metamodel consists of a common core with language extensions, the appropriate extension mechanisms must be available.

Implementation issues

If only one language, hybrid or not, needs to be supported, the metamodel typically contains constructs of that language in a straightforward one-to-one mapping. It gets more complex if multiple languages are supported. The constructs of both languages are modelled either separately or using a common abstraction. Separate modelling is typical in the case of languages that have dissimilar paradigms. In such a case the metamodel is often constructed with separate, but connected submetamodels for every paradigm [LS99]. If the supported languages have the same or an overlapping paradigm, common constructs are often modelled with a single abstraction. The typical structure of such a metamodel is *a language-independent core with multiple language extensions*. For instance, a core could contain a Class abstraction which would allow class concepts in Java, Smalltalk and C++ uniformly, but the C++ class template would be modelled in the C++ language extension. Treating similar constructs in a similar way results in language independence and reuse of analysis code. On the other hand, treating them explicitly decreases problems with semantic differences.

It is often useful to *store the mappings*. E.g, if Java interfaces and Java classes are modelled using a common Class abstraction, the information whether the element represents a class or interface is stored as an annotation to the Class abstraction. This allows, for instance, visualisation tools to colour Java interfaces differently from Java classes. Language-independent tools, however, can just treat the common concept without having to know about the language details. For similar reasons, if the modelled system is implemented in multiple implementation languages, it is often necessary to record what the implementation language is for every entity or every group of entities.

3.3 Level of Detail Axis

This axis describes the different levels of detail and the consequences of choosing one or the other. We distinguish the following levels:

1. **Abstract syntax tree (AST) level information** - a complete view of the source code. It is normally detailed enough to regenerate the source code and sufficient for control-flow analysis.
2. **Program entity level information** - abstracted but factual view on source code: classes, methods, functions, etc. It is normally used for a generating structural views on the target system. Typically only limited information about method bodies is available, normally consisting of method invocation and variable access information, which is sufficient for dependency analysis.

Dependencies

Language/Paradigm. The higher the level of detail, the more reasoning power you get, but the harder it is to support multiple languages in a common way.

Scalability. The higher the level of detail, the higher the memory consumption and load time of information from databases or files and the slower the response times of tools that use the information. To give an idea for the impact of different detail levels on resource consumption, we show the file sizes of the textual representation of a model using our own FAMIX metamodel (see CHAPTER 4) and the two standard exchange formats, namely CDIF [Com94] and XMI [OMG98]. The numbers are shown in Table 3.1. The default representation of both standards is not optimized for space consumption, which is confirmed by the huge compression achieved by zipping the files. The modelled system is the Java Swing framework, version 1.3.0, consisting of 7.2 MB of source code (225 KLOC). It has been parsed by SNIFF+ parser, version 3.2.1. It consists of ~2700 classes (including inner classes), ~11500 methods. The metric information contains up to 25 metrics per source code entity where every measurement is stored as a separate element in the file.

Swing 1.3.0: 225 KLOC~ 2700 classes	CDIF file, MB (zipped MB)	XMI file, MB (zipped MB)
classes, methods, attributes	8.3 (0.53)	21.8 (0.90)
+ invocations and accesses	12.1 (0.82)	28.8 (1.25)
+ formal parameters	15.5 (0.99)	39.7 (1.67)
+ metric information		84.7 (3.15)

Table 3.1: File sizes for different levels of detail of Swing 1.3.0

Implementation issues

Scalability is an important issue to consider, because the large amounts of information can make it hard to effectively analyse a system. We describe here two techniques to reduce the amount of information. The first one *incremental extraction* by using source anchors, i.e., pointers to the original source code. Source anchors allow one to go back to the source code and extract additional information only when needed. Important consideration is that the partial information is still sound. For instance, if method information is required, information about the classes that contain the methods typically needs to be stored as well. A second technique is to *collapse information*. For instance, instead of representing all invocations from methods of one class to methods of another class, this information can be collapsed into a single invocation relationship between the two classes. The details of which method calls which other method is lost, but the dependency between the classes is still represented and less information needs to be stored.

3.4 Multiple Models Axis

The possibility to analyse multiple models simultaneously is useful in evolution analysis, where multiple versions of the same system need to be analysed [LDS01] [JGR99]. Likewise, it is interesting to analyse parallel branches of similar applications, for instance, to develop a framework by abstracting common assets in these branches. Another aspect is that different models can have different metamodels. This is typically the case when multiple paradigms are modelled [LS99].

Dependencies

Entity Reference. Multiple models require the possibility to uniquely identify elements from different models, even if these elements are similar, for instance, the same class in multiple versions of the same system. Furthermore, it must be possible to identify which model an element belongs to.

Metamodelling. Information about models can be modelled as part of the model itself or as a meta-entity.

Implementation issues

Instead of making multiple models explicitly part of the metamodel, a tool environment can support multiple models in its implementation. This keeps the metamodel simpler, but makes it impossible to formally exchange mixed information. Furthermore, tools that use information of multiple models get dependent on the implementation of the tool rather than its underlying metamodel.

3.5 Grouping Axis

Grouping is an important technique in reengineering, primarily to build higher-level abstractions from low-level program elements or to categorize elements with a certain property. Two ways of grouping can be identified: intentional (description-based) and extensional grouping where the group acts as a bag of (references to) model elements [DD99] [MWD99]. A tool developer should consider the following options:

- the support for nested groups, i.e., groups that can contain other groups
- which entities can be grouped, e.g. can a group only contain named entities such as classes or also nameless entities such as relationships.
- groups of entities over multiple models, i.e., can a group contains entities belonging to different models.

Dependencies

Entity Reference. The entities that need to be groupable, must be uniquely referencable. This can include the groups themselves and entities in multiple models.

Implementation issues

If intentional groups are to be supported, the model must define how descriptions are modelled. Expressions using the Object Constraint Language (OCL) expressions [SMHP⁺97] or other formalisms could be used. Furthermore, intentional groups need strategies for recomputing the contents when a model is adapted or additional information is incrementally loaded [MWD99]. This all is, however, a topic that goes beyond the scope of this chapter.

3.6 Extensibility Axis

Extensibility is an important issue in modelling software as it allows additions to a model without having to change the model itself. Typically extensibility is needed for the following kinds of information:

- Language-specific information. In the case of a metamodel with a language-independent core, it is often still interesting to store language-specific information. For instance, a visualisation tool might want to colour all Java interfaces different from classes, even if both classes and interfaces are mapped to a common class concept in the language-independent core. Additionally, languages have their own specific problems that are interesting in themselves. An example is the analysis of include hierarchies in C++.
- Tool-specific information. Tools might want to store and exchange tool-specific information such as analysis results or layout information for graphs.
- Whatever information people find worth modelling. Not all information needs can be known in advance. Furthermore, being able to annotate any element in a model, can be a great help to store acquired knowledge in the process gaining understanding of a system.

The following subsections describe the extension mechanisms to be considered.

3.6.1 Adding new entities to a metamodel

To be able to represent constructs that are not yet covered in a metamodel and cannot be mapped in a sensible way to the existing elements, it must be possible to add entities to an existing metamodel. Examples are the addition of the C++ include relationship to a language-independent metamodel, or the addition of a special-purpose container that is relevant in the specific domain of system under investigation.

3.6.2 Adding attributes to existing entities.

Similar to the addition of new entities to an existing metamodel, it may be necessary to add new attributes to existing entities. An example is the information that a class in a model represents an Java interface. The extension adds an attribute `isInterface` to the `Class` entity. This can be done through subclassing or by explicitly adding an attribute. Figure 3.2 shows an example. The subclassing solution (a) demands for all clients to know the extension and how it relates to the original entity. It demands importers that are fully metamodel aware and can deal with extensions that it did not know yet. When the attribute is just added to the original entity (b) an importer that does not (want to) know the extension just recognizes the element it knows and can ignore any attribute of the element it does not recognize.

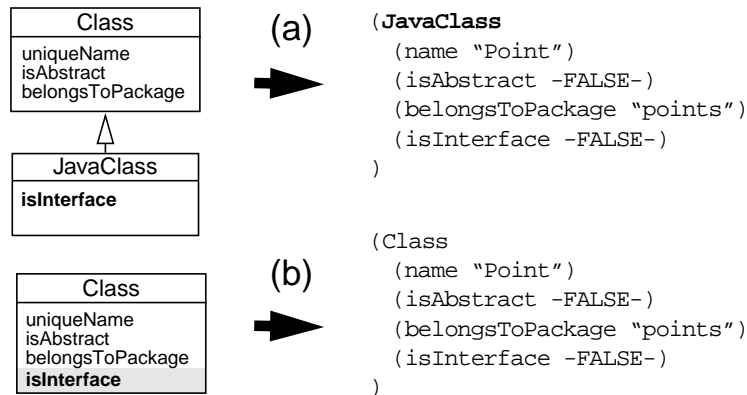


Figure 3.2 Extending Class with an attribute `isInterface` by (a) subclassing an Class and (b) adding the attribute to the existing Class element

A more severe problem with the subclassing solution is the fact the multiple orthogonal extensions (typically language and tool extensions) might exist which cannot be modelled by inheritance at all. This is illustrated in Figure 3.3. This is also a problem if an attribute must be added in an existing hierarchy where all existing subclasses should inherit the attribute.

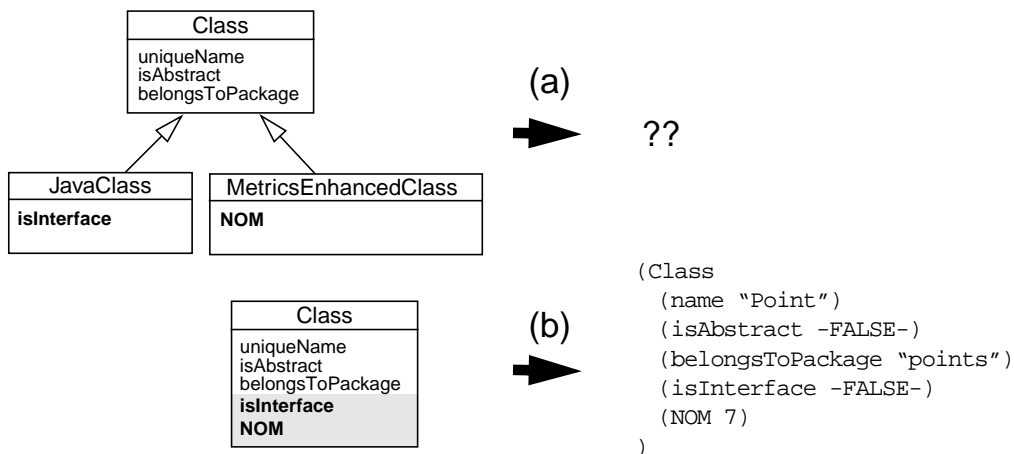


Figure 3.3 Extending Class with two orthogonal attributes by (a) subclassing Class and (b) adding the attributes to the existing Class element

3.6.3 Annotating entities

Annotations allow any information to be added to any entity. Entity properties, analysis results, human understanding in the form of notes can then be attached to the entity they refer to. Although a metamodel could support a way to support annotations, it is often considered the domain of the metamodel. In such a case the metamodel explicitly allows any entity to be in relation with some property, annotation or tag objects (see Figure 3.4).

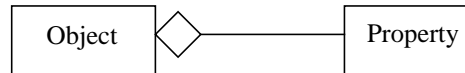


Figure 3.4 Object annotation

By following such an approach, the issue of the representation of the annotation object in memory and in the exchange format have to be answered. Indeed, from our experience of representing metrics as entity annotations, we learnt that annotation objects can constitute more than 90% of the data processed while loading and analysing a system. One solution is to only represent properties explicitly in the exchange format and use appropriately optimized data structures for the in-memory representation.

In the exchange format, annotations can be represented as an attribute of the annotated entity or as an explicit entity itself (see Figure 3.5 for an example). The first solution precludes loading of annotations separate from their containers, but requires less space.



Figure 3.5 Annotations as an attribute of the annotated entity (a) or as a separate entity (b)

3.6.4 Metamodel extensibility limits

The extensibility of the metamodel is restricted by the extensibility the metamodel allows you to have. This is especially an issue when using modelling standards such as the MOF [OMG97], because they restrict you to their extension mechanisms. The MOF and GXL [HWS00] do not support class extensions, i.e., the ability to add attributes to existing classes. CDIF [Com94] and RDF [WWWC99] both allow this kind of extension.

3.7 Incremental Loading Axis

Incremental loading of information is about the ability to load new entities or additional information for entities that already exist in a model. The reasons for considering incremental loading are resource optimisation and the merging of information from different sources.

Incremental loading generally allows a model to contain references to information that is not in the current model. This is interesting for the following reasons. It allows to load only parts of models. This is particularly useful if only part of a system is available, e.g., the source code of a library is not always accessible, or if only a part of a system is of interest. Extractors can also make mistakes that result in dangling references, especially for complex languages such as C++.

Dependencies

Entity Reference. The way model elements are referenced must support the ability that information elements in different files or databases can reference each other.

Scalability. Incremental loading can be used for resource optimization. Information about the same system can be stored (and thus transferred) in separate files or databases. Furthermore, information can be loaded on demand. For example, computed metrics or analysis results can be so space consuming that you only want to load them when necessary.

Tool integration. Code analysis can be performed by different tools that do not share a common repository [DDT99]. Results can be stored in different locations and merged if needed. For example, metric values computed by one tool can be loaded into a visualisation tool.

Storage. Incremental loading enables the storage of related information in different locations.

Implementation issues

Referenced information which is defined in a separate resource must be able to be filled in later and be correctly associated with the entity it refers to. A solution is to use stubs, i.e., empty placeholders, that represent the missing entities. If later the actual element is loaded, it replaces its stub. Note that referenced information often is left out on purpose. Frequently we want to model an application but not the complete libraries or frameworks it uses. In such cases the model ends up with references to non-represented entities.

3.8 Storage Medium Axis

Models, once created, need to be stored and exchanged between tools. One issue is the *storage medium*: different storage approaches exist such as a textual representation, databases or in-memory repositories that can be saved to disk, all with their own advantages and disadvantages:

Textual storage

- is simple (even if this depends on the format structure. See section 3.9 for details),
- supports easy information manipulation, for instance, with scripting languages,
- is a good base for information exchange.

Database storage

- scales better,
- has slower response times than an in-memory repository,
- allows tools to work together on a common information base,
- often has support for information exchange standards,
- but less easily serves as information exchange medium itself.

In-memory repository

- provides fast access to information,
- provides easy manipulation of information,
- scales only in the context of the available working memory,
- but its information exchange and storage capabilities are limited to the implementation language.

Dependencies

Scalability. As discussed above the different storage media have different scalability properties.

Tool integration. The storage medium determines how tools can integrate. Textual storage is a light-weight approach mostly for tools that just exchange information. Databases allow tools to work together on a common information base more easily.

Incremental Loading. Incremental loading allows file exchanges to be divided into multiple files and for different tools to produce information about a system that can be merged afterwards.

Implementation issues

The choice between textual representation and a database is a choice between a light-weight and a heavy-weight solution. A text file is easy to produce, to process and move between platforms. It is quite a low-level solution, however, because users have to deal with saving, parsing and entity reference themselves. A database, on the other hand, is not as easy and quick to set up and normally needs considerable tailoring for a specific task. Once set up, however, it normally provides integrated support for industry standard information exchange and schema transformation.

The in-memory solution is about using the runtime datastructure as a storage format. When loaded, it allows tools fast access and easy manipulation of information. The storage is not necessarily faster or easier than a textual solution or a database, as the data needs to be stored to a persistent medium such as a harddisk. Java serialization, for instance, can be used for this purpose. Some development environments, however, have built-in support for the storage of a complete working session, i.e., all the data and runtime state of the tools. This is the case for many Smalltalk dialects.

3.9 Exchange Format Structure Axis

Repository information is often stored in text files. It is a lightweight way of storage, which is particularly well suited for information exchange. St-Denis et al [SDSK00] describe a set of thirteen criteria that are important for an exchange format. Many of the criteria they describe are non-functional, such as reliability and completeness, which rather depend on the quality of the tool that produces the information than that it is an intrinsic property of the exchange format.

We describe three common format structures and discuss them in the light of three criteria, namely human readability, machine processability and the ability to incrementally load information. These are aspects that only depend on the actual structure of the exchange format. They are independent of the quality of the extractor and the actual encoding (i.e., if the line ends with a bracket or with an XML-tag).

The three format structures store model elements in a *nested* way, in *chunks* or in a completely *flat* way. We analyse them in a separate subsection and discuss their properties afterwards.

Dependencies

Tool integration. The structure of the textual format influences its fitness for information exchange. This is discussed in more detail in the following subsections.

Storage Medium. Textual storage is one of the storage approaches presented as part of the Storage Medium Axis. Consequently, it is influenced by the textual format structure.

3.9.1 Nested, chunk and flat formats

This subsection describes the nested, chunk and flat format and discusses their human readability, machine processability and incremental loading properties.

Nested. In the nested format every element physically contains its constituents. Relationships other than containment, are modelled by explicit relationship elements. The following example shows a class A which inherits from class B and contains two methods (M and N) and an attribute (X).

```
(class A
  (method M
    (isAbstract true))
  (method N
    (isAbstract false))
  (attribute X
    (visibility public)))
(InheritanceDefinition
  (subclass A)
  (visibility protected)
  (precedence 2)
  (superclass B))
```

Evaluation. The nested format is complex and elements can quickly get large due to the elements they contain. It only favours human readability as long as the elements do not get too large. Incremental loading is not well supported either, because elements cannot be stored without their containing element, and replacing an element requires finding it inside its containment hierarchy. The format is useful for a one-to-one representation of the tool-internal datastructure, which is typically optimised for information navigation rather than storage or exchange.

Chunk. In the chunk format, entities are not nested into their scoping entity. Simple relationships are stored as attributes of the contained entity. The example hereafter shows the same information as the nested example, but now in the chunk format. Now the methods and the class are stored as explicit entities and containment relationship is represented by the `belongsToClass` attribute in M and N.

For relationships that need additional information to be stored, explicit entities are created. In the example, the inheritance relationship is an explicit entity with `visibility` and `precedence` as attributes.

```
(class A
  (isAbstract true))
(method M
  (belongsToClass A)
  (isAbstract true))
(method N
  (belongsToClass A)
  (isAbstract false))
(attribute X
  (belongsToClass A)
  (visibility public))
```



```
(InheritanceDefinition
  (subclass A)
  (visibility protected)
  (precedence 2)
  (superclass B))
```

Evaluation. The chunk format favours human readability, because entities are self-contained with the containment relationships readily available as attributes. The format supports incremental loading better than the nested format, because, when loading or updating an element, the containing entity does not need to be found or even be available. Because the format is less complex, importers can be simpler as well.

The chunk format has as main disadvantage that relationships are stored in two different ways.

Flat. The flat format explicitly represents all entities and relations, typically using a line-based format. For example, RSF is based on this approach [Mul86]. Contrary to the two other formats, which support entities with attributes in pair-value form, here the attributes require another storage form. In the pure form they are all explicit relations. Some formats are more elaborate [Hol98] and allow attributes as pairs on the same line. This actually goes a step towards the chunk format. The following code shows the flat format with both kinds of attribute storage:

```
(class A)
(method M)
(methodBelongsToClass M A)
(attribute X)
(attributeBelongsToClass X A)
(inheritance A B)
(inheritance A B visibility protected)
(inheritance A B precedence 2)
```

Evaluation. With this approach, incremental loading is well supported and merging of models is easy. Machine processability is also easy, because the format is conceptually simple. Because it is line based, manipulation of models can be easily done by running a script (in Perl or similar scripting languages) to convert one file into another one. A drawback is that information representing one single entity can be scattered over the file, which hampers readability.

3.9.2 Discussion

Table 3.2 summarizes the evaluation of the three format structures. The nested structure does not score well against our criteria. We do not recommend it for information exchange purposes, although it can be a viable option to store complex datastructures. The chunk format does well in all three areas. The flat format is less readable, but easier to process. Note that human readability gets less important when importing and saving tools stabilize.

	readability	processability	incremental loading
nested	-	-	-
chunk	+	+	+
flat	-	++	+

Table 3.2: Textual formats compared

3.10 Entity Reference Axis

As can be seen in Figure 3.1, the entity reference axis directly and indirectly influences many other axes in the design space. The decision how entities are referenced comes down to a trade-off between heavyweight solutions that ensure that every element in the model can be uniquely identified, always and everywhere between different tools, or solutions that are more light-weight, but might not adequately support grouping, multiple models, or referenced information that is not in the same file or database.

Dependencies

Grouping. The entities that need to be groupable, must be uniquely referencable. This can include the groups themselves and entities in multiple models.

Multiple Models. Multiple models require the possibility to uniquely identify artifacts from different models, even if these artifacts are similar, for instance, the same class in multiple versions of the same system.

Incremental Loading. Incremental loading requires information elements in different files or databases to be able to reference each other.

Implementation issues

There are several possibilities to identify entities. One is to use the intrinsic unique name of entities that have one (like a class name or attribute name), or to use a meaningless unique identifier. We present both solutions in the next subsections and finish with a discussion.

3.10.1 Unique identifiers

One approach for model element identification is the use of meaningless identifiers¹. Every model element gets such an identifier attached to it and all references use this identifier as well. The most important issue is how ‘unique’ this identifier needs to be. Only within one database or file, only within a known set of files and databases, or always guaranteed unique. Light-weight approaches can be used, the simplest one probably the use of integers. However, if uniqueness with a broader scope is required, solutions such as Universal Unique Identifiers (UUIDs) [OG97] must be considered.

UUIDs are unique across both space and time, with respect to the space of all UUIDS. A UUID can be used for multiple purposes, from tagging objects with an extremely short lifetime, to reliably identifying very persistent objects across a network. UUIDs are generated using a combination of the network address and current time at the moment and place it was generated. Assuming that network addresses are unique and that time never runs backwards, this guarantees that UUIDs really are unique. Furthermore, the UUID generator must have access to an IP network address. An example of a UUID:

```
c842bf06-d202-0000-0282-5c410d000000
```

However, using UUIDs increases memory consumption and loading and saving times. Table 3.3 shows, for the same system as discussed in Table 3.1, the file sizes when using UUIDs and plain integers in XMI files. We see that the file size increases up to 9% when using UUIDs. Similarly the working memory usage of the same models in our reengineering environment (Moose, see CHAPTER 5) increased by up to 13%. For Smalltalk systems we measured memory usage increases of up to 40%. This is probably due to the fact

1. We use the word ‘meaningless’ to indicate that the value of the identifier does not have any meaning in itself.

that our Smalltalk parser extracts a lot more detailed information (such as arguments) than the parser we use for Java.

Swing 1.3.0	XMI file MB (zipped MB)	XMI + UUID MB (zipped MB)
classes, methods, attributes	21.8 (0.90)	23.6 (0.93) (+ 8.3 %)
+ invocations and accesses	28.8 (1.25)	31.2 (1.30) (+ 8.3 %)
+ formal parameters	39.7 (1.67)	43.1 (1.75) (+ 8.6 %)

Table 3.3: Textual representation of Swing 1.3.0 without and with UUIDs

The complexity and resources needed to compute and store these identifiers might however be an overkill. More lightweight approaches can be designed, but will rely on proprietary conventions.

3.10.2 Unique naming scheme

Another way of referencing elements is to reference them by their intrinsic unique names. For instance, the unique name of the class `Box` in the Java Swing library would have the unique name `javax.swing.Box`, which is known to be unique within a Java system, but not over time or different versions. Using unique names as identifier has the advantage that reference information is human readable. A second advantage is that every time a model is generated from a certain system, exactly the same unique name will be used instead of generating a completely new identifier or having to take earlier assigned identifiers into account. A problem is that model elements that do not have an intrinsic unique name such as relations, models and groups, do not easily fit this scheme:

Relations. Relations can only partly be identified with unique names. A simple example is when class `A` inherits from class `B`, a unique name for the relationship could be `A.InheritsFrom.B`. However, this only works for one-to-one relationships and not for any relationship with multiple targets. For instance, the two invocation relationships resulting from a method that invokes another method twice, would result in two elements with the same name.

Multiple models. A unique naming scheme in the context of multiple models needs to take the model into account, especially if the different models model the same system (for instance, multiple versions of this system). In such a case the model should somehow be integrated in the unique name of an element. We could, for instance, prepend a model name to the intrinsic unique name. For example, the `Box` class in the Java 2 Swing library in version 1.3.0 could get the unique name `Java2v130.javax.swing.Box`. Note that uniqueness of model names depends on convention rather than on a language-induced rule.

Grouping. A problem similar to model names exists for groupings. Although a grouping typically has a name to describe what is grouped, the uniqueness of those names is not intrinsic and therefore cannot be guaranteed.

It is important to normalize unique names. The unique naming scheme should be clearly defined so that unique names are always the same. An example rule is that unnecessary spaces are always removed. If the metamodel supports different languages, unique names can either be close to every supported language, or a common scheme can be defined for all supported languages. The latter requires more thought about the naming scheme, but allows tools that are independent of the supported languages deal with one naming scheme only.

3.10.3 Analysis

Unique names and meaningless identifiers have clear properties. Names can be used to reference entities with an intrinsic unique name, meaningless identifiers can be used to reference anything. The following example, expressed in simplified CDIF, shows the class `Point`, which is referred to by name in the `HasMethod` relationship.

```
(class
  (name Point))
(method
  (name intersect))
(Class.HasMethod.Entity Point intersect)
```

The same example using meaningless identifiers:

```
(class 111
  (name Point))
(method 112
  (name intersect))
(Class.HasMethod.Entity 111 112)
```

Another important difference is that unique names are reproducible and other identifiers not. Different tools generate different identifiers, which hampers incremental loading, because two model entities from different sources that represent the same code element cannot be linked through the unique identifiers.

A solution that leverages the best of both worlds is the following, is to use intrinsic unique names for the named entities and meaningless unique identifiers for the non-named entities. Such a combination preserves the advantages of naming and is able to uniquely identify any non-named element as well. Clearly, complexity increases when the two separate schemes are mixed.

A similar solution is to provide both solutions and use them where appropriate without mixing them. However, when an entity is referencable in two different ways, incremental loading is not possible. If a certain entity is not in the current model, it cannot be determined that a reference by name and by unique identifier are actually supposed to reference the same entity.

Comparison with industry standards

CDIF [Com94] and XMI [OMG98] promote an entity referencing schema that is *internal* to the file. As shown by the code example in section 3.10.3, CDIF relationships use an identifier that is associated to the entity in the context of only one single file. However, if entities need to be able to be referenced over multiple files, the referencing scheme needs to be unique over those files, if incremental loading is to be supported. One solution is to use UUIDs. Yet another approach is taken by XLink [DMO00], a standard for linking between elements in different XML files. It links elements in different resources by explicitly mentioning the resource the element is defined in. This implies that the exact location of the information must be known. The identifiers we have described, uniquely identify model elements within their scope, independent of the location of the file or database the information is stored in.

3.11 Metametamodeling Axis

Metametamodeling is about making the representation of the metamodel explicit. Hence, a metametamodel allows one to reason about, and possibly change the metamodel. Properties of a metamodel such as how it can be extended, can be defined using such an explicit meta layer. Two options need consideration:

- **Do you want to use an explicit metametamodel?** An explicit metametamodel results in a standard way of describing metamodels. This description can be used for the following capabilities:
 - *identify metamodels*, i.e., are different tools compliant to the same or compatible metamodels? This issue can be solved simply by providing meta-information representing the metamodel.
 - *auto-adapt platform*, i.e., tools can use an explicit metamodel description to customize themselves. For example, a user interface can show, or a saver can store, any information according to any metamodel, because it knows the structure of the metamodel through the metametamodel.
 - *create metamodels*, i.e., meta-tools use the metamodel description to create metamodels themselves. This requires an approach where the metamodel description is interpreted to create appropriate metamodel representation. For instance, it is possible to interpret UML description in MOF to create UML compliant tools.
- **Do you want use a standard metamodel?** A standard metametamodel such as the MOF [OMG97] has the advantage that you do not have to build it yourself. Furthermore, it allows you to use compliant tools and other metamodels that adhere to the standard. However, when you build your own metametamodel you have full control over its capabilities, for instance, that it supports all the extensibility mechanisms you need.

Dependencies

Extensibility. The extensibility capabilities of a metamodel are determined by its metametamodel. A standard metametamodel might restrict you too much.

Implementation issues

The implementation issues include the storage of model information and the use of explicit metamodel descriptions.

Model Information. To precisely identify a model, information such as the implementation language, the tool that extracted the information, the date of creation of the model and the metamodel to which the model conforms. Such model information can be represented in two different ways:

- in the metamodel. For example, headers of CDIF exchange files include such a kind of information in addition to the complete description of the metamodel itself described in terms of the CDIF metametamodel [Com94].
- as a common entity of the model. This approach is simpler as the model information is treated as any model element. It is also independent of any metametamodel. However, because the model information is information about the model rather than representing a source code element, tools sometimes need to be aware of the existence of such a specific entity.

Metamodel creation and tool adaptation. Using a metametamodel to generate model is straightforward as long as it deals with the structural aspect of the metamodel. The specification of entity behaviour,

however, for instance, how to compute all inherited methods of a certain class, additionally requires a language that is able to formally express behaviour in terms of the metamodel.

3.12 Conclusion

In this chapter we have defined a design space for the infrastructural aspects of metamodels and repositories for reengineering environments. The space spans multiple, non-orthogonal axes, which are either directly related to requirements for a repository (the requirement axes) or they are pure implementation aspects (the implementation axes). The design space makes explicit the options for every axis, the trade-off between the different options and the dependencies between the axes.

The next chapter describes FAMIX, a language-independent metamodel for modelling object-oriented software. We discuss the infrastructural design choices for FAMIX according to the axes of the design space of this chapter. As such the design space provides the infrastructural framework in which FAMIX must be viewed. At the same time FAMIX validates the suitability of the design space to describe its infrastructural aspects.

CHAPTER 4

FAMIX, a Language-Independent Metamodel for Modeling Object-Oriented Software

This chapter introduces FAMIX, a metamodel for modelling object-oriented software. The main goal is to support reengineering activities in a language-independent way. The aim is not to cover all aspects of all languages, but rather to capture the common features that we need for reengineering activities, so tools can be easily reused for multiple target languages.

This chapter gives an overview of the contents of the FAMIX metamodel, as well as how and why the information is modelled as it is. Firstly, the chapter discusses the organisation of the information. The metamodel consists of a language-independent core. Language mappings describe how language-specific constructs are mapped to the core model and how the core model is extended with language-specific information. Secondly, the chapter places FAMIX in the design space presented in chapter 3 by discussing the design choices we have made for the infrastructural issues that form this design space. The complete specification of the model can be found in appendix B.

One might wonder why we came up with our own model in the first place. One reason is that when we started we did not find any model that adequately modelled object-oriented source code in the way we needed it to. As section 2.3.2 shows, several source code models exist, but they are either focused on another language paradigm [Kos00], or focus on one language only [CGK98], or they are aimed at multiple languages but do not have explicit definitions of what is modelled and how the different languages map to the language-independent part of the model. We have also looked at models such as Unified Modelling Language (UML) [OMG99]. However, they are directed towards object-oriented analysis and design rather than source code representation. Section 4.8 compares the information FAMIX models to UML and discusses in-depth why UML currently does not fit our needs.

We start with a presentation of the requirements the metamodel should fulfil. Following we give an overview of what information is represented in the metamodel and how this information is modelled. We finish with a discussion why we did not use UML and a conclusion.

4.1 Requirements

This section introduces the requirements for the FAMIX metamodel. They are strongly influenced by the requirements of the FAMOOS project under which the major part of this work has been realised [DD99]. The FAMOOS project had multiple partners with large object-oriented legacy systems in different languages. Furthermore, the tools of the different partners should be able to interchange information. This brings us to the following list of requirements:

- *Support for multiple languages.* The metamodel must support multiple object-oriented implementation languages. It must abstract from those languages to allow tools to be used without adaptation for the different supported languages. In particular the metamodel needs to support C++, Java, Smalltalk and Ada.
- *Support for the whole reengineering lifecycle.* The metamodel is targeted at software analysis and reengineering. It should, therefore, contain relevant information for tasks such as metrics computation, grouping and reorganisation operations.
- *Extensibility.* The metamodel needs to be extensible to deal with language extensions, tool-specific extensions and other information that is not represented in the core metamodel that is considered useful. Furthermore, it is required that any information about a model element can be attached to it to store insights gained during exploration.
- *Scalability.* The metamodel needs to support multi-million line software systems.
- *Information exchange.* The metamodel must support textual information exchange. For the format human readability and machine processability aspects as well as industry standards support need to be addressed.

4.2 Overview of the FAMIX core

The FAMIX metamodel models *multiple object-oriented languages*, i.e., in terms of the design space of chapter 3, it supports multiple languages within one paradigm. It defines a *language-independent core*, which allows tools to be reusable without adaptation over the supported languages. How languages are mapped to the core and which language specifics can be stored, is specified in *language extensions*. This section presents the core part of FAMIX. The language extensions are discussed in detail in section 4.4.

The metamodel represents source code at the *program entity level* (see also section 3.3). First of all, this level of information is sufficient for the analysis tasks we want to support. The information allows one to perform structural analysis and dependency analysis. It supports metrics computation and heuristics. It does not support control flow analysis and the regeneration of source code from the model. We store, however, the location of the source code, allowing one to obtain additional information from the source code itself. A second reason to choose the program entity level is that more detailed information increases the size of models considerably which hampers scalability. Thirdly, the program entity level enables to abstract from language-specific details and as such allows for a clean language-independent metamodel.

Figure 4.1 shows the core entities and relations. All basic elements of an object-oriented languages are present (Class, Method, Attribute). Furthermore, FAMIX models dependency information, such as method

invocations (which method invokes which method) and attribute accesses (which method accesses which attribute). This is important information for, for instance, dependency and impact analysis [CGK98].

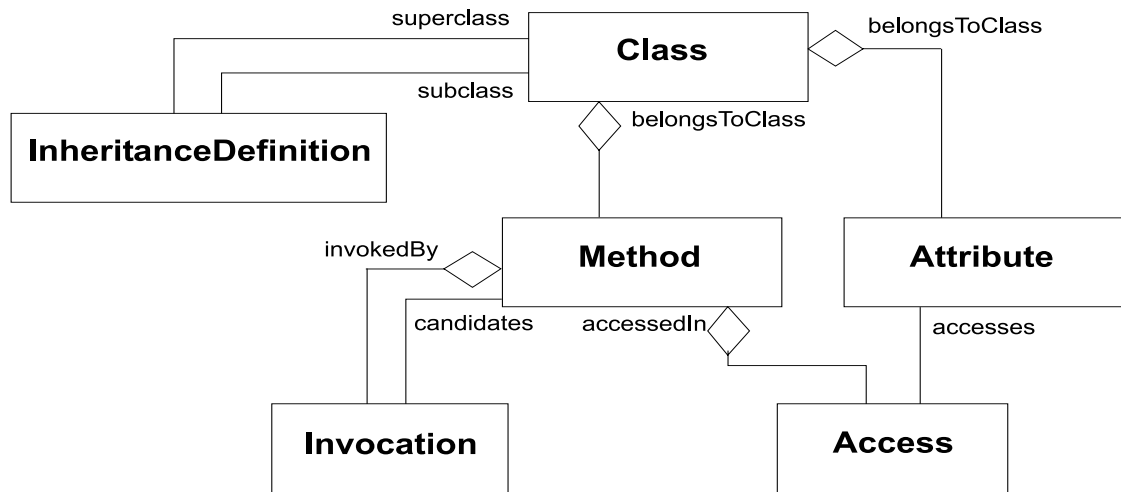


Figure 4.1 The core of the FAMIX model

The complete metamodel is not restricted to the above elements. Additionally it also models different kinds of variables, functions and arguments. We give here a short description of these elements. The exact specification can be found in appendix B. They are modelled in an object-oriented hierarchy, which is shown in Figure 4.2.

- Function - a definition of behaviour with global scope
- LocalVariable - a variable local to a method or function
- GlobalVariable - a variable with global scope
- ImplicitVariable - variables that are not explicitly defined such as self, this and super
- FormalParameter - a parameter of a method or function
- AccessArgument - an argument of an invocation that constitutes a simple variable access
- ExpressionArgument - an argument of an invocation which is an expression
- Package - a scoping mechanism
- Model - a meta entity containing information about a model such as creation time

Functions and global variable are modelled because they exist in several object-oriented languages we want to cover such as C++ and Smalltalk. This effectively makes FAMIX support hybrid object-oriented and procedural languages.

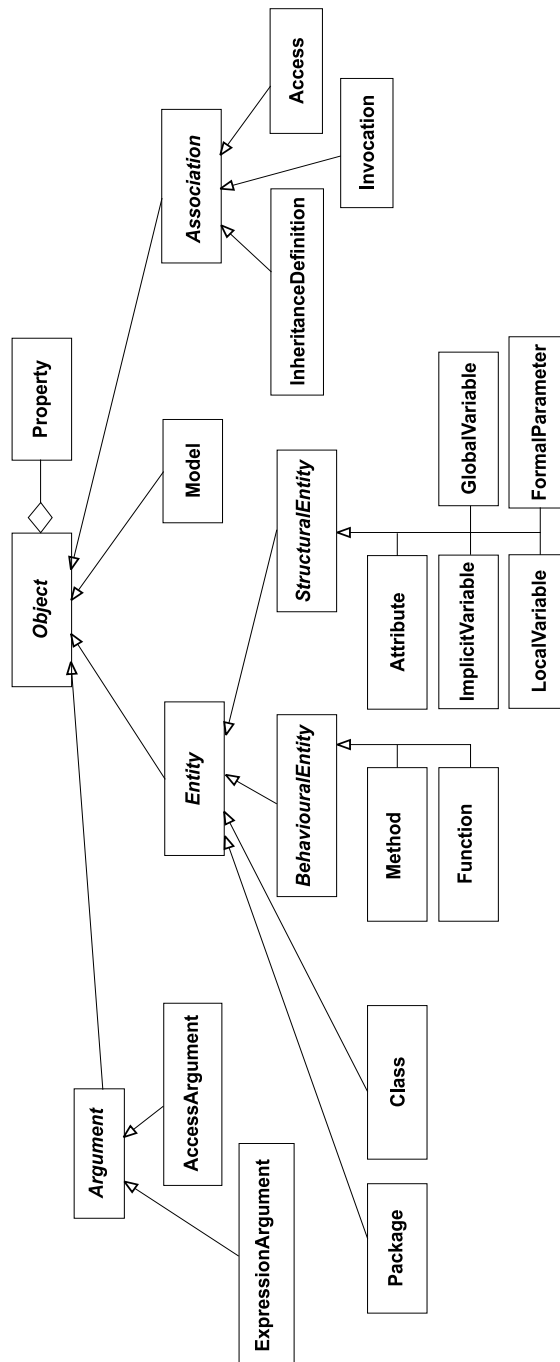


Figure 4.2 The complete hierarchy of the FAMIX model

For every model element the metamodel defines a set of attributes. A Method, for instance, has attributes such as `signature` and `isAbstract`. Figure 4.3 shows the Method entity in the FAMIX inheritance hierarchy.

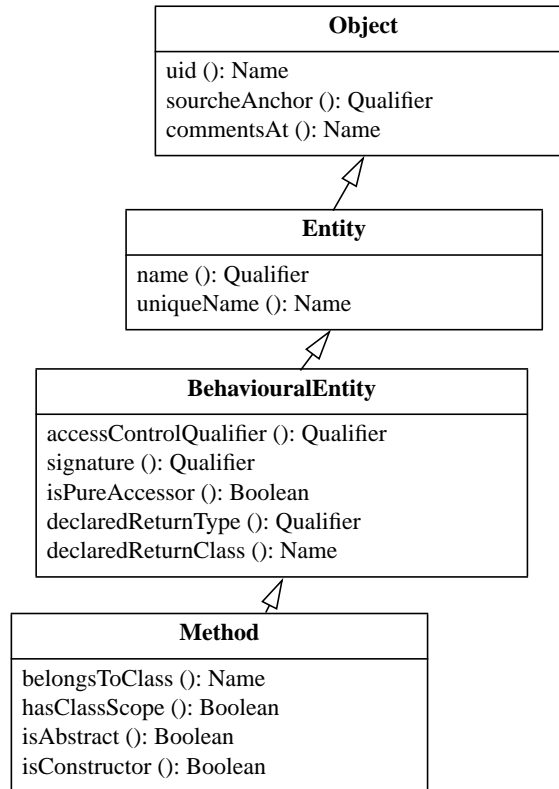


Figure 4.3 The Method entity in the FAMIX metamodel hierarchy

The semantics are well defined: for every entity and attribute it is described what it is supposed to model, what its value can be and how it should be interpreted. For example, the `isAbstract` attribute of method is describes as follows:

`hasClassScope: Boolean; optional`

Is a predicate telling whether the method has class scope (i.e., invoked on the class) or instance scope (i.e., invoked on an instance of that class). For example, static methods in C++ and Java have a `hasClassScope` attribute set to true.

The description of all attributes can be found in appendix B.

4.3 Extensibility

Before going into the details of the language extensions, we present the ways in which the FAMIX meta-model can be extended:

- *New model elements.* An extension can define new model elements. Examples are the Include relationship for the C++ extension [Bar99] and the Measurement element for the metrics extension.
- *New attributes to existing model elements.* Existing elements can be extended to allow one to store additional information. An example is the `isFinal` attribute that the Java extension adds to the definition of the Method element (see appendix D).

Section 3.6 discusses the advantages and disadvantages of class extension. One of the problems we have encountered is that not all standard metamodels support class extension. In the context of textual information exchange we have worked with CDIF [Com94] and XMI [OMG98]. The CDIF metamodel supports class extensions, the XMI metamodel, i.e., the MOF, does not. This means that class extensions cannot be expressed in XMI and exchanged with generic XMI compliant tools.

- *Annotations.* Any model element can be annotated by attaching a Property to it. This is shown in Figure 4.4 by the Object class, which can have zero or more Properties attached to it.

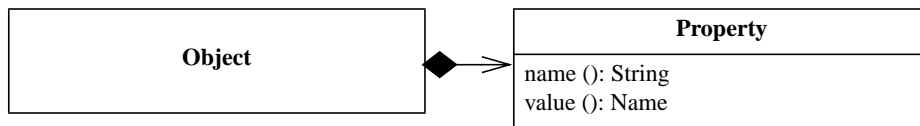


Figure 4.4 FAMIX model elements can be annotated with Properties

4.4 Multiple language support

The FAMIX metamodel supports multiple object-oriented languages. This section describes the design decisions that make it easier to support more rather than one specific language. Afterwards we describe the actual language extensions.

4.4.1 General multi-language design decisions

The following decisions in the design of FAMIX are relevant for the support of multiple languages.

Multiple inheritance. FAMIX supports multiple inheritance. This allows us to deal with single inheritance languages such as Smalltalk, but also with multiple inheritance languages such as C++. Java also fits this scheme by interpreting Java interfaces as abstract classes and interface implementation as common inheritance.

Statically typed and dynamically typed languages. Static type information is important to store, because it reveals important dependencies. If the information is not known, which is normally the case with dynamically typed languages such as Smalltalk, the information is left empty. For instance, Figure 4.3 shows that Method inherits the `declaredType` attribute. It is used to store the statically declared returntype for methods, like Point for the method declaration `Point getPoint() { ... }` in Java. It is left empty for Smalltalk methods.

Another example of supporting both dynamic and static typing is the candidate methods of an invocation. Figure 4.5 shows the Invocation entity. The `candidates` attribute stores the methods possibly invoked by this invocation. In Smalltalk, without static type information, the candidates are all methods in a

system that have the signature as stored in the `invokes` attribute¹. In Java the static type information reduces the possibly invoked methods to a single inheritance hierarchy or interface implementation hierarchy. By storing the candidates independent of the way the information is collected, tools can use the information independent if it concerns a dynamically or statically typed language.

Invocation
invokedBy (): Name
invokes (): Qualifier
base (): Name
receivingClass (): Name
candidatesAt (): Name

Figure 4.5 The FAMIX Invocation

Pointer, array and other non-Class types. FAMIX does not explicitly model pointer, primitive array and other primitive types. The BehaviouralEntity class in Figure 4.3 illustrates how FAMIX deals with such types. The class has two attributes `declaredReturnType` and `declaredReturnClass`. `declaredReturnType` stores the complete type as it is declared in the source code. This can be a regular type such as a class, but also primitive types, e.g., `int` in Java, or a primitive array type, e.g., `Point[]` in Java, or a pointer type, e.g., `Point*` in C++. The `declaredReturnClass` attribute, however, always stores the class that is implicit in the `declaredReturnType` and that exists as an entity in the model. Table 4.1 shows the declared type and declared class for the examples given before. The advantage of this approach is that the model does not need to model all complex types of the separate languages. At the same time the type information is not lost and the dependency information with classes in the model is retained. A disadvantage is that tools that want to use the complex type information, need to extract it themselves from the declared type.

source code	declared (return)type	declared (return)class
Point	Point	Point
Point[]	Point[]	Point
Point*	Point*	Point
int	int	

Table 4.1: Declared type and declared class in FAMIX

4.4.2 Language mappings and extensions

An important part of the usability of the metamodel depends on how the actual programming languages are mapped to language-independent constructs. The goal is to treat as many concepts of different languages as possible uniformly. On the other hand, we store information about the mapping, because the semantic difference of a similar concept in different languages might be of interest for certain tools. Figure 4.6 shows

1. There are three special cases where the set of candidate methods can be reduced, namely if the receiver of the invocation is `self`, `super` or a classname.

the Java mapping and extension to the Class entity. The common definition of Class includes the attributes `isAbstract` and `belongsToPackage`. The extension specifies additional properties specific to a Java class, namely if it is declared `public` or `final`. Furthermore, the Java interface concept is mapped to a FAMIX Class as well, with `isInterface` and `isAbstract` set to true. Language-independent tools just use the common interface and tools that want to use the language-specific information, have it available.

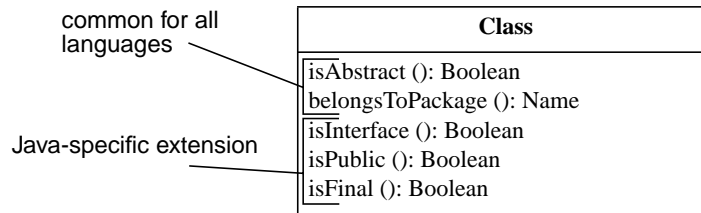


Figure 4.6 The Java mapping and extension to Class

Like in the core model the extensions describe in detail what the model elements and their attributes mean. Furthermore, for the common attributes the interpretation for the specific language is described. An example is the `isAbstract` attribute:

`isAbstract`

In Java a class is abstract if the class is declared abstract. This is obligatory if one or more of its methods are abstract. Even if the class does not contain any abstract methods, it can be declared abstract, preventing the class from being instantiated. Interfaces are always abstract, but do not have to be explicitly declared as such.

Extensions can also define new model elements. The `TypeCast` entity for Java is an example (see Figure 4.7).

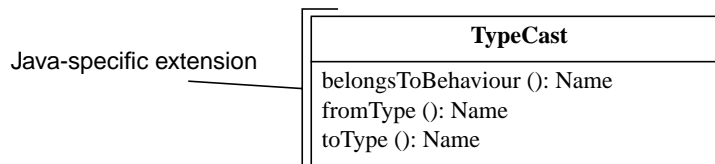


Figure 4.7 The Java TypeCast entity

Following we summarize the mappings that are most important. The complete specification of the separate mappings can be found, for Smalltalk in appendix C, for Java in appendix D, for Ada in [Neb99] and for C++ in [Bar99].

Classes, Interfaces, Metaclasses and Structs. Classes in the different languages are modelled as classes in FAMIX. Also Java interfaces, as discussed in the example above, Smalltalk metaclasses and C++ structs are mapped to classes. A Java interface is modelled as an abstract class that only contains abstract method definitions and final attributes. An implicit Smalltalk metaclass is modelled as an explicit class. Consequently, Smalltalk class methods are modelled as instance methods of the FAMIX class instance representing the metaclass. In Smalltalk there exist two kinds of class variables. One is the common class variable which is modelled as an attribute with class scope. The second is the class instance variable, which is

modelled as instance scope attribute of the metaclass. C++ structs are modelled as classes as well. They differ only from C++ classes in that their members have default public instead of private visibility [Bar99].

Methods, Constructors and Destructors. Apart from common methods the Method entity covers Java and C++ constructors as well. Special rules apply, however, that tools might need to be aware of. For example, a constructor has to have the same name as its class, it does not have a return type, and the syntax to invoke it is different from a normal method invocation. In Smalltalk constructors are common methods. A method can be interpreted to be a constructor if it is a class method returning an instance of its defining class.

The fact if a method represents a constructor or a destructor is stored in the `isConstructor`, respectively `isDestructor` attribute.

Global variables. FAMIX defines a `GlobalVariable` entity. Smalltalk and C++ have global variables where Java does not. In Smalltalk classes are global variables as well. Consequently, attributes in Smalltalk cannot have the same name as a class (or any other global variable), because this might hide these globals in the scope of the new attribute. In Java types and attribute names do not interfere. In FAMIX we model Smalltalk classes as `Classes` and not as `GlobalVariables`.

Abstractness. For Java and C++, abstractness is straightforward to determine as a class or method is explicitly tagged with the ‘abstract’ keyword respectively as pure virtual. In Smalltalk abstractness is implicit. A method is abstract when it invokes the method `subclassResponsibility`. This method throws an exception at runtime with the message that this method should not be called but a subclass method instead. Similarly a Smalltalk class can be interpreted to be abstract if it has one or more abstract methods. However, it is still possible to instantiate this class. Although abstractness is implicit for Smalltalk methods and classes, we explicitly model them as abstract in FAMIX.

4.5 Reference Schema

In FAMIX, model elements can be referenced in two ways. Firstly, every model element has a unique identifier (see the attribute `uid` in `Object` in Figure 4.2 and Figure 4.3). Secondly, all elements that have an intrinsic unique name, i.e., all instances of `Entity` and its subclasses, can be uniquely identified by that name (see the `uniqueName` attribute in `Entity` in Figure 4.2 and Figure 4.3). The advantages and disadvantages of naming and unique identification are extensively discussed in section 3.10.

FAMIX uses a unique naming scheme that is normalized over multiple languages. It uses the intrinsic unique name of a program element. It ensures that names generated by different tools are the same and tools need to deal with only one naming schema. For example, a method name must look like

```
package::subpackage::classname.methodname(para1,para2)
```

The details of the naming scheme can be found in section 2.1.2 of appendix B.

In FAMIX, we reference elements as much as possible by their unique name rather than their `uid`, because the advantages of human readability and the fact that multiple tools independently generate the same unique name. An example is the `belongsToClass` attribute of `Method` (see Figure 4.3). It will always reference a named element, namely a `Class`, and thus the unique name of that element is used rather than an unique identifier.

Non-named elements such as relations can be identified using unique identifiers. Basically any identifier can be used as long as it is unique in the scope that you need. We have experimented with plain integers and Universal Unique Identifiers (UUIDs) [OG97]. Plain integers are only locally valid. UUIDs allow us to uniquely identify model elements over multiple models and reference model elements over different storage media. They increase the size of exchange files up to 9% (see Table 3.3) compared to using plain integers. We find this increase a minor detriment compared to the advantage of universal uniqueness¹.

We have also experimented with using a ‘best of both worlds’ approach where we use the unique name as unique identifier for the named entities and only use non-intrinsic identifiers for the non-named elements. For example, a group referencing both named and non-named elements would then contain names for the named entities and meaningless identifiers for non-named entities. Such a mixed scheme keeps the advantages of using names for the named entities, but allows other elements to be referenced anyway. Furthermore, it solves the above incremental loading problem. Disadvantages are the increased complexity, because schemas with different properties are mixed. We find the advantages outweighing the disadvantages.

Generally unique names together with UUIDs support incremental loading and grouping because the identification is valid over multiple databases or exchange files. However, when an entity is referencable in two different ways, incremental loading is not possible. If an entity is not in the current model, it cannot be determined that a reference by name and by unique identifier are actually supposed to reference the same entity.

Generally unique identifiers support grouping of all elements. The FAMIX referencing schema does not take multiple models into account. Although the UUID is valid over multiple models as well, the unique naming schema does not take the model an entity belongs to, into account.

4.6 Support for information Exchange

FAMIX uses the chunk format described in section 3.9. This implies that the entities are not nested into their scoping entity and simple relationships are stored as attributes of the entity it is linked to. The format is chosen, because of its human readability and support for incremental loading (the latter together with the FAMIX entity reference schema discussed in section 4.5). Figure 4.8 shows an example using the CDIF standard format [Com94].

The chunk format defines its own way of storing relationships rather than using an existing standard such as XMI and how it encodes relationships. In the chunk format relationships are represented either as attributes of model elements for containment relationships (e.g., the `belongsToClass` attribute of `Method` in the example in Figure 4.8), or as explicit entities for the other, mostly attributed, relationships (e.g., the `InheritanceDefinition` entity in Figure 4.8). There is no specific relationship kind of elements.

Sofar we have exchanged information with the CDIF [Com94] and XMI [OMG98] standard. Their relationship representations are not expressive enough to cover our needs. Both XMI and CDIF reference ele-

1. Actually, the bigger problem we encountered was in the implementation of the Moose Reengineering Environment (see chapter 5). Our current implementation of UUID creation is very resource intensive causing considerable delays in load and save times. However, we have not yet seriously attempted to optimise it.


```
(Class FM1
  (uid "c842bf06-d202-0000-0282-5c410d000000")
  (name "Widget")
  (uniqueName "gui::Widget")
  (isAbstract -FALSE-)
  (sourceAnchor #[file "factory.h" start 260 end 653|]#)
)

(Method FM2
  (uid "c842bf06-d202-0000-0282-5c410d000001")
  (name "Widget")
  (uniqueName "gui::Widget.Widget()")
  (signature "Widget()")
  (belongsToClass "gui::Widget")
  (sourceAnchor #[file "factory.h" start 321 end 326|]#)
  (accessControlQualifier "public")
  (hasClassScope -FALSE-)
  (isAbstract -FALSE-)
  (declaredReturnType "")
  (declaredReturnClass "")
)

(InheritanceDefinition FM3
  (uid "c842bf06-d202-0000-0282-5c410d000002")
  (subclass "gui::ScrollBar")
  (superclass "gui::Widget")
  (accessControlQualifier "public")
  (index 1)
)

(Property FM4
  (uid "c842bf06-d202-0000-0282-5c410d000003")
  (name "LOC")
  (value "56")
  (belongsToID "c842bf06-d202-0000-0282-5c410d000000")
)
```

Figure 4.8 FAMIX information in CDIF format

ments only locally to a file¹. For instance, the belongsToClass relation as a CDIF relation rather than an attribute of Method looks like:

```
(Method.belongsToClass.Class REL1 FM2 FM1)
```

(with REL3, FM1 and FM2 being local CDIF references). Both standards, however, provide ways to represent entities with attributes, and thus can we encode our relationships like that.

4.7 Metametamodeling

This section discusses the metametamodeling issues introduced in section 3.11. FAMIX does not have an explicit metametamodel. Its metametamodel is an implicit entity-relationship model. In the context of textual information exchange the metamodel has been described using the CDIF metametamodel and the MOF of XMI. Both metametamodels do not have the expressive power needed to describe FAMIX and its extensions they way we want it. For instance, XMI does not support class extensions as discussed in section 4.3. Furthermore, both CDIF and XMI only support entity references local to a file. By defining our own relation entities and accompanying reference schema, we go around the latter problem (see section 4.6).

For metametamodel-based tasks such as tool generation either the CDIF or XMI description can be used. In our FAMIX-based tool environment we are also experimenting with an explicit metamodel description for the same purpose.

To store information about the model such as its creation time, FAMIX contains an explicit Model entity (see Figure 4.9). The choice to go with an explicit entity is motivated by the fact that it firstly explicitly defines what information can be stored about a model in a way that is independent of any metametamodel or information exchange standard.

Model
exporterName (): String exporterVersion (): String exporterDate (): String exporterTime (): String publisherName (): String parsedSystemName (): String extractionLevel (): String sourceLanguage (): String sourceDialect (): String

Figure 4.9 The Model entity in FAMIX

4.8 Why not UML?

The previous sections show how we model object-oriented software for the purpose of reengineering in the form of the FAMIX metamodel. We have also considered object-oriented analysis and design (OOAD)

1. XLink [DMO00] is a standard that is integrated in XMI that allows to link to entities 'somewhere else' (as specified with a Universal Resource Identifier (URI) [BLFIM98]). However, XLink stores a specific location, where our referencing schema stores a unique name or identifier that is independent of the location of storage.

models, but found them unfit to adequately model software for reengineering [DDT99]. In this section we discuss why, in particular looking at the Unified Modeling Language (UML) [OMG99], which is becoming the de facto standard for software modeling. Therefore, it seemed an interesting candidate for our purposes.

Primarily, the problem is that UML is specifically targeted towards OOAD and not at representing source code as such. The specification itself says [OMG99]:

The UML, a visual modeling language, is not intended to be a visual programming language, in the sense of having all the necessary visual and semantic support to replace programming languages. The UML is a language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system, but it does draw the line as you move toward code.

The problems are illustrated by Figure 4.10 and can be summarized as follows:

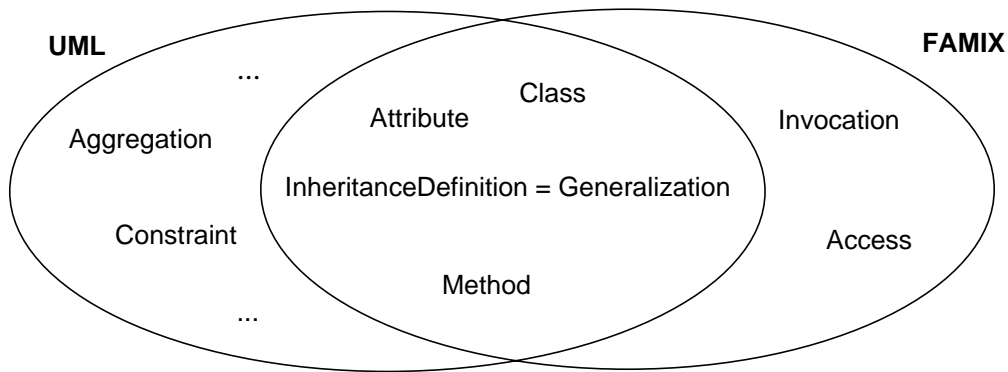


Figure 4.10 Comparison UML with the FAMIX core model

1. The UML metamodel defines a large number of concepts that are not relevant for an implementation model. ‘Aggregation’ and ‘Constraint’ are two examples but there are many more.
2. There is a substantial overlap between the FAMIX core model and UML. With some flexibility it is possible to map ‘InheritanceDefinition’ onto ‘Generalization’ and ‘Class’, ‘Method’ and ‘Attribute’ on their respective counterparts with the same name. However, non-standard interpretation of concepts breaks the standard and thus tools that expect the standard interpretation. For instance, inheritance in source code does not always represent a generalization relationship, but can represent an implementation inheritance relationship.
3. Due to its OOAD focus, UML lacks some concepts that are necessary in order to adequately model source code. Especially the Invocation and Access concepts are non-existent. There are several ways to extend UML to incorporate these concepts. We present the most plausible solutions:
 - Use the Usage dependency to model Invocation and Access. A Usage dependency represents “a relationship in which one element requires another element (or set of elements) for its full implementation or operation” [OMG99]. A Usage can be stereotyped to represent a *call*, which specifies that a source operation invokes a target operation. However, an UML Operation specifies a specification of an operation rather than its implementation. The implementation of an operation is represented by a UML Method. The body of a method contains the in-

vocations we want to represent and consequently we need a dependency between a Method and an Operation rather than between an Operation and an Operation to represent such an invocation. It is not clear from the UML specification if the intention of a *call* is the same as representing an FAMIX invocation. Furthermore, an additional dependency or stereotyped dependency would be needed between Methods and Attributes to represent an attribute access.

- Use CallAction to model Invocation and Access. However, because a CallAction is defined in the context of instances, i.e. runtime entities, rather than classes a non-standard interpretation is needed for this solution to work.
- Stereotype Association to let it represent an Invocation or Access. However, an Association normally associates to Classes, not two Methods or Methods and Attributes. Therefore, extra information needs to be stored in an Association to reference the actual contained entities involved.
- Use the MOF, the OMG Meta Object Facility [OMG00], the metamodel of UML to create to add new concepts to the language. However, this is a major UML extension which will not be supported by most UML-aware tools.

So modeling Invocations and Accesses requires either minor extensions that need non-standard interpretations, or major extensions that will break most tools. Apart from the harder Invocation and Access, there are no straightforward ways to model entities such as GlobalVariables, Functions and implementation ‘details’ such as LocalVariables either.

Concluding we can say that UML *in stricto sensu* is not sufficient for modelling source code for the purpose of reengineering.

4.9 Conclusion

This chapter describes the FAMIX metamodel, which models object-oriented source code at the program entity level. It is aimed at supporting tools for analysing and reorganising object-oriented legacy systems. We have discussed the properties of the FAMIX model are discussed according the design space defined in chapter 3. This is summarised in Table 4.2. Comparing to the requirements we posed in section 4.1, we see that FAMIX is designed to support multiple languages, extensibility and information exchange. Scalability is addressed by the support of incremental loading and choosing the program entity level of detail rather than the AST level of detail. From the reengineering tasks grouping is explicitly supported. The support for other tasks can only be validated by actually using the model a basis for tools that implement them. Multiple models are not supported. Although the unique identifiers allow one to uniquely identify entities in multiple models, the unique naming scheme does not incorporate model names. The Storage Medium Axis is not discussed, because it is a repository implementation issue rather than a metamodel design issue.

The applicability of FAMIX as a metamodel for reengineering has been validated in the following ways. A tool environment with a repository based on FAMIX has been implemented and used as a basis for a series of reengineering tools. This tool environment is called Moose and is described in chapter 5. Furthermore, an in-depth analysis has been performed to evaluate the suitability of FAMIX to support refactorings on a language-independent level. Refactorings strain the model to its limits in terms of the completeness of the provided information and the ability to abstract from the modelled implementation languages. The refactoring analysis is presented in chapter 6 and a tool prototype implementing and validating this analysis is presented in chapter 7.

Requirement Axes	
Language/Paradigm	Multiple object-oriented and object-oriented/procedural hybrid languages. The model has a language independent core with explicit language mappings and extensions.
Level of detail	Program-entity level
Multiple models	not supported
Grouping	supported
Incremental loading	supported
Extensibility	New entities can be added, new attributes to existing entities. Annotations are supported on the metamodel level (an Object can have Properties).
Implementation Axes	
Entity Reference	A unique naming scheme supports incremental loading and referencing over multiple files. FAMIX also uses UUIDs to support these features uniformly for all model elements, not only elements with an intrinsic unique name.
Exchange Format Structure	The model is designed to support the chunk format. N-ary attributes are supported, but its representation in the text files depends on the standard used (CDIF hacked, because it does only support multi-valued attributes for a restricted set of primitive types, XMI with multiple instances of single-valued attributes)
Metametamodeling	Explicit definitions of FAMIX as instance of the CDIF and the MOF metamodel exist that can be used for the purpose of tool generation. An explicit Model element stores context information about a model, such as the date and time of extraction, the name of the extracted system, its implementation language, etc. The Model element goes around exchange format facilities such as CDIF headers to store this kind of information

Table 4.2: FAMIX according to the design space from chapter 3

CHAPTER 5

The Moose Reengineering Environment

This chapter presents the Moose Reengineering Environment, a language-independent tool environment to reengineer object-oriented systems. Its repository is based on the FAMIX metamodel presented in chapter 4. Apart from the tool environment itself, we present some of the tools we have built on top of it and the industrial case studies we have performed.

In this thesis Moose functions as the validation of the ability of FAMIX to support multiple reengineering tools that work on real-world legacy systems. With real-world we mean that the tools (and thus the environment and thus the FAMIX-based repository) can deal with legacy software of industrial size and complexity while providing the user with useful results. In particular we discuss the relevance of the information FAMIX represents, its language independence, as well as its extensibility and information exchange properties.

The chapter starts with a set of general requirements for a reengineering environment based on literature and experience. We then give an overview of the architecture of Moose and discuss its built-in services for querying and navigation model information, metrics computation, refactoring, source code parsing and model exchange. For every service we discuss how well they are supported by FAMIX. Then we introduce some of the tools that are built around it. First, the tools show that Moose indeed facilitates concrete reengineering tasks. Secondly, they validate Moose as a tool environment: multiple tools use Moose as their information base and cooperation platform. After the tools we present the industrial case studies and a discussion to which extent Moose and FAMIX are up to the task. We review how well our requirements are met and present some general insights we gained while building and applying Moose. We wrap up the results in a conclusion.

5.1 Requirements for a Reengineering Environment

Based on our experiences and on the requirements reported in literature [MN97] [HEH⁺96] [Kaz96], these are our main requirements for a reengineering environment:

- **Support for reengineering tasks.** An obvious requirement which determines the focus of the tool. It determines the information to store and which services the environment provides. Typical reengineering tasks are metrics, grouping, visualisation and refactoring.
- **Extensible.** An environment for reverse engineering and reengineering should be extensible in many aspects:
 - The repository (and thus its underlying metamodel) should be able to represent and manipulate entities other than the ones directly extracted from the source code (e.g., measurements, associations, relationships).
 - To support reengineering in the context of software evolution the environment should be able to handle several source code models simultaneously.
 - The environment should be able to use and combine information from various sources, for instance the inclusion of tool-specific information such as runtime information, metric information, graph layout information, etc.
 - The environment should be able to operate with external tools like graph drawing tools, diagrammers and parsers.
- **Exploratory.** The exploratory nature of reverse engineering and reengineering demands that a reengineering environment does not impose rigid sequences of activities. The environment should be able to present the source code entities in many views, both textual and graphical, in little time. It should be possible to perform several types of actions on the views the tools provide such as zooming, switching between different abstraction levels, deleting entities from views, grouping entities into logical clusters, etc. The environment should also provide a way to easily access and query the entities contained in a model. To minimize the distance between the representation of an entity and the actual entity in the source code, an environment should provide every entity with a direct linkage to its source code. A secondary requirement in this context is the possibility to maintain a history of all steps performed by the reengineer and preferably allow him to return to earlier states in the reengineering process.
- **Scalable.** As legacy systems tend to be huge, an environment should be scalable in terms of the number of entities being represented. Furthermore, it should provide meaningful information at any level of granularity. An additional requirement in this context is the actual performance of such an environment. It should be possible to handle a legacy system of any size without long latency times.
- **Information Exchange and Tool Integration.** A reengineering effort is typically a cooperation of a group of specialised tools [DDT99]. Therefore, a reengineering environment needs to be able to integrate with external tools, either by exchanging information or ideally by supporting runtime integration.

In addition to these general requirements, the context of the FAMOOS project [DD99], in which Moose was originally developed, imposed the following requirement:

- **Support for multiple object-oriented languages.** This specific tool environment must support the reengineering of software systems written in C++, Java, Ada and Smalltalk.

5.2 Architecture

This section presents the architecture of Moose (see Figure 5.1). The repository is based on the FAMIX metamodel (see chapter 4). Consequently, the functionality of the services and import/export framework is

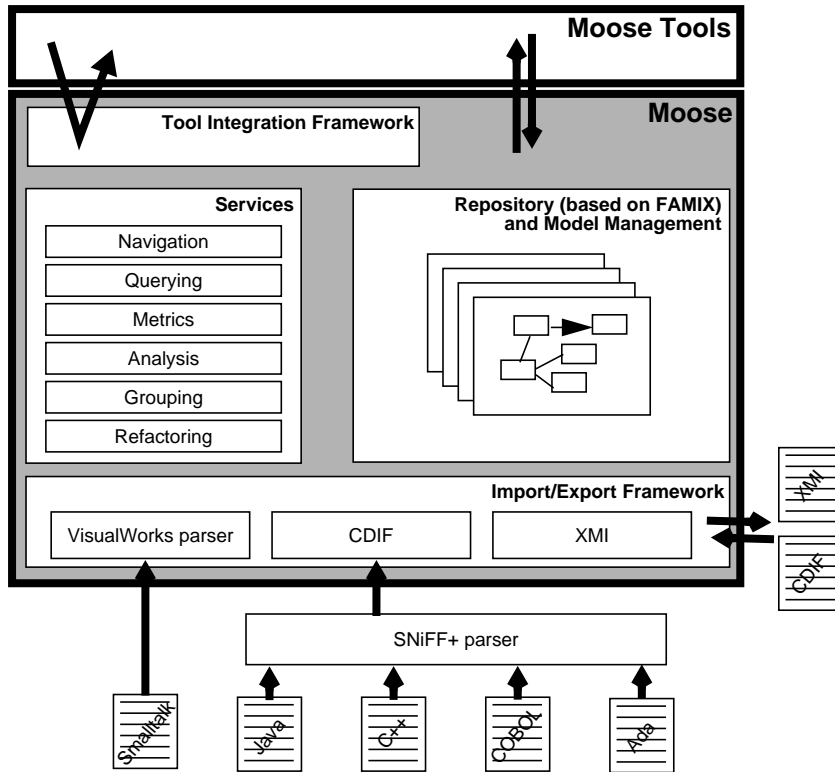


Figure 5.1 Architecture of Moose

tailored towards and constraint by the information that is defined by FAMIX. Moose has a layered architecture. The rest of this section gives an overview of the architecture. In subsequent sections we discuss the different functionalities in detail including how well FAMIX supports them.

Repository and Model Management. The repository stores models of software systems. They contain elements representing the software artifacts of the target system. This information can be analysed, manipulated and used to trigger code transformations by means of refactorings. Moose can maintain several models in memory at the same time. The models are based on the FAMIX metamodel. Consequently the stored information has the following properties. It is *language independent*. This allows tools that use the repository to work without adaptation with legacy systems in different implementation languages (C++, Java, Smalltalk, Ada). It is also *extensible*. This allows tools to deal with information not anticipated by the core metamodel. It also allows to store language-specific information (e.g., to analyse include hierarchies in C++) or tool-specific information such as analysis results and layout information for graphs.

Import/Export Framework. The import/export framework provides support to import information into and export information from the Moose repository. Import and export is possible in the following ways:

- In the case of VisualWorks Smalltalk — the language in which Moose is implemented¹ — sources can be directly extracted via the metamodel of the Smalltalk language or via the built-in parser.

- For other source languages Moose provides support for the import and export of CDIF [Com94] or XMI [OMG98] files based on the FAMIX metamodel. CDIF and XMI are industry-standard interchange formats for exchanging models via files or streams. Over this interface Moose uses external parsers for source languages other than VisualWorks Smalltalk. Currently C++, Java, Ada and several Smalltalk dialects other than VisualWorks are supported. Information exchange is discussed in more detail in section 5.7

Services. Moose provides several services that tools can use to perform their reengineering tasks:

- *Querying and Navigation.* Every element in a model is represented by an object, which allows direct interaction of elements, and consequently an easy way to query and navigate a model. The query and navigation support is discussed in detail in section 5.3.
- *Metrics and other Analysis support.* Moose's analysis services are mostly implemented as operators that can be run over a model to compute additional information regarding the software elements. For example, metrics can be computed and associated with the software entities. Section 5.4 provides more details.
- *Grouping.* Moose has a grouping mechanism, with which it can group several model elements into a group entity. This is useful to classify information or to provide views on the available information at different levels of abstraction. More details can be found in section 5.5.
- *Refactoring.* The Moose Refactoring Engine implements language-independent refactorings. Section 5.6 describes the engine in more detail.

Tools Layer and Tools Integration Framework. The functionality which is provided by Moose is to be used by tools. This is represented by the top layer of Figure 5.1. Tools can use the repository and services of Moose and use the Tools Integration Framework to find each other to interoperate. The Tools Integration Framework and examples of tools based on Moose are described in section 5.7.2.

The following sections discuss the different parts of Moose in more detail. For every part we discuss how well it is supported by FAMIX.

5.3 Querying and Navigation

One of the challenges when dealing with large complex metamodels is how to support their navigation and facilitate easy access to specific entities. In the following subsections we present two different ways of querying and navigating source code models in Moose.

5.3.1 Programming Queries

The fact that the metamodel in Moose is fully object-oriented together with the facilities offered by the Smalltalk environment, makes it simple to directly query a model in Moose. We show two examples of queries, both of them in the Smalltalk language. The first query finds all the methods accessing the attribute name of the class `Node`. It first asks the model for the entity with name `Node.name`. From the access information of this attribute, it then collects all Method entities that access the attribute.

1. Moose is implemented in VisualWorks 3.0 with Envy 4.0.

```
(MSEModel currentModel
  entityWithName: #'Node.name')
  accessedByCollect:
    [:each | MSEModel currentModel entityWithName: each accessedIn]
```

The second query collects all classes that have more than 10 descendants. It asks the current model for all classes that have a property called WNOC (the metric Weighted Number Of Children) that has a value higher than 10.

```
MSEModel currentModel allClassesSelect:
  [:each | (each hasPropertyName: #WNOC)
    ifTrue: [(each getNamedPropertyAt: #WNOC) > 10]]
```

Moose Finder

The Moose Finder [Ste01] is a tool that allows one to express, compose and store queries based on different criteria like element type, properties or relationships (see Figure 5.2). Such a query can in turn be combined with other queries to express more complex ones. Furthermore, the Moose Finder supports multiple models in the context of software evolution [LDS01].

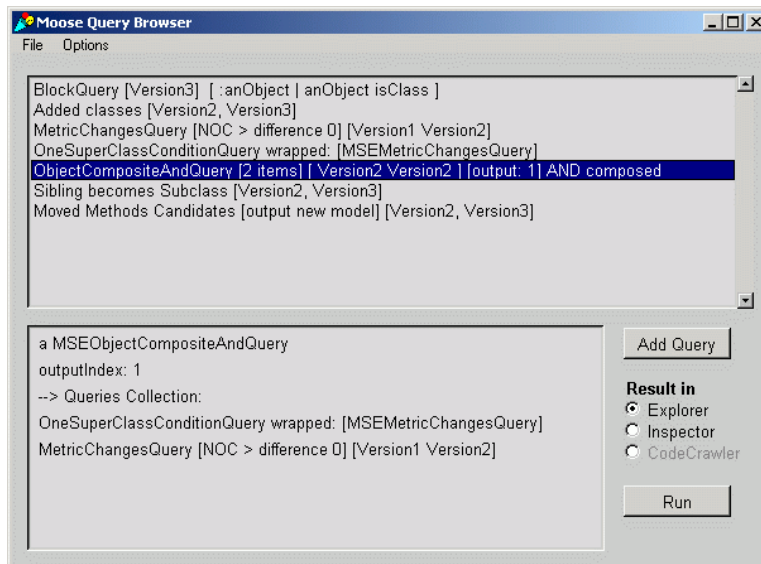


Figure 5.2 Moose Finder

Apart from the query support itself — which is independent of the metamodel — the Moose Finder defines a set of standard queries for reengineering based on FAMIX. These queries have been found useful in the context of system analysis, and validated on different industrial case studies in different implementation languages [Ste01]. First, FAMIX provides sufficient information to define these queries and secondly, it enables the queries to be applied without change on systems in different languages.

5.3.2 Querying and navigating using the Moose Explorer

Reengineering large systems brings up the problem of how to navigate large amounts of complex information. Well-known solutions are code browsers, which support browsing and editing code and navigating a system via senders and implementers of methods. However, for reengineering these approaches are not sufficient, because:

- The number of potentially interesting entities and their interrelationships is too large. A typical system can have several hundreds of classes which contain in turn several thousands of methods, etc.
- All elements need to be navigable in a *uniform way*.
 - In the context of reengineering no element is predominant. The relative importance of information depends heavily on the current focus and task of the reengineer.
 - In presence of an extensible metamodel, the navigation schema should take into account the fact that new entities and relationships can be added and should be navigable as well.

Moose Explorer proposes a uniform way to represent model information (see Figure 5.3). All entities, relationships and newly added entities can be browsed in the same way. From top to bottom, the first pane represents a current set of selected entities. Here we see all the classes of the current model. The bottom left pane represents all the possible ways to access other entities from the currently selected one, e.g., in Figure 5.3 the attributes of the class `Object`. The resulting entities are displayed in the right bottom pane. ‘Diving’ into the resulting entities puts them as the current selection in the top pane again, which allows for further navigation through the model.

FAMIX does not support navigation very well. The information is organised according to the chunk format structure, because of the incremental loading and information exchange properties of this structure (see

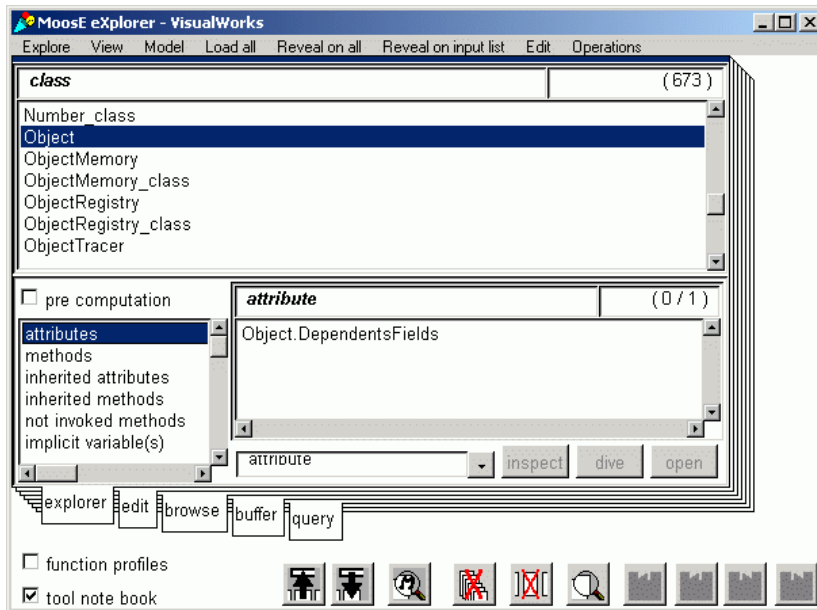


Figure 5.3 Moose Explorer

section 3.9 and section 4.6 for details). Consequently, FAMIX only defines containment relationships from a contained entity to its containing entity. Full navigation support, however, requires that all relations can be navigated in both directions. In Moose extra information is generated to enable the navigation of relationships in both directions. On top of that Moose Explorer defines navigable relationships that are not explicitly defined in FAMIX. These are mostly derived relationships such as the methods inherited by a class.

5.4 Metrics and other analysis support

Moose includes a metrics engine for the computation and storage of metric measurements. It supports so called Design Metrics, i.e., metrics which are extracted from the source entities themselves [LK94]. These metrics are used to assess the size and in some cases the quality and complexity of software. The current implementation of the metrics engine includes language-independent as well as language-specific metrics. The language-independent metrics are computed based on the core FAMIX metamodel. Examples are the number of methods or the number of attributes of class. Language-specific metrics are covered by the FAMIX language extensions. Examples for language-specific metrics are the number of method categories of a class in Smalltalk or the number of private attributes of a class in C++ or Java. FAMIX does not support metrics that need information about method bodies, e.g., the Method Complexity metric, which requires information about nested expressions [LK94].

Moose provides several other information revealers. Examples are the annotation of entities with inferred type information or the computation of the possible targets polymorphic invocations.

5.5 Grouping

Moose supports grouping of any model element. Groups can be described by intention, i.e., by describing the group in the form of a query, and by extension, i.e., by enumerating the elements the group contains. Furthermore, groups can be nested, i.e., a group is a model element itself. The following Smalltalk code shows a simple example of how all classes in a model can be grouped into groups representing the Smalltalk categories they belong to. For all classes in the current model, the category name is extracted from the class's source anchor and the class is added to a group that has the same name as the category name.

```
MSEModel currentModel allClassesDo:
[ :aMooseClass |
  | group categoryName |
  categoryName := MSEUtilities extractCategoryName: aMooseClass sourceAnchor.
  group := MSEModel currentModel enumeratedGroupWithName: categoryName.
  group add: aMooseClass ]
```

Note that this code does not manipulate Smalltalk classes directly, but Moose representations of these classes. This representation abstracts from the implementation language. However, the example illustrates that, while the core metamodel is language independent, language-dependent information can be used. The category concept from our example only exists in Smalltalk.

FAMIX does not support grouping explicitly, i.e., there is no concept in FAMIX to capture and exchange user-defined groups of model elements. The reference schema underlying FAMIX, however, supports grouping by providing unique identification for any element in a model. Grouping over multiple models is currently only supported if unique identifiers are used that are unique over multiple models. The unique name of entities is only unique within models of one single system. See section 4.5 for details.

5.6 Moose Refactoring Engine

The Moose Refactoring Engine is the part of Moose that takes care of behaviour preserving code transformations. It provides support for the fifteen low-level refactorings that are described in chapter 6. The engine uses the Moose repository to retrieve the information required to analyse what code needs to be changed and to check the preconditions. Only the final physical code transformations are performed directly on the source code. They cannot be applied on the language-independent model level, because the FAMIX meta-model, and thus Moose's repository, does not contain enough information to reproduce source code. For these physical code transformations the refactoring engine uses so-called *code transformation front-ends*. They only perform low-level code changes such as changing a single invocation in a certain method body. In its current implementation two languages are supported, namely Smalltalk and Java. The Smalltalk front-end uses the Refactoring Browser [RBJ97] to change Smalltalk code. The Java front-end currently uses a text-based approach based on regular expressions, which suffices to support all refactorings, but requires that the source code adheres to certain layout rules.

The Moose Refactoring Engine is described in more detail in chapter 7. In its function as in-depth validation of FAMIX in the area of supporting language-independent refactoring, it requires a detailed description of its inner workings and the experiments that have been performed with it, which goes beyond the purpose of this section.

5.7 Information Exchange and Tool Integration

Interoperability between reengineering tools is supported in two ways. First, there is the possibility to exchange information in text files using industry standard exchange formats. Second, tools written in VisualWorks Smalltalk can interoperate with the Moose repository, its services and each other at runtime.

5.7.1 Information Exchange with CDIF and XMI

To exchange FAMIX-based information between different tools, Moose provides two textual formats. One is CDIF [Com94], an industrial standard for transferring models created with different tools. The main reasons for adopting CDIF are, that it is an industry standard and has a standard plain text encoding which tackles the requirements of convenient querying and human readability. Next to that the CDIF framework supports the extensibility we need to define our model and plug-ins. As shown in Figure 5.1 we use CDIF to import FAMIX-based information about systems written in Java, C++ and other languages. The information is produced by external parsers such as SNiFF+ [Tak96] [TD99]. Next to parsers we also have integrations with external environments such as the Nokia Reengineering Environment [DD99].

Recently, we have adopted XMI (XML Metadata Interchange [OMG98]) as a second storage and exchange format [Sch01] [Fre00]. XMI is an OMG standard for exchanging models based on the MOF (Meta-Object Facility [OMG00]) and uses XML (Extensible Markup Language [BPSM98]) as the underlying technology to save this information. The main reason to support a second standard is that CDIF did not succeed in becoming a widely used standard. XMI seems to stand a better chance, especially because it is based on XML, which is likely to become the de facto standard for transferring information between applications and allows the use of XML-based technologies such as XSL. Secondly, XMI is based on the MOF, which is likely to become the de facto standard to describe metamodels and offers excellent integration to MOF-based metamodels such as UML.

A third format we plan to support is the Graph eXchange Language (GXL) [HWS00]. GXL is a collaborative effort from several academic and industrial research institutes to come up with an exchange format and a set of metamodels for information exchange for reengineering tools. We actively participate in the discussions and FAMIX is one of the input models that is looked at as a basis for the GXL standard program entity level metamodel.

5.7.2 Tool Integration Framework and Tools

Moose serves as a foundation for other tools. It acts as the central repository and provides services such as metric computation and refactorings to the reengineering tools built on top of Moose. To enable tools to interact with each other, Moose provides a tool registration and lookup mechanism. At this point in time the following tools have been developed:

- CodeCrawler supports reverse engineering through the combination of metrics and visualization [Lan99] [DDL99] (see Figure 5.4). Through simple visualisations which make extensive use of metrics, it enables the user to gain insights in large systems in a short time. CodeCrawler works best when a new system is approached for the first time and a quick insights are needed to get information on how to proceed. CodeCrawler has been successfully tested on several industrial case studies.

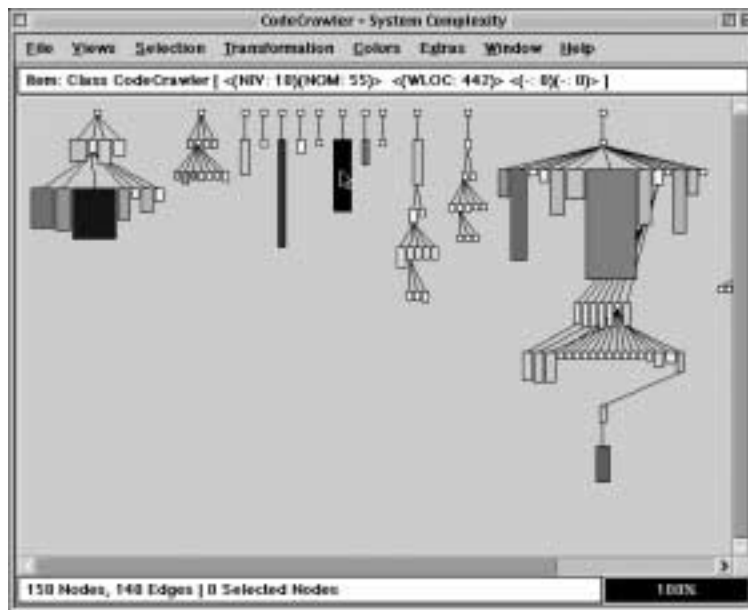


Figure 5.4 CodeCrawler

- Gaudi [RD99] combines dynamic with static information (see Figure 5.5). It supports an iterative approach by creating views, which can be incrementally refined by extending and refining logical queries on a repository. Moose is used as a source of static information about the target software

system. The dynamic information is generated independently from Moose. In any case, FAMIX does not currently cover dynamic information.

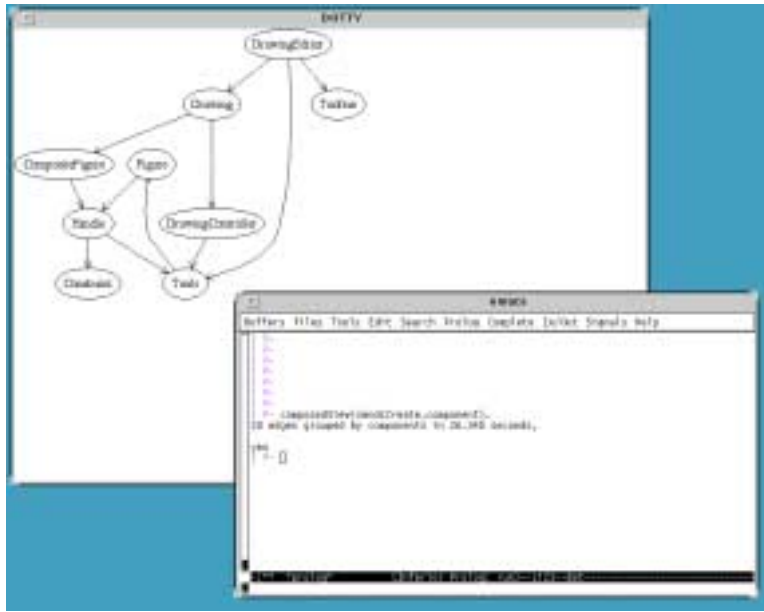


Figure 5.5 Gaudi

- Supremo [KN01] uses the Moose repository and the duplication detection tool Duploc [DRD99] to put duplication in context. Figure 5.6 shows an example: the dark nodes in the class inheritance tree represent the distribution of a recurring code sequence. The language independence of both Moose and Duploc allows the approach to function for multiple languages.

Except for providing the foundation for our own tools, Moose also interfaces with external tools. Examples are the Nokia Reengineering Environment [DD99] and (Q)Soul [Wuy01].

Runtime tool integration

Some of the above tools, namely Moose Explorer, Moose Finder and CodeCrawler, use Moose’s tool integration framework to find each other in order to use each others services. For instance, Moose Finder query results can be viewed and further navigated using the Moose Explorer. The interaction is *ad hoc* in the sense that the different tools need to know how to interact with each specific other tool. A set of standardised common interfaces for common services could improve the pluggability of the tools.

5.8 Industrial Case Studies

The ability of Moose to support reengineering of object-oriented software in a language-independent way has been validated by applying the environment and its tools on several industrial case studies¹. The approach with these case studies was that a team of reengineers was set to work on the industrial applications in a “let’s see what they can tell us about our system” way. There was no training of the system developers

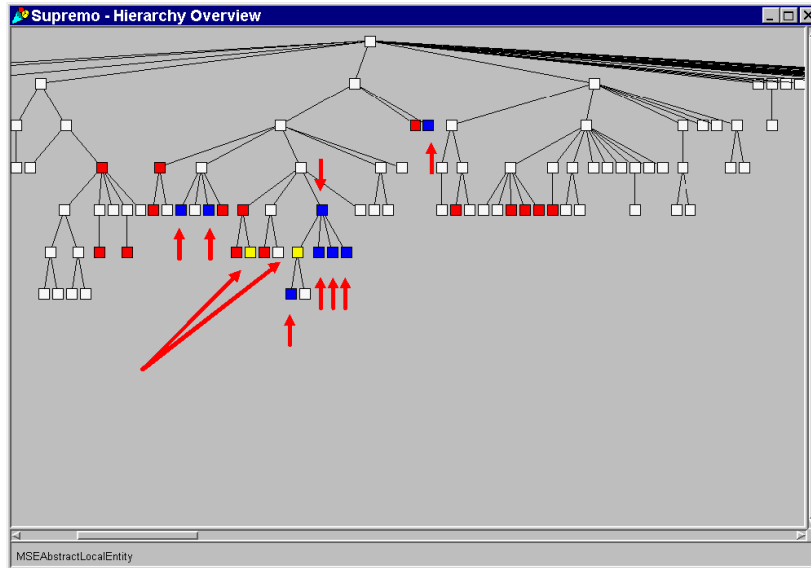


Figure 5.6 Supremo

with our tools. The common point about those experiences was that the subject systems were of considerable size and that there was a narrow time constraint to obtain results. The case studies consist of:

1. A very large legacy system written in C++. The size of the system was 1.2 million lines of code in more than 2300 classes. We had four days to obtain results.
2. A medium-sized system written in both C++ and Java. The system consisted of about 120,000 lines of code in about 400 classes. The time frame was again four days.
3. A large system written in Smalltalk. The system consisted of about 600,000 lines of code in more than 2500 classes. This time we had less than three days to obtain results.

The fact that all the industrial case studies were under extreme time pressure lead us to mainly get an understanding of the system and produce overviews [DDL99]. We were also able to point out potential design problems and on the smallest case study we even had the time to propose a possible redesign of the system. Taking the time constraints into account, we obtained very satisfying results. Most of the time, the (often initially sceptical) developers were surprised to learn some unknown aspects of their system. On the other hand, they typically knew already about many of the problems we found.

5.9 Discussion

This section discusses our experiences building Moose as well as the case studies we have carried out with it and the tools we have built on top of it. First we present some general observations we collected during the

1. Some of the case studies have been carried out together with FAMOOS project partners [DD99] and their tools. Most notably these are Duploc [DRD99], Goose [Ciu99], Audit-RE and the Nokia Reengineering Environment [DD99].

case studies. After that we discuss to which extent Moose fulfils the requirements we have set in the beginning of this chapter.

5.9.1 Observations

The case studies have given us the following insights:

Code Browsing. In addition to the views provided by our tools, code browsing was needed to get a better understanding of specific parts of the applications. The case studies show that combining metrics, graphical analysis and code browsing is a successful approach to get to a satisfying understanding of a system quickly.

Tool Adaptability. The tools needed to be adapted to deal with unexpected requirements of specific case studies. For instance, custom visualisations were created. Another example is complex invocation dependency information that is part of middleware layers such as CORBA. This is implemented in a set of classes from which the stubs and skeletons are normally generated. The interesting information are not these middleware classes, but the fact that the application calls some object via some communication means. We needed to abstract from the details of this communication.

Moose and its tools could be easily adapted, this partly due to the extensibility of FAMIX, which enables the easy capture of additional information. However, the tools were adapted in a static way, i.e., stopping the analysis process, changing the application, refill the repository and go on. A mechanism to dynamically add abstractions would allow the reengineer to record domain-specific information during the actual reverse engineering process.

Language Independence. Our tools did not need to be adapted to deal with systems implemented in different implementation languages. It shows the strength of Moose repository and the FAMIX metamodel with its language-independent core and carefully designed language mappings.

Incremental Loading and Partial Information. Although we have heavily used the information exchange capabilities of Moose, its incremental loading capabilities have not been used so far. We have only exchanged complete models. Incremental loading might get more important if tools integrate their activities more closely and need to update a common repository rather than exchange complete models, or when parts of systems are loaded only on demand for scalability reasons. The ability to deal with partial information, which is also a requirement for incremental loading, was crucial, however, because we did not have access to the source code of libraries used by the target applications.

Scalability. An important factor for a reengineering environment to function in an industrial context is that it can deal with large systems. The case studies show that Moose is able to do just that. We did not have problems regarding the number of elements we loaded into the in-memory code repository. In the industrial context we reached 300'000 elements, with the most limiting factor the small amount of RAM (128 MB) of the desktop computer Moose was running on. In another experiment on a workgroup server with 2GB of RAM, we loaded multiple models comprising of a total number of entities of around 700'000. As a comparison, the Java Swing 1.3.0 libraries modelled on the highest level of detail (i.e., classes, methods, attributes, formal parameters and their invocations and accesses) count only for about 48'000 elements.

The following considerations need to be taken into account when speaking about memory problems. First, the amount of available memory on the used computer system is, of course, an important factor. Secondly, we have never tried to heavily optimise our environment neither for access speed nor for memory consumption. Therefore, there is room for improvement, would it be needed in the future. Furthermore, we

have designed the code repository to support a possible database mapping easily, so that scalability can be improved by using a database instead of having all information in-memory. A last aspect is that tools that make use of the repository need memory of their own as well. For instance, CodeCrawler creates a lot of additional objects (representing nodes and edges) for the purpose of visualisation. But, as shown by the case studies, so far this proved to pose no problem.

5.9.2 The requirements revisited

In section 5.1, we have listed the main requirements for a reengineering environment. After presenting Moose, we now discuss how Moose evaluates in the context of those requirements.

1. **Support for reengineering tasks.** Moose supports all major reengineering tasks, which is primarily shown by the services it provides and the tools that are built on top of it. Furthermore, the industrial reengineering experiences show their successful application.
2. **Extensible.** The extensibility of Moose is inherent to the extensibility of FAMIX (see section 4.3). Its design enables extensions for language-specific features, for tool-specific information and annotations of model elements. Several of the Moose tools use these functionalities. For example, the metrics service extends the metamodel with a Measurement entity. Other analysis tools store their analysis results as annotations to the model elements.
3. **Exploratory.** Moose is an object-oriented framework and offers as such a great deal of possible interactions with the represented entities. It supports several ways to handle, manipulate and navigate entities contained in a model, as we have described in the previous sections.
4. **Scalable.** The industrial case studies presented at the beginning of this section have proven that Moose can deal with large systems in a satisfactory way: we have been able to parse and load large systems in a short time. Since we keep all entities in memory we have fast access times to the model itself. So far we have not encountered memory problems: the largest group of systems we loaded contained more than 700'000 entities and could still be completely held in memory without any notable performance penalties.
5. **Information Exchange and Tool Integration.** Integration with external tools has been repeatedly done without major problems. Information can be exchanged with other tool platforms using text-based standards such as CDIF and XMI. FAMIX clearly defines what information needs to be stored and its chunk structure supports human readability and incremental loading. Information has been exchanged with several external parsers and reengineering environments.

For runtime integration, Moose provides a small tool integration framework for tools to register themselves and find other tools. As presented in section 5.7.2, several Smalltalk-based tools use the tool integration framework to combine each others services. However, more elaborate interoperability requires standardised interfaces for provided and required services.

6. **Support for multiple object-oriented languages.** The industrial case studies show that Moose has been successfully applied to systems implemented in different object-oriented languages. The repository of Moose with its language-independent core allows the different reengineering tools to be applied without any change to the different systems.

5.10 Conclusion

In this chapter we have presented the Moose Reengineering Environment. Its facilities for storing, querying and navigating information, its extensibility and the set of services it provides, make it a solid foundation for reengineering tools. This is shown by Moose-based tools such as Supremo, Gaudi and CodeCrawler.

In this thesis Moose, the tools and the case studies have the main function to validate that FAMIX successfully supports multiple cooperating reengineering tools. In this context we can make the following observations. Firstly, Moose's built-in services and the Moose-based tools show that FAMIX successfully supports a whole range of reverse engineering tasks. They also validate the language independence of FAMIX: the Moose-based tools have been applied without adaptation on systems in different implementation languages. Furthermore, as the above discussion of the tool environment requirements shows, FAMIX supports information exchange and extensibility well and the amount of information generated for FAMIX-based models so far does not exceed the scalability needs of the tools.

The case studies show the limitations of FAMIX as well. Currently it does not support grouping and full multiple model support in FAMIX. These are both features we have already implemented and used in Moose. Other FAMIX features, such as the ability to incrementally load information, are available but not heavily used. Furthermore, the way information is structured in FAMIX does not favour information navigation. This is, however, a conscious design decision based on trade-offs between navigability on one side and support for information exchange, incremental loading and minimisation of redundant information on the other side.

Additionally to the validation of FAMIX, this chapter presents a set general set of requirements for reengineering environments. We have analysed Moose according to these requirements. Furthermore, we describe the lessons we have learned while building Moose and while performing the case studies. Summarizing the in-depth discussion of section 5.9, we can make the following observations:

- Multiple tools are needed to get useful results. Understanding a system quickly requires the ability to create different viewpoints and the application of multiple problem detection techniques. Good old code browsing is an important part of this process.
- Scalability is crucial to deal with the typical (large) size of legacy systems. Not only must a tool be able to handle the vast amounts of information, but it must be responsive as well.
- Tools need to be extensible and adaptable to deal with the specific requirements of reengineering projects.
- Language-independence has proven itself useful and worthwhile. The same tools have been used on Java, C++ and Smalltalk without adaptation.

Note that most of the tools and also the case studies cover reverse engineering activities only. System restructuring, in particular refactoring, is the focus of the next two chapters. They present an analysis of fifteen refactorings for Smalltalk and Java in the context of language independence. The analysis is based on the FAMIX metamodel. The analysis firstly shows that language-independent refactoring is feasible. At the same time it provides an in-depth validation of the language-independence of FAMIX.

CHAPTER 6

Language-Independent Refactoring

In recent years refactorings — behaviour preserving code transformations — have become a key topic in the context of reengineering object-oriented applications [SGMZ98] [TB99] or new development process models such as eXtreme Programming [Bec99]. Tools to support refactorings have been built such as the Refactoring Browser [RBJ97]. Refactoring semantics have been the topic of PhD theses for specific languages such as Smalltalk [Rob99], Java [Wer99] or C++ [Opd92]. However, an analysis of the proposed solutions is missing to understand the exact semantics of certain refactorings. Indeed depending on the tool used and the language, even with closely related languages like Smalltalk and Java, the semantics of certain refactorings are different.

In this chapter we analyse fifteen refactorings for the languages Java and Smalltalk. Based on FAMIX, the language-independent metamodel presented in chapter 4, we capture as many commonalities as possible. As such the work in this chapter is a validation of FAMIX: it shows the ability of FAMIX to support refactoring on a language-independent level. This involves complex semantical analysis and demands sufficient, complete and 100% correct information, because the result of a transformation should not result in a faulty software system. This is unlike the requirement for a typical reverse engineering task such as visualisation, which is normally not strongly affected if information is slightly incomplete or incorrect [MNGL98] [Bis92].

Additionally to the validation of FAMIX — and different from the approaches of the various authors and tool builders mentioned above — we discuss and compare the definition and required analysis of the separate refactorings rather than just presenting our solution. Where applicable, comparisons are made with the work presented in the PhD theses mentioned above as well as the Refactoring Browser¹.

The presented refactorings, listed in appendix A together with their pre- and postconditions, are what Opdyke [Opd92] calls *low-level* refactorings, i.e., primitive program transformations for adding, removing

1. We compare our work with the latest released version of the Refactoring Browser for VisualWorks at the time of writing, namely version 3.5.1 (August 2001).

and renaming entities and moving entities within their inheritance hierarchies. These low-level transformations can be combined to perform more complex transformations, called *high-level* refactorings, for instance to introduce design patterns [OCN99] [FBB⁺99]. The high-level refactorings are outside the scope of this thesis. They are typically a combination of low-level refactorings and therefore have not much to do with language issues that are handled on the lower level.

The presented analysis focuses on two issues, the first being *tool automation*. Tool automation has the strict requirement that the refactoring operation is behaviour preserving in the sense that input-output behaviour is the same before and after the refactoring. This is different from approaches such as Fowler's refactoring catalog [FBB⁺99] which focuses on manual refactoring. Fowler discusses every refactoring in an informal manner. Descriptions are imprecise and many special cases are left out. For manual refactoring this is not really a problem as there is always the developer who monitors the process and adapts it to his/her own needs. However, a tool does not have that luxury. It should perform a refactoring quickly and securely, taking away the need to test after every small change. It therefore cannot leave out any special case and/or should be conservative in complex cases.

The second focus is *multiple language support* [TDDN00]. The chapter discusses the refactorings for Smalltalk and Java, two relatively clean object-oriented programming languages. They are close enough to make a common refactoring definition useful and feasible, but they are different enough to make the comparison interesting from a research point of view.

The rest of the chapter is organised as follows. Before we present the fifteen refactorings themselves, we discuss the language subsets considered in the analysis as well as some general issues about the language mappings and information availability relevant for refactoring (section 6.1). Section 6.2 introduces the common template we use to describe the refactorings. Section 6.3 presents the actual refactorings, and section 6.4 presents how we have validated the presented analysis and in section 6.5 we discuss our findings.

6.1 Language subsets and mappings

In general the refactoring analysis is restricted to the information available in FAMIX. However, even if information is available in FAMIX, we do not always cover it in our analysis. Furthermore, the analysis must take language semantics into account that go beyond the plain information representation of FAMIX. In the following subsections we discuss the language subsets that are covered by the refactoring analysis, as well as the language mappings and level of detail issues that are relevant for the refactoring analysis.

For more information on the language extensions: section 4.4 extensively discusses them and their exact definitions can be found in the appendix C for Smalltalk and appendix D for Java.

6.1.1 Language subsets

In general we take all information that is available in FAMIX into account in our refactoring analysis. We list here the exceptions and the major omissions in FAMIX for both Java and Smalltalk.

Java. FAMIX does not cover inner classes and therefore they are not covered in the refactoring analysis either. We also do not support static and instance initialisers, inner classes and reflective use of the language. Furthermore, we do not take into account if attributes, methods and classes are final. The refactoring analysis takes casts and hiding of attributes into account.

Smalltalk. Smalltalk has many different dialects. In this thesis we concentrate on the common core. This means a single inheritance language without namespaces, with classes that are also global variables, with attributes that are only visible within the defining inheritance hierarchy and cannot hide each other. We have taken VisualWorks version 3.0 [Par98] as our reference implementation. Several times the analysis tells that a refactoring precondition is trivially preserved for Smalltalk, although it might not be for a specific dialect. Examples of language features not covered by the analysis in this chapter are the namespaces of VisualWorks 5i, or hiding variables in Enfin Smalltalk. Note that these language features are probably easily covered, because the appropriate elements are available in FAMIX (e.g., Package for namespaces) and the appropriate analysis is already existent, because it is required for Java. We do not cover pool variables and, like for Java, we do not cover reflective use of the language.

6.1.2 Language mappings

In this subsection we list some language mappings and information availability issues of FAMIX that are relevant for our refactoring analysis, namely typing information, abstractness, and the ability to semantical equivalence of methods.

Static vs dynamic typing. Section 4.4 discusses how FAMIX deals with type information of static and dynamically typed languages. In the context of refactoring three issues need attention:

- *Type related analysis.* In several refactorings there exists analysis for dealing with typed information. For Smalltalk much of that analysis is unnecessary. For instance, a query for all attributes with a certain type will return an empty set and this is known beforehand. This does not make the refactoring language-dependent. It just means that analysis is done that is unnecessary for Smalltalk: preconditions will not be violated and it will not result in any changes in the Smalltalk sources.
- Due to *the lack of static type information in Smalltalk*, invocations to a certain method name can be to any method with that name in the current system. A variable can reference an object of any type that understands the method that is invoked on that variable. In Java the static type information helps reducing that information to a single inheritance hierarchy and to the set of methods that have the same order of types of the parameters. In our metamodel an Invocation records the *candidate methods*, i.e., the methods that are possibly invoked, which abstracts from the issue whether static or dynamic information has been available to gather this information.

In the description of the refactorings the question ‘is this method invoked?’ means ‘is there an invocation that has this method in its set of candidates?’.

- *Default types.* Several creational refactorings (Add Method, Add Attribute and Add Parameter) need to provide type information for Java. The solution we have chosen is to assign default types (Object for new attributes and parameters, void for method return types). Another solution would be to ask the user for a type and ignore this information in the Smalltalk case.

Roberts [Rob99] includes an extensive discussion about how dynamic analysis and dynamic refactoring could solve the lack of static type information in dynamically typed languages. In this thesis we have limited ourselves to statically available information.

Abstractness. Abstractness of classes and methods for the different languages and the interpretation in FAMIX is discussed in section 4.4. The main difference between Java and Smalltalk is that abstractness is explicit in Java and implicit in Smalltalk. In this chapter we assume that implicitly abstract classes in Small-

talk are not instantiated, even though they can be instantiated and do not raise an error as long as their abstract methods are not invoked.

Abstractness of classes is important information for refactoring, because it gives information if the class can possibly be instantiated. If it cannot, you know that a method can not be invoked on an instance of that class and can therefore be pushed down to a subclass. If a class is not abstract, you can still know if it is not instantiated, namely if it is not referenced. The Refactoring Browser [RBJ97], for instance, tests if a class is referenced. Similarly, Compost [Uni96], a Java analysis tool, checks if a class is never instantiated, i.e., no constructor of the class is invoked. This is more precise than the Smalltalk test, because the explicit constructor concept in Java allows for a construction detection where in Smalltalk all references to a class make it possibly non-abstract.

Semantical equivalence. For refactorings that deal with overriding methods or multiple similar methods, it is interesting to be able to determine if two pieces of code are semantically equivalent. For instance, if a method overrides a semantically equivalent method, it can be removed without changing the behaviour of the system. Semantical equivalence, however, is hard to determine. Code needs to be transformed into some common representation and considerable dataflow analysis is required. FAMIX does not contain the detailed information needed for this analysis and consequently we cannot determine semantical equivalence. In contrast, the Refactoring Browser [Rob99] has access to full parse tree information. It detects a few cases of semantical equivalence by checking if two parse trees are equal with possibly different parameter and local variable names.

6.2 The Refactoring Template

Before discussing the Pull Up Method and Push Down Method refactoring in the following two sections, we present the template we use to describe and discuss the two refactorings:

- **Name:** The name identifies the intent and target entities involved in the refactoring.
- **A short description** including a figure describes our definition of the refactoring and the key problems involved.
- **Preconditions:** This part lists the preconditions that need to be checked to be able to safely apply the refactoring. The list starts with the preconditions that are independent of the language, followed by the language-specific preconditions for Java and Smalltalk.
- **Precondition analysis:** This section describes for every precondition why it is necessary and, if applicable, why we did not select an alternative solution.
- **Related work:** Different approaches of other authors are analysed.
- **Discussion:** Finally we discuss the language-independence and the different alternatives.

6.3 The refactorings in detail

This section lists fifteen low-level refactorings using the template presented in the previous section. The analysis of every refactoring is based on the assumption that a complete and correct model of the system that is the target of the refactoring, is available. A further assumption is that the this system compiles and runs correctly. Similar to Roberts [Rob99], we define correctness as that a program passes a test suite that covers its full specification.

ADD CLASS (*CLASSNAME*, *PACKAGE*, *SUPERCLASSES*, *SUBCLASSES*)

Inserts a new class with name *classname* in package *package* where *superclasses* are the superclasses of the new class and *subclasses* are subclasses of all *superclasses* that have to become subclasses of the new class.

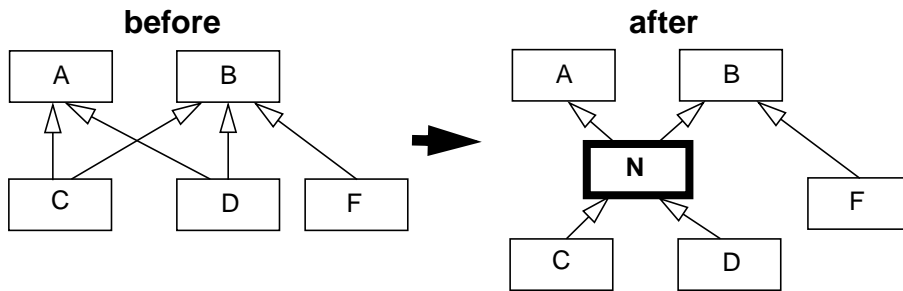


Figure 6.1 Add Class refactoring with *classname* N, *superclasses* A and B and *subclasses* C and D

Typically this is a simple refactoring, because the new class is not referenced yet, so no relationships need to be updated and only name clashes need to be checked. However, abstractness of classes and multiple inheritance need to be taken into account.

Preconditions

Language-independent preconditions

1. no class may exist with *new name* in the same scope.
2. no global variable may exist with *new name* in the same scope.
3. all *subclasses* must be subclasses of all *superclasses* or no subclasses are specified.

Language-dependent preconditions

4. *classname* must be a valid name.

Smalltalk-specific preconditions

5. *superclasses* (and therefore *subclasses*) must not be metaclasses.

Precondition analysis

1. no class may exist with *new name* in the same scope.

Classes within the same scope cannot have the same name.

2. *no global variable may exist with new name in the same scope.*

In Smalltalk it is not allowed for classes and global variables to have the same name. Actually, classes are global variables in Smalltalk. For Java the absence of global variables makes this precondition trivially fulfilled.

3. *all subclasses must be subclasses of all superclasses or no subclasses are specified.*

This precondition is necessary to support multiple inheritance. If fulfilled, inserting a class in the middle (like N in Figure 6.1) will have no impact on the outside behaviour, because the new class does not add, overwrite or hide any behaviour and the existing classes (C and D in Figure 6.1) still inherit from the same set of classes. Smalltalk's single inheritance is trivially supported by this scheme and also the Java interface concept.

4. *classname must be a valid name.*

The name of the new class should adhere to the naming rules of the implementation language.

5. *superclasses (and therefore subclasses) must not be metaclasses.*

Smalltalk has explicit metaclasses, which are modelled in FAMIX as classes. Every class has an accompanying metaclass and it is not allowed to create a metaclass independent of a class and thus to add a class in a metaclass hierarchy.

Related work

For this refactoring the main difference with the language specific approaches by Werner [Wer99], Roberts [Rob99] and Opdyke [Opd92] is that they only support single inheritance. For Smalltalk this just follows the language, for Java and C++ this is done for reasons of simplicity. Our approach goes a step further in supporting multiple inheritance and so Java interfaces.

Discussion

The preconditions are mainly language-independent. The first language-specific precondition (precondition 4 about the name of the new class) could be abstracted from if a useful common set of naming restrictions can be defined over the languages. The second language-specific precondition (precondition 5 about the Smalltalk metaclasses) is an example of where the mapping of metaclasses to classes results in an extra precondition rather than that it provides transparency of concepts.

There are some important issues for this refactoring that are not covered by the preconditions.

- When the new class inherits abstract methods without implementing them, it must be declared abstract. This does not need to be dealt with on the Smalltalk source code level, because abstractness of classes in Smalltalk is implicit (see section 4.4.2).
- In Java, depending on if the superclasses of the new class are interfaces or classes the new class needs to be a Java interface or a Java class.
- A new class in Java typically needs a new file to be created as well. This is transparently taken care of by the Java front-end.

Note that this refactoring is not defined to work for Java inner classes as well and that we currently do not cope with constructor chaining in Java.

REMOVE CLASS (CLASS)

Removes *class* from a system.

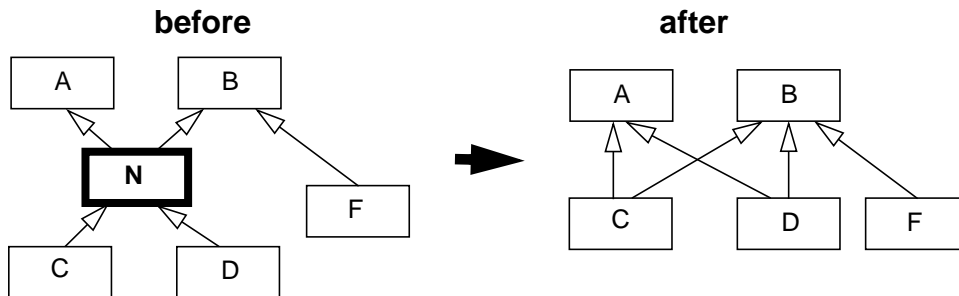


Figure 6.2 Remove class N

This refactoring removes an unreferenced class together with its unreferenced methods and attributes. After the refactoring all subclasses of *class* inherit from all superclasses of *class*. This ensures that all subclasses still inherit the same features as before the refactoring.

Preconditions

Language-independent preconditions

1. *class* must not have attributes or its attributes are only referenced from within *class*.
2. *class* must not have methods or its methods are only referenced from within *class*.
3. *class* must not be referenced.
4. *class* must not implement abstract methods from its superclass hierarchy or must not have non-abstract subclasses.

Smalltalk-specific preconditions

5. *class* must not be a metaclass.
6. the metaclass of *class* must not have referenced methods or classes.

Precondition analysis

1. *class must not have attributes or its attributes are only referenced from within class.*

If *class* has referenced (instance or class) attributes, removing the class breaks the system. References to attributes of *class* are allowed, if it is only *class* itself that references them, because removing the class removes the references with it.

2. *class must not have methods or its methods are only referenced from within class.*

If *class* has referenced (instance or class) methods, removing the class breaks the system. This includes references to implicit methods such as Java default constructors. Not referenced means more precisely that methods are not *possibly referenced*, because due to polymorphism it might not be statically determinable which method will be called at runtime. In terms of FAMIX this means that a method does not have *candi-*

date invocations (see section 6.1.2). Like in precondition 1, references to methods of *class* are allowed, if it is only *class* itself that references them, because removing the class removes the references with it.

3. *class must not be referenced.*

Obviously when a referenced class is removed it will break the system. A class can be referenced in the following ways:

- as type of a variable or parameter
- as returntype of a method
- as part of an array type or returntype. In FAMIX the `declaredClass` attribute of typed elements contains the class reference extracted from the type declaration. See section 4.4.1 for details.
- as part of array instantiations
- as a cast
- when class methods are invoked or class attributes are accessed on *class*. This is the only reference in this list that exists in Smalltalk, as Smalltalk is dynamically typed and does not have casts and primitive arrays.

4. *class must not implement abstract methods from its superclass hierarchy or must not have subclasses.*

If *class* implements an inherited abstract method and non-abstract subclasses are inheriting this method without overriding it, in Java compilation is broken after the refactoring. For Smalltalk the target system will work after the refactoring because the method is not referenced (according to precondition 2) but the resulting unimplemented abstract method is bad style and a possible cause for errors in the future. More complex analysis could be performed to check if all non-abstract subclasses implement or inherit an other implementation than a method in *class*, but we have not done this for reasons of simplicity.

5. *class must not be a metaclass.*

In Smalltalk every class has an accompanying metaclass and it is not allowed to create or remove a metaclass independent of a class and thus to remove a class in a metaclass hierarchy.

6. *the metaclass of class must not have referenced methods or attributes.*

Like instance and class methods and attributes of *class* any instance methods and attributes of the metaclass of *class* cannot be referenced. Removing *class* (and thus its metaclass) would break the system.

Related work

Roberts [Rob99] (Smalltalk) has as preconditions that the class is not referenced and that it is either empty or does not have subclasses. Inherited abstract methods are not further analysed, but in Smalltalk this will not break the system. Opdyke [Opd92] (C++) calls this refactoring `delete_unreferenced_class` and only allows the removal of non-referenced classes without subclasses. Werner [Wer99] (Java) analyses, similar to our approach, the references of methods and attributes from the class to be removed. His graph-based model actually allows to easily check any kind of reference to a class. Like Roberts and Opdyke he does not deal with abstract inherited methods, which can render the target system incompatible. Different from the other approaches he analyses if the class is only used as a pass through for superclass features and changes casts accordingly to be able to remove the class anyway.

All three of the above approaches only support single inheritance. For Smalltalk this just follows the language, for Java and C++ this is done for reasons of simplicity.

Discussion

The analysis for this refactoring is more complex for Java than for Smalltalk, because due to static typing together with the concepts of explicit casting and primitive arrays, many more possibilities exist to reference a class. On the other hand, like in the other class refactorings the mapping of Smalltalk metaclasses to FAMIX classes results in extra checks to be made rather than useful transparency.

Different from the other approaches presented in the related work, our approach supports both multiple inheritance and the analysis of references of methods and attributes of the class to be removed. This increases the complexity of the analysis, but provides for a wider applicability of the refactoring as well.

Currently FAMIX can detect all class references listed in the analysis of precondition 3, except for instantiations of primitive arrays. Although arrays are not explicitly modelled in FAMIX, a variable with a primitive array as type or a method with a primitive array as return type can be detected via the declared(Return)Type attributes (see section 4.4.1). Array instantiations, however, also contain a reference to the class name and are not currently modelled in FAMIX. FAMIX does not currently support nested classes either. This refactoring probably deals with nested classes without problem, but tests need to be performed to confirm that.

In Java every (non-nested) class is stored in a separate file with the same name as the class and the extension ‘.java’. In such a case removing a class means therefore the removal of the corresponding file as well. This is transparently taken care of by the Java code transformation front-end.

RENAME CLASS (*CLASS*, *NEW NAME*)

Renames the class *class* and all references to this class to *new name*.

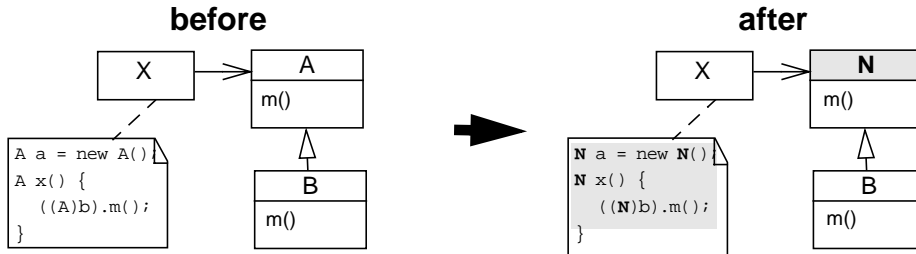


Figure 6.3 Rename Class refactoring renaming A to N including updating the references

Renaming a class does not have complex preconditions. Basically if the new name is valid and not used already within the same scope the refactoring can be applied. For this refactoring the differences between the supported languages are more in what needs to be updated. The following class references may occur:

- types of variables (Java)
- return types of behavioural entities (Java)
- inheritance definitions (Java and Smalltalk)
- class method invocations (Java and Smalltalk)
- class variable accesses (Java)
- constructor names (Java)
- explicit casts (Java)
- primitive array types (Java)
- primitive array instantiations (Java)
- import statements (Java)

Clearly Java has more kinds of references to be updated. This is partly because the static typing and partly because of simplicity of Smalltalk which does not have specific language constructs for primitive arrays and constructors.

Preconditions

Language-independent preconditions

1. no class may exist with *new name* in the same scope.
2. no global variable may exist with *new name* in the same scope.
3. classes that refer to *class* must not already contain or inherit a variable with *new name*.

Language-dependent preconditions

4. *new name* must be a valid class name.

Smalltalk-specific preconditions

5. *class* must not be a metaclass.

Precondition analysis

1. *no class may exist with new name in the same scope.*

Classes within the same scope cannot have the same name.

2. *no global variable may exist with new name in the same scope.*

In Smalltalk it is not allowed for classes and global variables to have the same name. Actually, classes are global variables in Smalltalk. For Java the absence of global variables makes this precondition trivially fulfilled.

3. *classes that refer to class must not already contain or inherit a variable with new name.*

The contained or inherited variable would hide the renamed class, which is a problem if the class is referenced in the scope of that variable. In Java it is allowed for variables and types to have the same name within the same scope. For instance,

```
String String;  
public String String() {  
    return String;  
}
```

is valid Java code. However, applying the renaming will result in less understandable code. Therefore enforce this precondition for both languages (with the additional advantage of keeping it language independent).

4. *new name must be a valid class name.*

The new name should adhere to the naming rules of the implementation language.

5. *class must not be a metaclass.*

In Smalltalk it is not allowed to rename a metaclass independently of its accompanying class.

Related work

Both Opdyke [Opd92] (C++) and Werner [Wer99] (Java) only check if there are no existing classes with the new name already. They also do not take namespaces, respectively packages and thus not scope into account. Roberts [Rob99] describes for Smalltalk the preconditions that an existing class or global variable does not have the name already (see precondition 1 and 2). He does not mention precondition 3 although the Refactoring Browser [RBJ97] implements this check.

Discussion

Most preconditions are language independent. The FAMIX unique naming scheme allows for easy checking of similar names in the same scope transparently of the underlying implementation language. About the two language specific preconditions. Precondition 4 will always depend on the naming rules of the specific implementation language. A common subset of those rules could be defined to abstract this precondition from the languages as well, but currently this is not the case in FAMIX. And like the other class refactorings metaclass definitions can not be changed independent of their accompanying classes (precondition 5).

ADD METHOD (*NAME*, *CLASS*)

Adds the method with *name* in *class*. The new method has an empty body.

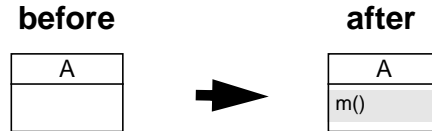


Figure 6.4 Add Method refactoring adding a method named *m* in class *A*

A simple refactoring. No references need to be updated as the new method did not exist before. The only check needed to be made is if a method with *name* does not already exist in *class* or its superclass hierarchy. The refactoring adds a method in *class* with no parameters, an empty body, public visibility and a default return type `void` for Java and no returntype for Smalltalk.

Preconditions

Language-independent preconditions

1. no (inherited) method with signature derived from *name* may exist in *class*.

Language-dependent preconditions

2. *name* must be a valid method name.

Precondition analysis

1. no (inherited) method with signature derived from *name* may exist in *class*.

Otherwise this method would be overridden (and the original thus hidden) or replaced.

2. *name* must be a valid method name.

Obviously should the new name adhere to the naming rules of the implementation language.

Related work

The Add Method refactoring for Java in [Wer99] (which is called Add Operation there) has, additionally to name and class, a set of parameter types and a return type as parameters. Opdyke [Opd92] and Roberts [Rob99] allow overriding if the overridden method is not referenced in class or its subclasses or when the new method is *semantically equivalent* with the overridden method. The new method can have a body already. The semantical equivalence is hard to check in practice. See section 6.1.2 for in-depth discussion.

Discussion

Apart from the ever existing valid name precondition (precondition 2), this refactoring only contains one language-independent precondition (precondition 1).

Different from Werner [Wer99] we do not have parameter definitions as part of the new method definitions. This for reasons of simplicity. This might be a problem when a user wants to add a new method with

some parameter and use Add Method and Add Parameter to achieve this. The method without parameter that is added using Add Method might be rejected because of an existing method with the same signature, but the combined refactoring would be perfectly valid.

It could be argued to loosen precondition 1 to allow for new methods that overload existing methods. The name and return type should be the same, the set of types of the parameters and/or their order should be different. A similar name can be unintentional though which would result in less understandable code, because a same name communicates a strong relationship. A compromise solution would be to prompt the user for a choice. In any case, Smalltalk does not allow overloading and therefore we have chosen not to allow this for reasons of language-independence.

Similarly precondition 1 does not allow for methods to override existing methods. To allow this, extensive analysis would be needed to be done to see if the now overridden method was never called in or through class or its subclasses. This to ensure behaviour preservation. Again, also because overriding might be unintentional, we do not allow overriding for reasons of simplicity.

Note that it is not checked if a method with *name* already exists in subclasses of *class*. This is not necessary to ensure behaviour preservation. However, if the name equivalence is unintentional, in a later stage unintended behaviour might be observed in subclasses due to overriding of the added method.

REMOVE METHOD (*METHOD*)

Removes method from its containing class.

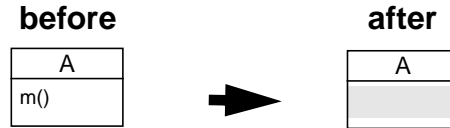


Figure 6.5 Remove Method refactoring removing a method named *m* from class *A*

A method can be removed if it is not possibly referenced.

Preconditions

Language-independent preconditions

1. *method* must not have candidate invocations unless method itself is the only candidate invoker.
2. if *method* is abstract it must not have static references.

Precondition analysis

1. *method must not have candidate invocations unless method itself is the only candidate invoker.*

A method cannot be removed if it is referenced, i.e., if there is a non-empty set of statically determinable candidate invocations of *method*. Due to polymorphism it is not always possible to determine which method will be the actual method invoked at runtime, hence the name ‘candidate’ invocations.

2. if *method* is abstract it must not have static references.

An abstract method cannot have candidate invocations. But in statically typed languages it can be explicitly referenced anyway, in which case the method cannot be removed. In Smalltalk this precondition always returns true.

Related work

Roberts [Rob99] does not allow any invocations in the system to methods with the same name as *method*. This reflects the situation in Smalltalk where polymorphism is not tight to a single inheritance hierarchy. Beyond our approach Roberts allows *method* to be removed anyway if a *semantically equivalent* superclass method exists that will be called instead after the refactoring. As already denoted in Add Method, semantic equivalence is very hard to determine in practice, which is discussed in more detail in section 6.1.2.

Werner [Wer99] does not allow any existing candidate invocations.

Opdyke [Opd92] describes a multiple method version of the Remove Method refactoring (`delete_member_functions`). This gives the possibility to delete a referenced methods if the reference is from another method in the set to be removed. He also allows removal if there is a ‘redundant’ method inherited from the superclass. Redundant meaning that the method have the same signature and body, which is the most basic case of semantical equivalence.

Discussion

This refactoring only contains two preconditions, which are both language-independent. This is mostly due to the way candidate invocations are treated in FAMIX. They are stored transparently from the fact how they are computed. The precondition therefore is not depending on the fact that in Smalltalk the set of candidate invocations is typically much larger than in Java, because the dynamic typing and polymorphism that goes beyond a single inheritance hierarchy. The possibly invoked methods by an invocation in Smalltalk are normally all methods in the system with the invoked name, where in Java the static type information restricts the possibly invoked methods to one inheritance hierarchy.

The second precondition deals with static references to abstract methods. In Smalltalk it is not possible, so the precondition is always fulfilled for that language. Note that abstractness is implicit in Smalltalk and it is indeed possible to invoke an abstract method, resulting in a runtime error. However, we presume well running systems and thus no abstract methods invoked at runtime.

RENAME METHOD (*METHOD*, *NEW NAME*)

Renames *method* and all method definitions with the same signature in the same hierarchy. All invocations to all changed methods are changed to refer to the new name.

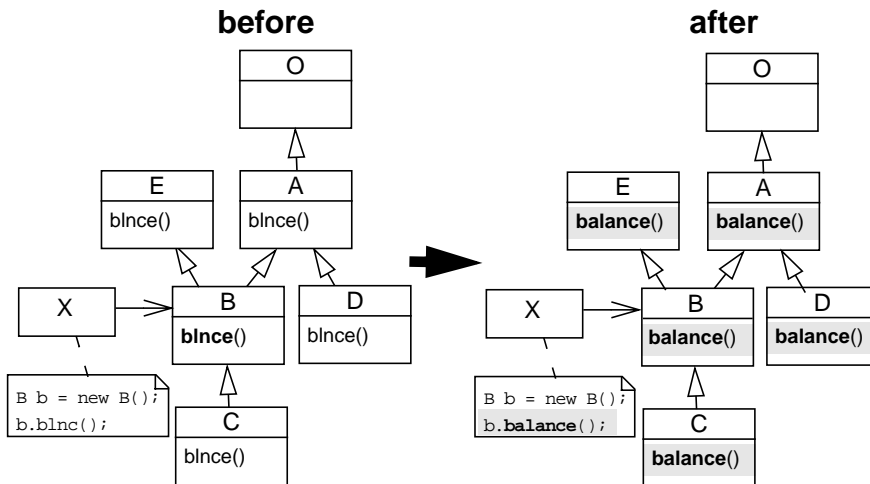


Figure 6.6 Rename Method refactoring renaming `blnc` in class `B` to `balance`

A method can only be renamed in a behaviour-preserving way if all overriding methods and overridden methods (and all their overriding and overridden methods) are renamed as well. Furthermore, all invocations to *all* changed methods need to be renamed accordingly.

Preconditions

Language-independent preconditions

1. all superclasses of the class containing *method* as well as the subclass hierarchies of the highest superclasses that define a method with the same signature as *method*, must not already contain a method with a signature implied by *new name* and the parameters of *method*.
2. the candidate invocations to the group of methods that need to be renamed, do not have candidates that are methods outside of this group.

Language-dependent preconditions

3. *new name* must be a valid method name.

Java-specific preconditions

4. when *method* is a constructor, the refactoring cannot be applied unless in the context of a Rename Class refactoring.

Precondition analysis

1. *all superclasses of the class containing method as well as the subclass hierarchies of the highest superclasses that define a method with the same signature as method, must not already contain a method with a signature implied by new name and the parameters of method.*

Firstly, the signature of the renamed methods cannot be the same as an existing method in all superclasses of the class containing *method* (O, A and E in Figure 6.6), because renamed methods would override existing methods. Secondly, a similar signature is also not possible in the full subclass hierarchies of the classes highest in the superclass hierarchies that define a method with the same signature as *method* (i.e., A and E in Figure 6.6 and all their subclasses). This covers overriding of existing methods, possible replacement or double definitions of methods in classes that define *method* or a polymorphic equivalent method, or overriding of renamed methods by an existing subclass method. Double definitions are not allowed, and method overriding that did not exist before potentially changes behaviour, because the scope of the newly overridden methods would change.

2. *the candidate invocations to the group of methods that need to be renamed, do not have candidates that are methods outside of this group.*

In Smalltalk it is possible that invocations invoke methods with the same signature in different inheritance hierarchies. These invocations cannot be changed to reference the *new name*, because they possibly invoke a method in another inheritance hierarchy that has the old name. In Java this precondition is trivially preserved, because its static type information and limited polymorphism restrict the candidate invoked methods to the targeted group of methods within the same inheritance hierarchy.

3. *new name must be a valid method name.*

The new name must adhere to the naming rules of the implementation language.

4. *when method is a constructor, the refactoring can not be applied unless in the context of a Rename Class refactoring.*

Java constructors have the same name as their containing class. Changing that name independently of changing the class name is not allowed.

Related work

The different approaches we had a look at have all a different definition of this refactoring. Opdyke [Opd92] (C++) renames *method* and equivalent methods in the subclasses. Werner [Wer99] (Java) only renames *method*. Fowler [FBB⁺99] (also Java) changes all methods with the same signature within an inheritance hierarchy, which is like our approach with the difference that he does not discuss Java interfaces. Roberts [Rob99] renames all methods with the same signature in a set of classes which must contain all classes that are polymorphically equivalent with respect to these methods. For Smalltalk, Roberts' target language, this set can contain classes from different inheritance hierarchies. Like our definition of precondition 2, Roberts' precondition is generic enough to cover both Smalltalk and Java. He does not specify however what the rules are to determine what polymorphically equivalent classes are.

Unlike our precondition 1, Opdyke [Opd92] and Werner [Wer99] allow for methods to be renamed to an already existing name from an inherited method when either this other method is not referenced from the

class containing *method* and its subclasses, or, only by Opdyke, if the methods are *semantically equivalent*. All of the above approaches only cover single inheritance.

Discussion

It is interesting to see that the approaches presented in the Related Work as well as ours have all a different interpretation of this refactoring.

We have chosen ours, because it changes all polymorphically equivalent methods detectable from *method* that are explicitly intended to be polymorphically equivalent. These are the methods with the same signature within the same inheritance hierarchy. In Smalltalk there can be more polymorphically equivalent methods outside of the same inheritance hierarchy, but this equivalence can be, and often is, unintentional. That is why we filter these cases out in precondition 2. The fact that the candidate methods of a certain invocation already take the difference between Java and Smalltalk in polymorphism into account, keeps it a language-independent refactoring. It is similar to Roberts' definition, which is also language-independent, only he changes all polymorphically equivalent methods instead of changing only when these methods are within the same inheritance hierarchy.

Another possibility is a stronger involvement of the user to restrict changes to method names and invocations to a certain hierarchy or package or other kind of part of a system. The Refactoring Browser [RBJ97] offers limited user interaction by asking to proceed or not in cases where the polymorphically equivalent methods are not within one inheritance hierarchy only. However, the focus in our research has been on automation, so this path has not been explored any further.

PULL UP METHOD (*METHOD*, *SUPERCLASS*)

Pulls up *method* to one of its superclasses (*superclass*).

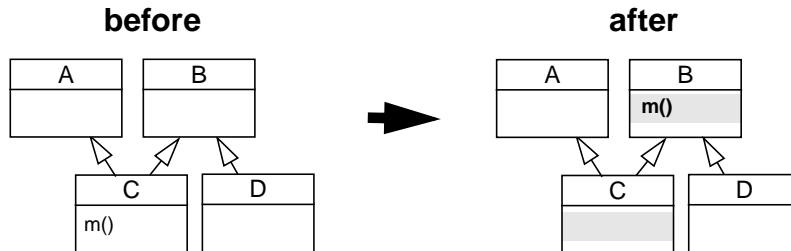


Figure 6.7 Pull Up Method refactoring pulling *method* C.m() up to *superclass* B

Pulling up a method is one of the more complex refactorings, because the set of possible violations of behaviour preservation is large. The main difficulty is the possible occurrences of methods with the same signature in the same inheritance hierarchy. The effect of the pull-up on the visibility of these methods and the possible (kinds of) invocations of them (self/this calls, super calls, scoped invocations from within and outside of the hierarchy) results in a great number of checks to be made.

Preconditions

Language-independent preconditions

1. *method* must not be private.
2. *method* should not directly access attributes from its defining class.
3. *method* should not directly invoke methods from its defining class unless all those invocations have self/this as receiver and are either to methods that are also defined or inherited in the superclass or to itself.
4. *superclass* may not contain or inherit a non-abstract method with the same signature as *method*.
5. *method* cannot have super references to *superclass*.

Java-specific preconditions

6. *method* must not be a constructor.
7. non-abstract method cannot be pulled up to an interface.

Smalltalk-specific preconditions

8. *method* should not access methods from its metaclass.

Precondition analysis

1. *method* must not be private.

If a private method is pulled up, it is not visible anymore in its original class. Possible invocations from within this class will break.

2. *method should not directly access attributes from its defining class.*

If *method* refers to attributes of its defining class, once it is pulled up, it will either reference a variable that is undefined in the superclass or if a variable with the same name exists in the superclass reference this other variable which is possibly breaks behaviour preservation.

3. *method should not directly invoke methods from its defining class unless all those invocations have self/this as receiver and are either to methods that are also defined or inherited in the superclass or to itself.*

A method cannot be pulled up if it refers to methods in its original class, because these methods cannot be referred to once the method is pulled up into *superclass* (see case (a) in Figure 6.8).

Only if *superclass* defines or inherits a method with the same signature, *method* will not refer to a method that is undefined in its new location, and at runtime it will be dynamically bound to the same method as before. In this case *method* can be pulled up without problems. In the example of Figure 6.8 this means that case (a) is allowed if class B contains a method x as well. A special case is if *method* invokes itself. In that case it can also be pulled up, because once pulled up it is still bound to itself (case (b) in Figure 6.8). Note that these exceptions must be valid for all invocations from *method* to local methods.

Any references from *method* to methods of the defining class could be allowed if the invocation is performed on a different instance. This is not determinable with the level of information available in FAMIX and thus do we not analyse this case (case (c) in Figure 6.8). We only analyse the cases where the receiver of the invocation is *self* in Smalltalk or *this* in Java¹. Even if we would have access to full parse tree information, considerable dataflow analysis would be needed.

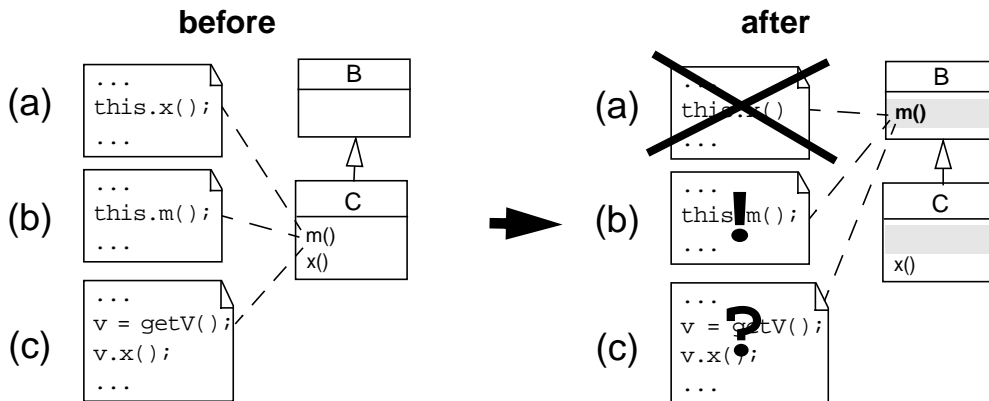


Figure 6.8 C.m() invokes methods of its defining class. In case (a) m() cannot be pulled up, because it would reference a subclass method afterwards. In case (b) m() references itself on the same instance (=this) and can be pulled up without problems. In case (c) m() invokes x() on a variable v which possibly references the same instance.

1. We consider both `this.m()` and `m()` in Java as an invocation of `m()` on `this`, as the latter is only a shorthand for the former.

4. superclass may not contain or inherit a non-abstract method with the same signature as method.

For this precondition we need to look at two cases. The first possibility is that *superclass* itself contains a method with the same signature (e.g., in Figure 6.7, if B would already contain a method `m()`). Pulling up `C.m()` would replace the existing method. This is allowed if `B.m()` is abstract, because then it is surely never invoked at runtime. If `B.m()` is not abstract, it could be replaced if it is never invoked, but this can only be determined for certain cases, or if `B.m()` is *semantically equivalent* to `C.m()`. This is, however, generally hard to determine in practice and cannot be determined with our metamodel, because it does not contain AST-level information (see also section 6.1.2). Therefore we do not allow non-abstract methods to be replaced.

The second possibility is that a method with the same signature as *method* is inherited by *superclass*. Pulling up *method* would hide inherited implementations to other subclasses of *superclass*. If the inherited method is abstract this is not a problem, because it is known not to be the method invoked at runtime. If it is concrete, behaviour changes if the newly hidden method was invoked from the other subclasses. In the example in Figure 6.9, `D.x()` calls `X.m()` before the refactoring and `B.m()` after the refactoring.

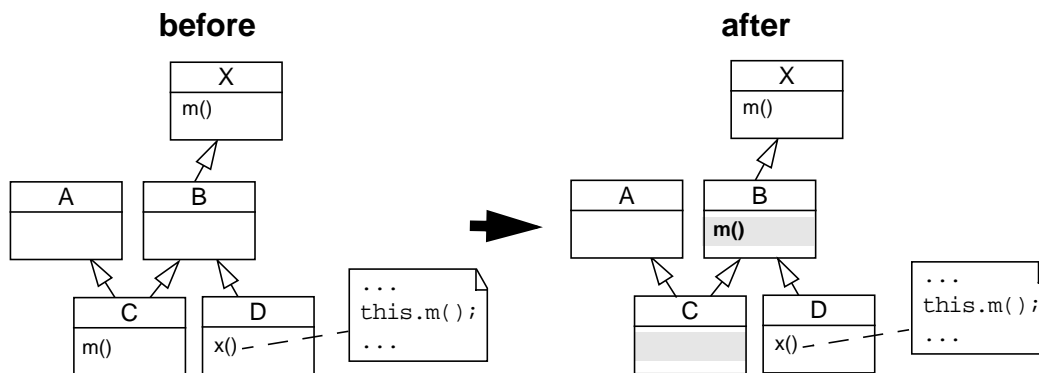


Figure 6.9 `B.m()` hides `X.m()` from some subclasses that use it (`D`)

The hidden method (`X.m()`) can be *semantically equivalent* to *method* (`C.m()`), but we cannot determine that with the available information. Another possibility is that the hidden method is not invoked in or through *superclass* or its subclasses (i.e., `X.m()` is not invoked on instances of `B` and `D` in the example). However, this is impossible to statically determine for Smalltalk unless methods with signature `m()` only exist in this hierarchy and are only invoked from within the hierarchy using `self` or `super`. We do not check for this special case for reasons of simplicity.

5. method cannot have super references to superclass.

Super references in *method* would, after pulling the method up, point to a different (set of) class(es). This possibly changes behaviour or the possibility to compile the target system. Figure 6.10 shows an example where `m()` refers to `B.x()` with a super reference. If `m()` would be pulled up to `B`, this reference is broken. It either references a method `x()` in a superclass of `B`, which possibly results in a change of behaviour, or it references no method at all, resulting in a compilation or runtime error.

For Java a super reference always points to the superclass and never to an *interface* it implements (see the mapping from Java interfaces to FAMIX classes in section 4.4.2). In Smalltalk `super` always points to the only one superclass. In Smalltalk only methods, not attributes, can be accessed with `super`.

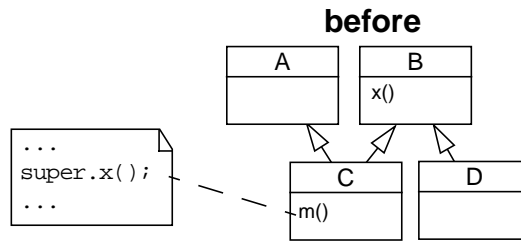


Figure 6.10 C.m() has a super reference to a method of B

There are several possibilities to relax this precondition, but we did not implement them for reasons of simplicity:

- if the super invocation invokes a method or accesses an attribute even higher up in the hierarchy of *superclass* (i.e., in a superclass of B in Figure 6.10). If that is the case for all super references the method could be pulled up anyway. In our metamodel this information is available for methods, not for attributes.
- the super references could possibly be replaced with self references and so point to the same class after the refactoring, preserving behaviour. However, because super is statically bound and self dynamically to the current instance, this can only work if subclasses do not define a method with the same signature.

6. *method must not be a constructor.*

A Java constructor is bound to its class, even by its name which is always the same as its defining class. Therefore it can not be moved out of its class in any way.

7. *non-abstract method cannot be pulled up to an interface.*

A Java interface can only contain abstract method declarations. Therefore, any method that is pulled up to an interface must be abstract already.

8. *method should not access methods from its metaclass.*

Similar to precondition 2, but for the metaclass methods of Smalltalk classes.¹

Related work

Opdyke [Opd92] and Roberts [Rob99] do not describe this refactoring.

Smalltalk

The Refactoring Browser [RBJ97] implements this refactoring. It has an interesting approach. If pulling the method up would result in replacing or hiding a method of the superclass hierarchy, i.e., a violation of our precondition 4, it gives the user the possibility to copy that method down to the other subclasses of *superclass*. The possible situations are depicted in Figure 6.11. The refactoring is applicable in more cases than

1. Currently this is not determinable in FAMIX as class method invocations have the form 'self class myMetaclassMethod' => the method 'class' is invoked on self, the method 'myMetaclassMethod' is recorded to be invoked on 'some' expression.

our approach and is especially useful when the special case method is in the superclass instead of the general one. However, if a method is copied down from superclasses of superclass, duplicated code scattered over the inheritance hierarchy is the result (case (b) in Figure 6.11).

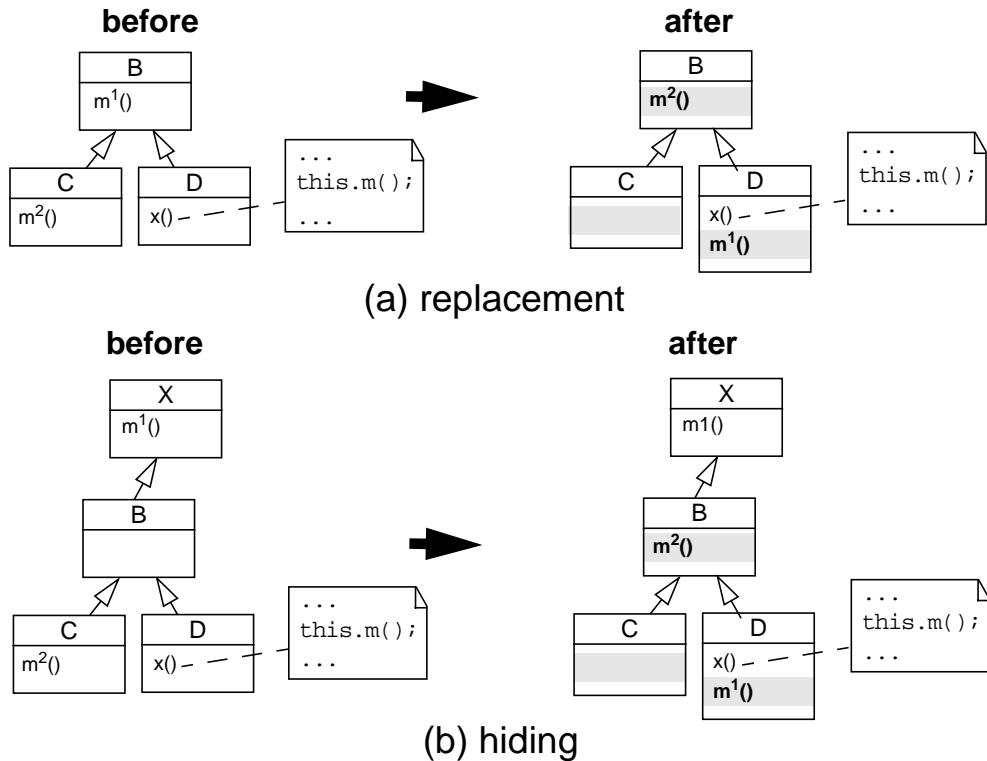


Figure 6.11 C.m() is pushed up to B. The Refactoring Browser copies down any method m that would be replaced (B.m¹() in (a)) or hidden (X.m¹() in (b)) from other subclasses of B (D in this figure).

Another useful feature of the Refactoring Browser is that it removes all duplicate methods from the subclasses of *superclass*. If such duplicated methods exist, typically a user starts to push up *method* to get rid of duplicated functionality in the first place, and without the tool support he would have to do manually. In FAMIX we do not have enough information available to determine if the subclass methods with the same signature are duplicated or not, and as such cannot provide similar functionality.

Furthermore, the Refactoring Browser performs the following checks:

- It allows replacing methods in the superclass or overriding of methods higher up in the hierarchy if the methods have the same parsetree (with possibly different local variable and parameter names). In that case they are verifiably semantically equivalent (see also section 6.1.2).
- It analyses super sends of *method* by checking if *superclass* defines the invoked method, in which case the refactoring cannot be applied, or if the invoked method is defined higher up in the hierarchy,

in which case *method* can be pulled up without any problem. This is equivalent to our precondition 5 described above.

- It checks if super invocations of other subclasses of *superclass* will invoke the pushed up method instead of the originally invoked method. If so, the refactoring is rejected. Self invocations are not checked, because that is not necessary as the possibly replaced or overridden methods are either semantically equivalent or copied to the subclasses.

We see that the Refactoring Browser does a thorough analysis of possible replacement and overriding of methods. It does not, however, completely analyse invocations performed by *method* as we discussed in precondition 2 of this refactoring. It allows the push up of a method that invokes methods in its originally containing class as depicted in case (a) of Figure 6.8. Behaviour will not break in Smalltalk, because *method* was only invoked on its originally containing class or its subclasses. However, unless a method with the same signature is defined higher up in the inheritance hierarchy, the moved code references methods in its subclass, which is bad style. In Java it would not even compile. We do not allow it for both these reasons.

Java

Werner [Wer99] includes a thorough investigation of invocations performed by *method*. He does not deeply investigate possible replacement and overriding of (inherited) methods of *superclass*. He just does not allow it, which is the same approach we have taken. He does not analyse super invocations as we do in precondition 5 and consequently breaks behaviour in the cases we describe there.

An interesting reference to mention with regard to precondition 6 about Java constructors, is Fowler's Pull Up Constructor Body refactoring¹. This refactoring is about pulling up part of constructor body to the superclass constructor and calling the superclass constructor from the subclass one (see Figure 6.12). How-

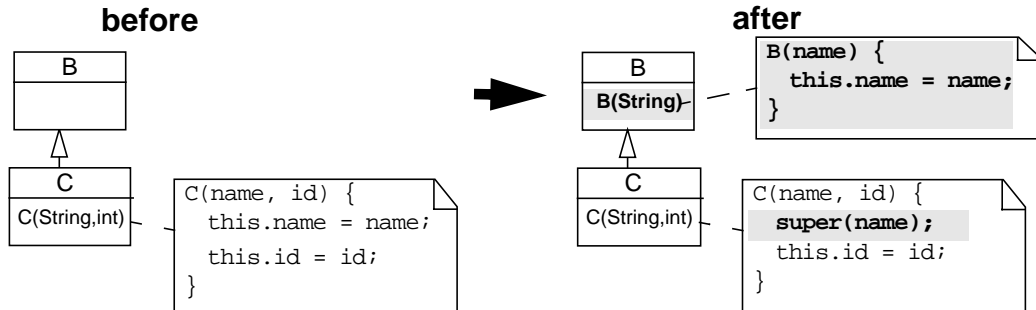


Figure 6.12 Pull Up Constructor Body as defined by Fowler [FBB⁺99]

ever, this is more a variant of the Extract Method refactoring than of the Pull Up Method refactoring. And for both the Extract Method Refactoring and the Pull Up Constructor Body refactoring information about the actual method bodies is needed that we do not have available in our metamodel.

1. The Extract Method refactoring takes a code fragment of a method and turns it into its own method [FBB⁺99].

Discussion

Most of the preconditions are language independent. Some checks need to be performed for the special cases of constructors (precondition 6), interfaces (precondition 7) and metaclasses (precondition 8). We see here that the semantical differences of the mapped entities (e.g., Java interface to FAMIX class) to their standard interpretation (e.g., Java class to FAMIX class) result in extra language-dependent checks. However, exactly the same mappings *are* useful in the language-independent preconditions. For instance, although an explicit check for the case of metaclasses needs to be made in precondition 8, preconditions 2, 4 and 5 are equally valid for Smalltalk classes *and* metaclasses.

Class-scope methods

The preconditions are also sufficient for the case of class-scope methods. In Smalltalk those are instance methods of the metaclass and all rules and transformations for instance methods can be applied. In Java static methods cannot have the same name as instance methods. Possible violations by the refactoring are therefore covered by precondition 4, because this precondition does not differentiate between instance and class-scope methods. Thereby static methods are not allowed to hide instance methods with the same signature anyway [GJSB00]. All other preconditions equally hold for both static and instance methods in Java. Even precondition 7 as abstract methods cannot be static [GJSB00]. The language-specific precondition 8 that deals with metaclass members could be removed if these metaclass members would be interpreted as class-scope members of the class rather than instance members of the metaclass. However, other problems with the mapping of Smalltalk to FAMIX — for instance the possibility for class and instance methods having the same name — would have to be solved (see also section 4.4.2).

Related work

From the related work we see that the major Smalltalk and Java approaches both have their own foci and omissions. Especially the copying down of replaced or newly overridden methods in the Refactoring Browser is interesting. The Pull Up Method refactoring we present does not involve a downcopy of any original superclass method. Partly because it does not appear in other approaches, especially in Fowler's refactorings catalog [FBB⁺99], and thus is against common perception of what the refactoring is supposed to do. Secondly, in cases like situation (b) in Figure 6.11, copying down results in duplicated code, effectively worsening code quality. In our solution, as a result from not copying down, we cannot ignore any invocations to methods that are possibly replaced or overridden. The advantage of the Refactoring Browser approach is that self invocations to methods that are copied down do not need to be checked and the refactoring is therefore applicable in a wider number of cases.

Multiple inheritance

Multiple inheritance is not explicitly dealt with in the preconditions. This is not necessary, because the only multiple inherited related issue that can crop up, namely the unhiding of a method with the same signature inherited from another superclass than *superclass*, does not pose a problem for the supported languages Java and Smalltalk. Consider the example depicted in Figure 6.13. In Smalltalk this cannot occur as it does not have multiple inheritance. In Java this situation can occur with the three following cases:

- B is a Java class and A is a Java interface. In that case there is no problem. The program still compiles and functions as before. The resulting design might be considered bad, because the interface

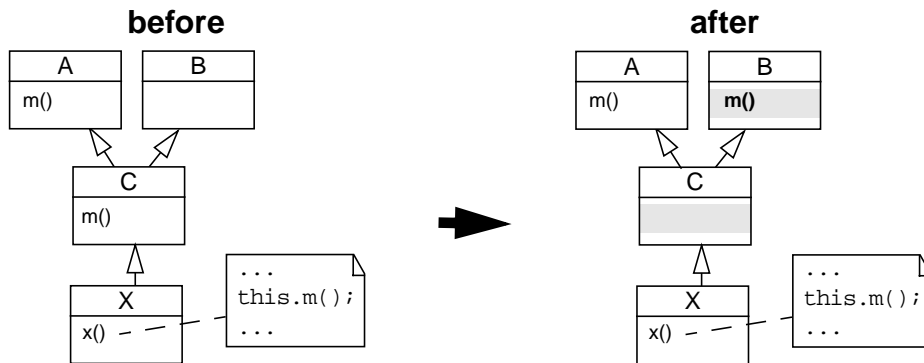


Figure 6.13 Pulling up C.m() to B unhides A.m() from C and its subclasses

defines a method that the implementing class inherits from somewhere else. But this is up to the developer to consider on a case-to-case basis.

- A is a class and B is an interface. This imposes the additional requirement on C.m() that it is abstract, which is covered by precondition 7. In this case there is no problem either.
- Both A and B are interfaces. Again the additional requirement that C.m() is abstract applies. And again there is no problem with compilation or behaviour preservation after the refactoring.

Support for C++ however would require reconsideration, because for that language the scenario of Figure 6.13 poses the problem that the call `this.m()` from `X.x()` would be ambiguous in the ‘after’ situation. The system would not compile anymore. Solutions would be to not allow the refactoring in that case resulting either in an additional language-dependent precondition or a language-independent precondition which would be too restrictive for Java. A second solution would be to explicitly scope all invocations such as the one in `X.x()` to explicitly invoke `B.m()`.

PUSH DOWN METHOD (*METHOD*)

Pushes down *method* to all direct subclasses that do not contain a method with the same signature.

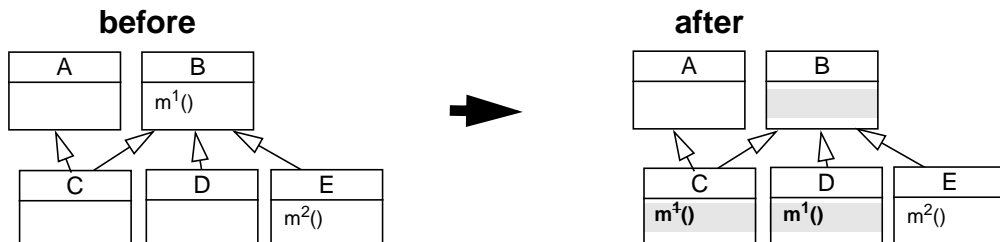


Figure 6.14 Push Down Method refactoring pushing *method* B.m() down to C and D

Similarly to Pull Up Method, the lack of type information in Smalltalk makes it hard to determine if *method* (B.m()) in Figure 6.14 is called on (instances of) its defining class (B in Figure 6.14). The same problem arises when trying to determine which subclass branches of B possibly invoke m(). Because of this, the refactoring is specified to push down methods to all subclasses rather than a single subclass. In Java, due to the static typing, we have sufficient information available to push down to a single subclass. This would, however, result in separate definitions of this refactoring for different languages.

Preconditions

Language-independent preconditions

1. *method* must not be invoked in or through its defining class unless it only invokes itself on self/this.
2. At least one direct subclass of the defining class of *method* may not already contain a method with the same signature as *method*.
3. self/this accesses to attributes that are also defined in one or more of the direct subclasses, may not exist in *method*.
4. self/this invocations and accesses to private members of the containing class may not exist in *method*.
5. no super invocations of *method* may exist in the direct subclasses of the defining class.
6. super invocations to methods that are also defined in the defining class may not exist in *method*, except if the invoked method has the same signature as *method* itself.
7. super accesses to attributes that are also defined in the defining class may not exist in *method*.
8. subclasses cannot inherit non-abstract methods with the same signature as *method* from other superclass branches.

Java-specific preconditions

9. *method* must not be a constructor.

Precondition analysis

1. *method must not be invoked in or through its defining class unless it only invokes itself on self/this.*

If *method* is invoked on its defining class, pushing down can have two results. One possibility is that the program is broken, because *method* will not be found anymore by the invocation either at compile-time in Java or runtime in Smalltalk. The other possibility is that another method with the same signature is defined somewhere in the superclasses of the defining class (e.g., in superclasses of B in the example in Figure 6.14). In this case behaviour preservation is not guaranteed. In the special case that *method* only invokes itself, it can be pushed down, because the self/this invocation will be moved with the method¹.

In Smalltalk if the receiving class (i.e., the statically determinable type of the receiving variable) of an invocation is not known (i.e., if the receiver is not self, super or a class), the method can be pushed down if the containing class is abstract or not referenced. In such a case the containing class is known not to be instantiated and method is thus never invoked on an instance of the containing class. In Java the static type of the receiving variable is always known.

Cast accesses of Java are covered by this precondition as well. E.g., `((B)a).m()` in Java is interpreted as being an invocation of `m()` on B.

2. *At least one direct subclass of the defining class of method may not already contain a method with the same signature as method.*

If all subclasses already define a method with the same signature as *method*, there is no subclass left where method can be pushed down to. Application of the refactoring would result in removing *method* without letting an equivalent method appear in a subclass. Because this code disappearance is possibly unwanted and unexpected, we do not apply the refactoring in such a case.

3. *self/this accesses to attributes that are also defined in one or more of the direct subclasses, may not exist in method.*

Otherwise another attribute will be accessed after the refactoring. The precondition is trivially fulfilled for Smalltalk as it allows only one attribute with the same name in the same inheritance hierarchy.

4. *self/this invocations and accesses to private members of the containing class may not exist in method.*

Private members cannot be referenced from outside their containing class and are statically linked. Pushing *method* down would invalidate these references: either a compile error occurs or another method is referenced, which breaks behaviour preservation. The precondition is trivially preserved for Smalltalk, which only has public methods.

5. *no super invocations of method may exist in the direct subclasses of the defining class.*

If this precondition is violated, after pushing *method* down any super references from subclass methods would not point to *method* anymore, but either to another method in the superclass hierarchy of the original defining class or to no method at all if no such superclass method exists. Figure 6.15 shows an example.

In Smalltalk only methods can be accessed with super, in Java both methods and attributes.

1. We consider both `this.m()` and `m()` in Java as an invocation of `m()` on `this`, as the latter is only a shorthand for the former.

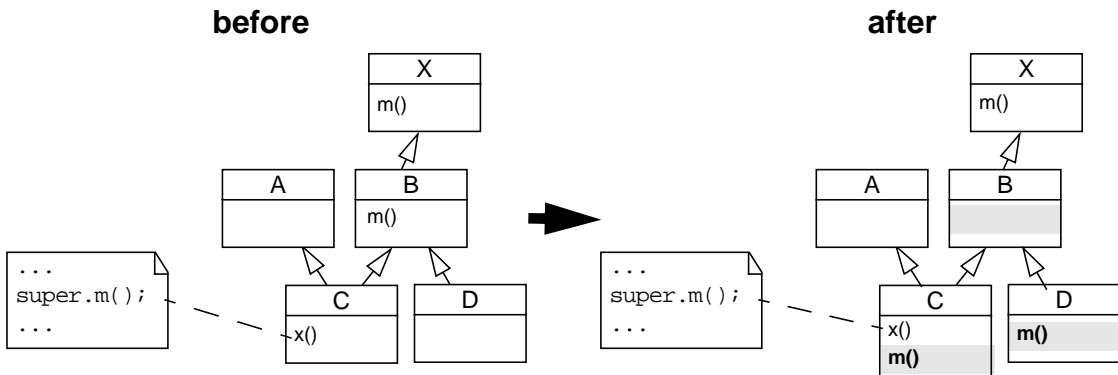


Figure 6.15 The super reference in `C.x()` points to `B.m()` before and to `X.m()` after the refactor-

This precondition could possibly be relaxed by replacing the super references with self references and so point to the same method after the refactoring. However, this is only possible if no subclasses define methods with the same signature as *method*. Because *self* is dynamically bound to the instance, it would invoke the subclass method on instances of the subclass rather than the method that was statically referenced by *super*.

6. *super* invocations to methods that are also defined in the defining class, may not exist in method, except if the invoked method has the same signature as *method* itself.

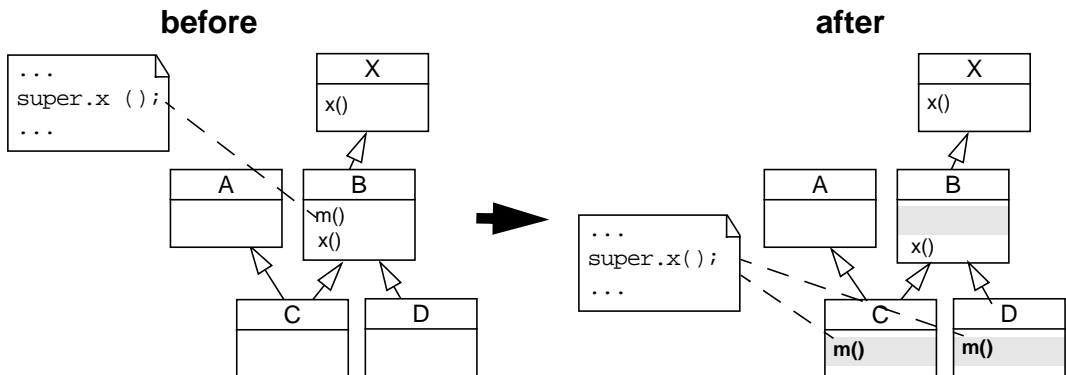


Figure 6.16 The super reference in `m()` points to `X.x()` before and to `B.x()` after the refactoring

If *method* (`B.m()` in the example in Figure 6.16) invokes methods (like `X.x()` in the example), these other definitions would be called instead afterwards. This possibly changes behaviour. If the method invoked via *super* has the same signature as *method* itself (which is actually the most common case), then there is no problem because *method* will be removed from its defining class and the invoked method will therefore be the same one.

7. *super* accesses to attributes that are also defined in the defining class, may not exist in method.

Idem as precondition 6 but for attributes instead of methods. The precondition is trivially fulfilled for Smalltalk as it allows only one attribute with the same name in the same inheritance hierarchy.

8. *subclasses cannot inherit non-abstract methods with the same signature as method from other super-class branches.*

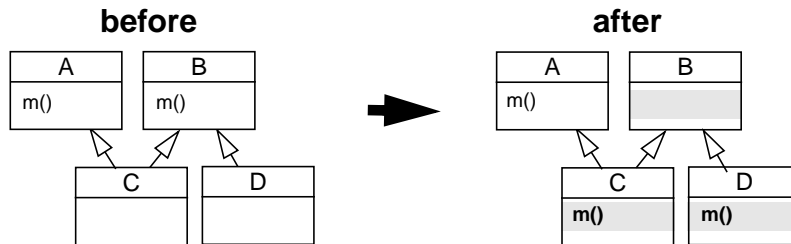


Figure 6.17 Pushing down B.m() causes A.m() to be overridden in C

This precondition is a general one for multiple inheritance systems. Figure 6.17 shows an example. In Smalltalk this situation cannot occur as it has single inheritance. For Java the following cases need to be considered:

- B is a Java class and A is (consequently) a Java interface and A.m() is abstract. In this case there is no problem, because the implementing method in C stays the same.
- A and B are Java interfaces. A.m() and B.m() is consequently abstract and C is an interface or an abstract class (as it does not implement m()). Therefore pushing down is not a problem, because no implementations are hidden.
- B is a Java interface and A is a Java class. Pushing it down will hide A.m(). Unless A.m() is abstract, this cannot be allowed, hence this precondition.

9. *method must not be a constructor.*

A Java constructor is bound to its class, even by its name which is always the same as its defining class. Therefore it can not be moved out of its class in any way.

Related work

Opdyke [Opd92] and Roberts [Rob99] do not describe this refactoring.

Smalltalk

The Refactoring Browser [RBJ97] implements this refactoring. It allows the pushing down of a method if the defining class is abstract (as defined for Smalltalk in section 4.4.2). In such a case the method is known to not be invoked on an instance of the defining class. However, invocations from within the class itself as discussed in precondition 1 are not considered. If the method is originally invoked from a method in its defining class, in its final position in a subclass it will be invoked from a superclass method. In a Smalltalk this is not a problem due to the dynamic typing and the fact that the defining (abstract) class is never instantiated anyway. But unless a method with the same signature is inherited the result is bad style code. And in Java, not relevant for the Refactoring Browser but relevant for our work, a compilation error will occur. Therefore we do not allow this.

The Refactoring Browser also checks if there are any super references to *method* in the direct subclasses, which is equivalent to our precondition 5. It does not check, however, method calls *from* the target method. This is no problem for self calls, because after the refactoring they will call the same methods as they did

before (as *self* is always dynamically bound to the current instance), but it breaks behaviour in the case of super references, because they are statically bound and therefore refer to a different class after the refactoring (see our precondition 6).

The other preconditions we describe are trivially preserved for the Smalltalk language.

The Refactoring Browser automatically expands the scope of pool dictionaries to retain visibility for the moved methods.

Java

Werner [Wer99] describes this refactoring as moving a method to one of its subclasses instead of all of them (as Fowler does [FBB⁺99]). He includes a thorough investigation of invocations by *method*, but does not check invocations to *method* as we do in precondition 1 and super invocations by *method* as we do in precondition 6. Werner also does not check for constructors (our precondition 9). These omissions can lead to a change in behaviour of the target system.

An interesting precondition he describes is that *method* is not allowed to invoke any method on the same instance (i.e., using *this* or implicitly) that is overridden in the subclass. This situation is depicted in Figure 6.18. However, *m()* cannot be called on instances of B (through precondition 1). Consequently it can only be called on instances of C. In both cases, before and after the refactoring, *this.x()* will call *C.x()*, because lookup with *this* always starts at the current instance. Therefore, it is an unnecessary precondition.

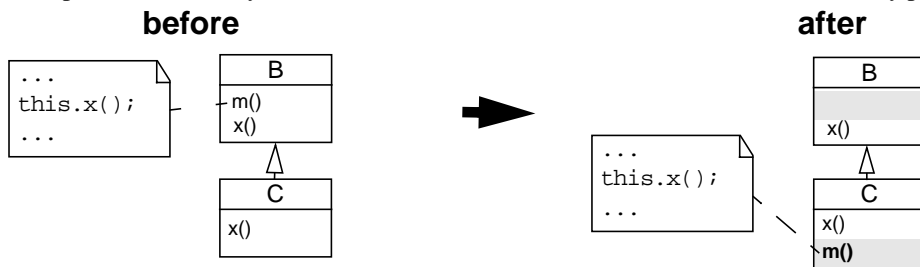


Figure 6.18 Both before and after *m()* will call *C.x()* on instances of *C*

Another choice Werner makes is, instead of not allowing the refactoring if accessed attributes exist that are accessed by *method* (as in our precondition 3), to replace the *self/this* references by explicit casts. See Figure 6.19 for an example. We do not do this, because it is a language-dependent solution and widely considered a bad coding style.

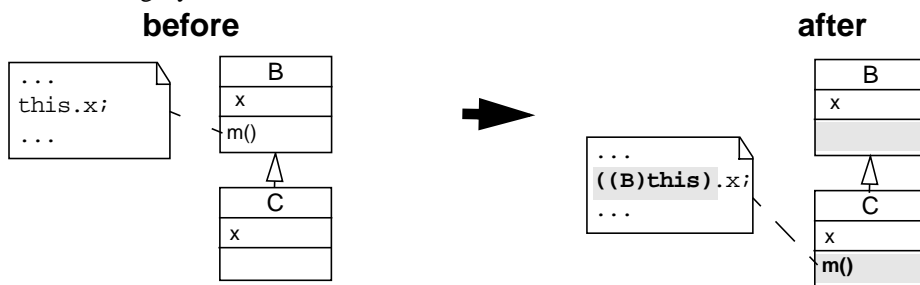


Figure 6.19 Solving attribute accesses by inserting casts

Discussion

For this refactoring a clear choice has been made to use a certain definition of it to be able to serve multiple languages. We have chosen to push down to all subclasses, because it can both be supported by Java and Smalltalk. This is also the way the Refactoring Browser implements this refactoring. However, as reflected by both Fowler's [FBB⁺99] and Werner's [Wer99] descriptions this is not what Java developers might want or expect from this refactoring. It might also not be the preferred way for visual tools that support this refactoring by dragging methods to a subclass.

This refactoring also does not deal explicitly with the pool dictionaries in Smalltalk. If a method has access to pool dictionaries and it is moved to another class, it still needs to have access to the same pool dictionaries. We let the Smalltalk code transformer take care of that transparently.

ADD PARAMETER (NAME, METHOD)

Adds parameter with name *name* to *method*. All invocations to the method get an added argument with a default value.

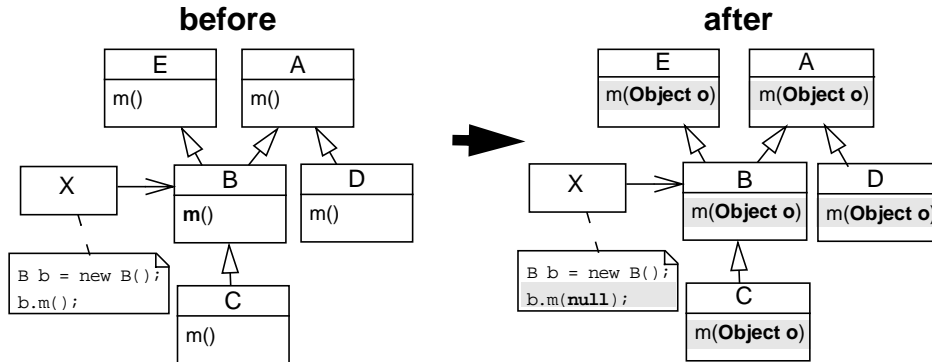


Figure 6.20 Add Parameter refactoring adding `o` to `m` in class `B`

This refactoring is a variant of the Rename Method refactoring. In both refactorings the signature of *method* changes. Analysis of methods and invocations that need to be changed is therefore the same in both cases. A difference with the Rename Method refactoring is that this refactoring can be applied to Java constructors without any problem, because, although the signature of *method* changes, the name of *method* does not.

In Smalltalk adding (or removing) a parameter requires the name of the method to be changed. Every parameter in the method name must be preceded by a part of the method name followed by a colon. Example:

create: anObject **with:** aValue

When adding a parameter, a method without any parameter needs at least a colon to be added, a method that already has parameters needs some text ending with a colon to be added. Some examples:

create becomes **create:** anObject,

create: anObject becomes **create:** anObject **with:** anObject

Preconditions

Language-independent preconditions

1. *method* must not already have a parameter with *name*.
2. *method* must not already have a local variable with *name*.
3. containing class must not have an attribute with *name*.
4. no classes or global variables with *name* may exist in the system.
5. all superclasses of the class containing *method* as well as the subclass hierarchies of the highest superclasses that define a method with the same signature as *method*, must not already contain a method with a signature implied by adding a parameter with *name* to *method*.
6. the candidate invocations to the group of methods that need to be renamed, do not have candidates that are methods outside of this group.

Language-dependent preconditions

7. *name* must be a valid parameter name.

Smalltalk-specific preconditions

8. no methods with same original signature as *method* may exist outside the inheritance hierarchy of *method*.

Precondition analysis

1. *method must not already have a parameter with name*.

Otherwise name clashes will occur.

2. *method must not already have a local variable with name*.

Otherwise name clashes will occur.

3. *containing class must not have an attribute with name*.

For Smalltalk a violation of this precondition will result in a compile error independent of the fact if the original attribute is accessed in *method* or not. The language just does not allow it. In Java, a parameter can have the same name as an attribute. The original attribute will be hidden in *method*. If the original attribute is not accessed in *method* behaviour will be preserved. If the original attribute is accessed, the system will compile if the new parameter has the same actual interface as the attribute with the same name, but behaviour is not preserved.

We do not allow this situation for both languages, even though in some cases for Java this precondition is not a problem. Obviously this conveniently keeps the precondition the same for both languages. For the few cases it is allowed for Java we argue that the resulting code is confusing, because two entities have the same name in a partly overlapping scope. Code quality decreases by allowing it.

4. *no classes or global variables with name may exist in the system*.

In Smalltalk a global variable or class with a certain name is hidden by a parameter with the same name¹. A second point similar to precondition 3: hiding is confusing and the resulting code is worse than if another name is chosen. Therefore, we do not allow the hiding of global variables. Java does not have global variables so that part of the precondition is trivially conserved. The case of classes with similar names as variables is allowed in Java and does not hide the class from the scope of the variable with the same name. The following is valid Java code:

```
public void aMethod( int String ) {
    String x = new String();
    x = "Sander";
    System.out.println("int String = " + String);
    System.out.println("String x = " + x);
}
```

Again confusing code is the result, so we do not allow this.

1. Although the Smalltalk compiler (at least in VisualWorks 3.0) warns the user about the hiding.

5. *all superclasses of the class containing method as well as the subclass hierarchies of the highest superclasses that define a method with the same signature as method, must not already contain a method with a signature implied by adding a parameter with name to method.*

This precondition is similar to precondition 1 of Rename Method, because both adding a parameter and renaming a method result in a change in the signature of the method and thus may result in a name clash with existing methods. See the analysis of precondition 1 of Rename Method on page 81 for details.

6. *the candidate invocations to the group of methods that need to be renamed, do not have candidates that are methods outside of this group.*

This is similar to precondition 2 of Rename Method. In Smalltalk it is possible that invocations (possibly) invoke methods with the same signature in different inheritance hierarchies. These invocations cannot be adapted to reference a changed method, because they possibly invoke a method in another inheritance hierarchy that has not been changed. In Java this precondition is trivially preserved, because its static type information and limited polymorphism restrict the candidate invoked methods to the targeted group of methods within the same inheritance hierarchy.

7. *name must be a valid parameter name.*

The new name should adhere to the naming rules of the implementation language.

Related work

Opdyke [Opd92], Roberts [Rob99] and Werner [Wer99] do not describe this refactoring. However, the Refactoring Browser [RBJ97] implements this refactoring for Smalltalk. It uses a default name for the new parameter (anObject or anObject<nr> if the name ‘anObject’ already exists in the scope of the method). It does, therefore, not need to check the clashes with existing local and global variables and attributes. Different from us, it allows the user to enter an initialization expression which is inserted in every invocation of *method* and a thorough analysis of validity of this expression is performed. We just insert an default empty value (null or nil depending on the language).

Discussion

Clearly most of the preconditions have to do with possible name clashes. A parameter has method scope and if any existing variable exists that has the same name and the same or a wider scope (and therefore is accessible within the containing method), the new parameter will hide this variable and any use of this variable in *method* will break.

All except one of the preconditions are language-independent, partly because they are just language-independent and partly because there are some restrictions that would not be necessary for one of the supported languages (like in precondition 4). These restrictions, however, contribute to keeping the quality of the final code higher by enforcing good practices. Some users, however, may not like the fact that they are restricted and cannot make the decision themselves.

A language dependent issue not reflected in the preconditions, is the method name change required in Smalltalk. The implementation of precondition 4 depends on the signature after the parameter addition and thus on a changed name in the case of Smalltalk. Similarly, the Smalltalk code transformer uses a changed name when it applies the actual code changes. In our implementation we add some default text to the method

name to be able to add the parameter. A nicer, but for this research irrelevant, approach is taken in the Refactoring Browser [RBJ97]. It gives the user the possibility to change the name of the method and the order of the parameters.

REMOVE PARAMETER (*PARAMETER*)

Removes *parameter* from its method. Corresponding arguments are removed from all invocations.

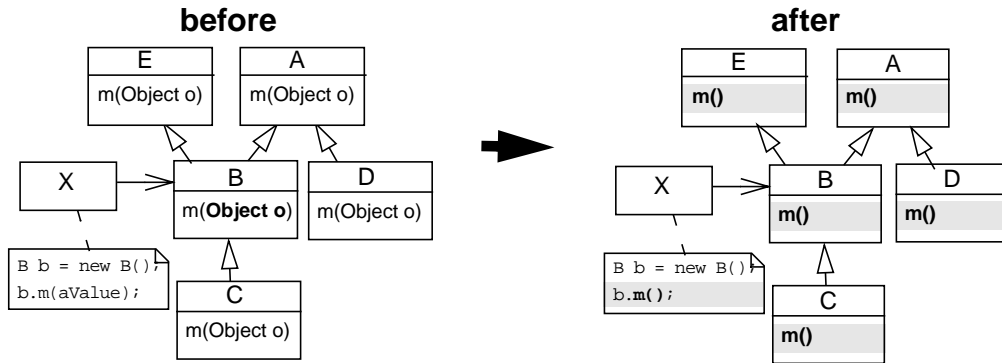


Figure 6.21 Remove Parameter refactoring removing `o` from `m` in class B

This refactoring is a variant of the Rename Method refactoring. In both refactorings the signature of *method* changes. Analysis of methods and invocations that need to be changed therefore is the same in both cases. A difference with the Rename Method refactoring is that this refactoring can be applied to Java constructors without any problem, because, although the signature of *method* changes, the name of *method* does not.

In Smalltalk, similar to the Add Parameter refactoring, removing a parameter requires a name change of the method. We change the name of the method by removing only the colon if the first parameter is being removed, or the text with colon directly preceding the parameter being removed. Some examples:

invokes: `aClass` becomes **invokes**

invokes: `aClass with: anArgument` becomes **invokeswith:** `anArgument`

invokes: `aClass with: anArgument` becomes **invokes:** `aClass`

Preconditions

Language-independent preconditions

1. *parameter* must not be referenced in the containing method or equivalent parameters in any over-riding or overridden method.
2. all superclasses of the class containing the method *parameter* belongs to, as well as the subclass hierarchies of the highest superclasses that define a method with the same signature that method, must not already contain a method with a signature implied by removing *parameter* from its method.
3. the candidate invocations to the group of methods that need to be renamed, do not have candidates that are methods outside of this group.

Precondition analysis

1. *parameter must not be referenced in the containing method or equivalent parameters in any overriding or overridden method.*

If one of parameters is used in the bodies of the containing methods the refactoring would break the system.

2. *all superclasses of the class containing the method parameter belongs to, as well as the subclass hierarchies of the highest superclasses that define a method with the same signature that method, must not already contain a method with a signature implied by removing parameter from its method.*

This precondition is similar to precondition 1 of Rename Method, because both removing a parameter and renaming a method result in a change in the signature of the method and thus may result in a name clash with existing methods. See the analysis of precondition 1 of Rename Method on page 81 for details.

3. *the candidate invocations to the group of methods that need to be renamed, do not have candidates that are methods outside of this group.*

This is similar to precondition 2 of Rename Method and precondition 6 of Add Parameter. In Smalltalk it is possible that invocations (possibly) invoke methods with the same signature in different inheritance hierarchies. These invocations cannot be adapted to reference a changed method, because they possibly invoke a method in another inheritance hierarchy that has not been changed. In Java this precondition is trivially preserved, because its static type information and limited polymorphism restrict the candidate invoked methods to the targeted group of methods within the same inheritance hierarchy.

Related work

Opdyke [Opd92], Roberts [Rob99] and Werner [Wer99] do not describe this refactoring. However, the Refactoring Browser [RBJ97] implements this refactoring for Smalltalk. It essentially implements the same preconditions as we describe here.

Discussion

Basically issues worth mentioning for this refactoring are issues that have been discussed extensively for other refactorings already. The element to be removed, i.e., *parameter* here, cannot be removed if it is still referenced, the resulting method signature may not clash with existing methods within the inheritance hierarchy and then there is the issue of being able to link invocations to a certain method implementation in precondition 3. Similar to precondition 4 in the Add Parameter refactoring, the implementation of precondition 2 is language dependent in the sense that removing a parameter requires a method name change in Smalltalk and not in Java.

ADD ATTRIBUTE (*NAME*, *CLASS*)

Adds the attribute with *name* in *class*.

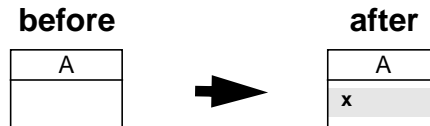


Figure 6.22 Add Attribute refactoring adding an attribute named *x* in class *A*

A simple refactoring. No references need to be updated as the attribute did not exist before. The only check needed to be made is if an attribute with *name* does not already exist in the class hierarchy of *class*. For Java we insert the default type `Object`, for Smalltalk we do not need to provide any type information.

Preconditions

Language-independent preconditions

1. the inheritance hierarchy of the containing class must not already contain an attribute with *name*.
2. no global variable with *name* may exist.
3. no class with *name* may exist.

Language-dependent preconditions

4. *name* must be a valid attribute name.

Precondition analysis

1. *the inheritance hierarchy of the containing class must not already contain an attribute with name.*

There are three ways an existing attribute that already has *name* can occur in the inheritance hierarchy of *class*. Firstly, in *class* itself. Adding an attribute with *name* would result in two attributes with the same name, which is not allowed. Secondly, in a superclass. The new attribute would hide the existing attribute from the subclasses, changing behaviour if the superclass attribute is accessed in or through those subclasses. Thirdly, an attribute with *name* might already exist in a subclass. The new attribute would be hidden from the such a subclass.

In Java the second and third case are allowed and accesses to hidden attributes can be resolved through the use of explicit scoping. However, this requires extensive analysis of the accesses. Furthermore, compatibility of the precondition with Smalltalk is broken, because in Smalltalk two attributes with the same name in the same inheritance hierarchy are not allowed.

2. *no global variable with name may exist.*

The new attribute would hide the global variable from the containing class and its subclasses, changing behaviour if that global variable is referenced in those classes. Java does not have global variables and thus this precondition is for Java trivially preserved.

3. *no class with name may exist.*

In Smalltalk the class would be hidden in the classes where the new attribute is visible, because in Smalltalk every class is also a global variable. In Java classes and variables can have the same name. They do not hide each other. The following is valid Java code:

```
public class X {  
    int X;  
    public X newInstance() { return new X(); }  
}
```

However, because the resulting code in such a situation is not considered good style and to keep the precondition language independent, we enforce this precondition for Java as well.

4. *name must be a valid attribute name.*

name should adhere to the naming rules of the implementation language.

Related work

Roberts [Rob99] describes the same set of preconditions for this refactoring, which is to be expected because all presented preconditions are hard preconditions for Smalltalk. For Java and C++ different decisions can be made. Werner [Wer99], for instance, allows attributes with the same name in the hierarchy of *class* and resolves existing access clashes with explicit scoping. Opdyke [Opd92] has as only precondition that there is ‘no name collision with an existing member or global variable’ which includes inherited member variables. He does not analyse subclass attributes and existing classes, but he does not have to in C++. As discussed above we do not allow both classes and subclass attributes with the same name for reasons of good coding style and compatibility of the approach with both Smalltalk and Java.

Discussion

Apart from the valid name precondition (precondition 4), which is always present when a new named entity is added, this refactoring only contains language-independent preconditions.

REMOVE ATTRIBUTE (*ATTRIBUTE*)

Removes *attribute* from its containing class.

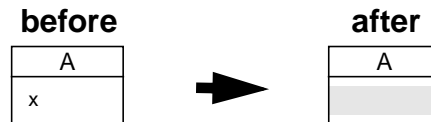


Figure 6.23 Remove Attribute refactoring removing the attribute x from class A

If not accessed by any method the attribute can be removed from its containing class.

Preconditions

Language-independent preconditions

1. *attribute* must not be accessed.

Precondition analysis

1. *attribute must not be accessed.*

If *attribute* is accessed, either from methods in its containing class or subclasses of this class (in Java and Smalltalk), or from any other method outside the inheritance hierarchy (only in Java), removing it breaks compilation or changes behaviour.

If accesses to *attribute* exist, it is a possibility to analyse attributes that would be unhidden by the removal of *attribute*. Accesses to *attribute* would access the superclass attribute instead. However, the only case in which behaviour does not change is if the superclass attribute is not used for other purposes. Figure 6.24 shows an example where both superclass attribute (A.i) and the attribute to be removed (B.i) are used in parallel. Removing B.i would result in a change in behaviour. We do not analyse unhidden attributes.

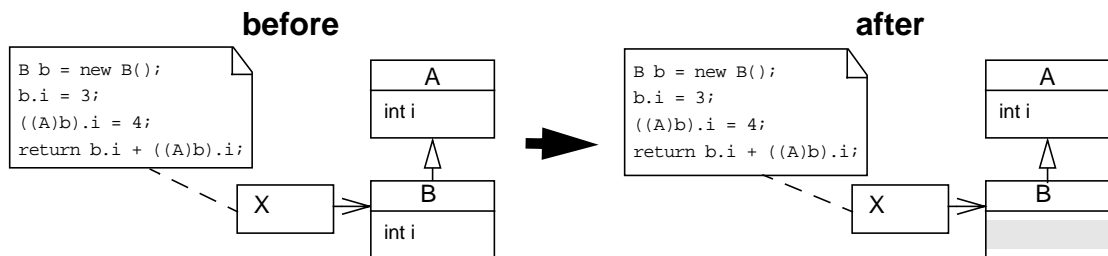


Figure 6.24 The code returns 7 before and 8 after the removal of B.i

Related work

Opdyke [Opd92] (for C++) and Roberts [Rob99] (for Smalltalk) describe the same precondition. Roberts is slightly more restrictive in the sense that in his version the attribute is not to be accessed *within the inheritance hierarchy*, which is an obvious restriction as in Smalltalk an attribute can only be accessed within the inheritance hierarchy. With hierarchy Roberts means the subclasses, superclasses and the containing class

itself. The superclasses need not really to be checked because an attribute cannot be accessed from a superclass, but Smalltalk allows only one attribute with a certain name in a hierarchy so it does not really matter.

Werner [Wer99] allows the removal of an attribute even if it is accessed if the removal unhides an attribute with the same type of a superclass that was not used in the subclass hierarchy before (through casting). The unhidden attribute would take over the role of the removed attribute thereby preserving behaviour. However, a simple scenario shows that this is not always the case (see Figure 6.24). Also parallel use of the hiding attributes outside of the containing inheritance hierarchy needs to be analysed.

Discussion

This refactoring is simple and completely language-independent. Only one — language-independent — precondition exists. The approach of Werner discussed above is firstly not sufficient to ensure behaviour preservation and secondly FAMIX does not provide any dataflow information to be able to (attempt to) analyse a version of that approach that would take accesses external to the inheritance hierarchy into account. We also feel that the complexity does not outweigh the advantage of being able to apply the refactoring in the special case of non-clashing parallel use of hiding attributes.

Note that removing B.i in Figure 6.24 is not a problem if B.i is not accessed (which is expressed in the only precondition) or if C.i is not accessed and has the same type as B.i (which is a case we do not cover).

RENAME ATTRIBUTE (*ATTRIBUTE*, *NEW NAME*)

Renames *attribute* to *new name*. All accesses of *attribute* are changed to refer to the new name.

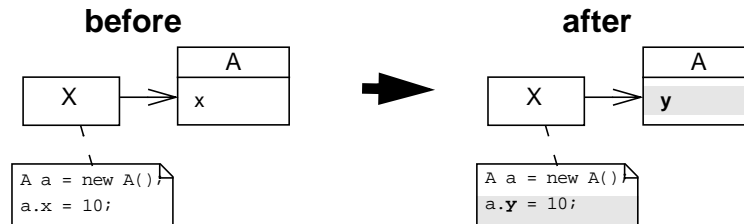


Figure 6.25 Rename Attribute refactoring renaming A.x to A.y

An attribute can be renamed to a new name if there is not yet an attribute with the same name in the same scope already. The accesses to *attribute* need to be updated to refer to the new name.

Preconditions

Language-independent preconditions

1. the inheritance hierarchy of the containing class must not already contain an attribute with *new name*.
2. no global variable with *new name* may exist.
3. no class with *new name* may exist.
4. methods of the containing class and its subclasses that access *attribute* must not contain a local variable with *new name* already.

Language-dependent preconditions

5. *new name* must be a valid attribute name.

Precondition analysis

1. *the inheritance hierarchy of the containing class must not already contain an attribute with new name.*

This precondition is identical to precondition 1 of Add Attribute. The refactoring would either result in two attributes with the same name in the same class, or the new attribute would hide existing attributes or be hidden by it. See Add Attribute for details.

2. *no global variable with new name may exist.*

This precondition is identical to precondition 2 of Add Attribute. The renamed attribute would hide the global variable. See Add Attribute for details.

3. *no class with new name may exist.*

This precondition is identical to precondition 2 of Add Attribute. In Smalltalk the renamed attribute would hide the class. To keep the precondition language independent and because the result would be bad style code, we also enforce this precondition for Java. See Add Attribute for details.

4. *methods of the containing class and its subclasses that access attribute must not contain a local variable with new name already.*

A local method variable with the same name as an attribute in the same scope hides this attribute from (part of) the method scope. This is not a problem as long as *attribute* is not accessed in this method. If it is, renaming would give it the same name as the local variable, turning the access of the attribute into an access of the local variable and changing behaviour.

5. *new name must be a valid attribute name.*

new name should adhere to the naming rules of the implementation language.

Related work

Opdyke [Opd92] (for C++) describes the refactoring in general terms. Any variable (if global or an attribute or a local variable or a parameter) can be renamed including an update of the references to it, as long as there are no name clashes in the scope of the renamed attribute. Consequently he does not allow local method variables to hide the renamed attribute, even if the attribute is not referenced in this method. We do allow hiding if the method does not reference *attribute* (see precondition 4).

Roberts [Rob99] (for Smalltalk) checks the same as we do except for the local variables, which might lead to a behaviours change (see precondition 4). The Refactoring Browser [RBJ97], however, does check for local variables.

Werner [Wer99] allows hiding of existing attributes with *new name*. Name clashes are resolved with explicit scoping of the attributes involved. We do not allow this (see precondition 1). Hiding and explicit scoping decreases code quality and is not a viable solution for Smalltalk. Furthermore, Werner, like Roberts, does not take local variables into account.

Discussion

The same discussion as for the Add Attribute applies with a small addition. The Rename Attribute refactoring has one additional precondition (namely precondition 4), because it needs to deal with possible name clashes of existing accesses to the attribute. Similarly Rename Attribute needs to update the attribute accesses on top of renaming the attribute, where Add Attribute only needs to add the attribute.

PULL UP ATTRIBUTE (*ATTRIBUTE*, *SUPERCLASS*)

Pulls up *attribute* to one of its superclasses (*superclass*) removing all attributes with the same name and type from all subclasses of *superclass*.

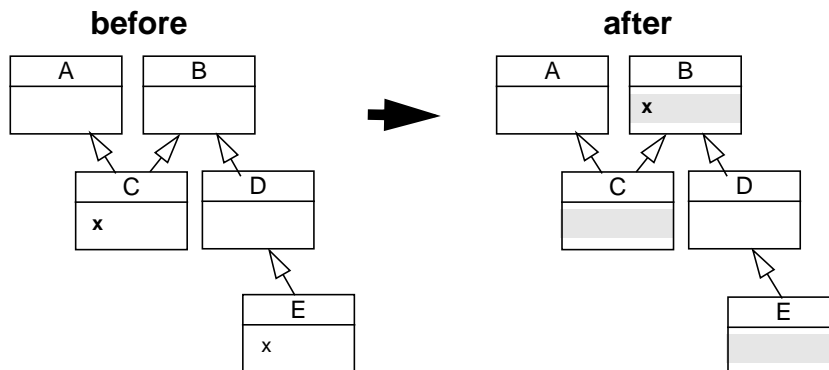


Figure 6.26 Pull Up Attribute refactoring pulling *attribute* C.x up to *superclass* B

Not only is the attribute pulled up, but all attributes with the same name and type in other subclasses of superclass are pulled up as well and merged into one declaration. The visibility of the attribute in its final location is at least protected to keep it visible in its previous location, and public if one or more of the pulled up attributes was public before the refactoring. To resolve visibility differences between the different attributes that are pulled up the widest visibility needs to be chosen. All original accesses to the attributes still work because they have either the original access to the attribute or an even wider one.

Preconditions

Language-independent preconditions

1. *superclass* must not contain an attribute with the same name as *attribute*.
2. any attribute in the subclasses of superclass with the same name as *attribute* must have the same type as *attribute*. These attributes must not hide each other.
3. pulling up *attribute* may not hide in *superclass* another attribute with the same name.
4. pulling up *attribute* may not unhide any attribute with the same name from other superclass branches of the containing class.

Precondition analysis

1. *superclass* must not contain an attribute with the same name as *attribute*.

Otherwise the superclass attribute would be replaced (which is equivalent to removing *attribute*). As *attribute* and the superclass attribute can be used in parallel as discussed in the motivation of precondition 1 of Remove Attribute on page 105, this cannot be allowed unless either *attribute* or the superclass attribute is not accessed within the system, which we do not check for. The precondition is trivially preserved for Smalltalk as no two attributes with the same name are allowed in an inheritance path.

2. *any attribute in the subclasses of superclass with the same name as attribute must have the same type as attribute. These attributes must not hide each other.*

The attributes with the same name that are pulled up into a common attribute in *superclass* need to have the same type, because in Java the resulting system will not compile otherwise¹. In Smalltalk the precondition is trivially preserved as Smalltalk is dynamically typed. Even if the attribute holds attributes with completely different types at runtime, the system will still function as before, because independent of the types the same objects are used at the same places before and after the refactoring.

Hiding of those attributes is not allowed for the same reasons as in the previous precondition.

3. *pulling up attribute may not hide in superclass another attribute with the same name.*

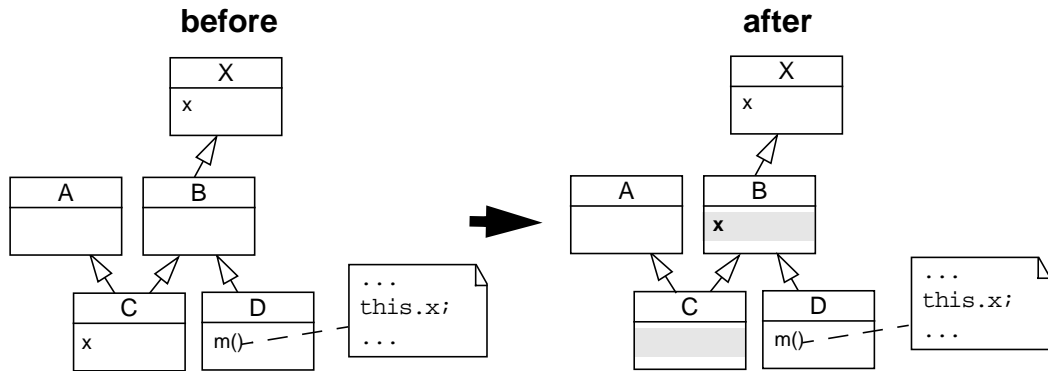


Figure 6.27 B.x hides X.x from subclasses that use it (D)

This situation could change behaviour, because any references in the subclasses to the now hidden attribute would reference the pulled up attribute instead (e.g., the reference to X.x in D.m() in Figure 6.27). In Java this can be resolved by explicitly scoping to the now hidden attribute, i.e., by changing the code in D.m() from `this.x` to `((X)this).x`. We do not do this, because the resulting code is more complex than the original. Another solution would be to give the user a choice.

1. The precondition can be relaxed by allowing attributes to have substitutable types and giving the pulled up attribute the general type. We do not analyse this possibility.

4. pulling up attribute may not unhide any attribute with the same name from other superclass branches of the containing class.

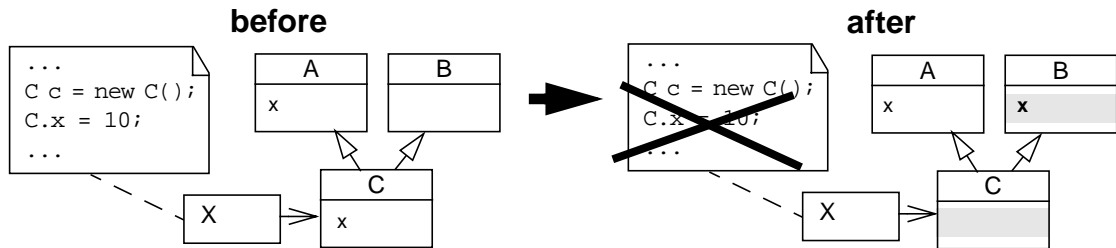


Figure 6.28 The access to C.x before is ambiguous after the refactoring

Any original access to *attribute* would become ambiguous otherwise. Figure 6.28 shows an example. The presented situation can only occur in Java as Smalltalk only has single inheritance. Explicitly casting the accesses of C.x to access B.x after the refactoring is not possible in Java. This is not contradictory to the statement in the analysis of precondition 3. Casting successfully circumvents hiding variables, but does not resolve ambiguous inherited attributes. In C++ explicitly scoping the variable is possible [Str97].

Related work

Werner's definition of this refactoring [Wer99] pulls the attribute up even if it hides an attribute with the same name higher up in the inheritance hierarchy (as depicted in the example in Figure 6.27). Any access to that attribute in or through B is explicitly scoped to use the same attribute after the refactoring. The fact that other attributes might hide the pulled up attribute is not analysed, but can be safely ignored in Java.

Roberts [Rob99] describes the Pull Up Attribute refactoring from the point of view of the superclass (which is actually the most consistent view given the name of the refactoring). He only checks if there is a direct subclass defining the attribute that is intended to be pulled up. If yes, this attribute is pulled up and all other attributes with the same name in other subclass branches of the superclass are removed as well. Because his work is focused on Smalltalk he does not need check the type of the attributes (because the attributes do not have a static type) and replacing and hiding (because only one attribute in an inheritance chain is allowed in Smalltalk).

Opdyke [Opd92] describes this refactoring under the name 'move_member_variable_to_superclass'. The only difference with our approach is that Opdyke does not analyse hiding and the possibility of parallel use of hidden and hiding attributes.

Discussion

The chosen definition of this refactoring fits both languages. The choice to remove all attributes in the subclass branches of *superclass* is needed for Smalltalk, which does not allow multiple attributes with the same name in a inheritance chain. For Java it would not be necessary but the resulting code would possibly contain hidden attributes, which is bad coding style.

All preconditions are Java-specific but trivially preserved for Smalltalk. They deal with type information, hiding and scoped accesses and visibility of attributes. Which are Java specific features. Smalltalk attributes are (implicitly) protected meaning they are not visible outside of their inheritance hierarchy. Also

only one attribute with a particular name may exist in an inheritance chain, which takes away all the difficulties that appear in Java due to hiding and scoping. Furthermore, the attributes are dynamically typed and are all interpreted to have the most general type `Object`.

PUSH DOWN ATTRIBUTE (*ATTRIBUTE*)

Push *attribute* down to its subclasses.

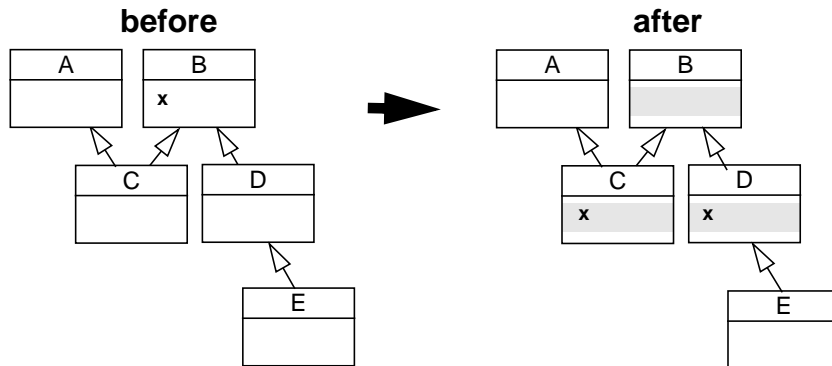


Figure 6.29 Push Down Attribute refactoring pulling *attribute* B.x down to its subclasses

This refactoring pushes *attribute* down to all its subclasses.

Preconditions

Language-independent preconditions

1. *attribute* must not be accessed in or through its containing class.
2. the direct subclasses of the containing class must not contain an attribute with the same name as *attribute*.

Precondition analysis

1. *attribute must not be accessed in or through its containing class.*

By pushing down the attribute the scope of this attribute is narrowed by taking the containing class away from the scope. However, if there is any access in this class it will break afterwards, because attribute is not available in that scope anymore. In Smalltalk this precondition is easier preserved, because attributes can only be accessed from the class it is defined in or its subclasses. In Java, although often regarded as bad practice, it is possible to access an attribute from outside the class. The receivingClass attribute of the Access records the class on which an attribute is accessed, because they might have been used in parallel with *attribute* using scoping.

2. *the direct subclasses of the containing class must not contain an attribute with the same name as attribute.*

This is trivially preserved for Smalltalk, because there will never be an attribute with the same name in the subclasses since the language does not allow it. In Java it is possible and these subclass attributes cannot just be overwritten, because both attributes may be used in parallel to store different values (see also the analysis of precondition 1 of the Remove Attribute refactoring).

Related work

We see different definitions of this refactoring. Opdyke [Opd92] and Roberts [Rob99] use the same definition we use, namely pushing *attribute* down to all subclasses. Werner [Wer99] pushes down to only one user-definable subclass. Fowler [FBB⁺99] and the implementation of the Refactoring Browser [RBJ97] push down to all subclass branches where the attribute is used.

Opdyke checks accesses to attributes in its containing class (i.e., our precondition 1) and does not allow *attribute* to be private. Although it can be argued not to push down private attributes, because there is a difference in reducing the scope of an attribute (which is what happens when a non-private attribute is pushed down) and moving an attribute from one scope to the other (which is what happens when a private attribute is pushed down), there is no real harm in pushing down a private attribute as it is not referenced anyway according to precondition 1. Opdyke does not check for subclass attributes with the same name and no multiple inheritance issues.

Roberts [Rob99] only checks if attribute is not referenced in class. This is sufficient for attributes in Smalltalk as they can only be referenced from its containing class and its subclasses, only one attribute with a certain name may exist in inheritance chain and Smalltalk does not have multiple inheritance.

Werner [Wer99] pushes down to a one distinct subclass. He checks if that subclass contains an attribute with the same name already. He does not check accesses on the original containing class, which might render the refactoring behaviour breaking (see precondition 1).

Discussion

The refactoring is defined to push the attribute down to all subclasses rather than to only one subclass or the subclasses in which hierarchies the attribute is used (see also the Related Work section above). It makes the analysis slightly easier, because no checks are needed to find out in or though which subclasses the attribute is accessed.

As we model refactorings in a multiple-inheritance environment it is necessary to have a look at possible consequences of hiding other attributes by pushing down *attribute*. Figure 6.30 shows a scenario. A.x would be hidden by C.x after the refactoring possibly breaking accesses to A.x from C or its subclasses. However, the ‘before’ situation cannot occur. Smalltalk has single inheritance, in Java any access is ambiguous and cannot be explicitly scoped. In C++ the accesses are ambiguous.

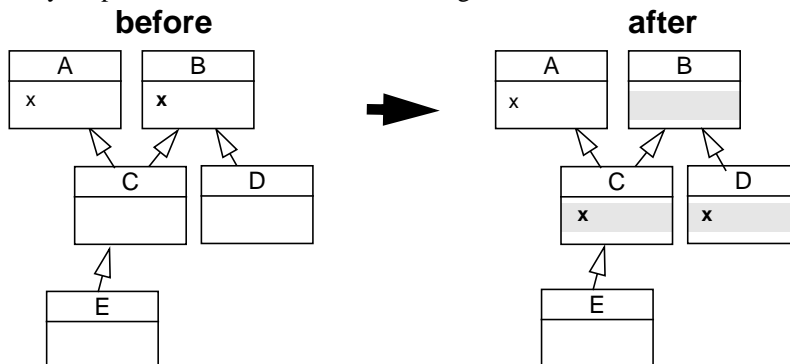


Figure 6.30 Pushing down B.x to C hides A.x from (subclasses of) C

6.4 Validation

Validation of this kind of work is a difficult point. Without a formal definition of the target languages, every statement will be based on some assumptions somehow. Basically the question that needs answering is, if the presented preconditions are correct, necessary and sufficient. Firstly, the discussions of the refactorings in section 6.3 clearly aim at providing confidence in the correctness and necessity of the preconditions. The sufficiency cannot be proven as such, as we cannot go further than carefully check the affected program elements in the original scope of the target methods before the refactoring and the possibly final scope after the refactoring.

Secondly we have compared our findings with existing work in the field [Opd92] [RBJ97] [Rob99] [Wer99], as shown in the related work sections of the refactorings. We have found several holes in these other approaches in the process.

Thirdly, we have verified the work by doing experiments. We have built a prototype, the Moose Refactoring Engine, that supports the fifteen refactorings described in this chapter. It is part of the Moose Reengineering Environment, a tool environment for reengineering object-oriented systems, which is discussed in more detail in chapter 5. We have used the refactoring engine in two ways. One is a non-trivial sequence of refactorings on two similar toy banking system, one implemented in Smalltalk and one in Java. Secondly we have applied the refactorings on the code of our Moose environment in Smalltalk and on the JUnit framework in Java. These experiments, including a detailed description of the engine itself, are presented in chapter 7.

6.5 Discussion

The refactorings are to the greater extent language-independent. It is hard to sensibly quantify the language independence, but the following numbers give an impression.

The 15 refactorings define 67 preconditions, from which 51 are language independent (~ 76%), 5 are Smalltalk dependent, 4 are Java dependent and the remaining 7 define the language-dependent precondition that a entity name should conform to the rules of the implementation language of the target system. The 11 Smalltalk- and Java-specific preconditions can be classified as follows:

A class is not a Smalltalk metaclass	3
Instance methods of Smalltalk metaclass rather than class methods	2
Method is not a Java constructor	3
Superclass is not a Java interface	1

Mostly the language-dependent checks that need to be performed are simple checks that certain operations are not possible for certain specific entities. Examples are that the class refactorings (Add Class, Remove Class and Rename Class) cannot be applied to Smalltalk metaclasses. And some of the method refactorings (Rename Method, Pull Up Method and Push Down Method) cannot be applied to Java constructors. However, Remove Method as well as the parameter refactorings (Add Parameter and Remove Parameter) can be applied to constructors without any problem. And preconditions for other refactorings transparently take constructors (as just another method) and their invocations into account. For instance, in the Push Down Attribute refactoring it is checked if no method (i.e., all methods including constructors) in its defining class accesses the attribute.

A few preconditions are really specific for the semantics of the language-specific construct. An example is precondition 7 of Pull Up Method, which states that a non-abstract method cannot be pulled up into an interface. However, this is the only precondition that is interface specific. In all other cases the interface as class can be treated as any other class, including the multiple inheritance. For the actual code transformations the fact if a FAMIX class represents an Java interface or a Java class, needs to be taken into account more often, because it influences the syntax of the resulting code. An example is the use of the `implements` keyword in the case of Java interfaces and the `extends` keyword for Java classes.

The following additional observations can be made:

Language independence brings useful reusability. Major parts of the refactorings are described and analysed on a language-independent level. Similar concepts in the different languages are treated in a uniform way, resulting in reuse of analysis and reducing the language specifics to only the changes in the source code. However, in some cases the advantages of reuse come at a cost:

- *Increased complexity of algorithms.* To deal with multiple languages the underlying model needs to be general enough to cover the supported languages. For instance, the model supports multiple inheritance, which involves more complexity than would be needed, for instance, for single inheritance in Smalltalk alone.
- *Mapping back to the actual code.* The actual code transformations are, naturally, language specific. Therefore, in some cases the concepts that are generalized at the language-independent level (e.g., Java constructors are methods, Java interfaces are classes) need to be mapped back to their language-specific kind, because at the code level they need to be dealt with differently than their ‘normal’ counterparts. For example, on the code level invocations of Java constructors are different from invocations to ‘normal’ methods. This implies that the language-specific information about how an entity has been mapped needs to be stored, because it is necessary information when mapping back.
- *Language-independent defaults.* To keep some refactorings as language independent as possible, some defaults are used. Typical examples are types: some refactorings use the most general type, i.e., Object for both Smalltalk and Java. This works well for both languages, although it is clear that support for defining or changing types would be desirable for statically typed languages such as Java.
- *Definitions of refactorings are tuned for compatibility over multiple languages.* There is (only) one refactoring in the presented set, namely Push Down Method, which is defined the way it is, because of one of the implementation languages. It pushes down to all subclasses, although it could have been defined otherwise if in Smalltalk there would be enough information to push down to only one subclass. See Push Down Method on page 91 for more details.
- *Checks are not necessary for all supported languages.* Depending on the implementation language some of the presented language-independent checks would not need to be carried out. An example is the analysis for attributes hiding each other, which cannot happen in Smalltalk. Optimisations in the responsiveness of a refactoring tool could be realised by, in the case of the above example, not checking the language-independent Java-specific preconditions for Smalltalk. However, as one of the goals of this research is to maximise reusability and language independence we did not pursue this path.

Not all language differences can be abstracted from. i.e., most refactorings cannot be completely described at a language-independent level. We see the following kinds of issues:

- Standard issues that are apparent in all languages, but need a language-specific interpretation, like if a name of a class is a valid class name *for that language*.
- Issues that are caused by the *mapping* from the language to FAMIX. For example, the metamodel does not know the concept of metaclasses or interfaces. Rules that apply to these specific concepts need to be checked nonetheless and are inherently language specific.
- The most problematic issues are in the *core differences* between the languages. The fact that Smalltalk is dynamically and Java statically typed, means that there is less information available at compile-time. Especially for dependency analysis through invocations and accesses, the type information tells much more precisely which method is invoked or which attribute is accessed. In dynamically typed languages a certain method invocation can be any method with that signature, no matter what class it is defined in. Therefore, some refactorings can only be applied for dynamically typed languages when more severe restrictions are taken into account. An example is the Rename Method refactoring which can only be applied when there is no method with the same signature as the method to be renamed outside of the targeted inheritance hierarchy. Note that the type information for dynamically typed languages can be refined through additional analysis (for instance, using type inference techniques) [Rob99], but this is outside the scope of this thesis.

All in all we can say that the presented model is adequate to represent refactorings for multiple object-oriented languages. The program entity level of information is sufficient for refactorings that do not need detailed information about method bodies. Some language-dependent details, however, must be coped with.

Influence of metamodel design decisions. Many design decisions for the model—to apply a language-independent naming scheme including scoping and the different mappings to allow to treat similar constructs in different languages in a similar way—result in language independence and reuse of analysis code. However, especially with the mappings, it is always a trade-off between reuse and complexity. Instead of mapping similar constructs to one representation, the two constructs can be both modelled explicitly. Naturally this decreases problems with differences between the constructs, but it also makes the model less general and opportunities for reuse could be missed. Another possibility is to not model a construct at all. This typically allows to get rid of language specifics, but also makes the model less useful.

In the context of refactoring the chosen mappings, most notably those of Java constructors to methods and Java interfaces and Smalltalk metaclasses, have worked out well. We especially found both Java mappings to easily fit and allow to exploit the similarities with other constructs. For the metaclass mapping the advantages are less clear. Method and Attribute refactorings can be applied to (members of) metaclasses without any problems, but the class refactorings are not applicable at all. An alternative would be to not model metaclasses explicitly and model metaclass methods and attributes as class (in Java static) methods and attributes of the class the metaclass is representing. We have chosen not to do this, because, as said, some refactorings do work with this scheme and the alternative mapping results in problems with name clashes between class methods and instance methods and problems with the equal treatment of instance level class attributes and class level instance attributes which are different concepts in Smalltalk.

One of the modelling decisions that has worked out particularly well is the way candidate invocations are modelled. As can also be read in section 4.4.1 every invocation lists the possibly invoked methods. In Smalltalk this list can be considerably larger than in Java, because in Smalltalk there is no static type information is available to restrict an invocation to a certain class or hierarchy of classes. However, the list of candidates

of an invocation abstracts from the differences in static and dynamic typing in the analysis of possible targets of invocations and so hides one of the main differences between Java and Smalltalk at the conceptual level of defining refactorings in terms of FAMIX. An example is precondition 2 of the Rename Method refactoring.

Support for other languages. A word about supporting other languages than the ones discussed in this chapter. The FAMIX model is already set up to support more languages than Smalltalk and Java. Explicit mappings are defined for both C++ [Bar99] and Ada [Neb99] and our FAMIX-based toolset has been actively used to analyse systems in these languages. Furthermore, a project is underway to build a C++ refactoring tool based on the FAMIX metamodel [Bor01]. Therefore, we are confident we can use our model and extend our tool to support these and other languages without too many problems. All standard object-oriented features are supported, most notably in comparison with other approaches multiple inheritance and a combination of static and dynamic typing. Of course, for every newly supported language the model and the definition of the refactoring need to be carefully checked to see if the semantics of already modelled features is different and if this difference demands a change in the model and/or a change in the refactoring definition, for instance in the form of adapted preconditions or an added language-dependent precondition.

CHAPTER 7

The Moose Refactoring Engine

The Moose Refactoring Engine is the part of Moose that provides code transformation support. It implements the fifteen refactorings described in chapter 6. Consequently, the analysis performed by the refactoring engine, i.e., checking the preconditions and determining what pieces of code need to be changed, is completely based on the Moose repository, and thus on the information available according to the FAMIX metamodel and its language extensions. The Moose Refactoring Engine currently supports Smalltalk and Java refactorings.

The Moose Reengineering Environment, described in chapter 5, validates the FAMIX metamodel for its adequacy to support multiple cooperating reverse engineering tools. This chapter focuses on validating the refactoring theory presented in chapter 6. As such it also provides in-depth validation of the language-independence of FAMIX. Refactoring requires complex semantical analysis with information that is sufficiently precise, complete and correct. Otherwise a transformation cannot not be safely applied. This must be seen in contrast to the analysis tasks that can often work with partial or slightly imprecise information [MNGL98] [Bis92]. Furthermore, the refactorings change the (inherently language-specific) source code and therefore require the metamodel to supply sufficient information about the language mappings.

The practical goal of the refactoring engine is to integrate refactoring support in Moose. In this way reverse engineering and reengineering can seamlessly work together to, on the one hand, support system analysis and on the other hand to propose solutions for problems found in terms of (semi-)automated code transformations.

We start with a description of the architecture in section 7.1. Afterwards we describe the experiments we have done with the engine in section 7.2. We finish with a discussion (section 7.3).

7.1 Architecture

The architecture of the Moose Refactoring Engine is depicted in Figure 7.1. The different parts are (see also the numbers in the figure):

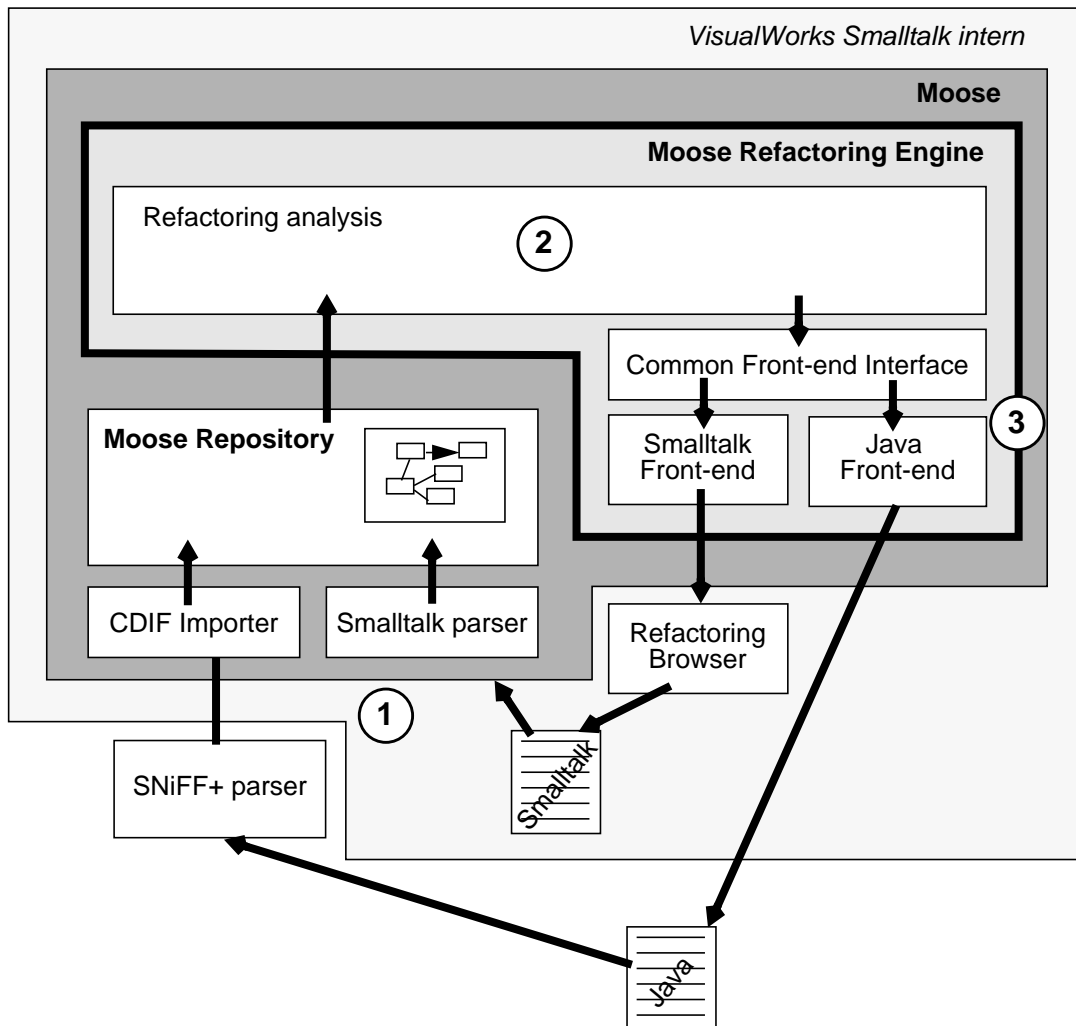


Figure 7.1 Architecture of the Moose Refactoring Engine

Parsing and importing of the target system (1). First a system is parsed by either an external parser (SNiFF+ [Tak96] for Java) or an internal parser for Smalltalk, and imported into the repository. This is standard functionality of Moose and is not specific for refactoring.

Refactoring analysis (2). The analysis part implements the analysis that is presented in section 6.3. It gathers its data from the Moose repository with which it checks the preconditions of the refactoring. It also collects the relevant model elements that represent the source code that needs to be transformed. If all preconditions are fulfilled, the engine uses the gathered information to trigger the actual code transformers, the *code transformation front-ends* we discuss hereafter.

The Code Transformation Front-ends (3). The code transformation front-ends perform the final low-level code transformations. They work directly on the source code, hence they are language specific. They cannot work on the level of the model, because it does not contain enough information to regenerate source code. Instead they use the source anchor information in the model to determine where a specific transformation must take place.

The front-end implementation classes have a common interface for all supported languages. This makes it easy to exchange and add front-ends. Figure 7.2 shows an example. The `MethodTransformer` interface, used by the Pull Up Method, Push Down Method, Add Method and Remove Method refactorings, contains three methods that must be implemented by all transformation front-ends. Implementing classes only need to take care of a local implementation of the action, e.g., `removeMethod`: only physically removes a method. It does not check any preconditions or update any references.

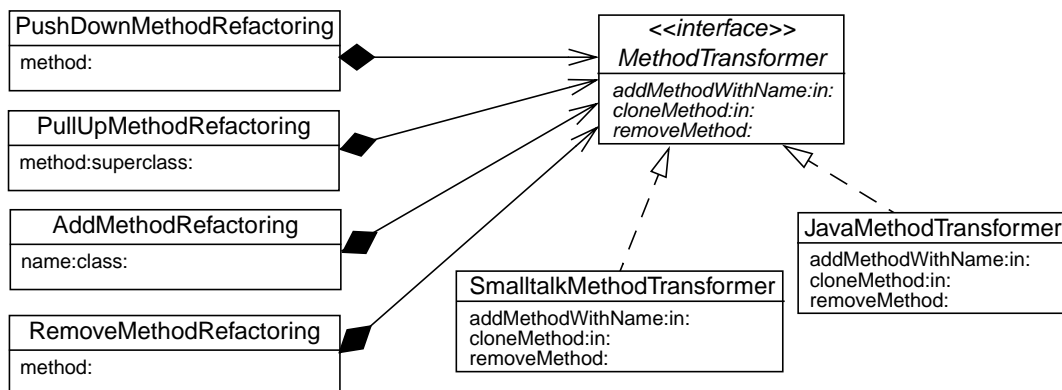


Figure 7.2 Front-end interface for the method refactorings

In some of the front-end classes methods exist in the public interface that do not have an implementation for some of the supported languages. As an example Figure 7.3 shows the required interface for Rename Class front-end classes. The Java front-end implements all methods. In the Smalltalk front-end only three methods have a meaningful implementation, namely `changeClassName`, `changeSuperClassReferenceOf:` and `changeClassMethodInvocation:Of:`. The other four methods have empty implementations and are there only for interface compatibility reasons. These methods deal with changing type declarations — which Smalltalk does not have — and accesses to class attributes using the classname to reference the containing class¹ — which cannot occur in Smalltalk. However, the Rename Class refactoring invokes all methods of the `RenameClassTransformer`, because it is independent of the targeted language.

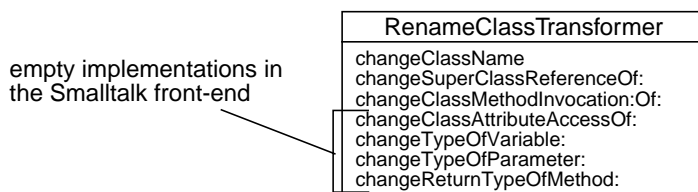


Figure 7.3 Front-end interface for the Rename Class refactoring

1. Like `A.b` in Java to access the static variable `b` of class `A`

The Smalltalk front-end uses the parts of the Refactoring Browser [RBJ97] for low-level code transformations to change Smalltalk code. The Java front-end currently uses a text-based approach based on regular expressions. It supports all our refactorings as long as the source code adheres to certain layout rules. We plan to move to an abstract syntax tree based approach in the future, because it better abstracts from these layout details and better fits the more complex code transformations. However, for the purpose of validating the refactoring analysis of chapter 6 the current implementation suffices.

7.2 Validation

We have validated the prototype in two ways. One is the application of a non-trivial sequence of refactorings on a toy banking system, one time implemented in Smalltalk and one time in Java. Secondly, we have applied all refactorings on real world code, namely on our Moose environment in Smalltalk and on the JUnit testing framework in Java [JUn]. We discuss these experiments now in detail.

7.2.1 A non-trivial refactoring sequence on a toy banking system

The case study consists of the application of a sequence of refactorings to two small pieces of similar Smalltalk and Java code. Both implementations have a testsuite included that thoroughly tests the application. This allows us to apply exactly the same transformation sequence on both Smalltalk and Java and test behaviour preservation and language independence. The sequence includes all fifteen refactorings of the refactoring engine.

The code implements a toy banking system with a Bank, Customers and Accounts (see Figure 7.4 (1)). The refactorings are applied to gradually add transaction support to the Customer and Account class. After every refactoring in the sequence, a testsuite is run to test if the adapted software still functions as expected. This testsuite is adapted by the refactorings as well, as it contains references to the classes and methods of the application.

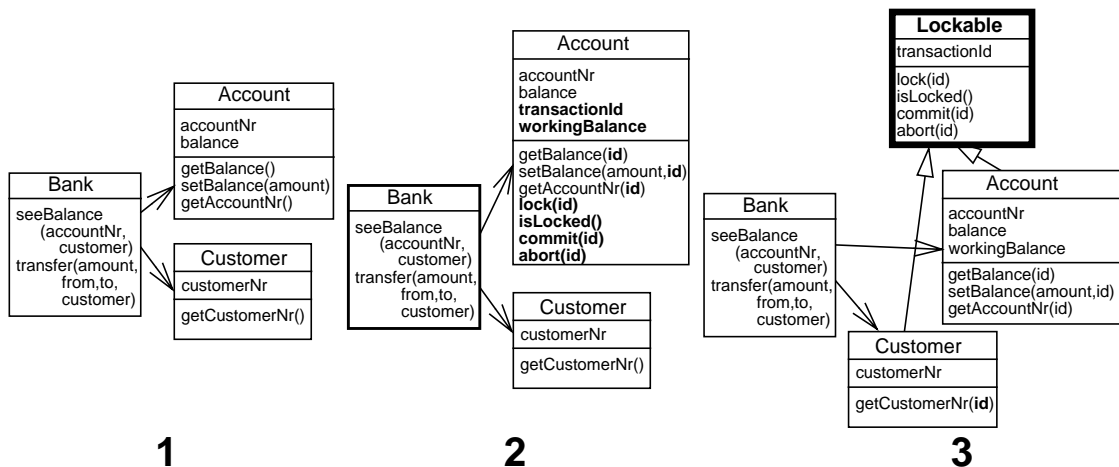


Figure 7.4 Refactoring scenario introducing transactional support to a toy banking system

Figure 7.4 shows the scenario in a nutshell. The transactional support that is added, comprises the adaptation of the Account and the Customer class with locking functionality for a two-phase commit protocol.

The functionality is added in two steps. In the first step the Account class gets transactional support (from **1** to **2**). In the second step (from **2** to **3**) the generic part of the transactional support is lifted into a newly added common superclass of Account and Customer, so that a customer can be locked as part of a transaction as well.

We now list sequence of refactorings in detail starting with the transformation from **1** to **2**:

1. Add Attribute: `transactionId` and `workingBalance` to Account
2. Add Method: `lock()`, `isLocked()`, `commit()` and `abort()` to Account
3. Add Parameter: `id` to `getBalance()`, `setBalance(amount)` and `getAccountNr()`, `lock()`, `commit()` and `abort()`.

Method bodies need to be added to the new methods `lock()`, `isLocked()`, `commit()` and `abort()` and the method bodies of `getBalance`, `setBalance` and `getAccountNr` need to be adapted. This is not covered by the refactorings and is therefore done by hand. We also add code to Bank to create transaction ids, but we do not describe the details here. Furthermore, we adapt the tests to cover the added transactional behaviour. This finishes the first step. The step from **2** to **3** involves the following refactorings:

4. Add Class: Lockable with subclasses Account and Customer
5. Pull Up Attribute: `transactionId` to Lockable
6. Pull Up Method: `isLocked(id)` to Lockable

To pull up `lock(id)`, `commit(id)` and `abort(id)` the method bodies of these methods need to be adapted to separate the account specific functionality (such as setting the `workingBalance`) from the generic transactional functionality. For instance, from the `commit` method the account specific functionality, namely committing the working balance to the balance, is extracted in a separate method called `commitWorkingState` (see Figure 7.5). This separation can be realised using an Extract Method refactoring [FBB⁺99], but this refactoring requires information that is not available in FAMIX. Consequently, it is not covered by our engine. In this experiment we extract the method by hand.

The next action is to create abstract template methods in the superclass for the extracted methods such as `commitWorkingState`. This allows us to pull up the transactional methods.

7. Add Method: `lockWorkingState`, `commitWorkingState` and `abortWorkingState` to Lockable and make them abstract by hand.
8. Pull Up Method: `lock(id)`, `commit(id)` and `abort(id)` to Lockable

And finally we change the `getCustomerId` method to use the now inherited transactional support:

9. Add Parameter: `id` to `getCustomer(id)` and adapt its body by hand.

The scenario does not inherently include the renaming refactorings. We just add them to cover the complete set of chapter 6:

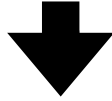
10. Rename Class: Customer to Client
11. Rename Method: `Lockable.isLocked()` to `locked()`
12. Rename Attribute: `Account.accountNr` to `accountNumber`

All refactorings that are not covered yet, are covered by reversing the scenario from **3** to **1** to bring the Bank application back to its original, non-transactional state. Renamed entities are given their original name again. Pushed up attributes and methods are pushed down and added parameters, methods, attributes and the Lockable class are removed. Where necessary method bodies are again adapted by hand.

```

commit: id
    self require: [self isLocked: id] usingException: #lockFailureSignal.
    balance := workingBalance.
    workingBalance := nil.
    transactionIdentifier := nil.

```



```

commit: id
    self require: [self isLocked: id] usingException: #lockFailureSignal.
    self commitWorkingState
    transactionIdentifier := nil.

commitWorkingState
    balance := workingBalance.
    workingBalance := nil.

```

Figure 7.5 Separating Account specific from generic transactional code

We have applied the above scenario successfully on implementations in Java and Smalltalk. FAMIX supports fifteen of the sixteen refactorings needed to perform the scenario. Only the Extract Method refactoring is not supported, because FAMIX does not contain sufficiently detailed information about method bodies. Obviously, the parts of the scenario that change the behaviour of the bank application, for instance, changing the `getBalance` method to check calling provides the right transaction identifier, are not covered by the engine as well.

7.2.2 Experiments on Moose and JUnit

We have applied all fifteen refactorings on the Moose reengineering environment in Smalltalk (see also chapter 7) and the JUnit testing framework in Java [JUn]. The goal is to show that the refactorings are also applicable to applications that solve real world problems, rather than having been engineered for the purpose of testing the refactoring engine. Criteria for these case studies are that the software comes with a comprehensive testsuite that can be run before and after the refactorings to test behaviour preservation, and that the source code is available to us. Moose and JUnit fulfil these requirements. Moose is a middle sized system (~350 classes, ~2200 methods), JUnit is small (78 classes (inner classes not counted), ~700 methods)¹. We have applied all refactorings of the refactoring engine and, similarly to the experiment in section 7.2.1, we have tested behaviour preservation by running the available testsuites. In several cases refactorings were rightfully rejected, because their preconditions were not fulfilled.

The experiments show no more than that separate refactorings can be successfully applied on larger unprepared software systems. The only way to do a more comprehensive real world test, is to use the refactor-

1. These numbers take the test code into account.

ing engine in day-to-day software development, so that it is verified in many different situations. However, the refactoring engine is a research prototype and as such not ready for developers to use in their daily work.

7.3 Discussion

The goal of the Refactoring Engine is to provide an in-depth validation of the analysis presented in chapter 6. It is a thorough test for the fitness of the FAMIX metamodel as a language-independent metamodel for reengineering object-oriented software.

The sequence of refactorings in the toy banking example shows that it is possible to express and execute a sequence of refactorings that covers all supported refactorings for the languages Java and Smalltalk. Although the example is engineered, we feel the example presents a non-trivial sequence of refactorings, which realistically mimics usages of refactorings in real world development. Secondly, we have tested the refactorings successfully on two real world systems. The experiments show that the information FAMIX provides is sufficient for checking the preconditions and determining what code must be transformed.

To assess the Moose Refactoring Engine from the tool point of view, we discuss it in the context of a series of success criteria that Don Roberts describes in his thesis [Rob99]. Roberts poses the following technical criteria:

- **Program Database.** “A refactoring tools needs a programming database to be able to search for various program elements across a program” [Rob99]. With Moose and its repository this requirement is obviously fulfilled. However, for instance, Smalltalk environments have their own code database, which is continuously updated with the latest changes to the code, i.e., the repository and the code are *causally connected* [Mae87]. Currently our code transformation front-ends do not update the model in the Moose repository in parallel to code changes. The changed code needs to be reparsed and effectively a new, updated, model is created. This is something we want to change, because it clearly hampers the usability of the tool. However, a reparse eases the ability to test if a refactoring has been applied correctly, because the gathered information completely independent of the refactoring implementation. This was especially convenient in the early stages of development of our prototype.
- **Abstract Syntax Trees (ASTs).** Refactorings require access to method bodies, mainly to update references to code elements that have been changed. This normally requires ASTs. Our Smalltalk front-end uses ASTs by means of the Refactoring Browser. In contrast, our Java front-end currently uses a text-based approach based on regular expressions, because it was easy and quick to set up. Although this approach is more powerful than we initially expected and supports all code transformation we need to apply, it requires code layout rules to be taken into account. We plan to move to an abstract syntax tree based approach in the future, because it better abstracts from these layout details and better fits the more complex code transformations.

Furthermore, we do not support refactorings that require detailed information below the method level, such as Extract Method, because FAMIX does not provide this level of information. Although it could be added to the metamodel, it is unlikely to happen in the near future, because it is not a focus of our environment. More detailed information makes it harder to support multiple languages (see also section 3.3).

- **Accuracy.** A refactoring tools must “reasonably preserve the behaviour of programs” [Rob99]. As discussed in chapter 6, we define behaviour preserving as that input-output behaviour is the same before and after the refactoring. We do not consider real-time constraints or code that uses reflective

features of a language. The accuracy of the tool depends on the accuracy of the analysis the tool is based upon. The accuracy of the analysis is discussed in section 6.4, its experimental validation in section 7.2.

Roberts also discusses three practical criteria, namely *speed*, an *undo* mechanism and a tight *integration with the environment*. The Moose Refactoring Engine is research tool and as such is not aimed at providing industrial-strength speed and usability to developers. We shortly discuss the practical criteria anyway, for completeness and to give an impression of the current status of the engine.

- **Speed.** For a tool used in the daily work of a developer, the automatic refactoring must execute the refactoring faster than the developer can do it by hand. The Moose Refactoring Engine is not optimised for speed in daily use. The execution of the refactoring is not particularly slow, but issues mentioned before such as reparsing the model instead of updating it in parallel with the code transformation, increase the time consumption and make it currently unfit to use as a tool in daily work.
- **Undo.** A multiple undo mechanism increases the support for an exploratory approach as to which refactorings to apply to increase a piece of code. The Moose Refactoring Engine does not have undo support.
- **Integration with Environment.** The Refactoring Engine is part of Moose. It can be called by any tool that knows how to use it. However, the engine still really is an engine. It is not, like the Refactoring Browser, integrated with code browsers. This is future work. Beyond browser integration we would like to integrate more strongly with analysis tools. Integrated tools would not only detect problems, for instance in the design of a system, but also propose a sequence of refactorings to resolve such a problem.

Finally, the Moose Refactoring Engine can be judged on its ability to support other languages than the currently supported Java and Smalltalk. Assuming FAMIX and Moose support the newly targeted language already, two issues need to be considered. First, for every refactoring the analysis of chapter 6 must be checked to see if it needs adaptation due to semantics specific to the new language. This is discussed in section 6.5 in more detail. Secondly, a code transformation front-end must be developed. The architecture of the Moose Refactoring Engine supports easy additions of such front-ends, but, as the front-ends deal with low-level code transformations, implementation can be complex, mostly depending on the complexity of the syntax of the new language and the available tool support for parsing and code generation.

CHAPTER 8

Conclusion and Future Work

Designing a metamodel that successfully supports a reengineering environment requires explicit knowledge not only about the relevance of the metamodel contents for reengineering, but also about infrastructural aspects such as scalability, interoperability and extensibility.

This thesis provides a better understanding of these issues by making explicit a set of possible design choices including their trade-offs, firstly, with a design space for infrastructural aspects of reengineering metamodels in general, and secondly, in the particular case of large object oriented systems, with a metamodel that supports reverse engineering and refactoring in a language-independent way.

The metamodel that underlies the repository of a reengineering environment, determines to a large extent how well such an environment supports multiple cooperating reengineering tools. However, for only a few of the existing environments the metamodel design choices are explicitly discussed, and these discussions mostly focus on the support for one particular reengineering task. Indeed, no general comprehensive overview exists that describes possible metamodel design decisions, their trade-offs and interdependencies. Consequently, developers who build tools for reengineering need to gather this knowledge over and over again. This thesis solves a part of this problem in the following ways:

- It makes explicit the infrastructural aspects of reengineering metamodels, i.e., the design aspects that deal with how information is organised and stored. We capture these aspects, the available design choices, their trade-offs and interdependencies in a so-called design space.
- It makes explicit how to model multiple object-oriented languages for the purpose of reengineering. Not only does it show the contents of one metamodel (FAMIX), it also makes explicit what choices this metamodel incorporates to handle multiple object-oriented languages in a common way. In particular:
 - it makes explicit the aspects of object-oriented systems that are relevant to reengineering.
 - it shows how the use of a language-independent core together with mappings of multiple object-oriented languages to this core, provides an effective common coverage of these languages. The mappings explicitly define how a certain model of a software system in a specific implementation language must be interpreted and also capture relevant language-specific information.

- it shows how a metamodel can effectively deal with language differences such as static versus dynamic typing and single versus multiple inheritance versus Java interfaces.

Our approach has the following limitations:

- FAMIX does not model detailed information about method bodies. Consequently, we do not support sophisticated control flow analysis. We have chosen not to pursue that path, because we regard the considerable effort to abstract from the many subtle differences on such a detailed level not weighting up against the advantages of language independence. We consider language-specific tools with full AST information such as Datrix for C++ [BC00], as more appropriate for cases that require this detailed level of information. Furthermore, the additional amount of information would have seriously affected the scalability of our approach.
- Language independence increases complexity for certain analysis operations that would be simpler for a particular language. An example is the refactoring analysis that takes type information and multiple inheritance into account, which are both unnecessary for Smalltalk.

We have validated the ability of FAMIX to support multiple cooperating reverse engineering tools by building a reengineering environment (Moose) with a repository based on FAMIX. Several services and tools have been built using this environment and we have used it to perform several case studies on large industrial software systems. The case studies show that FAMIX indeed supports a whole range of reverse engineering tasks, that it effectively abstracts from the supported implementation languages and that its information level scales well for large systems.

For a more in-depth validation of FAMIX as a metamodel for reengineering, we have analysed its ability to support refactorings on a language-independent level for Smalltalk and Java. Information requirements for refactoring are tighter than those for most reverse engineering tasks. Refactoring requires sufficient, complete and precise information to be able to ensure that the transformations can be applied correctly. This is in contrast to most reverse engineering tasks, which are typically not strongly affected if information is slightly incomplete or incorrect [MNGL98] [Bis92]. In particular our refactoring analysis shows the following:

- The analysis a refactoring requires to perform — to determine what low-level code transformations it needs to apply and to check if these can be applied safely — can be expressed for the greater part in a language-independent way.
- The metamodel must provide information about the language-specific interpretation of metamodel elements. This firstly enables the execution of the language-specific part of the analysis. Secondly, it enables the code transformation engine to apply the correct changes on the (language-specific) code level. Our work provides this language-specific information through well-defined language extensions to the core metamodel.
- While some of the design decisions of FAMIX work out particularly well (e.g., the way polymorphic calls are modelled), some are less convenient (e.g., the decision to model metaclasses as classes).
- The language-specific transformation front-ends define the basic code transformations that need to be implemented for each supported language separately.

The refactoring analysis has the following limitations:

- Refactorings that need control flow information cannot be easily abstracted to multiple languages, because they need access to detailed information about method bodies that is not available in

FAMIX. Consequently, our approach does not cover refactorings such as Extract Method and Inline Method [FBB⁺99].

- Similarly, the FAMIX metamodel does not contain sufficient information to regenerate the complete source code from a model. Consequently, the actual code transformations must be applied directly on the source code rather than on a model and are therefore language dependent.

The refactoring analysis is firstly validated by a thorough comparison with other, language-specific definitions of the same refactorings. Secondly, we have implemented the refactorings in the Moose Refactoring Engine and applied them on several case studies.

Future Work

First of all, we are still refining FAMIX based on requirements of new reengineering tasks we want to support and the experience we get while building the tools that support these tasks. This includes the addition of language features currently not supported such as nested classes and the support for more languages beyond the four (C++, Smalltalk, Java and Ada) we currently support. The addition of more detailed information below the method body level, such as conditional statements, may seem an obvious extension as well, but, as said above, it is not likely we will pursue this path. Another possibility is to extend FAMIX with multiple model support. Moose already supports multiple models, which is used for evolution analysis [LDS01] [Ste01], but it has not yet been formalized in the metamodel.

Apart from refining the metamodel itself we are also looking at explicit metamodel support. The goals are to be able to generate generic tools such as model browsers and exchange format savers as well as the integration with other metamodels such as UML [OMG99]. An explicit metamodel also allows us to better explore the dynamic adaptation and extension of metamodels.

A means to get a wider audience for the knowledge about modelling object-oriented software that FAMIX represents, is the standardisation of such a metamodel. We are currently involved in the constitution of the Graph eXchange Language (GXL) [HWS00]. GXL is a collaborative effort from several academic and industrial research institutes to come up with an exchange format and a set of metamodels for information exchange between reengineering tools. We actively participate in the discussions to come to a standardized program entity level metamodel with FAMIX being one of the main input metamodels.

In the context of language-independent refactoring, support for more languages is an obvious direction to take. For every new language, not only the contents of FAMIX, but especially the refactoring analysis needs careful checking, because it depends more on the actual semantics of the language than on the mere representation of facts that FAMIX provides. Some work has already been done to define C++ refactorings based on FAMIX [Bor01].

Beyond FAMIX and the refactoring analysis, the addition of refactorings to Moose opens a whole new class of possibilities, namely the combination of problem detection and analysis with refactorings. We want to explore to which extent tools can propose a developer solutions to detected problems and perform transformations based on that analysis. An example is the adherence of a software system to a certain architecture. Instead of only signalling mismatches between an expected architecture and the actual architecture, an analysis tool can propose a set of refactorings that resolves the mismatch. For instance, it could move a method from its containing class to a class in another layer of the architecture. Similarly, we are exploring the domain of component mining. We want to support the identification of potential components, as well as the transformation of legacy software to component-based frameworks.

A final direction we would like to mention, is the integration of reengineering techniques in forward engineering tools. The emergence of round-trip engineering tools is a step in that direction. They provide seamless integration between design diagrams and source code, between modelling and implementation [Ree96] [JBR99]. However, round-trip engineering does not need to be restricted to the integration of models and code alone. Reverse engineering and refactoring techniques enable a much more sophisticated round-trip engineering cycle. In addition to model extraction, developers can apply problem detection analysis and create different views on the software to increase their understanding. Furthermore, built-in refactorings allow the developer to quickly and safely adapt software. Many of the technologies already exist, but we see a lot of potential in a tighter integration.

APPENDIX A

Table of Refactorings

This appendix shows an overview of the pre- and postconditions of the refactorings presented in chapter 6.

Refactoring	precondition	postcondition
Add Class (<i>classname</i> , <i>package</i> , <i>superclasses</i> , <i>subclasses</i>)	<ul style="list-style-type: none">no class may exist with <i>new name</i> in the same scope.no global variable may exist with <i>new name</i> in the same scope.all <i>subclasses</i> must be subclasses of all <i>superclasses</i> or no subclasses are specified[dependent] <i>classname</i> must be a valid name.[Smalltalk] <i>superclasses</i> (and therefore <i>subclasses</i>) must not be metaclasses.	<ul style="list-style-type: none">new class is added into the hierarchy with <i>superclasses</i> as superclasses and <i>subclasses</i> as subclasses.new class has name <i>classname</i>.<i>subclasses</i> inherit from new class and not any more from <i>superclasses</i>.
Remove Class (<i>class</i>)	<ul style="list-style-type: none"><i>class</i> must not have attributes or its attributes must not be referenced.<i>class</i> must not have methods or its methods must not be referenced.<i>class</i> must not be referenced.<i>class</i> must not implement abstract methods from its superclass hierarchy or must not have non-abstract subclasses.[Smalltalk] <i>class</i> must not be a metaclass[Smalltalk] the metaclass of <i>class</i> must not have referenced methods or classes.	<ul style="list-style-type: none"><i>class</i> is removed (including non-referenced attributes and methods).superclasses of <i>class</i> are now superclasses of its subclasses.[Smalltalk] corresponding metaclass is deleted as well.

Refactoring	precondition	postcondition
Rename Class (<i>class</i> , <i>new name</i>)	<ul style="list-style-type: none"> no class may exist with <i>new name</i> in the same scope. no global variable may exist with <i>new name</i> in the same scope. classes that refer to <i>class</i> must not already contain or inherited a variable with <i>new name</i>. [dependent] <i>new name</i> must be a valid class name. [Smalltalk] <i>class</i> must not be a metaclass. 	<ul style="list-style-type: none"> <i>class</i> has <i>new name</i>. all references (types, class method calls, superclass references) are updated with the new name. [Java] constructors are updated with the new name. [Java] casts to <i>class</i> have been updated [Smalltalk] the corresponding meta-class of <i>class</i> has been renamed as well.
Add Method (<i>name</i> , <i>class</i>)	<ul style="list-style-type: none"> no (inherited) method with signature derived from <i>name</i> may exist in <i>class</i>. [dependent] <i>name</i> must be a valid method name. 	<ul style="list-style-type: none"> <i>class</i> has a method called <i>name</i> with an empty body or is abstract if <i>class</i> represents a Java interface.
Remove Method (<i>method</i>)	<ul style="list-style-type: none"> <i>method</i> must not have candidate invocations unless method itself is the only candidate invoker. if <i>method</i> is abstract it must not have static references. 	<ul style="list-style-type: none"> <i>method</i> is removed from its containing class.
Rename Method (<i>method</i> , <i>new name</i>)	<ul style="list-style-type: none"> all superclasses of the class containing <i>method</i> as well as the subclass hierarchies of the highest superclasses that define a method with the same signature a <i>method</i>, must not already contain a method with a signature implied by <i>new name</i> and the parameters of <i>method</i>. the candidate invocations to the group of methods that need to be renamed, do not have candidates that are methods outside of this group. [dependent] <i>new name</i> must be a valid method name. [Java] when <i>method</i> is a constructor, the refactoring cannot be applied unless in the context of a Rename Class refactoring. 	<ul style="list-style-type: none"> <i>method</i> has <i>new name</i>. relevant methods in the inheritance hierarchy have <i>new name</i>. invocations of changed method are updated to <i>new name</i>.

Refactoring	precondition	postcondition
Pull Up Method (<i>method</i> , <i>superclass</i>)	<ul style="list-style-type: none"> • <i>method</i> must not be private. • <i>method</i> should not directly access attributes from its defining class. • <i>method</i> should not directly invoke methods from its defining class unless all those invocations have self/this as receiver and are either to methods that are also defined or inherited in the superclass or to itself. • <i>superclass</i> may not contain or inherit a non-abstract method with the same signature as <i>method</i>. • <i>method</i> cannot have super references to <i>superclass</i>. • [Java] <i>method</i> must not be a constructor. • [Java] non-abstract method cannot be pulled up to an interface. • [Smalltalk] <i>method</i> should not access methods from its metaclass. 	<ul style="list-style-type: none"> • <i>method</i> defined in <i>superclass</i>. • <i>method</i> not defined in original containing class.
Push Down Method (<i>method</i>)	<ul style="list-style-type: none"> • <i>method</i> must not be invoked in or through its defining class unless it only invokes itself on self/this. • At least one direct subclass of the defining class of <i>method</i> may not already contain a method with the same signature as <i>method</i>. • self/this accesses to attributes that are also defined in one or more of the direct subclasses, may not exist in <i>method</i>. • self/this invocations and accesses to private members of the containing class may not exist in <i>method</i>. • no super invocations of <i>method</i> may exist in the direct subclasses of the defining class. • super invocations to methods that are also defined in the defining class may not exist in <i>method</i>, except if the invoked method has the same signature as <i>method</i> itself. • super accesses to attributes that are also defined in the defining class may not exist in <i>method</i>. • subclasses cannot inherit non-abstract methods with the same signature as <i>method</i> from other superclass branches. • [Java] <i>method</i> must not be a constructor. 	<ul style="list-style-type: none"> • <i>method</i> not defined in original containing class. • <i>method</i> defined in subclasses of the containing class.

Refactoring	precondition	postcondition
Add Parameter (<i>name</i> , <i>method</i>)	<ul style="list-style-type: none"> <i>method</i> must not already have a parameter with <i>name</i>. <i>method</i> must not already have a local variable with <i>name</i>. containing class must not have an attribute with <i>name</i>. no classes or global variables with <i>name</i> may exist in the system. all superclasses of the class containing <i>method</i> as well as the subclass hierarchies of the highest superclasses that define a method with the same signature as <i>method</i>, must not already contain a method with a signature implied by adding a parameter with <i>name</i> to method. the candidate invocations to the group of methods that need to be renamed, do not have candidates that are methods outside of this group. [dependent] <i>name</i> must be a valid parameter name. 	<ul style="list-style-type: none"> <i>method</i> and all relevant methods in the inheritance hierarchy have an extra parameter with <i>name</i>. invocations of <i>method</i> are updated to invoke it with an extra parameter with a default value.
Remove Parameter (<i>parameter</i>)	<ul style="list-style-type: none"> <i>parameter</i> must not be referenced in the containing method or equivalent parameters in any overriding or overridden method. all superclasses of the class containing the method <i>parameter</i> belongs to, as well as the subclass hierarchies of the highest superclasses that define a method with the same signature that method, must not already contain a method with a signature implied by removing <i>parameter</i> from its method. the candidate invocations to the group of methods that need to be renamed, do not have candidates that are methods outside of this group. 	<ul style="list-style-type: none"> <i>method</i> and all relevant methods in the inheritance hierarchy have <i>parameter</i> removed. invocations of <i>method</i> are updated to invoke it without <i>parameter</i>.
Add Attribute (<i>name</i> , <i>class</i>)	<ul style="list-style-type: none"> the inheritance hierarchy of the containing class must not already contain an attribute with <i>name</i>. no global variable with <i>name</i> may exist. no class with <i>name</i> may exist. [dependent] <i>name</i> must be a valid attribute name. 	<ul style="list-style-type: none"> <i>class</i> has attribute named <i>name</i>.
Remove Attribute (<i>attribute</i>)	<ul style="list-style-type: none"> <i>attribute</i> must not be accessed. 	<ul style="list-style-type: none"> <i>attribute</i> is removed from its containing class.

Refactoring	precondition	postcondition
Rename Attribute (<i>attribute</i> , <i>new name</i>)	<ul style="list-style-type: none"> the inheritance hierarchy of the containing class must not already contain an attribute with <i>new name</i>. no global variable with <i>new name</i> may exist. no class with <i>new name</i> may exist. methods of the containing class and its subclasses that access <i>attribute</i> must not contain a local variable with <i>new name</i> already. [dependent] <i>name</i> must be a valid attribute name. 	<ul style="list-style-type: none"> <i>attribute</i> has name <i>new name</i>. all accesses to <i>attribute</i> are to use the new name.
Pull Up Attribute (<i>attribute</i> , <i>superclass</i>)	<ul style="list-style-type: none"> <i>superclass</i> must not contain an attribute with the same name as <i>attribute</i>. any attribute in the subclasses of <i>superclass</i> with the same name as <i>attribute</i> must have the same type as <i>attribute</i>. These attributes must not hide each other. pulling up <i>attribute</i> may not hide in <i>superclass</i> another attribute with the same name. pulling up <i>attribute</i> may not unhide any attribute with the same name from other superclass branches of the containing class. 	<ul style="list-style-type: none"> <i>superclass</i> contains <i>attribute</i>. all attributes in the subclasses of <i>superclass</i> with the same name and type as <i>attribute</i> have been removed.
Push Down Attribute (<i>attribute</i>)	<ul style="list-style-type: none"> <i>attribute</i> must not be accessed in or through its containing class. the direct subclasses of the containing class must not contain an attribute with the same name as <i>attribute</i>. 	<ul style="list-style-type: none"> <i>attribute</i> is removed from its containing class. all subclasses that need it (i.e. that have a reference to the attribute somewhere in its hierarchy) define an attribute with the same name and type as <i>attribute</i>.

APPENDIX B

The FAMIX 2.1 specification

This appendix describes the FAMIX metamodel version 2.1. It starts with an overview of the metamodel before describing the metamodel in detail.

2.1 Overview

Figure B.1 shows an overview of the core FAMIX metamodel. Section 2.2 describes all the shown elements in detail. In this section we introduce some information that is necessary for the rest of the metamodel definition, such as some basic data types, unique naming conventions and extraction levels.

2.1.1 Basic Data Types

Besides the usual primitive data types (String, Integer, Boolean, ...) we have a number of extra data types in our metamodel that are considered 'basic'. These are `Name`, `Qualifier` and `Index`:

Name vs. Qualifier

A `Name` is a string that bears semantics inside the metamodel, while a `Qualifier` is a string that gets its semantics from outside the metamodel. A `String` does not bear any semantics. For instance, a `uniqueName` may be used to refer to another object, hence bears semantics inside the metamodel. However, a `sourceAnchor` will store some information that must be interpreted by applications outside the metamodel, hence is a qualifier. Finally, a comment line is a string, since it does not bear any semantics understandable by a computer. In CDIF these types are simply represented by `Strings`, or `TextValues` if they are multi-valued (see section 2.3.1 for a description of multi-valued strings in CDIF).

Index

An `Index` represents a position in some sequence. Indices always have a base of 1. In CDIF this type is represented by an integer.

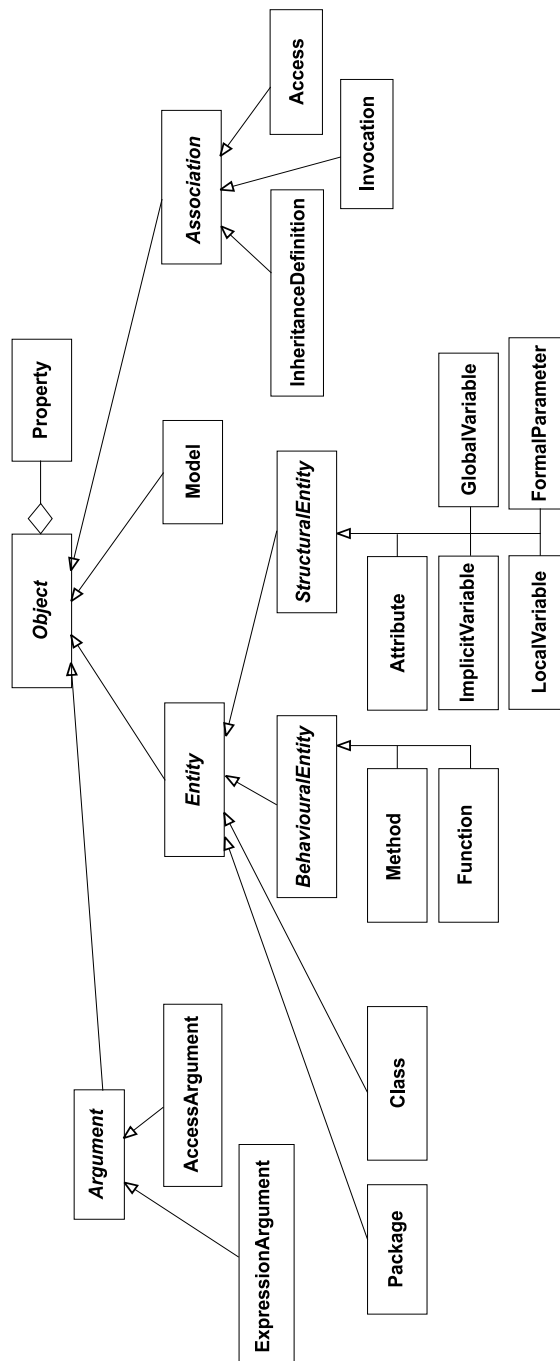


Figure B.1 The complete hierarchy of the FAMIX model

2.1.2 Unique Naming Conventions

The naming conventions used in the FAMIX metamodel is as much as possible compliant with UML [OMG99]. This means that the following rules apply:

1. **Scoping via packages.** Global entities, such as classes, functions, global variables and packages themselves receive a unique name by concatenating with the containing package name using "::" as a separator. They will typically look like "package::subpackage::classname".
2. **Naming of variables.** Variables, such as attributes, local variables, etc. receive a unique name by concatenating with the containing entity using a "." as a separator. They will typically look like "package::subpackage::classname.attributename",
"package::subpackage::classname.method().localvariablename".
3. **Naming of methods and functions.** Methods and functions distinguish themselves from variables because they have an parameter list. Therefore, they are named by concatenating their scope and their signature. For functions we follow the convention of package scoping, thus separate the scope and the signature via a "::". For methods we follow the convention of variable naming, thus separate the scope and the signature via a ".".

The signature of a method and a function contains the name of the method or function, followed by its parameter list surrounded by parentheses. The return type is not part of the signature. They will typically look like

"package::subpackage::functionname(para1,para2)",

"package::subpackage::classname.methodname(para1,para2)"

To achieve a normal form for signatures, parameter lists should not contain unnecessary spaces.

Thus

"functionname(para1,para2)"

instead of

"functionname(para1, para2)".

However, sometimes languages include keywords in their parameter list, and then spaces can not be avoided. For instance, the C++ const parameters will be represented like

"functionname(const para1,const para2)".

2.1.3 Level of Extraction

The core metamodel contains entities that not all parsers may provide. Next to that, some tools do not always need all of this information (e.g., a metrics tool might not need Invocation and Access, because many metrics can already be gathered from Class and Method alone). To allow focused models, we introduce the *level of extraction*.

Basically, the level of extraction is an integer, representing how much of the core metamodel is available in a model. The higher the number, the more information is available. The levels are set up in such a way that no information is available on a level that needs information from higher levels (for instance, Access is not usable if there are no Attribute's available). Next to that, it is possible that on the higher levels parts of the information are not necessary for a certain task, or simply not computable by a certain tool. Therefore it is allowed to only provide parts of the information (designated by the "+/-"). Table B.1 gives an overview of the levels of extraction.

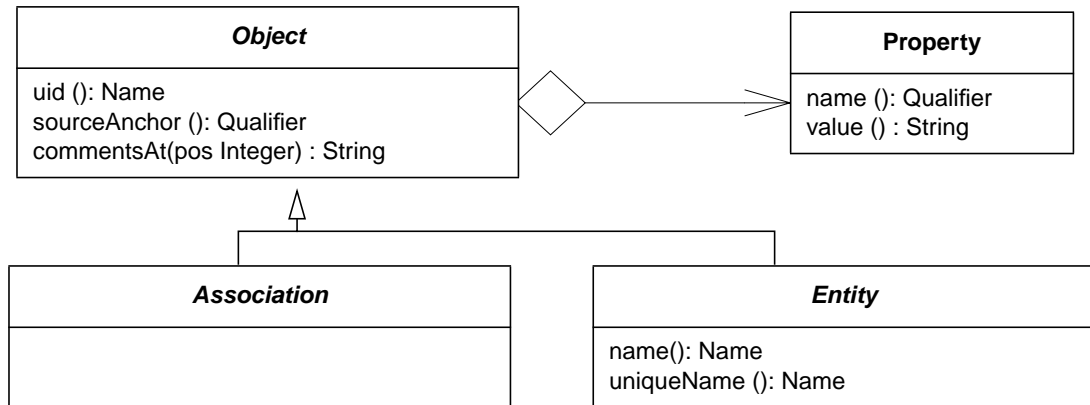
Level 1	Class, InheritanceDefinition, BehaviouralEntity (Method, Function) +/- Package Level 1 is the minimal information that parsers should be able to provide and corresponds with what is usually understood as the interface of a class.
Level 2	Level 1 +/- Attribute +/- Package
Level 3	Level 2 +/- Access +/- Invocation
Level 4	Level 3 +/- Argument +/- FormalParameter +/- LocalVariable +/- ImplicitVariable

Table B.1: Levels of Extraction

2.2 Definition of FAMIX

This part describes the various classes that together specify the FAMIX metamodel. The following subsections describe the different elements with their attributes, and give examples in the CDIF transfer format. Mandatory attributes must always be present. Optional attributes may be omitted. Some optional attributes have a default value.

2.2.1 The abstract part: Object, Entity and Association

**Figure B.2** The basic classes Object, Entity and Association

The classes **Object**, **Association**, **Entity** and **Property** capture the commonalities in the design of FAMIX. Furthermore, they provide the hooks to extend the core metamodel with new elements and, through the **Property** element, with the ability to annotate any **Object** in a model. This are the different classes in detail:

Object
uid (): Name sourceAnchor (): Qualifier commentsAt (pos Integer): String

Object is an abstract class without a superclass. The attributes of Object are:

- uid: Name; mandatory

Denotes an identifier that is unique for every element in a model. FAMIX does not impose a schema or format. It is recommended to use Universal Unique Identifiers (UUIDs) [OG97], which is a standard way of constructing identifiers that are unique over space and time. The resources to store and compute UUIDs, however, might clash with scalability needs (see also section 3.10).

- sourceAnchor: Qualifier; optional

Identifies the location in the source where the information is extracted. The exact format of the qualifier is dependent on the source of the information. Usually, it will be an anchor in a source file, in which case the following format should be used

```
file "<filespec>" start <start_index> end <end_index>
```

where <filespec> is a string holding the name of the source-file in an operating system dependent format (preferably a filename relative to some project directory). Note that filenames may contain spaces and double quotation marks. A double quotation mark in a filename should be escaped with a \". <start_index> and <end_index> are indices starting at 1 and holding the beginning respectively ending character position in the source file. Extra position indices or whole source anchors may be added to handle anchors in files that may need to be displayed with external editors. For instance, the line and column of the character (startline, startcol, endline, endcol). Or the negative offset counting from the end of the file instead of from the beginning (negstart, negend). In CDIF a basic source anchor looks as follows (delimited with a '|', see section 2.3.1 for a description of multi-valued strings in CDIF):

```
(sourceAnchor #[file "factory.h" start 260 end 653|]#)
```

- comments: 0..N String; optional

Entities and associations may own a number of comments, where developers and tools store textual information about the object. In CDIF we represent this with a CDIF TextValue, where the blocks are delimited by a '|' (see section 2.3.1 for a description of multi-valued strings in CDIF):

```
(comments #[commentLines|]#,[commentLines|]#,...)
```

Property
name (): Qualifier value (): String belongsToUid (): Name

Entities and associations may own a number of properties where extensions of the core metamodel may be stored. Property has the following attributes:

- name: Qualifier; mandatory

Is a string that identifies a Property within an Object. Thus, the name should be unique for all properties of a single Object.

- `value: String; mandatory`
Contains the value of the property. The meaning of the value is not defined within this metamodel.
- `belongsToUid: Name; mandatory`
Contains the uid that identifies the Object this Property is a property of.

CDIF example showing a class `Widget` with a `Property` containing the value 5 for the number-of-methods metric. They are related by the `belongsToUid` attribute.

```
(Class ENT001
  (name "Widget")
  (uid "c842bf06-d202-0000-0282-5c410d0000")
  ....
)

(Property PR005
  (name "metric_NOM")
  (value #[5]#)
  (belongsToUid "c842bf06-d202-0000-0282-5c410d0000")
)
```

Entity
name (): Qualifier
uniqueName (): Name

To enable a global referencing scheme based on names, the key classes in the metamodel should respect the minimal interface of `Entity`. `Entity` is an abstract class inheriting from `Object`. Besides inherited attributes, it has the following attributes:

- `name: Qualifier; mandatory`
Is a string that provides some human readable reference to an entity.
- `uniqueName: Name; mandatory`
Is a string that is computed based on the name of the entity. Each class of entities must define its specific formula. The `uniqueName` serves as an external reference to that entity and must be unique for all entities in a model.

Association

`Association` is an empty common superclass for all associations in the metamodel. `Association` is an abstract class inheriting from `Object`. It defines no new attributes itself.

2.2.2 Model

Model
exporterName (): String exporterVersion (): String exporterDate (): String exporterTime (): String publisherName (): String parsedSystemName (): String extractionLevel (): String sourceLanguage (): String sourceDialect (): String

Figure B.3 Model

A Model represents information about the particular system being modelled. Extractors must ensure that there is only instance of a Model in a model. Information exchange standards often provide means to exchange similar information in special sections of the exchange stream. However, having an explicit Model element in FAMIX allows us to transfer this information independent of the chosen exchange format.

Model is a concrete class inheriting from Object. Besides inherited attributes, it has the following attributes:

- `exporterName: String; mandatory`
Represents the name of the tool that generated the information.
- `exporterVersion: String; mandatory`
Represents the version of the tool that generated the information.
- `exporterDate: String; mandatory`
Represents the date the information was generated.
- `exporterTime: String; mandatory`
Represents the time of the day the information was generated.
- `publisherName: String; mandatory`
Represents the name of the person that generated the information. Provide an empty string if this information is not known.
- `parsedSystemName: String; optional`
Represents the name of the system where the information was extracted from.
- `extractionLevel: String; mandatory`
Represents the level of extraction used when generating the information (see Table on page 139).
- `sourceLanguage: String; mandatory`
Identifies the implementation language of the parsed source code, for instance 'C++', 'Ada', 'Java', or 'Smalltalk'.
- `sourceDialect: String; optional`
Identifies the dialect of the implementation language of the parsed source code. The exact contents of the string is a language-dependent issue, e.g., 'Borland' or 'ANSI' for C++.

CDIF example of a Model instance for a WidgetLibrary system implemented in Java:

```
(Model FM0
  (exporterName "sniff2famix")
  (exporterVersion "2.0")
  (exporterDate "1999/10/19")
  (exporterTime "00.00.01")
  (publisherName "Sander Tichelaar")
  (parsedSystemName "WidgetLibrary")
  (extractionLevel "3")
  (sourceLanguage "Java")
  (sourceDialect -NULL-)
)
```

2.2.3 Package

Package
belongsToPackage (): Name

Figure B.4 Package

A Package represents a named sub-unit of a source code model, for example namespaces in C++, and packages in Java. What exactly constitutes such a sub-unit is a language-dependent issue. Packages and other entities can only belong to zero or one Package, and their name must be unique within their containing Package.

Package is a concrete class inheriting from Entity. Besides inherited attributes, it has the following attributes:

- belongsToPackage: name; optional

Is the unique name of the package containing this package. A null value represents the fact that there is no containing package.

Formula for uniqueName (see also section 2.1.2 “Unique Naming Conventions” on page 139):

```
if isNull (belongsToPackage(package)) then
  uniqueName (package) = name (package)
else
  uniqueName (package) = belongsToPackage (package) + "::" + name (package)
```

CDIF example of a package gui:

```
(Package FM1
  (name "gui")
  (belongsToPackage -NULL-)
  (uniqueName "gui")
)
```

2.2.4 Class

Class
isAbstract (): Boolean
belongsToPackage (): Name

Figure B.5 Class

A Class represents the definition of a class in source code. What exactly constitutes such a definition is a language-dependent issue.

Class is a concrete class inheriting from Entity. Besides inherited attributes, it has the following attributes:

- `isAbstract: Boolean; optional`
Is a predicate telling whether the class is declared abstract. Abstract classes are important in object-oriented modelling, but how they are recognised in source code is a language-dependent issue.
- `belongsToPackage: Name; optional`
Is the unique name of the package defining the scope of the class. A null `belongsToPackage` is allowed, it means that the class has global scope. The `belongsToPackage` concatenated with the name of the class must provide a unique name for that class within a model.

Formula for `uniqueName` (see also section 2.1.2 “Unique Naming Conventions” on page 139):

```
if isNull (belongsToPackage (class)) then
  uniqueName (class) = name (class)
else
  uniqueName (class) = belongsToPackage (class) + "::" + name (class)
```

CDIF example of a non-abstract class `Widget` in package `gui` (note the difference between `name` and `uniqueName`):

```
(Class FM1
  (name "Widget")
  (uniqueName "gui::Widget")
  (isAbstract -FALSE-)
  (sourceAnchor #[file "factory.h" start 260 end 653|]#)
)
```

2.2.5 BehaviouralEntity Hierarchy

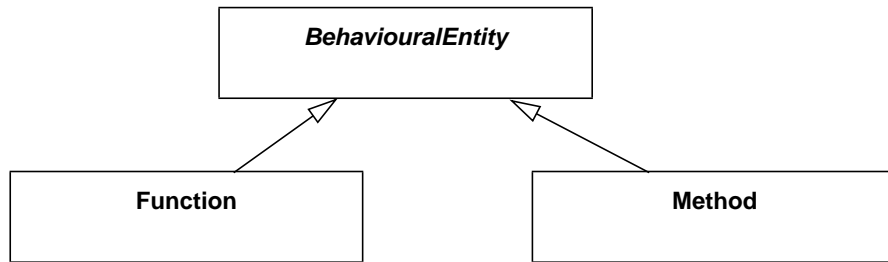


Figure B.6 The BehaviouralEntity hierarchy

The entities that define behaviour in our metamodel are all subclasses of BehaviouralEntity.

2.2.6 BehaviouralEntity

BehaviouralEntity
accessControlQualifier (): Qualifier signature (): Qualifier isPureAccessor (): Boolean declaredReturnType (): Qualifier declaredReturnClass (): Name

Figure B.7 BehaviouralEntity

A BehaviouralEntity represents the definition in source code of a behavioural abstraction, i.e., an abstraction that denotes an action rather than a part of the state. Subclasses of this class represent different mechanisms for defining such an entity.

BehaviouralEntity is an abstract class inheriting from Entity. Besides inherited attributes, it has the following attributes:

- `accessControlQualifier: Qualifier; optional`
Is a string with a language-dependent interpretation that defines who is allowed to invoke it (for instance, 'public', 'private').
- `signature: Qualifier; mandatory`
Is a string that allows to uniquely distinguish a behavioural entity. This is necessary, because object-oriented languages exist that allow to overload methods, so that the same method name may be associated with different parameter lists, each with its own method body (e.g., C++, Java). The way a signature string is composed is language-dependent, but it should at least include the name of the method. The UML [OMG99] compliant notation will be used, which will typically look like (see also section 2.1.2 “Unique Naming Conventions” on page 139)
“package::subpackage::classname.methodname(parameters)”.
- `isPureAccessor: Boolean; optional`
Is a predicate telling whether the behavioural entity is a pure accessor. There are two kinds of accessors, a reader accessor and a writer accessor. A pure reader accessor is an entity with a single receiver parameter, only returning the value of an attribute of the class the method is defined on. A

pure writer accessor is a method with one receiver parameter and one value parameter, only storing the value inside the attribute of a class. How accessor methods are recognised in source code is a language-dependent issue.

- `declaredReturnType`: `Qualifier`; optional

Is a qualifier that refers to the declared type of the returned object. Typically this will be a class, a pointer or a primitive type (e.g., `int` in Java). `declaredReturnType` is null if the return type is not known or the empty string (i.e., `""`) if the `BehaviourialEntity` does not have a return type¹.

Note that the `declaredReturnType` does *not* have meaning in a model (although it obviously has meaning in the context of the specific implementation language). We need a language-dependent interpretation to link a type name to a class name, because in most object-oriented languages, types are not always equivalent to classes. How the declared return type can be recognised in source code and how the return type matches to a class or another type are language-dependent issues. The declared return class is stored in the `declaredReturnClass` attribute (see below for the definition of `declaredReturnClass` and see section 4.4.1 for an in-depth discussion).

- `declaredReturnClass`: `Name`; optional

Indicates the unique name of the class that is implicit in the `declaredReturnType`, with the goal of capturing the dependency to the corresponding `Class` instance in a model. The `declaredReturnClass` always contains the name of a class, or null if it is unknown if there is an implicit class in the `declaredReturnType`, and the empty string (i.e., `""`) if it is known that there is no implicit class in the `declaredReturnType`. What exactly is the relationship between `declaredReturnClass` and `declaredReturnType` is a language-dependent issue.

2.2.7 Method

Method
<code>belongsToClass ()</code> : <code>Name</code> <code>hasClassScope ()</code> : <code>Boolean</code> <code>isAbstract ()</code> : <code>Boolean</code> <code>isConstructor ()</code> : <code>Boolean</code>

Figure B.8 Method

A `Method` represents the definition in source code of an aspect of the behaviour of a class. What exactly constitutes such a definition is a language-dependent issue.

`Method` is a concrete class inheriting from `BehaviouralEntity`. Besides inherited attributes, it has the following attributes:

- `belongsToClass`: `Name`; mandatory

Is a name referring to the class owning the method. It uses the `uniqueName` of the class as a reference.

1. In C++ the fact that a function does not have a return type is denoted by the keyword `void`. We do not use `void` in FAMIX to denote ‘no type’, because this causes problems for languages where it is possible to define a class called “void”, like for instance Smalltalk and Ada. Note that this is consistent with UML 1.3 [OMG99].

- `hasClassScope: Boolean; optional`

Is a predicate telling whether the method has class scope (i.e., invoked on the class) or instance scope (i.e., invoked on an instance of that class). For example, static methods in C++ and Java have their `hasClassScope` attribute set to true.

- `isAbstract: Boolean; optional`

Is a predicate telling whether the method is declared abstract, i.e., when non-abstract subclasses are forced to provide an implementation for this method. Abstract methods are important in object-oriented modeling, but how they are recognised in source code is a language-dependent issue.

- `isConstructor: Boolean; optional`

Is a predicate telling whether the method is a constructor. A constructor is a method that creates an (initialised) instance of the class it is defined on. Thus a method that creates an instance of another class is not considered a constructor. How constructor methods are recognised in source code is a language-dependent issue.

Formula for `uniqueName` (see also section 2.1.2 “Unique Naming Conventions” on page 139):

```
uniqueName (method) = belongsToClass (method) + "." + signature (method)
```

CDIF example (constructor for a class `Widget`. This method has no return type and therefore also no ‘return class’, hence both attributes are empty):

```
(Method FM2
  (name "Widget")
  (belongsToClass "gui::Widget")
  (sourceAnchor #[file "factory.h" start 321 end 326|]#)
  (accessControlQualifier "public")
  (hasClassScope -FALSE-)
  (signature "Widget()")
  (isAbstract -FALSE-)
  (declaredReturnType "")
  (declaredReturnClass "")
  (uniqueName "gui::Widget.Widget()")
)
```

2.2.8 Function

Function
<code>belongsToPackage ()</code> : Name

Figure B.9 Function

A Function represents the definition in source code of an aspect of global behaviour. What exactly constitutes such a definition is a language-dependent issue.

Function is a concrete class inheriting from BehaviouralEntity. Besides inherited attributes, it has the following attributes:

- belongsToPackage: Name; optional

Is the unique name of the package defining the scope of the function. A null belongsToPackage is allowed, meaning that the function has global scope. The belongsToPackage concatenated with the name of the function must provide a unique name for that class within a model.

Formula for uniqueName (see also section 2.1.2 “Unique Naming Conventions” on page 139):

```
if isNull (belongsToPackage (function)) then
    uniqueName (function) = name (function)
else
    uniqueName (function) = belongsToPackage (function) + "::" + name (function)
```

CDIF example (of a global function testFactory without arguments and return type in subpackage test of package widgetfactory):

```
(Function FM2
  (name "testFactory")
  (sourceAnchor #[file "factory.h" start 321 end 326|]#)
  (accessControlQualifier "public")
  (signature "testFactory()")
  (belongsToPackage "widgetfactory::test")
  (declaredReturnType "")
  (declaredReturnClass "")
  (uniqueName "widgetfactory::test::testFactory()")
)
```

2.2.9 StructuralEntity Hierarchy

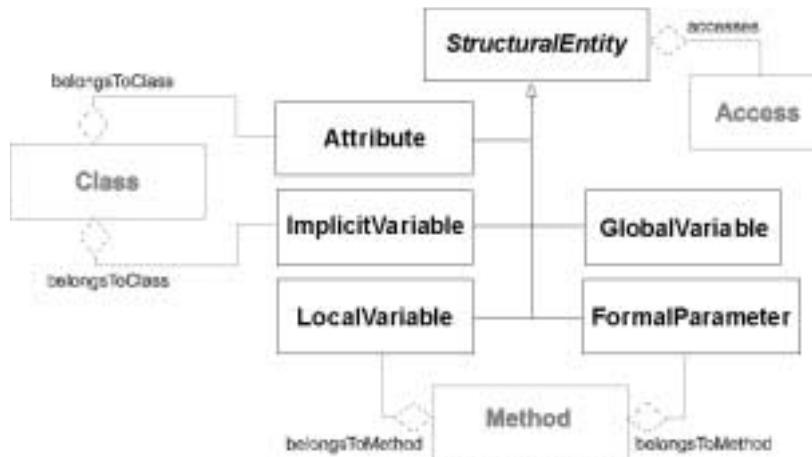


Figure B.10 The StructuralEntity hierarchy

All possible variable definitions are subclasses of the class StructuralEntity. StructuralEntity itself participates in the Access association.

2.2.10 StructuralEntity

StructuralEntity
declaredType (): Qualifier
declaredClass (): Name

Figure B.11 StructuralEntity

A StructuralEntity represents the definition in source code of a structural entity, i.e., it denotes an aspect of the state of a system. The different kinds of structural entities mainly differ in lifetime: some have the same lifetime as the entity they belong to, e.g., an attribute and a class, some have a lifetime that is the same as the whole system, e.g., a global variable. Subclasses of this class represent different mechanisms for defining such an entity.

StructuralEntity is an abstract class inheriting from Entity. Besides inherited attributes, it has the following attributes:

- `declaredType: Qualifier; optional`

Is a qualifier that refers to the declared type of the structural entity. Typically this will be a class, a pointer or a primitive type (e.g., `int` in Java). `declaredType` is null if the return type is not known or the empty string (i.e., `""`) if the BehaviourialEntity does not have a return type¹.

Note that the `declaredType` does *not* have meaning in a model (although it obviously has meaning in the context of the specific implementation language). We need a language-dependent interpretation to link a type name to a class name, because in most object-oriented languages, types are not always equivalent to classes. How the declared type can be recognised in source code and how the return type matches to a class or another type are language-dependent issues. The declared class is stored in the `declaredClass` attribute (see below for the definition of `declaredClass` and see section 4.4.1 for an in-depth discussion).

- `declaredClass: Name; optional`

Indicates the unique name of the class that is implicit in the `declaredType`, with the goal of capturing the dependency to the corresponding Class instance in a model. The `declaredClass` contains the name of a class, or null if it is unknown if there is an implicit class in the `declaredType`, and the empty string (i.e., `""`) if it is known that there is no implicit class in the `declaredType`. What exactly is the relationship between `declaredClass` and `declaredType` is a language-dependent issue.

2.2.11 Attribute

Attribute
belongsToClass (): Name
accessControlQualifier (): Qualifier
hasClassScope (): Boolean

Figure B.12 Attribute

1. In C++ the fact that a function does not have a return type is denoted by the keyword `void`. We do not use `void` in FAMIX to denote ‘no type’, because this causes problems for languages where it is possible to define a class called “void”, like for instance Smalltalk and Ada. Note that this is consistent with UML 1.3 [OMG99].

An Attribute represents the definition in source code of an aspect of the state of a class. What exactly constitutes such a definition is a language-dependent issue.

Attribute is a concrete class inheriting from StructuralEntity. Besides inherited attributes, it has the following attributes:

- `belongsToClass: Name; mandatory`
Is a name referring to the class owning the attribute. It uses the `uniqueName` of the class as a reference.
- `accessControlQualifier: Qualifier; optional`
Is a string with a language-dependent interpretation that defines who is allowed to access it (for instance, 'public', 'private').
- `hasClassScope: Boolean; optional`
Is a predicate telling whether the attribute has class scope (i.e., a shared memory location for all instances of the class) or instance scope (i.e., separate memory location for each instance of the class). For example, static attributes in C++ and Java have a `hasClassScope` attribute set to true.

Formula for `uniqueName` (see also section 2.1.2 “Unique Naming Conventions” on page 139):

```
uniqueName (attribute) = belongsToClass (attribute) + "." + name (attribute)
```

CDIF example of a private attribute `wTop` in class `Widget`:

```
(Attribute FM22
 (name "wTop")
 (belongsToClass "gui::Widget")
 (sourceAnchor #[file "factory.h" start 281 end 284|]#)
 (declaredType "int")
 (declaredClass "")
 (accessControlQualifier "private")
 (uniqueName "gui::Widget.wTop")
)
```

2.2.12 GlobalVariable

GlobalVariable
<code>belongsToPackage ()</code> : Name

Figure B.13 GlobalVariable

A GlobalVariable represents the definition in source code of a variable with a lifetime equal to the lifetime of a running system, and which is globally accessible. What exactly constitutes such a definition is a language-dependent issue.

GlobalVariable is a concrete class inheriting from StructuralEntity. Besides inherited attributes, it has the following attributes:

- `belongsToPackage: Name; optional`
Is the unique name of the package defining the scope of the variable. A null `belongsToPackage` is allowed, it means that the variable has global scope. The `belongsToPackage` concatenated with the name of the variable must provide a unique name for that class within a model.

Formula for `uniqueName` (the second branch of the if statement is necessary because a global variable can have package scope) (see also section 2.1.2 “Unique Naming Conventions” on page 139):

```
if isNull (belongsToPackage (globalVariable)) then
    uniqueName (globalVariable) = name (globalVariable)
else
    uniqueName (globalVariable) = belongsToPackage (globalVariable)
    + "::" + name (globalVariable)
```

CDIF example of a global variable called ‘TRUE’ with type ‘int’:

```
(GlobalVariable FM23
  (name "TRUE")
  (sourceAnchor #[file "factory.h" start 287 end 291|]#)
  (declaredType "int")
  (declaredClass "")
  (accessControlQualifier "public")
  (uniqueName "TRUE")
)
```

2.2.13 ImplicitVariable

ImplicitVariable
belongsToContext (): Qualifier

Figure B.14 ImplicitVariable

An `ImplicitVariable` represents the definition in source code of context dependent reference to a memory location (i.e., `this` and `super` in C++ and Java, `self` and `super` in Smalltalk). What exactly constitutes such a definition is a language-dependent issue.

`ImplicitVariable` is a concrete class inheriting from `StructuralEntity`. Besides inherited attributes, it has the following attributes:

- `belongsToContext: Qualifier; optional`
Is a string with a language-dependent interpretation, that defines a possible scope of the variable. A null `belongsToContext` is allowed, it means that the variable has global scope. The `belongsToContext` concatenated with the name of the variable must provide a unique name for that variable within a model.

Formula for `uniqueName` (see also section 2.1.2 “Unique Naming Conventions” on page 139):

```
if isNull (belongsToContext (implicitVariable)) then
    uniqueName (implicitVariable) = name (implicitVariable)
else
    uniqueName (implicitVariable) = belongsToContext (implicitVariable)
    + "." + name (implicitVariable)
```

Example of an implicit variable `super`:

```
MotifWidget.print () {
    super.print();
    System.out.print("Motif");
}
```

In CDIF this gives the following result:

```
(ImplicitVariable FM77
  (name "super")
  (declaredType "gui::Widget")
  (declaredClass "gui::Widget")
  (belongsToContext "gui::MotifWidget")
  (uniqueName "gui::MotifWidget.super")
)
```

2.2.14 LocalVariable

LocalVariable
belongsToBehaviour (): Name

Figure B.15 LocalVariable

A LocalVariable represents the definition in source code of a variable defined locally to a behavioural entity. What exactly constitutes such a definition is a language-dependent issue.

LocalVariable is a concrete class inheriting from StructuralEntity. Besides inherited attributes, it has the following attributes:

- belongsToBehaviour: Name; mandatory

Is a name referring to the BehaviouralEntity owning the variable. It uses the uniqueName of this entity as a reference.

Formula for uniqueName (see also section 2.1.2 “Unique Naming Conventions” on page 139):

```
uniqueName (localVar) = belongsToBehaviour (localVar) + "." + name (localVar)
```

Example of a local variable position_:

```
Class ScrollBar {
  computePosition(int x,int y,int width,int height) {
    int position_;
    . . .
  }
}
```

In CDIF:

```
(LocalVariable FM76
  (name "position_")
  (sourceAnchor #[file "factory.h" start 85 end 89]|#)
  (declaredType "int")
  (declaredClass "")
  (belongsToBehaviour "ScrollBar.computePosition(int,int,int,int)")
  (uniqueName "gui::ScrollBar.computePosition(int,int,int,int).position_")
)
```

2.2.15 FormalParameter

FormalParameter
belongsToBehaviour (): Name
position (): Index

Figure B.16 FormalParameter

A FormalParameter represents the definition in source code of a formal parameter, i.e., the declaration of what a behavioural entity expects as an argument. What exactly constitutes such a definition is a language-dependent issue.

FormalParameter is a concrete class inheriting from StructuralEntity. Besides inherited attributes, it has the following attributes:

- `belongsToBehaviour: Name; mandatory`
Is a name referring to the BehaviouralEntity owning the variable. It uses the `uniqueName` of this entity as a reference.
- `position: Index; mandatory`
Indicates the position of the parameter in the list of parameters. Language extensions should specify what the position of a parameter is and this should be consistent the `position` attribute of Argument (see page 158).

Formula for `uniqueName` (see also section 2.1.2 “Unique Naming Conventions” on page 139):

```
uniqueName (formalPar) = belongsToBehaviour (formalPar) + "." + name (formalPar)
```

Example (`w` is the formal parameter):

```
Window::addWidget(Widget& w) { ..... };
```

In CDIF:

```
(FormalParameter FM41
  (name "w")
  (declaredType "gui::Widget&")
  (declaredClass "gui::Widget")
  (belongsToBehaviour "gui::Window.addWidget(Widget&)")
  (position 1)
  (uniqueName "gui::Window.addWidget(Widget&).w")
)
```

2.2.16 InheritanceDefinition

InheritanceDefinition
subclass (): Name
superclass (): Name
accessControlQualifier (): Qualifier
index (): Index

Figure B.17 InheritanceDefinition

An `InheritanceDefinition` represents the definition in source code of an inheritance association between two classes. One class then plays the role of the superclass, the other plays the role of the subclass. What exactly constitutes such a definition is a language-dependent issue.

`InheritanceDefinition` is a concrete class inheriting from `Association`. Besides inherited attributes, it has the following attributes:

- `subclass: Name; mandatory`
Is a unique name referring to the class that inherits. It uses the `uniqueName` of the class as a reference.
- `superclass: Name; mandatory`
Is a unique name referring to the class that is inherited from. It uses the `uniqueName` of the class as a reference.
- `accessControlQualifier: Qualifier; optional`
Is a string with a language-dependent interpretation, that defines how subclasses access their superclasses (for instance, 'public', 'private').
- `index: Index; optional`
In languages with multiple inheritance, this is the position of the superclass in the list of superclasses of one subclass. The information is of interest for object-oriented languages with multiple inheritance that resolve name collisions via the order of the superclasses (e.g., CLOS). For most languages the index does not have any semantics and the attribute will have a null value.

CDIF example of an inheritance relationship between `Scrollbar` and its superclass `Widget`:

```
(InheritanceDefinition FM27
  (subclass "gui::ScrollBar")
  (superclass "gui::Widget")
  (accessControlQualifier "public")
  (index 1)
)
```

2.2.17 Access

Access
<code>accesses (): Name</code> <code>accessedIn (): Name</code> <code>isAccessLValue (): Boolean</code> <code>hasArgument (): Name</code>

Figure B.18 Access

An `Access` represents the definition in source code of a `BehaviouralEntity` accessing a `StructuralEntity`. Depending on the level of extraction (see Table on page 139), that `StructuralEntity` may be an attribute, a local variable, an argument, a global variable.... What exactly constitutes such a definition is a language-dependent issue. However, when the same structural entity is accessed more than once in a method body, then parsers should generate a separate access-association for each occurrence.

Access is a concrete class inheriting from Association. Besides inherited attributes, it has the following attributes:

- `accesses: Name; mandatory`
Is a unique name referring to the variable being accessed. It uses the `uniqueName` of the variable as a reference.
- `accessedIn: Name; mandatory`
Is a unique name referring to the method doing the access. It uses the `uniqueName` of the method as a reference.
- `isAccessLValue: Boolean; optional`
Is a predicate telling whether the value was accessed as Lvalue, i.e., a location value or a value on the left side of an assignment. When the predicate is true, the memory location denoted by the variable might change its value; false means that the contents of the memory location is read; null means that it is unknown.
Note that LValue is the inverse of RValue.
- `hasArguments: 0 .. N Name; optional`
An Access can have arguments. Typically this will be one, namely the receiving, argument. For instance, in the case of `x.a`, `x` is the receiving argument of the access of `a`. The `hasArgument` attribute denotes the uid of the argument.

Example of the method `print()` accessing `wTop` (both defined in class `Widget`):

```
virtual print () { cout << "top of widget " << wTop; };
```

In CDIF:

```
(Access FM18
  (accesses "gui:Widget.wTop")
  (accessedIn "gui:Widget.print()")
  (isAccessLValue -FALSE-)
)
```

2.2.18 Invocation

Invocation
<code>invokedBy (): Name</code> <code>invokes (): Qualifier</code> <code>base (): Name</code> <code>candidatesAt (pos Integer): Name</code>

Figure B.19 Invocation

An Invocation represents the definition in source code of a BehaviouralEntity invoking another BehaviouralEntity. What exactly constitutes such a definition is a language-dependent issue. However, when the same behavioural entity is invoked more than once in a method body, then parsers should generate a separate invocation-association for each occurrence.

It is important to note that due to polymorphism, there exists at parse time a one-to-many relationship between the invocation and the actual entity invoked: a method, for instance, might be defined on a certain class, but at runtime actually invoked on an instance of a subclass of this class. This explains the presence of the `base` attribute and the `candidates` aggregation.

Invocation is a concrete class inheriting from Association. Besides inherited attributes, it has the following attributes:

- `invokedBy: Name; mandatory`
Is a unique name referring to the BehaviouralEntity doing the invocation. It uses the `uniqueName` of the entity as a reference.
- `invokes: Qualifier; mandatory`
Is a qualifier holding the signature of the BehaviouralEntity invoked. Due to polymorphism, the signature of the invoked BehaviouralEntity is not enough to assess which BehaviouralEntity is actually invoked. Further analysis based on the arguments is necessary. Concatenated with the `base` attribute this attribute constitutes the unique name of a behavioural entity.
- `base: Name; optional`
Is the unique name of the entity where the invoked entity is defined on. Null means unknown and an empty string means the attribute has no base (the invoked entity may be a global function). Together with the `invokes` attribute, this attribute constitutes the unique name of a behavioural entity.
- `candidates: 0 .. N Name; optional`
Is a multi-valued attribute holding a number of names of BehaviouralEntities. Each name refers to a BehaviouralEntity that may be the actual one invoked at run-time. See section 2.3.1 for a description of multi-valued strings in CDIF.
- `hasArguments: 0 .. N Name; optional`
An Invocation has arguments. The `hasArgument` attribute denotes the uids of the arguments.

CDIF example. The method `widget.print()` is invoked according to the source code. The actual method invoked at runtime, however, could be the `print()` method of one of the subclasses `MotifWidget` or `SwingWidget`:

```
(Invocation FM35
  (invokedBy "gui::ScrollBar.print()")
  (invokes "print()")
  (base "gui::Widget")
  (candidates#[gui::Widget.print()|]#,
    #[motif::MotifWidget.print()|]#,
    #[javax::swing::SwingWidget.print()|]#)
)
```

2.2.19 Argument Hierarchy

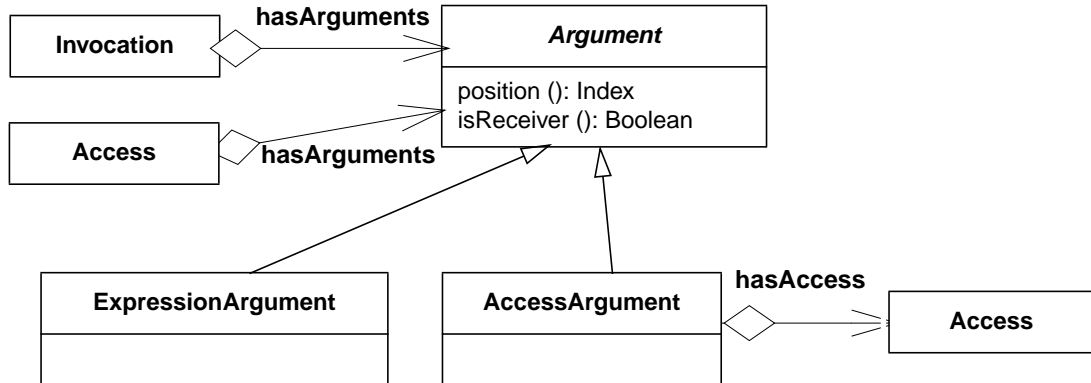


Figure B.20 Argument Hierarchy

An *Argument* represents the passing of an argument when invoking a *BehaviouralEntity* or accessing a *StructuralEntity*. What exactly constitutes such a definition is a language-dependent issue.

Argument is an abstract class inheriting from *Object*. Besides inherited attributes, *Argument* has the following attributes:

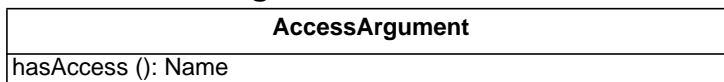
- `position: Index; mandatory`
The position of the argument in the list of arguments. Language extensions should specify what the position of a argument is and this should be consistent with the `position` attribute of *FormalParameter* (see section 2.2.15 on page 154).
- `isReceiver: Boolean; mandatory`
Is a predicate telling whether this argument plays the role of the receiver in the containing invocation. Knowing which argument plays the role of the receiver may help resolving polymorphic invocations.

2.2.20 ExpressionArgument



An *ExpressionArgument* models an argument that is a complex expression. This expression is not modelled in further detail (at least in the context of arguments; any access that is part of the expression should be modelled anyway). *ExpressionArgument* is a concrete subclass of *Argument*. It does not define any new attributes itself.

2.2.21 AccessArgument



An *AccessArgument* models an argument that is a reference to a *StructuralEntity*.

AccessArgument is a concrete subclass of Argument. Besides inherited attributes, it has the following attributes:

- `hasAccess`: Name; mandatory

Denotes the unique identifier (uid) of the Access instance that models the access of by the argument of a StructuralEntity.

Example of a method `print()` with two method invocations and their arguments. Note that the first call has one argument (namely `super`) and the second call has two (namely `System.out` and `"Motif"`):

```
MotifWidget.print () {
    super.print(a+3);
}
```

In CDIF:

```
#| FM90 expresses the access of the super implicit variable |#
(Access FM90
  (uid "c842bf06-d202-0000-0282-5c410d00000")
  (accesses "gui::MotifWidget.super")
  (accessedIn "gui::MotifWidget.print()")
)

#| FM91 expresses the passing of super as an argument to print |#
(AccessArgument FM91
  (uid "c842bf06-d202-0000-0282-5c410d000001")
  (position 1)
  (isReceiver -TRUE-)
  (hasAccess "c842bf06-d202-0000-0282-5c410d00000")
)

#| FM92 expresses existence of the a+3 expression |#
(ExpressionArgument FM92
  (uid "c842bf06-d202-0000-0282-5c410d000002")
  (position 2)
  (isReceiver -FALSE-)
)

#| FM101 expresses the invocation of print with argument super.
  Note that gui::Widget is the only candidate of the invocation. |#
(Invocation FM101
  (uid "c842bf06-d202-0000-0282-5c410d000002")
  (invokedBy "gui::MotifWidget.print()")
  (invokes "print()")
  (base "gui::Widget")
  (candidates #[gui::Widget|]#)
  (hasArguments #[ "c842bf06-d202-0000-0282-5c410d000001" |]#)
)
```

2.3 Miscellaneous

This section contains some miscellaneous topics.

2.3.1 CDIF Multi-valued String Attributes

CDIF is one of the standard exchange formats we use to transfer FAMIX-based models. In this appendix it is mainly used as a means to give examples for the different elements of the metamodel. One of the problems CDIF poses is that it does not provide multi-valued string attributes. We need those to deal with many-to-1 relationships (e.g., the `candidates` attribute of `Invocation`). Indeed, using the chunk format we encode relationships through unique names stored in attributes. However, using a string attribute to encode a relationship only allows for 1-to-many relationships.

CDIF provides `IntegerList` and `PointList` in its set of basic data types, thus — in principle — CDIF permits the use of multi-valued attributes. Unfortunately, there is no basic data type that copes with multi-valued strings. Yet, the `CDIF TextValue` data type comes near, thus in some rare occasions we interpret `TextValue` as a multi-valued text attribute.

In the original CDIF standard, a `TextValue` denotes a set of characters which is divided into blocks with a maximum of 1024 characters. The beginning of each block is marked by “#” while the end is marked by “]#”. The actual value of the text is the concatenation of the blocks. To represent a multi-valued string attribute with a `TextValue`, we interpret each block in a `TextValue` as a separate string. Also, we require that each one of those strings must append a special delimiter character (which is “[”) to its end so that the original multi-valued strings can be retrieved from the concatenated blocks. In the (unlikely) situation that a “[” appears in a string value it should be escaped with “\”. Thus we get

```
(Invocation FM35
  (invokedBy "ScrollBar.print()")
  (invokes "print()")
  (candidates#[Widget.print()|]#,
    #[MotifWidget.print()|]#,
    #[SwingWidget.print()|]#)
)
```

instead of (using CDIF relationships):

```
(Invocation FM35
  (invokedBy "ScrollBar.print()")
  (invokes "print()")
)
(Candidate FM45
  (value "Widget.print()")
)
(Candidate FM46
  (value "MotifWidget.print()")
)
(Candidate FM47
  (value "SwingWidget.print()")
)
(Invocation.HasCandidate.Candidate FM87 FM35 FM45)
```

APPENDIX C

Smalltalk Extension to FAMIX

This appendix describes the Smalltalk extension to FAMIX. Before discussing the specific interpretations and extensions, it gives a short overview of how FAMIX can be extended.

3.1 Extending FAMIX

The basic FAMIX model is modified in three different ways to handle Smalltalk sources:

- New classes are added to the common exchange model to model entities and associations unique to Smalltalk. These classes are marked as new entities respectively associations.
- New attributes are added to existing classes of the basic FAMIX model. In this case the class is marked “extended” and only the new and modified (see below) attributes are listed in the definition of the modified class.
- The definition of attributes of existing classes are modified or are made more specific. In this case the corresponding class is marked “interpreted” and the interpreted attributes are listed in the definition of the modified class. To discriminate new from interpreted attributes, new attributes are explicitly tagged as being new and interpreted attributes are listed without any type information since that information hasn't changed anyway.

3.2 Modified classes

3.2.1 Model (interpreted)

Model
sourceLanguage
sourceDialect

The new or modified attributes are:

- `sourceLanguage`
For Smalltalk models this attribute always contains the string “Smalltalk”.
- `sourceDialect`
The Smalltalk language has different dialects, with different versions. If known, this version can be stored in this attribute. Regarding FAMIX, certain Smalltalk dialects such as VisualWorks 5i or Quasar Smalltalk have the notion of namespaces.

3.2.2 Package (interpreted)

Package

Smalltalk namespaces (only available in certain dialects) are mapped to FAMIX Packages. Namespaces in Smalltalk have the following properties:

- Namespaces can contain classes and namespaces. Both classes and packages can belong to only one package.
- Namespaces names should be unique within their encapsulating package.

These properties are in sync with the expected properties in the core FAMIX definition.

3.2.3 Class (interpreted and extended)

Class
<code>isAbstract</code> <code>categories #new</code> <code>isMetaclass #new</code> <code>metaclass # new</code>

Both Smalltalk classes and metaclasses are modelled using the FAMIX Class concept. In Smalltalk every class has a metaclass associated with it. The metaclass does not have its own name, so we create the name from the sole metaclass instance concatenated with “_class”.

The new or modified attributes are:

- `isAbstract`
In Smalltalk a class is abstract if at least one of its methods is declared abstract or if it inherits abstract methods and does not provide an implementation for them.
- `categories`
A class has a list of categories to which methods can be associated.
- `isMetaclass: Boolean; required`
Returns true if the class represents a Smalltalk metaclass.
- `metaclass: Name; required`
if `isMetaclass` is false, `metaclass` returns the unique name of a associated metaclass, null otherwise. Although in Smalltalk asking the class of a metaclass or the metaclass of a metaclass is possible, we cut the recursion at the level of the class. Otherwise the whole reflective kernel of Smalltalk would have to be represented.

3.2.4 BehaviouralEntity (interpreted and extended)

BehaviouralEntity
declaredReturnClass declaredReturnType inferredReturnClasses #new

In Smalltalk returntypes of methods are not explicit. Therefore, `declaredReturnClass` and `declaredReturnType` contain the most general type available, namely `Object`. As in Smalltalk there are only classes and no primitive types or pointers, `declaredReturnClass` and `declaredReturnClass` always denote a class.

The new attribute is:

- `inferredReturnClasses 0..N Name; optional`
Contains possible inferred returned types of the BehavioralEntity.

3.2.5 Method (interpreted and extended)

Method
accessControlQualifier signature isPureAccessor hasClassScope isAbstract isConstructor isPrimitive (): Boolean #new belongsToCategory (): String #new

Each definition of a method in source code constitutes this entity.

The new or modified attributes are:

- `accessControlQualifier`
In Smalltalk, all methods are public.

- `signature`

The signature follows the FAMIX conventions (see section 2.1.2). For instance, `m: anInt` is stored as: `m: (Object).`

- `isPureAccessor`

A pure read accessor in Smalltalk normally looks like (accessing a variable name):

```
name
^name
```

A pure write accessor normally looks like:

```
name: aString
name := aString
```

- `hasClassScope`

In Smalltalk, class methods are instance methods of a metaclass. As metaclasses are represented in FAMIX as classes and thus distinguished from the class with which they are associated, `hasClassScope` can only be false.

- `isAbstract`

A method is abstract, if it invokes the method `subclassResponsibility`. Consequently, an abstract method in Smalltalk has an implementation (contrary to languages such as Java). An example:

```
name
    self subclassResponsibility
```

- `isConstructor`

In Smalltalk there is no special constructor concept. Every class method that returns an instance of that class is normally considered a constructor. However, there are no special rules. It is just another method. Therefore, `isConstructor` is false by default and may be set to true if further analysis is performed.

- `isPrimitive: Boolean; optional`

Is a predicate telling if the method is a primitive or not.

- `belongsToCategory: String; optional.`

In Smalltalk, a method is defined into a category, a name for a group of methods.

3.2.6 StructuralEntity (interpreted and Extended)

StructuralEntity
<code>declaredType</code> <code>declaredClass</code> <code>inferredClasses #new</code>

In Smalltalk types of variables are not explicit. Any object of any type can be stored in a variable. Therefore, `declaredType` and `declaredClass` contain the most general type available, namely `Object`. As in Smalltalk there are only classes and no primitive types or pointers, the `declaredType` and `declaredClass` always denote the same FAMIX class.

The new attribute is:

- `inferredClasses 0..N Name; optional`

Contains possible inferred returned types of the `StructuralEntity`.

3.2.7 Attribute (interpreted)

Attribute
<code>hasClassScope</code> <code>accessControlQualifier</code>

The new or modified attributes are:

- `hasClassScope`

An attribute in Smalltalk has class scope if it is defined as `ClassVariable`, e.g., shared by all the instances of a class and its subclasses. Metaclass instance variables are stored as instance variables of the metaclass (which is modelled as just another class) and hence do not have class scope.

- `accessControlQualifier`

In Smalltalk all attributes are protected, i.e., only accessible within the class that defines the attribute and its subclasses.

3.2.8 GlobalVariable (interpreted)

GlobalVariable

Smalltalk global variables are mapped to the GlobalVariable concept with the exception of classes. These are modelled as Classes, although a Smalltalk class is also a global variable.

3.2.9 ImplicitVariable (interpreted)

ImplicitVariable

Implicit variables in Smalltalk are `self`, `super` and `thisContext` in certain Smalltalk dialects (such as VisualWorks, Squeak and Dolphin). `self` is an implicit instance variable which refers the current object a method is executing in. `super` refers to the superclass of the current class defining the method in which `super` is used. `thisContext` is a variable that represents the execution stack. Implicit variables will only appear in a model when they are explicitly referred to.

3.2.10 LocalVariable (interpreted)

LocalVariable

Smalltalk local method variables are mapped to the LocalVariable concept. Variables that are local to a subscope of a method body (f.i., a block definition) are also considered LocalVariables. This leads the problem of multiple entities with the same unique name if multiple subsopes define local variables with the same name. For instance, the following code results in two LocalVariables with the unique name `myClass.myMethod.each`.

```
myClass>>myMethod
  collection do: [ :each | each doSomethingNice ].
  collection do: [ :each | each doSomethingBad ]
```

A possible solution is the explicit representation of blocks.

3.2.11 FormalParameter (interpreted)

FormalParameter

In Smalltalk, formal parameters are read-only. For instance,

```
name: aString
aString := ''
```

is not allowed.

3.2.12 InheritanceDefinition (interpreted)

InheritanceDefinition
accessControlQualifier index

In Smalltalk classes always inherit from a single class (except the root class `Object` which does not inherit from any class). Class and metaclass inheritance hierarchies are parallel. So a metaclass always inherits the metaclass of the superclass of its associated instance (which is a class).

The new or modified attributes are:

- `accessControlQualifier`

The access control in Smalltalk is always ‘public’. It means that all public methods and protected attributes are inherited by the subclass and keep their declared visibility.

- `index`

The index is always ‘null’ as Smalltalk has single inheritance, so an index has no meaning.

3.2.13 Invocation (interpreted)

Invocation
base receivingClass candidatesAt

The new or modified attributes are:

- `base`

Due to the lack of static type information, in Smalltalk the class of the receiver (and the base through that class) can only be statically determined when a method is sent to the `self` or `super` implicit variables (note that this does not take the runtime aspect into account, but the static definition of methods).

- `receivingClass`

Similar to `base`, the receiving class can only be statically determined when the receiver is `self`, `super` or a class.

- `candidates`

For invocations the candidates attribute holds any method with the same signature as invokes. Note that if the base or receiving class are known the candidates can be restricted to the hierarchy of `base`.

3.3 Miscellaneous

Smalltalk does not have functions, thus those entities will never appear in a FAMIX model of a Smalltalk system.

3.4 Pending Issues

Issues not yet covered in this extension are:

- In Smalltalk classes are also global variables. Normally any references to classes are covered through method of the class that are invoked or attributes that are accesses. However, FAMIX does not record the dependencies when a class is used only as a global variable (i.e., without a message send). Some examples:

```
x := Object
^Object
self m: Object
Object := OrderedCollection
```

If a class would be modelled as a kind of StructuralEntity (like any other variable) then the above cases would be modelled as accesses to this StructuralEntity. However, this requires a change in the FAMIX core.

- Pool dictionaries and pool variables are currently not covered. Possibilities to model these concepts are
 - to create two new StructuralEntity's called PoolDictionary (as a subclass of GlobalVariable) and PoolVariable. Additionally, to capture which classes use which pool variables, a relation between Classes and PoolDictionaries is must be created.
 - to model a pool dictionary as a class with only class-scope attributes (the pool variables). Access to pool variables would be an Access to the class attribute. Note that in Smalltalk it is not possible to access normal attributes from outside the hierarchy they are defined in.
- Currently it is not determinable in FAMIX that an invocation of the form 'self class myMetaclass-Method' is an invocation of a metaclass method. It is interpreted as follows: the method 'class' is invoked on self, the method 'myMetaclassMethod' is recorded to be invoked on 'some' expression. It might be desirable to interpret this special differently for the purpose of recording class method accesses.
- Depending on the dialect, other grouping concepts such as 'parcels' and 'applications' exist. We currently do not cover these.
- We do not cover block closures. As these are submethod-level concepts, it is not likely that we will cover them in the near future.

APPENDIX D

Java Extension to FAMIX

This appendix describes the Java extension to FAMIX. Before discussing the specific interpretations and extensions, it gives a short overview of how FAMIX can be extended.

4.1 Extending FAMIX

The basic FAMIX model is modified in three different ways to handle Java sources:

- New classes are added to the common exchange model to model entities and associations unique to Java. These classes are marked as new entities respectively associations.
- New attributes are added to existing classes of the basic FAMIX model. In this case the class is marked “extended” and only the new and modified (see below) attributes are listed in the definition of the modified class.
- The definition of attributes of existing classes are modified or are made more specific. In this case the corresponding class is marked “interpreted” and the interpreted attributes are listed in the definition of the modified class. To discriminate new from interpreted attributes, new attributes are explicitly tagged as being new and interpreted attributes are listed without any type information since that information hasn't changed anyway.

4.2 Modified classes

4.2.1 Model (interpreted)

Model
sourceLanguage
sourceDialect

The new or modified attributes are:

- `sourceLanguage`

For Java models this attribute always contains the string “Java”.

- `sourceDialect`

The Java language does not have dialects, but it has versions. If known, this version can be stored in this attribute. The possibly interesting issues for FAMIX on the language-feature- and-syntax level (as opposed to added libraries) between the different versions are:

- 1.0.x -> 1.1.x: - Addition of inner classes (including anonymous ones)
- Final method parameters and local variables
- 1.1.x -> 1.2.x: - Addition of a new keyword (`strictfp`)

4.2.2 Package (interpreted)

Package

A Package maps in Java to the Java package construct. Packages in Java have the following properties:

- packages contain classes and packages. Both classes and packages can belong to only one package.
- package names should be unique within their encapsulating package.

These properties are in sync with the expected properties in the core FAMIX definition.

Normally packages in Java map directly to the directory structure of source code, i.e. the source code for a certain class in a certain package appears in a directory with the same name as the package. Nested packages appear as subdirectories of the directory with the source code of the encapsulating package.

4.2.3 Class (interpreted and extended)

Class
<code>isInterface ()</code> : Boolean # new <code>isPublic ()</code> : Boolean # new <code>isFinal ()</code> : Boolean # new <code>isAbstract</code> <code>belongsToPackage</code>

Both classes and interfaces in Java are mapped to the FAMIX entity Class. Interfaces differ from classes in that they can only define abstract methods and final static variables. Interfaces cannot inherit from classes (for a full discussion, see “InheritanceDefinition (interpreted)” on page 174).

The new or modified attributes are:

- `isInterface`: Boolean; optional

Is a predicate telling if the entity is an interface as opposed to a normal class.

- `isPublic`: Boolean; optional

Is a predicate telling if the class is defined public or not. Public (as opposed to default) visibility means the class is visible outside its containing package.

- `isFinal: Boolean; optional`

Is a predicate telling if the class is defined final or not. Final classes cannot be subclassed (and subsequently its methods cannot be overridden). Interfaces cannot be final.

- `isAbstract`

In Java a class is abstract if the class is declared abstract. This is obligatory if one or more of its methods are abstract. Even if the class does not contain any abstract methods, it can be declared abstract, implying that the class is not allowed to be instantiated. Interfaces are always abstract, but don't have to be declared as such (although you may if you want to).

- `belongsToPackage`

The package to which a class belongs is defined by the package statement at the beginning of a Java source file that also contains the class definition.

4.2.4 BehaviouralEntity (interpreted and extended)

BehaviouralEntity
<code>declaredReturnClass</code> <code>declaredReturnType</code>

The following attributes are interpreted as follows:

- `declaredReturnType`

In Java this attribute can contain any primitive types, array types or classes (and interfaces).

- `declaredReturnClass`

This attribute contains the unique name of the FAMIX class entity (which is a Java class or interface) if the `declaredReturnType` denotes such an entity.

4.2.5 Method (interpreted and extended)

Method
<code>isFinal (): Boolean # new</code> <code>isSynchronized (): Boolean # new</code> <code>isNative (): Boolean # new</code> <code>accessControlQualifier</code> <code>signature</code> <code>isPureAccessor</code> <code>hasClassScope</code> <code>isAbstract</code> <code>isConstructor</code>

Each definition of a method in source code constitutes this entity.

The new or modified attributes are:

- `isFinal: Boolean; optional`

Is a predicate telling if the method is defined final or not. Final methods cannot be overridden.

- `isSynchronized: Boolean; optional`

Is a predicate telling if the method is defined synchronized or not. Only one of the synchronized methods of an instance of a class can be accessed at once at runtime.

- `isNative: Boolean; optional`

Is a predicate telling if the method is defined native or not. Native methods are implemented in an external language (for instance, C++) and therefore do not have an implementation in the Java side of the code.

- `accessControlQualifier`

The allowed access specifiers for methods are: `public`, `protected`, `private`. An empty specifier means default visibility, which denotes that the method is visible for all classes within the same package.

- `signature`

In Java a method is uniquely distinguished by its name and the number, the types and the position of its formal parameters. Therefore, the signature string takes the form `methodName(T1, ..., Tn)` where `T1..n` are the types of the formal parameters of the method (see also the FAMIX naming conventions in section 2.1.2). Note that parameters can be declared `final`, but that this finalness is not part of the method signature. A subclass can override a method and add or drop any final parameter modifiers you wish. You can also add or drop final modifiers in a method's parameters without causing any harm to existing compiled code that uses that method [GJSB00].

- `isPureAccessor`

A pure reader accessor in Java normally looks like (accessing a variable name):

```
String getName {
    return name;
}
```

A pure writer accessor normally looks like:

```
void setName(String name) {
    this.name = name;
}
```

- `hasClassScope`

A method in Java has class scope if it is defined static.

- `isAbstract`

A method is abstract, if it is declared abstract with the `abstract` keyword. An abstract method in Java doesn't have an implementation.

- `isConstructor`

A constructor in Java has the form of a method with no declared return type and a name identical to the name of the class it belongs to.

4.2.6 StructuralEntity (interpreted and Extended)

StructuralEntity
declaredType
declaredClass

The following attributes are interpreted as follows:

- `declaredType`

In Java this attribute can contain any primitive types, array types or classes (and interfaces).

- `declaredClass`

This attribute contains the unique name of the FAMIX class entity (which is a Java class or interface) if the `declaredType` denotes such an entity.

4.2.7 Attribute (interpreted)

Attribute
<code>isFinal ()</code> : Boolean # new <code>isTransient ()</code> : Boolean # new <code>isVolatile ()</code> : Boolean # new <code>hasClassScope</code> <code>accessControlQualifier</code>

The new or modified attributes are:

- `isFinal`: Boolean; optional

Is a predicate telling if the attribute is defined final or not. Final attributes are set only once and cannot be changed afterwards.

- `isTransient`: Boolean; optional

Is a predicate telling if the (non-static) attribute is defined transient or not. Transient indicates that an attribute is not part of an object's persistent state and thus needs not to be serialized with the object.

- `isVolatile`: Boolean; optional

Is a predicate telling if the attribute is defined volatile or not. Volatile specifies that an attribute is used by synchronized threads and that the compiler should not attempt to perform optimisations with it.

- `hasClassScope`

An attribute in Java has class scope if it is defined static.

- `accessControlQualifier`

The allowed access specifiers are: `public`, `protected`, `private`. An empty specifier means default visibility, which denotes that the attribute is visible for all classes within the same package.

4.2.8 ImplicitVariable (interpreted)

ImplicitVariable

Implicit variables in Java are `this`, `super` and `class`. `this` is an implicit instance variable which refers the current object a method is executing in. `super` refers to the superclass of the current object. `class` is not an implicit variable in the strict sense of the word (as it is also a keyword in Java). An expression like `String.class` evaluates to a reference to the `String` class object. This works for all types, including the primitive types. It is close enough, however, to an implicit static variable to be modelled as an implicit variable. Implicit variables will only appear in a transfer when they are explicitly referenced by other entities.

4.2.9 LocalVariable (interpreted)

LocalVariable
isFinal (): Boolean # new

The new or modified attributes are:

- `isFinal: Boolean; optional`

Is a predicate telling if the attribute is defined final or not. Final local variables are set only once and cannot be changed afterwards.

4.2.10 FormalParameter (interpreted)

FormalParameter
isFinal (): Boolean # new

The new or modified attributes are:

- `isFinal: Boolean; optional`

Is a predicate telling if the attribute is defined final or not. Final parameters cannot be changed within the body of the method it is a parameter of. Note that the finalness of a parameter is not part of the method signature — it is simply a detail of the implementation. A subclass can override a method and add or drop any final parameter modifiers you wish. You can also add or drop final modifiers in a method's parameters without causing any harm to existing compiled code that uses that method.

4.2.11 InheritanceDefinition (interpreted)

InheritanceDefinition
accessControlQualifier index

In Java classes always inherit from a single class (except the root class Object that does not inherit from any class). A class can *implement* multiple interfaces, which simulates some kind of multiple inheritance, but as interfaces do not have any implementation, resolving which method needs to be executed, is not a problem. Interfaces can inherit from multiple interfaces. In FAMIX classes and interfaces are treated similarly, as shown by the fact that they are both represented as classes, therefore both class inheritance and interface implementation is represented by an InheritanceDefinition in FAMIX.

The new or modified attributes are:

- `accessControlQualifier`

The access control in Java is always 'public'. It means that all public and protected attributes and methods are inherited by the subclass and keep their declared visibility.

- `index`

The index is always 'null' as Java has single inheritance and therefore name collisions cannot appear. Java classes can implement multiple interfaces, but as interfaces do not implement any behaviour name collisions do not cause any problems. Interfaces can contain constants, but a class cannot implement multiple interfaces that contain constants with the same name with possibly different values.

4.2.12 Invocation (interpreted)

Invocation
base receivingClass candidatesAt

The new or modified attributes are:

- **base**
In Java this attribute contains the statically determinable class of the expression receiving the invocation. For example:

```
MyClass r = new MyClass();
...
r.m();
```

Then *r* is the receiver and therefore *MyClass* the receiving class. If the receiving class also contains a method with the invoked signature, it is the base. Otherwise the class defining the inherited method with that signature it the base.
- **receivingClass**
See the definition of base.
- **candidates**
For invocations the candidates attribute holds either all methods overriding the method `base::invokes`, or if base is a Java interface it holds all methods with the same signature in the class hierarchies that implement that interface.

4.3 New classes

4.3.1 TypeCast

TypeCast
belongsToBehaviour (): Name fromType (): Name toType (): Name

TypeCast is a subclass of Association. It models Java type cast (e.g., `(MyClass)variable`). Type casts are interesting for reengineering as they often point to problems in the design of a system. There will be an instance of this class for every type cast occurring in the source code, even if the cast is between the same types, because we are interested in all the places where casts occur.

The attributes of TypeCast are:

- **belongsToBehaviour: Name; mandatory**
Refers to the BehaviouralEntity the cast appears in.
- **fromType: Name; optional**
Refers to the unique name of the type the cast expression has. This is the declared type of variable in the above example.
- **toType: Name; optional**
Refers to the unique name of the type the expression is cast to (*MyClass* in the above example).

4.4 Miscellaneous

Java does not have functions or global variables, thus those entities will never appear in a FAMIX model of a Java system. Next to that, arrays and primitive types are not handled explicitly in this FAMIX extension either.

Then there is a minor issue about file visibility. Normally a class with default visibility is visible within its package. However, when such a class is defined in the same file of another class and the name of the file is the same as the name of the other with the `.java` extension and these classes are not defined in the default package, then the class is not visible outside the file, even to classes in the same package that are defined in other files. This issue is not dealt with in this Java language plug-in, because it's a minor issue and in model transfers we assume a compilable system anyway.

4.5 Pending Issues

Issues not yet covered in this plug-in are:

- Nested classes, inner classes, anonymous classes. A solution for this needs to be synchronized with other language plugins (most notably C++).
- Implicit methods. In Java there are certain methods defined implicitly. These are the default constructors and the methods `this(..)`, `super(..)` (with or without parameters), which are some kind of aliases to constructors of either the current class or its superclass. If introduced in the FAMIX extension, these implicit methods should only appear in a transfer when they are explicitly referenced by other entities. Implicit methods can be introduced as Methods with a boolean `isImplicit` attribute set to true. Another possibility is to create a new entity called `ImplicitMethod` (similar to `ImplicitVariable`). However, this causes problems on the language independent level, as entities and associations on the language independent level (such as in `Invocations`) may reference this language specific entity. For that to work, `ImplicitMethod` should be defined on the FAMIX core level rather than the Java extension level.
- Static and instance initialisers. A possibility is to model these as a special kind of method.

Bibliography

- [AT98] M. N. Armstrong and C. Trudeau. Evaluating architectural extractors. In *Proceedings of WCRE'98*, pages 30–39. IEEE Computer Society, 1998.
- [Bar99] Holger Bär. FAMIX C++ language plug-in 1.0. Technical report, University of Berne, September 1999.
- [BBC+99] Philip A. Bernstein, Thomas Bergsträsser, Jason Carlson, Shankar Pal, Paul Sanders, and David Shutt. Microsoft Repository Version 2 and the Open Information Model. *Information Systems*, 24(2):71–98, 1999.
- [BC00] Bell Canada. DATRIX abstract semantic graph reference manual (version 1.4). Technical report, Bell Canada, May 2000.
- [Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
- [Bis92] Walter R. Bischofberger. Sniff: A pragmatic approach to a c++ programming environment. In *C++ Conference*, pages 67–82, 1992.
- [BLFIM98] T. Berners-Lee, R. Fielding, U. C. Irvine, and L. Masinter. Uniform Resource Identifiers (URI): Generic syntax. Technical report, RFC 2396, August 1998. <http://www.ietf.org/rfc/rfc2396.txt>.
- [BMMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *Antipatterns*, 1998.
- [Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.
- [Bor01] Michael Borchardt. A feasibility study for a C++ refactoring engine. Master’s thesis, University of Antwerp, August 2001.
- [BPSM98] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 - w3c recommendation 10-february-1998. Technical Report REC-xml-19980210, World Wide Web Consortium, February 1998.
- [Cas91] Eduardo Casais. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach*. Ph.D. thesis, Centre Universitaire d’Informatique, University of Geneva, May 1991.

-
- [Cas92] Eduardo Casais. An incremental class reorganization approach. In O. Lehrmann Madsen, editor, *Proceedings ECOOP'92*, LNCS 615, pages 114–132, Utrecht, The Netherlands, June 1992. Springer-Verlag.
- [Cas98] Eduardo Casais. Re-engineering object-oriented legacy systems. *Journal of Object-Oriented Programming*, 10(8):45–52, January 1998.
- [CC90] Elliot J. Chikofsky and James H. Cross, II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.
- [CEK+00] Jörg Czeranski, Thomas Eisenbarth, Holger M. Kienle, Rainer Koschke, Erhard Plödender, Daniel Simon, Yan Zhang, Jean-François Girard, and Martin Würthner. Data exchange in Bauhaus. In *Proceedings WCRE'00*. IEEE Computer Society Press, November 2000.
- [CGK98] Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios. A C++ data model supporting reachability analysis and dead code detection. *IEEE Transactions on Software Engineering*, 24(9):682–693, September 1998.
- [Ciu99] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of TOOLS 30 (USA)*, pages 18–32, 1999.
- [Com94] CDIF Technical Committee. CDIF framework for modeling and extensibility. Technical Report EIA/IS-107, Electronic Industries Association, January 1994. See <http://www.cdif.org/>.
- [DD99] Stéphane Ducasse and Serge Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Berne, October 1999. See <http://www.iam.unibe.ch/~famoos/handbook>.
- [DDHL96] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On automatic class insertion with overloading. In *Proceedings of OOPSLA'96*, pages 251–267, 1996.
- [DDL99] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In Françoise Balmas, Mike Blaha, and Spencer Rugaber, editors, *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering)*. IEEE, October 1999.
- [DDT99] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In Bernhard Rumpe, editor, *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, LNCS 1723, Kaiserslautern, Germany, October 1999. Springer-Verlag.
- [DLT00] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.
- [DMO00] Steve DeRose, Eve Maler, and David Orchard. XML Linking Language (XLink) version 1.0 - w3c proposed recommendation 20 december 2000. Technical Report PR-xlink-20001220, World Wide Web Consortium, December 2000.

-
- [DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, September 1999.
 - [Fav01] Jean-Marie Favre. G^{see}: a generic software exploration environment. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 233–244. IEEE, May 2001.
 - [FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
 - [FHK+97] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.
 - [FR98] Richard Fanta and Vaclav Rajlich. Reengineering object-oriented code. In *Proceedings of the International Conference on Software Maintenance*, 1998.
 - [Fre00] Michael Freidig. XMI for FAMIX. Informatikprojekt, University of Berne, June 2000.
 - [FY00] Brian Foote and Joseph W. Yoder. Big ball of mud. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design*, volume 4, pages 654–692. Addison-Wesley, 2000.
 - [GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.
 - [HEH+96] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, and D. Roland. Database reverse engineering: From requirements to CARE tools. *Automated Software Engineering*, 3(1-2), June 1996.
 - [Hol98] Richard C. Holt. An introduction to TA: The Tuple-Attribute language. Technical report, University of Waterloo, November 1998.
 - [HWS00] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a standard exchange format. In *Proceedings WCRE'00*, November 2000.
 - [HYR96] D. R. Harris, A. S. Yeh, and H. B. Reubenstein. Extracting architectural features from source code. *Automated Software Engineering*, 3(1-2):109–139, 1996.
 - [JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
 - [jFa] jFactor. <http://www.instantiations.com/jfactor/>.
 - [JGR99] Mehdi Jazayeri, Harald Gall, and Claudio Riva. Visualizing software release histories: The use of color and third dimension. In *ICSM'99 Proceedings (International Conference on Software Maintenance)*. IEEE Computer Society, 1999.
 - [JO93] Ralph E. Johnson and William F. Opdyke. Refactoring and aggregation. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742 of *Lecture Notes in Computer Science*, pages 264–278. Springer-Verlag, November 1993.

-
- [jRe] JRefactory. <http://jrefactory.sourceforge.net/>.
- [JUn] JUnit. <http://www.junit.org>.
- [Kaz96] R. Kazman. Tool support for architecture analysis and design, 1996. Proceedings of Workshop (ISAW-2) joint Sigsoft.
- [KN01] Georges Golomingi Koni-N'sapu. A scenario based approach for refactoring duplicated code in object oriented systems. Diploma thesis, University of Berne, June 2001.
- [Kos00] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.
- [KSRP99] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse engineering of design components. In *Proceedings of ICSE'99*, May 1999.
- [Lan99] Michele Lanza. Combining metrics and graphs for object oriented reverse engineering. Diploma thesis, University of Bern, October 1999.
- [LB85] M. M. Lehman and L. Belady. *Program Evolution - Processes of Software Change*. London Academic Press, 1985.
- [LDS01] Michele Lanza, Stéphane Ducasse, and Lukas Steiger. Understanding software evolution using a flexible query engine. In *Proceedings of the Workshop on Formal Foundations of Software Evolution*, 2001.
- [Leh96] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.
- [Let98] Timothy C. Lethbridge. Requirements and proposal for a Software Information Exchange Format (SIEF) standard. Technical report, University of Ottawa, November 1998. <http://www.site.uottawa.ca/~tcl/papers/sief/standardProposal-v1.html>.
- [LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.
- [LLB+98] Bruno Laguë, Charles Leduc, André Le Bon, Ettore Merlo, and Michel Dagenais. An analysis framework for understanding layered software architectures. In *Proceedings IWPC'98*, 1998.
- [LS99] Panagiotis K. Linos and Stephen R. Schach. Comprehending multilanguage and multiparadigm software. In *Proceedings of the short papers of ICSM'99*, pages 25–28, August 1999.
- [MADSM01] C. Best, M.-A. D. Storey and J. Michaud. Shrimp views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC 2001)*, 2001.
- [Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA'87, ACM SIGPLAN Notices*, pages 147–155, December 1987. Published as Proceedings OOPSLA'87, ACM SIGPLAN Notices, volume 22, number 12.
- [MN97] Gail C. Murphy and David Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 8:29–36, 1997.

-
- [MNGL98] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An emperical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, April 1998.
- [Moo96] Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of OOPSLA'96 Conference*, pages 235–250. ACM Press, 1996.
- [Mul86] H. A. Müller. *Rigi - A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.
- [MWD99] Kim Mens, Roel Wuyts, and Theo D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pages 33–45, June 1999.
- [Neb99] Robb Nebbe. FAMIX Ada language plug-in 2.2. Technical report, University of Berne, August 1999.
- [OCN99] Mel Ó Cinnéide and Paddy Nixon. A methodology for the automated introduction of design patterns. In *Proceedings ICSM'99*. IEEE Computer Society Press, August 1999.
- [OG97] Open Group. DCE 1.1: Remote procedure call. Technical Report C706, Open Group, August 1997.
- [OJ93] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In *Proceedings of the 1993 ACM Conference on Computer Science*, pages 66–73. ACM Press, 1993.
- [OMG97] Object Management Group. Meta object facility (MOF) specification. Technical Report ad/97-08-14, Object Management Group, September 1997.
- [OMG98] Object Management Group. XML Metadata Interchange (XMI). Technical Report ad/98-10-05, Object Management Group, February 1998.
- [OMG99] Object Management Group. Unified Modeling Language (version 1.3). Technical report, Object Management Group, June 1999.
- [OMG00] Object Management Group. Meta Object Facility (MOF) specification (version 1.3). Technical report, Object Management Group, March 2000.
- [Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.
- [Par98] ParcPlace. *VisualWorks 3.0*, 1998. User Guide, Cookbook, Reference Manual.
- [RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.
- [RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 13–22. IEEE, September 1999.
- [Ree96] Trygve Reenskaug. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996.

-
- [Rob99] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.
- [RSK00] Sébastien Robitaille, Reinhard Schauer, and Rudolf K. Keller. Bridging program comprehension tools by design navigation. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, 2000.
- [Sch01] Andreas Schlapbach. Generix XMI support for the MOOSE reengineering environment. Informatikprojekt, University of Bern, June 2001.
- [SDSK00] Guy Saint-Denis, Reinhard Schauer, and Rudolf K. Keller. Selecting a model interchange format. the SPOOL case study. In *Proceedings of the Thirty-Third Annual Hawaii International Conference on System Sciences*, 2000.
- [SGMZ98] Benedikt Schulz, Thomas Genssler, Berthold Mohr, and Walter Zimmer. On the computer aided introduction of design patterns into object-oriented systems. In *Proceedings of the TOOLS 27 Conference (Asia 1998)*. IEEE Computer Society Press, 1998.
- [SMHP+97] Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, iLogix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, and Softeam. *Object Constraint Language Specification (version 1.1)*. Rational Software Corporation, September 1997.
- [Som96] Ian Sommerville. *Software Engineering*. Addison-Wesley, fifth edition, 1996.
- [SS00] Susan Elliott Sim and Margaret-Anne D. Storey. A structured demonstration of program comprehension tools. In *Proceedings of WCRE 2000*, pages 184–193, 2000.
- [Ste01] Lukas Steiger. Recovering the evolution of object oriented software systems using a flexible query engine. Diploma thesis, University of Bern, June 2001.
- [Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.
- [Tak96] TakeFive Software GmbH. *SNiFF+*, 1996.
- [TB99a] Lance Tokuda and Don Batory. Automating three modes of evolution for object-oriented software architecture. In *Proceedings COOTS'99*, May 1999.
- [TB99b] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. In *Proceedings of Automated Software Engineering*, 1999.
- [TD99] Sander Tichelaar and Serge Demeyer. *SNiFF+* talks to Rational Rose – interoperability using a common exchange model. In *SNiFF+ User's Conference*, January 1999. Also appeared in the "Proceedings of the ESEC/FSE'99 Workshop on Object-Oriented Re-engineering (WOOR'99)" – Technical Report of the Technical University of Vienna (TUV-1841-99-13).
- [TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000*. IEEE, 2000.
- [Tic99] Sander Tichelaar. FAMIX Java language plug-in 1.0. Technical report, University of Berne, September 1999.

-
- [Tur81] W. Turski. Software stability. In *Proceedings for the 6th ACM Conference on Systems Architecture*, 1981.
- [Uni96] University of Karlsruhe. *COMPOST*, 1996. <http://i44www.info.uni-karlsruhe.de/compost>.
- [URE] Unisys Universal Repository (UREP). <http://www.unisys.com/marketplace/urep/>.
- [Wer99] Michael M. Werner. *Facilitating Schema Evolution With Automatic Program Transformation*. PhD thesis, Northeastern University, July 1999.
- [WH92] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.
- [Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.
- [WWWC99] World Wide Web Consortium. Resource Description Framework (RDF) model and syntax specification. Technical report, World Wide Web Consortium, February 1999.
- [YHC97] A. S. Yeh, D. R. Harris, and M. P. Chase. Manipulating recovered software architecture views. In *Proceedings of ICSE'97*, 1997.

Curriculum Vitae

Personal Information

Name: Sander Tichelaar
Date of Birth: 07-03-1972
Place of Birth: Bergambacht, The Netherlands
Nationality: Dutch

Education

1997-2001 PhD student in the Software Composition Group, University of Berne, Switzerland. During this time I have been mainly working on a European Esprit project (FAMOOS) on reengineering object-oriented systems.

1990-1997 Rijksuniversiteit Groningen (University of Groningen, The Netherlands), Master's degree in Computer Science, May 1997, Master's thesis title: "A Coordination Component Framework for Open Distributed Systems".

1984-1990 Christelijk Gymnasium Utrecht (Grammar School in Utrecht, The Netherlands).

