Vrije Universiteit Brussel - Belgium

Faculty of Sciences

In Collaboration with Ecole des Mines de Nantes - France

2003

Towards round-trip engineering using logic metaprogramming

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: João Guilherme Del Valle

Promotor: Prof. Dr. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Prof. Dr. Kim Mens (Université Catholique de Louvain)

Abstract

Round-trip engineering is a popular technique for software development. A development environment with round-trip engineering support normally provides two views of the software system: one view is the source code, containing the implementation of the software itself, and the other view is more abstract, giving a global picture of the design of the system. If changes are made in the source code, the environment is responsible for propagating them to the diagram view. Similarly, if the diagram view changes, the changes are propagated to the code.

Many industry-level tools provide support for round-trip engineering. However, there is no general approach to it. Each tool implements its own strategy for mapping code constructs to diagrams and vice versa. Moreover, tools provide very limited options for customization, enforcing the developer to adapt his development practices to the tool instead of being able to adapt the tool to his needs.

We propose logic metaprogramming (LMP) as an approach for constructing more customizable round-trip engineering tools. LMP is the use of logic programming to reason about source code at meta-level. In this thesis, we show how it can be used for extracting diagrams from the code and generating code from diagrams. Besides the advanced reasoning power, we believe that the use of logic rules can lead to a clear, general and more flexible approach map between source code and diagrams.

In this thesis, we focus on UML class diagrams to describe the design of the software system, and Java as the source code language. We validate our approach with the implementation of LURE, a prototype tool for round-trip engineering.

Acknowledgements

Thanks to my advisors, Kim Mens and Johan Fabry, for kindly pushing me forward, for criticizing me when it was needed, for telling me the possible paths, for discreetly pointing me out the most recommended, for leaving the choice up to me, for helping me to arise my best.

Thanks to Theo D'Hondt, Jacques Noyé and all the people that make the EMOOSE program a (wonderful) reality.

Thanks to all my new and old friends. You know how important you are to me.

Thanks to my father, my mother and my brother, without whose love I would never have made it.

Contents

1 Introduction **1**

1.1 Context 2

1.1.1 Unified Modelling Language 2

1.1.2 Logic metaprogramming 3

1.1.3 Round-trip engineering 3

1.2 Approach 3

1.3 Contribution 4

1.4 Structure 5

2 Round-trip engineering **6**

2.1 Overview 6

2.2 Tool review 7

2.2.1 Together Control Center 9

2.2.2 Fujaba 11

2.2.3 Poseidon 12

2.3 Discussion 13

3 Logic metaprogramming **16**

3.1 Logic metaprogramming 16

3.2 Logic metaprogramming with SOUL 17
3.2.1 Library for Code Reasoning — LiCoR 22
3.2.2 SOULJava 25

4 Code generation using LMP 27

4.1 Overview 27
4.2 The *pull* approach 28
4.3 Example — generating a visitor design pattern 31
4.4 Reverse-engineering using LMP 34

5 Round-trip engineering using LMP 37

5.1 From diagram to code and back again 37
5.1.1 A small example 39
5.1.2 Discussion 44
5.2 Achieving round-trip engineering 45
5.2.1 Round-trip engineering with updates 46
5.2.2 Refining the updates 49

6 Mapping strategies 52

6.1 UML Class Diagram representation 52
6.2 A mapping strategy 57
6.2.1 Conceptual mapping 58
6.2.2 Code generation mappings 58
6.2.3 Reverse-engineering mappings 71
6.2.4 Verifying the extraction 89
6.3 Language independence 93

6.3.1 Language independence issues 95
6.4 Lure — our tool for round-trip engineering 97
6.4.1 Starting to use Lure 97
6.4.2 Round-trip engineering with Lure 99
6.4.3 Editing code and diagram 100

7 Case Study — GSMCase 103
7.1 Case defintion 103
7.1.1 Requirements 103
7.1.2 Initial diagram 104
7.2 Round-trip engineering 105
7.2.1 First code generation 105
7.2.2 Implementing the simulation data 107
7.2.3 Verifying the changes 107
7.2.4 Changing the design 109
7.2.5 Finishing the implementation 109

8 Conclusion and future work 112
8.1 Conclusion 112
8.2 Future work 113

A Final code for GSMCase 120

Chapter 1

Introduction

Co-evolution [10] is a major concern in Software Engineering. This research area studies the synchronization between the different views of a software system during its lifetime, in order to manage its evolution. Throughout the evolution of a piece of software, there is no doubt that higher-level views of the code play a key role. Some examples of the uses of these views are architectural analysis, design refactoring or simply documentation.

A particulary interesting sub-area of co-evolution is round-trip engineering. This software development technique keeps higher-level views (e.g., diagrams) of the source code up to date while the code is being implemented, as well as it updates the source code when these views change. The term "round-trip" comes from the fact that the developer can perform changes both in the higher-level views and in the code, and the development environment is responsible for propagating these changes to the other part. This development technique turns out to be very comfortable, since it allows the developer to modify the code and have an ever-updated design view of it, as well as to modify a design view and have the modification reflected in the code [9].

But this propagation of changes is far from trivial and it is an important research problem how to do this translation from source code to higher-level views and vice versa. Taking the example of class diagrams from the Unified Modelling Language (UML) [32], the question is how to consistently map entities in the class diagram (e.g., classes, attributes, associations, operations etc) to those present in the source code (classes, instance variables, methods etc). For instance, what should be done when an instance variable is added to the code? Should it be represented in the diagram as an association, as an attribute or as something else? And, conversely, how should an association class in the diagram be mapped to the source code? Should it be implemented as a regular class, as a key-value dictionary in one of the sides of the association or as something else?

Similar problems are also faced by reverse-engineering and code generation tools. A reverse-

engineering tool that extracts class diagrams from source code usually defines a general strategy for mapping code constructs to class diagrams. In an analogous way, a code generation tool needs to know what code to generate for each entity of the diagram.

Much research has been done on reverse-engineering (e.g., [33, 38, 20]) and a large amount of tools exist for generating code from higher-level views (e.g., [34, 2]). Especially in the former, popular approaches emphasize on static and dynamic analysis of programs to try to extract better abstract views from it. State-of-art tools that support round-trip engineering, such as Together [7], Rose [34] or Fujaba [28], follow advanced code generation and reverse-engineering strategies to transform diagrams into source code and vice versa.

An important problem, however, is that there is no *standardized* mapping between higher-level views and code. Even if we restrict ourselves to specific languages such as the Unified Modelling Language (UML, [32]) for describing the views, and Java [16] as implementation language, there is no unique way to perform reverse-engineering or code generation. The mappings that allow this usually depend on issues like developer expertise or company conventions. For example, a company may, for some reason, decide to use its own collection framework, and this would significantly impact the way in which associations (with multiplicity n) are represented in the code. Moreover, modelling methodologies are still evolving, as well as programming languages, so fixing strategies for mapping between both do not seem to be a good idea at all.

Therefore, the contribution of this thesis is to propose an approach a tool for round-trip engineering which makes these mapping strategies explicit, and suitable for customization. We want to provide the developer with the ability to indicate how the diagrams should be extracted from the source code and how source code should be generated from diagrams.

1.1 Context

In this section, we describe the context in which we work and some important assumptions that were made for this thesis.

1.1.1 Unified Modelling Language

The Unified Modelling Language (UML) [32] is by far the most popular standardized language for modelling object-oriented software systems. The language is still evolving and it does not yet completely satisfy all requirements for an object-oriented modelling language [9]. However, taking into account its popularity and the lack of a better *standardized* modelling language, we use UML in this thesis.

As a reduction of scope, our approach is restricted to UML *class diagrams*.

1.1.2 Logic metaprogramming

Logic metaprogramming (LMP) is the use of a logic programming language at meta-level to reason about source code. As it is confirmed by recent research (e.g., [10, 25, 23]), LMP is a very suitable approach for extracting higher-level views from object-oriented source code. Since UML class diagrams are such a kind of higher-level view, we believe that LMP is the appropriate technology to build our round-trip engineering tool.

Therefore, in this thesis, we use LMP as the underlying technology to build our tool.

1.1.3 Round-trip engineering

We believe that round-trip engineering can be achieved by the composition of code generation and reverse engineering. Whenever either diagram or code are changed, the environment is responsible for propagating such changes to the other part through, respectively, code generation or reverse engineering.

We are working in a context of round-trip engineering *without* parallel changes. That is, we do not support changes to be done in the diagram and in the code at the same time. We think that this is a rare scenario. Moreover, when possible, it should be avoided, since consistency checking between versions is an ever error-prone process.

During the rest of this thesis (specifically, in chapter 5), we show that a consistent approach to round-trip engineering can be achieved without support for parallel changes.

1.2 Approach

We divide round-trip engineering into two main parts: code generation and reverse-engineering. The implementation of these processes are independent from each other but both rely on the semantics stated in the diagram-code mapping strategies. These mapping strategies describe how each UML entity is mapped to source code, and, consequently, what are the source code patterns that imply the presence of a UML entity. For example, an instance variable `i` of type `Integer` in the code of a class can, depending on the chosen strategy, imply the presence of an attribute `a` in this class in the diagram.

Logic metaprogramming (LMP) has proven to be an adequate paradigm to reason about object-oriented source code [10]. SOUL [40] is a LMP system built on top of the object-oriented programming language Smalltalk. It provides an advanced reasoning mechanism with direct reification of entities from the underlying Smalltalk system. Given its ability to recover high level descriptions from programming language structures, we use SOUL for extracting UML

class diagrams from object-oriented source code. The identification of simple UML entities in the code, like classes or operations, is rather straightforward, but when it comes to more complicated UML constructs, the complete power of LMP has to be used to identify them in the source code.

In addition to reasoning about code, we need also to generate code from class diagrams. Besides its ability to reason about code, some experiments [8, 24] have shown that, when properly applied, LMP can provide clear and extensible code generators. These generators essentially consist of user-defined predicates which define *what* has to be generated, leaving the actual generation of code entities for a separate stage, which has a more imperative nature.

The use of the same paradigm, LMP, for both code generation and diagram extraction has some interesting advantages. For instance, while generating code using LMP, we can use LMP itself to query about the structures which are already present in the code.

1.3 Contribution

Our main contribution is a tool called Lure for doing round-trip engineering of object-oriented programs using logic metaprogramming. Lure consists of customizable engines for code generation and for UML class diagram extraction. Such customization is achieved through the use of declarative rules that state the desired mapping for UML entities in the code.

Lure allows the development of software systems in two languages: Smalltalk and Java. It is constructed using Visual Works and SOUL. The class diagrams are represented by logic rules, and facilities are provided to import them from and export them to popular CASE tools, where they can be graphically manipulated by the user. To ensure correct extraction of diagrams from code, Lure provides the facility of verifying equality between class diagrams.

In order to materialize our approach into a tool, the conception and implementation of other components needed to be carried out and some of them can be mentioned as contributions of our work too. Firstly, a generic object-oriented code generation engine was implemented which can be easily extended to generate code starting from other information than class diagrams. This generator can therefore be used independently from the round-trip tool. Secondly, to be able to reverse-engineer the source code to UML in a customizable way, we developed a Library for UML Code Reasoning. It consists of a set of predicates which use the UML-code mapping strategies to reason about source code in terms of UML entities. This library can be used independently from the round-trip tool. Using this library, the user can, for example, import a portion of source code into the system and perform a query asking what are the associations present on it.

Thirdly, we have improved the language-independence characteristics of SOUL's Library for Code Reasoning (LiCoR), making it more suitable for transparent reasoning about object-

oriented source code, especially Smalltalk and Java.

1.4 Structure

In the next chapter, we make a brief review of current research work and tools in the area of round-trip engineering, in order to better contextualize our work. In chapter 3, we introduce logic metaprogramming, the technique which is the basis for our approach to round-trip engineering. A description of how we use logic metaprogramming to perform code generation and reverse-engineering is in chapter 4. The main chapters of this thesis are chapters 5 and 6. The first contains a general view of how we do round-trip engineering using LMP, and the second contains detailed descriptions of how to declare mappings between class diagrams and source code using LMP.

Chapter 7 derives from the experiments we made to validate our approach. It describes a case study: the development of a GSM billing application application using LURE, our prototype tool for round-trip engineering.

Finally, in chapter 8, we discuss some conclusions that we achieved in this thesis and also some interesting subjects for related future work.

Chapter 2

Round-trip engineering

The term "round-trip engineering" is not new, but, in recent years, it has become very popular in the software development community. A confirmation for that can be seen in the number of tools that support this development technique (e.g., Together Control Center [7], Rational XDE [35], Gentleware Poseidon For UML [2], Fujaba [28]). This chapter performs a brief analysis of the current context of round-trip engineering and how it is related to our work.

2.1 Overview

Although round-trip engineering is a popular software development technique, there are no standards for implementing it. In fact, it seems that, in spite of its promising power and productivity increase for the developers, there has not been much research performed specific to round-trip engineering, as far as we could investigate.

Aßmann, in [3], presented a systematic method for round-trip engineering called ARE (Automatic Round-trip Engineering). He models round-trip engineering as a method of domain transformation. So, if a modification to a system is too difficult to be performed in a certain domain (e.g., text), a domain transformation can be applied that maps the system to another domain (e.g., diagrams), where the user can more easily modify this system. Given that a inverse domain transformation exists to map the system back to the original domain, then the modification has also been applied to this domain.

Using Aßmann's approach, besides simple transformation composition systems, Burbeck and Larsson propose the Model-View-Controller based Engineering (MVARE, [5]), which is an application of the well-known MVC design pattern to software construction. The *model* is the system itself, the data to be constructed. The *views* are the different views in which the system can be presented or edited (text, diagrams etc). And the *controller* is the responsible

for maintaining relations between views and model, ensuring that every change in every view is propagated to the other views and to the model. Aßmann still suggests that his approach can be applied also to aspect-oriented programming (AOP, [19]). He claims that AOP can be achieved using ARE, given that a *deweaver* exists. In other words, one view would contain the original code and separated aspects, and the other view would contain the woven code. To transform from the first view to the second one, an aspect *weaver* is used, and to transform from the second view to the first one, a deweaver is used. According to him, this can be seen as round-trip engineering.

Henriksson and Larsson, in [18], give a definition for round-trip engineering systems: *Let D be the design of the product P, let r be a reverse-engineering procedure such that $r(P) = D$ and let g be a product generation procedure such that $g(D) = P$. Iff $g(r(P)) \equiv P$ (and thus $r(g(D)) \equiv D$) then $\langle g, r \rangle$ is a round-trip engineering system.* From this on, they build some more definitions and, using the example of a relational database system, they show how round-trip engineering with synchronization of multiple views can be achieved by the formalization of the domain of these views.

Demeyer, Ducasse and Tichelaar, in [9], discuss round-trip engineering with a more pragmatic approach. They defend that round-trip engineering tools should support a tight integration between reverse and forward engineering, with an adequate representation of source code. Moreover, according to them, round-trip engineering tools never are used alone. The developer usually has a set of tools to perform design, coding, testing etc, implying in a need for interoperability among different tools. In this publication, they expose the shortcomings of UML to support round-trip engineering (mainly the absence of representations for *method invocation* and *attribute access*) and propose possible extensions for overcoming that.

Research on round-trip engineering is much related to reverse-engineering. On this area, advanced techniques have been developed to extract higher-level view diagrams, especially, UML class diagrams, from source code.

Gogolla and Kollmann [15, 21], for example, have achieved good results on reverse-engineering of Java source code to UML class diagrams. For this, they use information about the static structure of the system, and also dynamic runtime information collected using the Java debug interface (JPDA). Using this approach, they were able to discover, for example, fine-grained details about class associations, such as association classes and aggregations.

2.2 Tool review

The construction of IDEs is a fast-evolving industry area and, from simple class diagrams to detailed sets of design descriptions, round-trip engineering seems to be a compulsory feature to make a tool competitive. In this section, we briefly evaluate three tools with support to round-

trip engineering, trying to get a hint about the real current state of round-trip engineering tools in the industry. The tools we evaluated were Borland Together Control Center [7], Fujaba [28] and Gentleware AG Poseidon For UML [2].

Criteria

More than theoretical, this evaluation is practical. We tried to analyze the features we regard as most important for round-trip engineering tools and IDEs in general. We used Java as programming language and UML for design language, giving emphasis to class diagrams.

Our test case is a small Java application for the billing of GSM carriers, called *GSMCase* [1]. The application is simple and has 13 classes with 15 attributes, 18 operations, 14 associations and 4 generalizations. The associations with multiplicity N are represented by Java collections, as it is common, and the accessors follow usual Java naming conventions (get- and set-).

The first activity we performed in all tools was to import the code of GSMCase. After that, we performed some iterations of modification in the code and in the diagram, analyzing the facilities provided by the tool. The items in which we focused our attention were:

Reverse-engineering How powerful is the reverse engineering technique used by the tool. For example, does it recover association ends correctly, or how detailed is the extraction.

Customization As we pointed out in the introduction of this thesis, we regard the customization of round-trip engineering tools as a major concern. So we analyzed the possibility of customization offered by each tool in three different aspects: code generation (forward engineering), reverse engineering and round-trip engineering itself (synchronization etc).

Supported diagrams Which are the diagrams supported by the tools. For example, class diagrams, use cases, collaboration diagrams etc.

Round-trip synchronization How tightly integrated are model and source code editing. For example, if the environment synchronize everything automatically, or if the user has to request every synchronization.

Tagging If the tool interferes in the source code putting tags to express design information.

Variable abstraction level One of the main problems of reverse-engineering is filtering irrelevant information. Since the developer wants to use the design model to get a higher-level view of the system, too much fine-grained diagrams are not suitable for this purpose. In this item, we verify if the diagram view can be filtered to show only the relevant information and, if so, up to which level.

[1] This same application was used for all the experiments of our approach, and will be described in chapter 7.

Programming language support We evaluated the round-trip features for Java, but we also verified what are the languages supported by each IDE.

Model editing We tried to determine how powerful and complete is the user interface for UML editing, and how these editing operations affect the code.

Code editing What are the facilities provided for code editing and how they are related to the UML model.

2.2.1 Together Control Center

Together [7] was probably the first popular tool to support UML round-trip engineering for Java. The product has evolved and now belongs to a traditional producer of Java IDEs, Borland Software Corporation. The version evaluated, 6.1, integrates development from analysis to deployment in one tool.

Together implements the typical idea of round-trip engineering. As it can be seen in figure 2.1, it provides a full-fledged code editor together with an always up-to-date diagram view of it. This kind of environment seems to be quite handy for the developer, since, for example, the addition of an association in the code immediately creates that association also in the diagram, where the developer can have a more adequate view of the impact of this change in the global picture of the system.

Let us list the items we evaluated in Together Control Center.

Reverse-engineering Recovered all classes, attributes and operations. Recovered only associations with multiplicity `1`. Accessors and mutators of instance variables were directly mapped to properties, following naming conventions (get- and set-). From the associations, only the end classes and the direction were extracted. Features such as roles and multiplicities must be edited manually by the user.

Customization Together is a very configurable IDE. Its options dialog has a large amount of items allowing the configuration of low-level details of the IDE. However, the IDE provides limited configuration about how the source code is mapped to diagram and vice-versa. There are simple templates for instance variable comments (tags) which help the environment to identify the characteristics of the association represented by this variable.

Supported diagrams Together supports eight types of UML diagrams: class, use case, sequence, collaboration, state chart, activity, component and deployment. Apart from that, it provides another ten diagrams for different purposes (e.g., web application, entity-relationship).

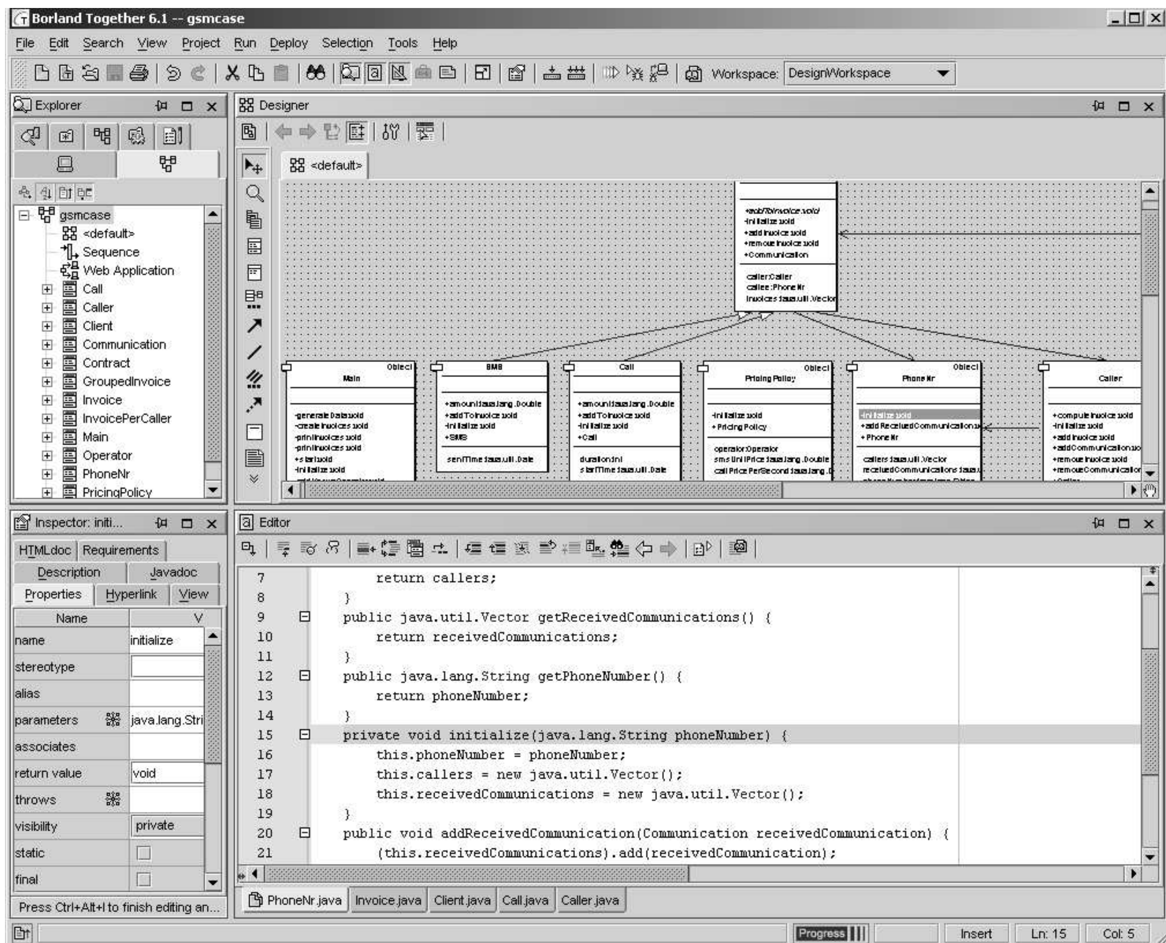

Figure 2.1: Round-trip engineering in practice with Together Control Center.

Round-trip synchronization Together performs automatically all the synchronization between code and diagram. When the user modifies existing code in the IDE or in another program, or imports new classes into the system, these changes are immediately reflected in the design view.

Tagging Together tags the source code to identify meta-information about the diagram. For example, when the user changes an association to be an aggregation, a tag is created in the instance variable which represents this association stating: `@link aggregation`.

Variable abstraction level Together has many configurations to allow the user to hide specific constructs, such as attributes, dependencies and links to self, from the diagram. It provides three pre-defined diagram detail levels: analysis, design and implementation.

Programming language support Besides Java, Together supports C++, C# and Visual Basic.

Model editing Together contains a quite complete diagram editor called Designer, which allows most of the graphic manipulation of all the supported diagrams.

Code editing Together provides some usual coding facilities like auto-complete, formatting and simple optimizations. When the code (the Java file) is saved the modifications are propagated to the diagram.

2.2.2 Fujaba

Fujaba [28] is an open source UML CASE tool project started at the software engineering group of Paderborn University in 1997. Fujaba is the acronym for "From Uml to Java And Back Again" and originally aims to provide an environment for round-trip engineering using UML as visual programming language. During the last years, the environment has become a base for several research activities, e.g., distributed software, database systems, modelling mechanical and electrical systems.

As an IDE, Fujaba has limited functionalities, however its internal architecture and abilities to re-engineer and generate source code have been subject to many publications (e.g., [29, 27, 30]). Besides class diagrams, Fujaba can generate code from collaboration and state chart diagrams, as well as to extract diagrams from its generated code.

The version we evaluated was Fujaba 4.0.0alpha. A list of the observed items follows.

Reverse-engineering All the classes, attributes and operations were extracted. However, surprisingly *no* association was recovered from the code. Accessors and mutators were considered regular methods. The only features extracted from the attributes were the ones that can be represented by Java modifiers (final, static etc). Besides a class diagram, Fujaba extracts activity diagrams for each method of the system.

Customization Fujaba does not provide configurations on how to map source code to diagram.

Supported diagrams From UML, Fujaba supports class diagram, state chart diagram and activity diagram. Apart from that, it provides also an interesting kind of diagram, called Dobs ([14]), which allows the visualization of memory objects as a graph.

Round-trip synchronization When a construct is added to (modified in or remove from) the diagram, the environment automatically propagates these changes, reorganizing the patterns that represent the modified construct in the code.

Tagging Fujaba makes no tagging comments in the source code.

Variable abstraction level Fujaba offers no configurations for setting the abstraction level of the design view. The only related functionalities are hiding all attributes and/or all operations of specific classes.

Programming language support Fujaba is specific to the Java language.

Model editing The design-level editing of Fujaba is still evolving and is currently quite limited. It does not support, for example, aggregations, compositions or association classes.

Code editing For code editing, Fujaba provides a simple text editor with syntax highlighting.

2.2.3 Poseidon

Gentleware AG Poseidon For UML [2] was derived from an open-source tool called ArgoUML [1], and its original purpose is UML modelling. Among the evaluated tools, it is the one which best adheres to the UML specification, providing, for example, all the standard features of associations and attributes (e.g., changeability values, ordering). Recently, the tool has evolved and become very popular. The UML editing is now extensible through an architecture of plug-ins. One of the plug-ins that is offered by the tool producers themselves serves round-trip engineering.

Poseidon does not aim to be a complete IDE as their focus is on UML modelling, so the round-trip plug-in works as a watcher for source files. These source files can be edited by Poseidon itself using its internal editor or by other tools. Poseidon maintains the UML diagrams consistent with the source code in the files and, if the diagrams are modified, source code can be re-generated.

Let us show the observations we made while evaluating Poseidon For UML Professional Edition version 1.6.1_02.

Reverse-engineering Poseidon recovered all classes, attributes and operations. It recognized only associations with multiplicity 1. In cases where there were two association ends between the same classes but with inverted direction, it merged the ends into a single association.

Customization With respect to reverse-engineering, Poseidon allows simple configurations of the mapping, such as the option to model Java attributes as UML associations or as UML attributes. For code generation, it contains files with code templates which can be edited by the user to customize, for example, the way accessors are generated.

Supported diagrams Class, Sequence, Collaboration, State chart, Activity, Use Case and Deployment/Component/Object diagrams.

Round-trip synchronization Poseidon does not automatically re-generate code when the diagram is changed, user intervention is needed for that. For reverse-engineering, Poseidon monitors the source code files with a configurable interval and, in case of change, the diagram is updated.

	Classes	Generalizations	Associations	Attributes	Operations
Originally	13	4	14	15	18
Together	13	4	11	28	66
Poseidon	13	4	9	6	123
Fujaba	13	4	0	28	128

Table 2.1: UML constructs extracted from the code of GSMCase.

Tagging Poseidon does not use any special tags in the source code.

Variable abstraction level Poseidon provides a set of options to hide diagram constructs (e.g., accessors, operation parameters, private features).

Programming language support Poseidon is specific to Java.

Model editing Poseidon has a very comfortable and intuitive graphic user interface to edit diagrams.

Code editing Poseidon provides a simple Java source code editor with syntax highlighting.

2.3 Discussion

As we could see, the three tools we reviewed have quite different approaches to round-trip engineering, especially with respect to the way they extract a UML diagram from existing code. Of course, one cause for this is that they focus on different goals: Fujaba is a proof of concept, a tool that is derived from academic research on round-trip and reverse-engineering; Poseidon is focused on UML modelling, providing round-trip engineering as a plug-in; and Together is a tool originally directed to provide an environment for round-trip engineering.

However, even having different purposes, these tools are related to the same subject: mapping source code to UML diagrams. Each one of them has its own approach for doing that. Fujaba uses strict code conventions and programming styles for identifying UML constructs in the code. If the code adheres to this conventions and styles, Fujaba can recognize advanced patterns of code and reconstruct detailed class diagrams. But this patterns are very particular to Fujaba, so using it to import “legacy” code or simply code that uses different conventions may cause Fujaba round-trip engineering to fail.

Poseidon extracts class diagrams from code using simple strategies. For example, all methods are extracted as operations or all attributes with primitive types are extracted as attributes. It also uses some heuristics for merging association ends. Together also uses simple code-diagram mappings.

Table 2.1 shows a more concrete view of the results of the evaluated tools for doing reverse-engineering. The line labelled as *Originally* shows how many UML constructs really exist in the code. The other lines of the table list how many instances each tool extracted for the different kinds of UML constructs contained in the code of GSMCase This shows some relevant discrepancies between the reverse-engineering strategies. For example, Together extracted eleven associations while Poseidon extracted 9 and Fujaba did not extract any. In fact, none of these represent the real number of associations in GSMCase, since none of the tools was able to identify associations with multiplicity N represented by collection classes. Poseidon recognizes only *arrays* as N-multiplicity associations and Together needs special tags to mark collections as associations.

Another discrepancy in this simple evaluation was the number of operations. Together brings the diagram to a more abstract level, transforming all variables with accessors into properties with read-only, write-only or read-write access. This reduces significantly the number of operations. The evaluated version of Poseidon recognizes accessors only for attributes with primitive types, leaving the ones for collections and associations displayed as regular methods. Fujaba did not distinguish accessors and regular methods.

All this variance among the reverse-engineering mappings of these tools is aggravated by the fact that all of them support very limited configurations regarding code-diagram mappings. Fujaba, for example, has a very advanced mechanism for identifying associations and their features, but if its strict code conventions are not followed, this identification of associations are totally useless.

Conclusion

In this chapter, we have seen a part of current context of round-trip engineering. We mentioned some research work on the area and reviewed three popular tools with round-trip engineering support.

As we discussed in the previous section, different round-trip engineering tools have different approaches to map source code to UML diagrams and vice versa. Each tool imposes its style of representing the source code as diagrams, since none of them gives appropriate configurations with respect to their way of mapping between code and diagram. This enforces the developers have to adapt themselves to the tools, and not the other way around as it is expected.

In the next chapter, we introduce logic metaprogramming (LMP), a technique for reasoning about object-oriented source code. In the following chapters, we show how LMP can be used to define mapping strategies between source code and class diagrams. LMP has the great advantage of being a declarative approach, which is very well suited for stating this kind of mapping, making strategies clearly readable. Moreover, the ease for declaring strategies using

LMP makes it more feasible the customization of mapping strategies accordingly to the needs of the developer, or for LMP-experienced users to conceive their own strategies by themselves.

Chapter 3

Logic metaprogramming

Logic metaprogramming (LMP) is used by our tool (Lure) as the paradigm to support round-trip engineering. In this chapter, we give an overview of LMP, describing its motivation and principles.

3.1 Logic metaprogramming

Class-based object-oriented programs are usually structured as a complicated web of intercommunicating classes. Developers who want to use or improve these programs need to understand how they are organized. As programs get bigger, this is a very difficult task, since source code contains a huge amount of very low-level detailed information, not allowing the developer to get a higher-level view of the design.

Some techniques for program documentation and design provide global overviews of the program, but mostly without linking it to the actual implementation. Design patterns [13] provide design information without completely ignoring implementation aspects. However, they are only available on paper in a reference guide. Cookbooks or tutorials provide only a very limited view and are more targeted towards learning the basic features of the system.

Other development concerns also raise the need for a mechanism to extract — and ensure — well-formed program structuring. For example, some programming conventions (e.g., 'every persistent class should have a *store* method') represent the only correct way in which the system can be implemented, and their violation is an error. These errors have to be dealt with and non-trivial programming conventions can be quite difficult to verify.

Another motivation for the extraction of a consistent design-level view from the software system is the analysis of change impact. How would a local change on the source code impact on the

global design view? Would a certain design change be feasible given the current source code structure?

The common issue in the mentioned problems is the incapability to easily express high-level information about the program. To solve this, LMP introduces a logic programming language as meta-language to express and reason about the structural information of software systems. [40]

Logic metaprogramming (LMP) is the name given to a particular kind of multi-paradigm programming. The starting point for LMP is an existing development environment for an object-oriented programming language augmented with a declarative meta layer. [10]

This declarative layer is the implementation of a Prolog-like logic language which allows the reasoning about the underlying system. The prefix "meta" is put before programming because the logic language is used at meta-level with respect to the source code it reasons about.

The typical use of LMP is to declare logic rules and facts about the source code and perform queries on the logic inference engine to extract information about this code. LMP implementations have a strong relation with the underlying system, using it as part of the knowledge base which is queried.

3.2 Logic metaprogramming with SOUL

SOUL [41, 40] is a LMP implementation built on top of Smalltalk VisualWorks[6]. It provides the basic features of a logic language and allows the direct reification of the objects from the underlying Smalltalk system.

The syntax of SOUL is much like Prolog [36], therefore it can be used to solve regular logic programming examples. Table 3.1 shows a simple Prolog example [4] and the correspondent code using SOUL. The main difference is the notation for variables: in Prolog, they are capitalized symbols; in SOUL, they start with a question mark (`?`).

This example declares four *facts* stating direct flights between two cities. [1] Then, two *rules* are declared expressing what are the conditions for the existence of a flight (connection) between two cities: either there are direct flights from the first city to the second one or the first city has direct flights to some city which has connection to the second one. Finally, two examples of queries are given: the first one asks if there are flights from Amsterdam to Fairbanks; and the second one asks where can one fly to from Seattle. If we execute these queries, the first query succeeds with `YES` (or `true`, in SOUL), and the second one succeeds with two solutions, informing the names of the cities Anchorage and Fairbanks.

[1] This example and the rest of this thesis assume that the reader has basic knowledge about logic programming or Prolog.

	Prolog

Facts	`direct(amsterdam, seattle).`
	`direct(amsterdam, paramaribo).`
	`direct(seattle, anchorage).`
	`direct(anchorage, fairbanks).`

Rules	`connection(X, Y) :- direct(X, Y).` `connection(X, Y) :- direct(X, Z), connection(Z, X).`
Queries	`?- connection(amsterdam, fairbanks).` `?- connection(seattle, X).`
	SOUL

Facts	`direct(amsterdam, seattle).`
	`direct(amsterdam, paramaribo).`
	`direct(seattle, anchorage).`
	`direct(anchorage, fairbanks).`

Rules	`connection(?x, ?y) if direct(?x, ?y).` `connection(?x, ?y) if direct(?x, ?z), connection(?z, ?x).`
Queries	`if connection(amsterdam, fairbanks).` `if connection(seattle, ?x).`

Table 3.1: Prolog example and SOUL correspondent.

```
1st solution: {?x → Anchorage}
2nd solution: {?x → Fairbanks}
```

Interacting with Smalltalk

An important feature of SOUL is the manipulation of objects from the underlying Smalltalk system. Smalltalk code can be inserted into the declarations of terms and this code is evaluated when the query is executed. Let us discuss the following SOUL logic query:

```
if equals(?x, [ OrderedCollection new ]).
```

The predicate `equals` performs the logic unification of two values. Expressions between brackets (‘`[`’ and ‘`]`’) are blocks of Smalltalk code, and the value of these expressions are the result of the evaluation of the block. This means that `[ OrderedCollection new ]` evaluates to an instance of the class `OrderedCollection`, and this is the value bound to the variable `?x` in the only solution for this example query:

```
1st solution: {?x → anOrderedCollection}
```

Besides their use as terms, blocks of Smalltalk code can be used also as clauses. In this case, the blocks are required to always return either true or false. Let us consider the following simple rule which verifies if an integer is odd:

```
odd(?anInteger) if
  [ ?anInteger odd ].
```

The clause [?anInteger odd] calls the method odd of that object, which is supposed to be of type Integer. This method returns true or false, implying, respectively, the success or failure of the query.

Another point to notice in the previous example is the insertion of the logic variable ?anInteger inside the Smalltalk code. This is the way Smalltalk receives the values from the logic engine. The variable is replaced by the value to which it is bound at the time of clause evaluation.

Other features

As mentioned, SOUL provides the basic features of logic languages. In order to help the reader to understand the examples throughout this thesis, let us describe some of them.

Lists Similarly to Prolog, SOUL contains a native data structure called *list* which serves for the manipulation of sequences. As notational facilities, the symbols ‘<’ and ‘>’, for list delimitation, and ‘|’, for head and tail separation, are provided.

The following example shows the extraction of the first element of a list.

```
if equals(<?head | ?tail>, <a, list, of, five, elements>).
```

The result of this query would have the variable ?head bound to the value a, and the variable ?tail bound to the rest of the list, <list, of, five, elements>.

SOUL provides also helper predicates for list manipulation. Some examples of these are:

- append(?aList, ?anotherList, ?concatenatedList): ?concatenatedList is the concatenation of ?aList and ?anotherList. This predicate can be used to form the list ?concatenatedList or it can be used to find ?aList and/or ?anotherList from a given ?concatenatedList.
- list(?aList) for testing if ?aList is a SOUL list.
- member(?x, ?aList) for testing if ?x is an element of the list ?aList. This predicate can also be used to generate multiple solutions with ?x bound to a different element of the list in each of them.

- `noDuplicates(?aList, ?result)` for removing all the duplicates from `?aList` and storing the result in `?result`.
- `reverse(?aList, ?reversedList)` for reversing the order of the elements of the list `?aList`.
- `sameElements(?aList, ?anotherList)` for verifying whether `?aList` contains the same elements as `?anotherList`, regardless of their order in the list.

Smalltalk collections can be used transparently as SOUL lists. The result of the following query has three solutions, in one of them `?x` would be bound to 1000, in the other, to 2000, and in the other, to 3000.

```
if member(?x, [ Array with: 1000 with: 2000 with: 3000 ]).
```

The above query is equivalent to:

```
if member(?x, <1000, 2000, 3000>).
```

This kind of behavior gives some hints about how powerful is the ability of SOUL to directly reify objects from the Smalltalk system.

Functors A functor is another type of data structure which can be used in SOUL. Its syntax is the same as in Prolog. The following example uses a functor to express the information about a person, containing its name and gender. The third predicate is a rule which consults the gender of the parent of a person.

```
parentOf(person(Marcheline, female), person(Angelina, female)).
parentOf(person(Jon, male), person(Angelina, female)).
fatherOf(?father, ?someone) if
  parentOf(?father, ?someone),
  equals(?father, person(?fatherName, male)).
```

Quoted Code As we will see on chapter 4, SOUL can be used for generating any kind of source code. For that purpose, a special notation is provided for making code concatenation much simpler. The quoted code is put between curly braces (‘`{`’ and ‘`}`’) and logic variables can be used inside it. These variables are replaced by a string representation of the value to which they are bound in evaluation time, similarly as for blocks of Smalltalk code.

The following rule generates the code for a Java instance variable accessor:

```
generateAccessor(?TypeName, ?VariableName, ?Code) if
  capitalizedStringOf(?CapitalizedVariableName, ?VariableName),
  equals(?Code, {public ?TypeName get?CapitalizedVariableName() {
    return ?VariableName;
  }}).
```

The predicate `capitalizedStringOf` is used here to capitalize the name of the instance variable, in order to follow the usual convention for Java accessors.

So, when the following query is executed:

```
if generateAccessor(Person, caller, ?Code).
```

one solution is returned, in which the variable `?Code` is bound to the quoted code:

```
public Person getCaller() {
  return caller;
}
```

More syntax Here are some other syntax constructs of SOUL:

- The symbols `+` and `-` before a variable in the header of a rule mean that this variable must be, respectively, bound and unbound. This is often used when the variable must be substituted in a block of code, situation in which it must be bound. If the following rule is queried with `?anInteger` as an open (unbound) variable, it fails even before the clause `[ ?anInteger odd ]` is evaluated.

  ```
  odd(+?anInteger) if
    [ ?anInteger odd ].
  ```

  This is equivalent (syntactic sugar) to:

  ```
  odd(?anInteger) if
    atom(?anInteger),
    [ ?anInteger odd ].
  ```

- A single question mark (`?`) as variable name means that that term is not important at this point. It is the correspondent in SOUL to the underscore variables (‘`_`’) in Prolog. Every occurrence of `?` in the same rule represents another unique unbound logic variable. In this example, we are not concerned about the name of the father:

  ```
  fatherOf(?father, ?someone) if
    parentOf(?father, ?someone),
    equals(?father, person(?, male)).
  ```

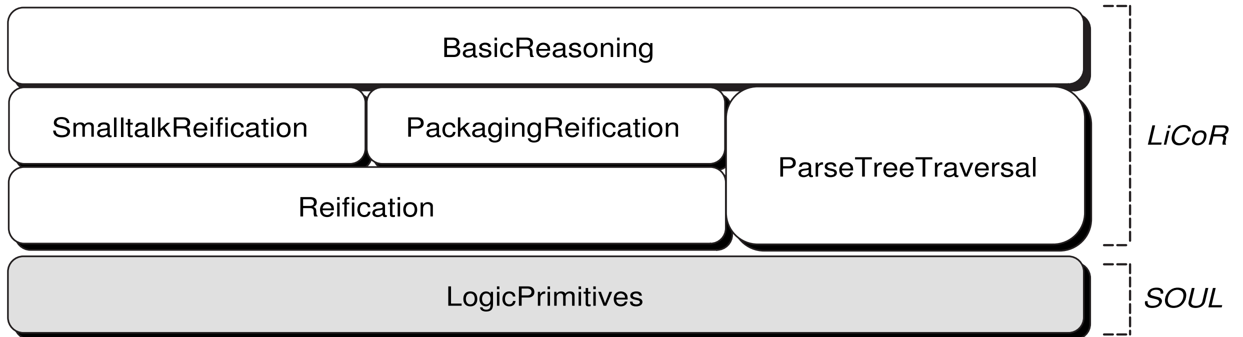

Figure 3.1: The structure of LiCoR.

3.2.1 Library for Code Reasoning — LiCoR

The main application of SOUL is source code reasoning. For this purpose, a Library for Code Reasoning (or LiCoR, for short) is provided. This library consists of a set of logic predicates to reason about Smalltalk code.

Originally, these predicates were a part of SOUL [41], but afterwards they were factored out in a separate package and baptized as LiCoR. In the experiments for this thesis, we used NewLiCoR, which is a refactored and optimized version of the previous one.

LiCoR has a layered structure. The lower layers contain the more primitive and fine-grained predicates, and the higher layers use the lower ones to create more abstract reasoning predicates. Figure 3.1 illustrates this structure.

Let us discuss the characteristics of each of these layers, from the lowest to the highest level:

LogicPrimitives This layer contains general logic predicates, such as `findall`, `forall`, `not`, `append`, `list` etc. Actually, it is not a part of LiCoR but of SOUL.

Reification This layer represents the predicates which reify entities from the underlying language. It is split up in two other layers: SmalltalkReification and PackageReification.

SmalltalkReification This layer does the actual interaction with the classes of Smalltalk, reifying classes, methods, instance variables etc. Two examples of predicates from this layer are `class` and `methodInClass`.

PackagingReification This is an environment-specific layer which deals with the packaging constructs of VisualWorks (name spaces, bundles, protocols etc). Example of predicates: `bundle` and `classInCategory`.

ParseTreeTraversal This layer contains predicates to deal with the parse tree of methods. The parse tree of a method is represented through a functor and the predicates of this layer traverse this functor looking for specific information, such as assignments, temporary variables etc.

Using LiCoR

The predicates of LiCoR can be used for verifying entities or relations, or for extracting them. The predicate `class`, for example, can be used for verifying if a certain value is a class or for getting all classes from the system. Let us consider the four following queries:

```
if class([ String ]).
if class([ String new ]).
if class(String).
if class(?class).
```

The first one verifies if `[ String ]` is a class. This query succeeds, because the evaluation of this Smalltalk block returns the class `String`. The second one fails, since the value returned by the block is not a class, but an instance of the class `String`. The third query also fails, because the value passed to the predicate is merely a symbol and not a Smalltalk class [2]. The last query succeeds with a huge amount of solutions, since there is a solution for every class of the system:

```
1st solution: {?class → [ Array ]}
2nd solution: {?class → [ String ]}
3nd solution: {?class → [ Object ]}
...
```

The layers of LiCoR provide a rather complete set of predicates to reason statically about object-oriented source code. Some of these predicates are described in table 3.2. Most of them can be used both to verify or to lookup entities and relationships in the way we saw in the previous example. For example, the predicate `methodInClass(?M, ?C)` can be called with both variables bound and, in this case, the predicate verifies if the class bound to `?C` contains the method bound to `?M`. If the predicate is called with both `?M` and `?C` as unbound variables, it returns a solution for each method `?M` in the system, associated with its corresponding class `?C`. The predicate can also be called with only one of the variables bound. In case `?M` is the bound one, this returns the class of the method bound to `?M`. In case `?C` is the bound one, all the methods of the class bound to `?C` are returned.

[2] `class(String)` is equivalent to `class([ #String ])`.

Representational Mapping Predicate	Description
`class(?C)`	`C` is a class
`instanceVariableInClass(?V,?C)`	`V` is an instance variable of class `C`
`subclassOf(?C,?P)`	class `C` is a subclass of class `P`
`superclassOf(?C,?P)`	class `C` is the superclass of class `P`
`methodInClass(?M,?C)`	`C` implements the method `M`
`methodWithName(?M,?N)`	`M` is a method called `N`
`abstractMethodInClass(?M,?C)`	`C` implements an abstract method `M`
`argumentsOfMethod(?A,?M)`	`A` is the list of the arguments of `M`
`temporariesOfMethod(?A,?M)`	`A` is the list of the temporary variables of `M`

Table 3.2: Representational Mapping Predicates

A more complex example

To finish this section about LiCoR, let us take a look at a more complex example of its use [39], which demonstrates the power and simplicity of using LMP to reason about code. This example checks the system for all the method parameters which are not being used in its whole class hierarchy. In this case, LMP helps the developer which may not have realized yet that a certain parameter of a method is not being useful and can be removed. This is a way of discovering a very specific kind of "dead code".

Detecting whether a formal parameter is not used by a method boils down to checking whether the method itself and none of its overriding or overridden methods uses this parameter. We can use the following two logic rules to implement such an algorithm:

```
unusedParameter(?class, ?selector, ?parameter) if
  methodInClass(?selector, ?class),
  parameterOf(?parameter, ?selector, ?class),
  forall(hierarchyImplements(?subclass, ?selector, ?class),
    not(selectorUsesParameter(?subclass, ?selector, ?parameter))).
```

First, we retrieve the methods implemented by a given class (`methodInClass` predicate), and we retrieve all the parameters of each method by means of the `parameterOf` predicate. For each class in the hierarchy of the given class (including the class itself) that implements the given method (gathered by means of the `hierarchyImplements` predicate), we check if it uses the given parameter or not, using the `selectorUsesParameter` predicate. The `selectorUsesParameter` predicate itself is implemented as follows:

```
selectorUsesParameter(?class, ?selector, ?parameter) if
```

```
methodWithNameInClass(?method, ?selector, ?class),
parseTreeOfMethod(?parsetree, ?method),
parseTreeUsesVariable(?parsetree, ?parameter).
```

It uses a variant of the `methodInClass` predicate, the `methodWithNameInClass` predicate, which returns the method identified by the class and the selector. The parse tree of this method is obtained through the predicate `parseTreeOfMethod`. The `parsetreeUsesVariable` predicate (from the `ParseTreeTraversal` layer of LiCoR) is then used to traverse this parse tree and look for uses of the specified parameter.

All the predicates from this example, except `unusedParameter` and `selectorUsesParameter`, are part of LiCoR.

3.2.2 SOULJava

SOUL and LiCoR were build using Smalltalk and can reason only about this language. However, one of our research goals was to build a round-trip engineering tool which supports more than one language, namely Smalltalk and Java. To do that, we use an add-on to SOUL called SOULJava [12]. SOULJava takes advantage of the layered structure of LiCoR to replace some components and provide the ability to reason about Java using SOUL.

LiCoR is organized in such a way that its predicates do not interact directly with the metaclasses of Smalltalk to get information about them. Instead of this, all the meta-level calls are delegated to a Meta-Level Interface (MLI), which is responsible for consulting the code repository for such information. In the case of Smalltalk, this code repository is the Smalltalk image itself.

To enable the reasoning about Java, SOULJava swaps the MLI of LiCoR replacing it for a one that is made to reason about Java. This MLI is connected to a code repository which it stores information about Java source code. This structure is shown in figure 3.2.

SOULJava also provides a Java parser which enables the import of external Java code, and other environment utilities, such as a Java code browser and editor.

Conclusion

As seen in this chapter and in the cited publications, LMP is a very suitable technology for reasoning about the structure of object-oriented systems. The extraction of higher-level views from source code can be clearly performed using or extending the predicates of LiCoR.

Since the subject of this thesis, round-trip engineering, is exactly about the management of

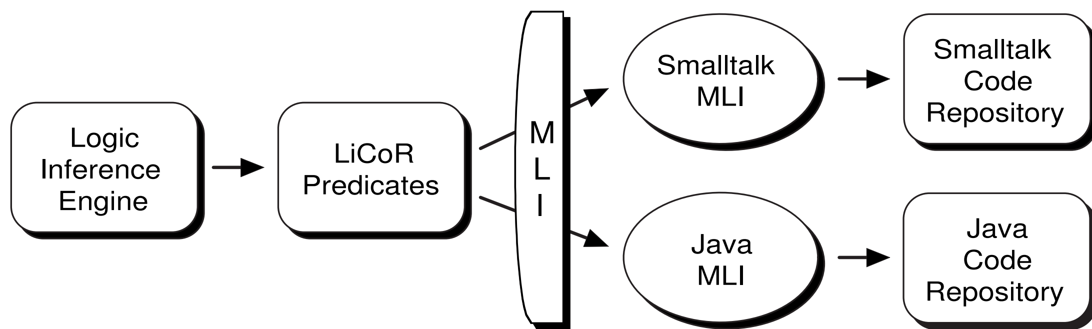

Figure 3.2: Swapping of the MLI of LiCoR to Java.

a specific kind of higher-level view of source code, UML diagrams, LMP seems to be a very appropriate choice as our main paradigm.

In the next chapter, another use of LMP is shown: code generation. The union of the ability of LMP to extract higher-level views from the code with its ability to generated code are the two main foundations of our approach to round-trip engineering.

Chapter 4

Code generation using LMP

Our approach for round-trip engineering has two foundations: code generation and reverse-engineering. As we will discuss more deeply in chapter 6, reverse-engineering can be achieved using the abilities of logic metaprogramming to extract design-level views from source code that we saw in the previous chapter. In this chapter, we describe how we use logic metaprogramming — in particular, SOUL — to generate code from design descriptions.

This chapter is mainly based on some experiments we conducted in [24].

4.1 Overview

The idea of using logic metaprogramming for generating code is not new; it was explored in [8] under the exotic name TyRuBa. TyRuba is a LMP system designed for generating Java code. SOUL, the LMP system which we use in this thesis, was originally conceived for reasoning about the structure of Smalltalk programs [40]. Today, the two systems have grown much closer. SOUL has taken inspiration from TyRuBa and has included similar code generation constructs, while TyRuBa has been extended to allow better reasoning about existing program code. Until recently, the only essential remaining difference was the target language they are considering: Java for TyRuBa and Smalltalk for SOUL. However, even that difference has disappeared with the implementation of SOULJava, which, as presented in the previous chapter, is an add-on to SOUL for reasoning about and generating Java code.

Both in TyRuBa and SOUL, a code generator is written as a logic program (i.e., a set of logic facts and rules) that builds an object-oriented program. In SOUL, for example, this is not a difficult task, since SOUL is built on top of a Smalltalk system and we can easily interact with the highly reflective nature of this system to add, remove or modify classes. To provide similar behavior in Java, SOULJava contains its own code repository, which is manipulated through the

predicates of LiCoR, allowing addition, removal or modification of Java code constructs. From now on, we will focus on the use of SOUL and SOULJava to generate code, since these are the tools we use in our thesis.

The code generation using SOUL consists of a set of logic predicates which form a layer of the Library for Code Reasoning (LiCoR) called CodeGeneration. This logic layer is a façade for the multi-language meta-level interface of LiCoR, which currently allows code generation for Smalltalk (default) and Java (SOULJava add-on). These predicates normally receive a number of parameters describing the entity to be generated (or removed, or modified) and succeed if, and only if, the operation was successfully performed.

For example, for generating a class, one can use the simplest predicate for that purpose executing the following logic query:

```
if compileClass(Foo).
```

This adds a class named `Foo` in the current code repository. In the case of Smalltalk, this adds the class to the Smalltalk image. In Java, using SOULJava, the class is added to the internal Java code repository.

4.2 The *pull* approach

In this section, we discuss the structure of our code generation engine.

If appropriate care is not taken, generating code with a logic metaprogramming language can be tricky. This is because the reasoning subject changes while the logic program is being executed: the logic program reasons about the program code while this program code itself is being changed by code generation. One way to handle this is to arrange all the logic rules for code generation in such a way that their evaluation sequence is ensured. This is not recommended as a good logic programming strategy, given that such an approach is quite imperative and would be spread out throughout the logic code, making it very difficult to be managed. Moreover, this conflicts with the declarative nature of logic programming, in which *what* is preferred rather than *how*. [4] The developer of a logic program should state what has to be generated, not having to worry about how it is generated.

Therefore, to make a more appropriate use of SOUL for code generation, we use a different approach that separates the generation process in two well-defined phases, collecting the logic query part in one phase and the imperative part in the other. The major advantage of this is that we can cleanly write the actual generation code in a completely declarative way. We call this the *pull* approach. [1]

[1]The term "pull" comes from the fact that the information is pulled in the first phase.

The logic rule that implements our general code generation engine is the parameterless logic predicate below:

```
generateCode if
  accumulate(classToGenerate, ?Classes),
  accumulate(instanceVariableToGenerate, ?InstanceVariables),
  accumulate(methodToGenerate, ?Methods),
  doAll(generateClass, ?Classes),
  doAll(generateInstanceVariable, ?InstanceVariables),
  doAll(generateMethod, ?Methods).
```

The first phase (the first three lines) queries the repository to accumulate all necessary information about the entities to be generated. The second phase (the last three lines) takes this accumulated information and actually generates the program code for each of these entities. This separation is especially useful when the code is being updated rather than being generated, because the current code can be freely consulted by the accumulation phase without needing to care about possible side-effects of the generation phase. The implementation of the general `accumulate` and `doAll` predicates is given below. The former takes as argument a `?Query` that returns, one by one, all entities of a certain kind (classes, instance variables, methods) to be generated, and groups all these results in a `?List`. The latter iterates over this list and, for each entity in that list, invokes the appropriate code generation `?Action` for that kind of entity. Their code follows:

```
accumulate(?Query, ?List) if
  findall(?Argument, ?Query(?Argument), ?List).

doAll(?Action, ?List) if
  forall(member(?El, ?List), ?Action(?El)).
```

To make the logic program above work, two more things are needed: for each kind of entity to be generated, a predicate that knows how to generate code for that entity, and a predicate that tells us which are all the entities of that kind to be generated. The former is defined below and basically delegates the responsibility on how to generate entities to the predicates in the CodeGeneration layer of LiCoR:

```
generateClass(<?ClassName, ?SuperClassName, ?NamespaceName>) if
  namespaceWithName(?Namespace, ?NamespaceName),
  compileClass(?ClassName, ?SuperClassName, ?NamespaceName).
generateClass(<?ClassName, ?SuperClassName>) if
  compileClass(?ClassName, ?SuperClassName).
generateClass(<?ClassName>) if
```

```
    compileClass(?ClassName).

  generateMethod(<?ClassName, ?MethodCode, InstanceMethod>) if
    classWithName(?Class,?ClassName),
    compileMethod(?Class, ?MethodCode).
  generateMethod(<?ClassName, ?MethodCode, Constructor>) if
    classWithName(?Class,?ClassName),
    compileConstructor(?Class, ?MethodCode).

  generateInstanceVariable(<?ClassName, ?InstVarType, ?InstVarName, ?Visib>) if
    classWithName(?Class, ?ClassName),
    compileInstanceVariable(<?Visib, ?InstVarType, ?InstVarName>, ?Class).
```

To customize this generic code generation engine to create a specific code generator, the only thing remaining to be done is to provide an implementation for each of the unary logic predicates `classToGenerate`, `instanceVariableToGenerate` and `methodToGenerate` that declare the classes, instance variables and methods to be generated. These predicates have the following form:

```
classToGenerate(?Description)
instanceVariableToGenerate(?Description)
methodToGenerate(?Description)
```

The parameter `?Description` of each of these predicates will be bound to a list describing the entity to be generated. For classes, it is a list `<?ClassName, ?SuperClassName>` consisting of the name of the class to be generated and the name of the superclass of which it should inherit, or a list `<?ClassName>` containing only the name of the class if it is a root class. Another alternative form is `<?ClassName, ?SuperClassName, ?NamespaceName>`, which takes also the name of the namespace in which the class should be generated. For methods, we need the class name and method code: `<?ClassName, ?MethodCode>`. For instance variables, we need a list consisting of the class name, the type of the instance variable, its name and visibility of the variable: `<?ClassName, ?InstVarType, ?InstVarName, ?Visib>`. Upon generation, the meta-level interface for Java uses the given type declaration when adding a variable, while the meta-level interface for Smalltalk discards this information.

This code generation engine cleanly separates the code generation algorithm from the declaration of the entities to be generated, which has also advantages for composability of code generators. When composing code generators, the code generation algorithm remains the same; we only need to combine the facts or rules that declare the entities to be generated (and add / modify a few rules to specify how the code generators being composed should interact).

4.3 Example — generating a visitor design pattern

To illustrate how code generators are built using LMP, this section shows an example of a generator which generates Java code for visitor design pattern.

The visitor design pattern [13] is fairly well known in the object-oriented community and is complex enough to show the strengths of our approach to code generation without becoming too large an example. The visitor design pattern is extremely useful when we need to perform a variety of operations on a tree structure. Instead of implementing the functionality in each type of tree node, all code for a given operation is grouped within one visitor class. The tree nodes, or visited nodes, use a double dispatch method to *accept* a visitor object (representing the operation to be performed) and call the corresponding *visit* method in that visitor object. [2] New operations can be added easily by defining new visitors, without touching the implementation of the tree.

We start by declaring, as logic facts, a description of the visited hierarchy, the name of the root of the visitor hierarchy, and the required visitor operations.

```
visitedRoot(Node).
visitedNode(ExpressionNode, Node).
visitedNode(AssignmentNode, ExpressionNode).
visitedNode(VariableNode, Node).

visitorRoot(NodeVisitor).
visitorOperation(PrettyPrinting).
visitorOperation(Parsing).
visitorOperation(Compiling).
```

In other words, the class named `NodeVisitor` will be subclassed by three classes with names containing “PrettyPrinting”, “Parsing” and “Compiling”. These visitor classes will interact with the classes of the `Node` hierarchy — as declared by their name and superclass name in the `visitedNode` facts — which will also be generated.

We first discuss the logic code that declares the classes to be generated, and then show the logic rules which declare the methods to be generated for these classes.

Visited and visitor classes

Before actually generating classes (in the second phase), the generic code generator engine first calls the classToGenerate rules to accumulate all required information on the classes to be

[2] It is convention to use the words “visit” and “accept” in the method names of, respectively, the visitor and the visited nodes of the tree. [13]

generated. For the visitor design pattern, we need to generate both the classes of the visitor and the visited hierarchy. The logic rules that declare the visited classes are straightforward:

```
classToGenerate(<?Class>) if visitedRoot(?Class).
classToGenerate(<?Class, ?SuperClass>) if visitedNode(?Class, ?SuperClass).
```

In other words, a class with the name declared as `visitedRoot` will be generated, as well as all classes declared as `visitedNode`, for which also the superclass is given, thus constructing the hierarchy of visited classes.

For the classes of the visitor hierarchy, we have chosen to suffix the name of the classes declared as `visitorOperation` with the string "Visitor", to emphasize that these are visitor classes. This is achieved in the `visitorOpClass` rule by making a meta-call to Smalltalk and using the string concatenation operator (`,`) to append the suffix "Visitor" to the name of the operation. All operation classes are to be generated as subclasses of the class declared as `visitorRoot`.

```
classToGenerate(<?Class>) if
  visitorRoot(?Class).
classToGenerate(<?Class, ?SuperClass>) if
  visitorRoot(?SuperClass),
  visitorOpClass(?Class).

visitorOpClass(?Class) if
  visitorOperation(?VisitorOp),
  equals([ ?VisitorOp , 'Visitor' ], ?Class).
```

Visitor and visited methods

Next, we need to declare the methods to be generated. For each visitor class, a visit method is needed for each of the possible visited classes. A visitor class is either the `visitorRoot` or a `visitorOpClass`. A visited class is either the `visitedRoot` or a `visitedNode`.

```
methodToGenerate(<?VisitorClass, ?Code>) if
  visitorClass(?VisitorClass),
  visitedClass(?ClassName),
  visitorCodeForClass(?Code, ?ClassName).
visitedClass(?Class) if
  visitedRoot(?Class).
visitedClass(?Class) if
  visitedNode(?Class,?).
visitorClass(?Class) if
```

```
    visitorRoot(?Class).
  visitorClass(?Class) if
    visitorOpClass(?Class).
```

The predicate `visitorCodeForClass` generates the code of the visitor method (visit), which is specific to each visited class. It contains a quoted code template for visitor methods which is filled in at evaluation time with the value of the variable `?Visited`, which corresponds to the name of the visited class.

```
visitorCodeForClass(?Code, ?Visited) if
  equals(?Code, {public void visit?Visited(?Visited vis) {}}).
```

Here is an example of the generated code of a visitor method:

```
public void visitAssignmentNode(AssignmentNode vis) {}
```

Finally, we still need to describe the double-dispatch methods to be generated on the visited hierarchy. These methods are generated in each class visited class.

```
methodToGenerate(<?Visited, ?Code>) if
  visitedClass(?Visited),
  visitorRoot(?Root),
  visitedCodeForClass(?Code, ?Visited, ?Root).
```

The process of formatting the visited methods is the same as for visitor methods, the only difference is the quoted code template, which now corresponds to an "accept" method:

```
visitedCodeForClassForJava(?Code, ?Visited, ?VisitorRoot) if
  equals(?Code, {public void accept(?VisitorRoot visitor) {
    return visitor.visit?Visited(this); }})
```

An example of the generated code follows:

```
public void accept(NodeVisitor visitor) {
  return visitor.visitAssignmentNode(this); }
```

4.4 Reverse-engineering using LMP

We have discussed about SOUL and its abilities to extract higher-level views from source code. In our case, these higher-level views are class diagrams. In this thesis, we will show the definition of code-diagram mapping strategies which allow one to perform ad hoc queries about UML entities present in the code, such as a query to discover which are attributes present in the class `Foo`:

```
if dcAttributeInClass(?Attribute, [ Foo ]).
```

However, since our round-trip engineering tool does *automatic* extraction of diagrams from code, we need more than the ability to make queries about UML constructs present in the code. We need a way to somehow save these diagrams. In other words, we need to *generate* diagrams from code.

Taking into account the same reasons that we discussed for code generation, we also use the *pull* approach for the reverse engineering part of our approach to round-trip engineering. The obvious difference is that now we are not generating code, but extracting a class diagram. So, in the first phase, we query for all the information about the UML constructs which have to be extracted, and, in the second phase, we generate a new diagram with this information.

Our class diagrams are internally represented by logic facts. For example, a class is represented by a fact in the following form: [3]

```
umlClass(?Diagram, ?Name, ?Concreteness)
```

Our goal is to extract facts with exactly the same form. Instead of performing queries like `classToGenerate`, in reverse-engineering, we perform queries related to UML constructs, like `classToExtract`, `attributeToExtract` or `associationToExtract`. When all this information has been collected, we store it in the predicate repository of SOUL, allowing the extracted predicate to be normally queried. The summarized code of the main predicate for diagram extraction is the following:

```
extractDiagram if
  accumulate(classToExtract, ?Classes),
  accumulate(attributeToExtract, ?Attributes),
  accumulate(associationToExtract, ?Associations),
  ...
  doAll(extractClass, ?Classes),
```

[3]The complete description of our internal representation of UML diagrams by means of logic facts is given later on, in section 6.1.

```
    doAll(extractAttribute, ?Attributes),
    doAll(extractAssociation, ?Associations),
    ...
```

The predicates of the second phase take each of the results in the lists returned by the first phase and add the appropriate fact for it in the predicate repository. For doing that, they use the predicate `compileUmlRule`, which concatenates the extraction suffix (e.g., "euml") to the predicate name and calls the predicate `compileRule` of SOUL to finally add the rule to the predicate repository.

```
extractClass(<?Diagram, ?Name, ?Concreteness>) if
  compileUmlRule(Class, <?Diagram, ?Name, ?Concreteness>).
extractAttribute(<?Diagram, ?ClassName, ?Type, ?Name, ?Visibility>) if
  compileUmlRule(Attribute, <?Diagram, ?ClassName, ?Type, ?Name, ?Visibility>).
...

compileUmlRule(?PredicateSuffix, ?Description) if
  extractedUmlPrefix(?ExtractedPrefix),
  umlRepository(?LogicRepository),
  compileRule(?LogicRepository, [ ?ExtractedPrefix , ?PredicateSuffix ],
              ?Description).
```

Conclusion

In this chapter, we have shown our approach to code generation using LMP, the *pull* approach. In fact, what happened is that, after some unsuccessful attempts to build LMP code generators for UML using ad hoc techniques, we realized that a much more general approach was needed, and this is how this approach was conceived. The clear division between a phase with declarative rules to state what to generate, and an imperative phase actually generating the code was decisive to create a usable and general code generator.

As we have shown in the example of the visitor design pattern, this code generator turns out to be quite simple to use, since the developer just declares rules about the entities that should be generated. The approach is also general, given that it can be used to generate any kind of code, and not only object-oriented. An indication of that is the way we completely inverted it to generate class diagrams from source code instead of source code from class diagrams.

In the next chapter, 5, we describe our general approach to round-trip engineering using LMP, which has code generation and reverse-engineering as foundations. Later on, in chapter 6, we explain how the pull approach for code generation and diagram extraction is used to create

clear and extensible mapping strategies between code and class diagram. In section 6.3, we will discuss about how this approach can be used to support multi-language code generation.

Chapter 5

Round-trip engineering using LMP

In this chapter, we describe our general approach to round-trip engineering. We discuss important issues for synchronizing class diagrams and source code and we introduce the way in which logic metaprogramming can be used to map constructs from source code to class diagrams and vice versa.

5.1 From diagram to code and back again

A typical way to start the development of a software system is to create a UML class diagram and start writing code based on that diagram. The process of writing the initial code is often automated in a process called code generation (or forward engineering).

Reverse-engineering is the process of extracting more abstract views from source code. In our case, these views are UML class diagrams.

Both code generation and reverse-engineering can be seen as a kind of code transformation. When generating, we have a class diagram — which can be regarded as code in a higher-level language — that is transformed into programming language source code. And when extracting, we have a portion of source code and it is transformed into a diagram.

These transformations, implicitly or explicitly, follow some well-defined *mapping strategies* for mapping concepts from class diagrams to source code, and vice versa. These mapping strategies define how the code is generated from a given diagram and how a diagram is extracted from a given portion of code.

Let us say that we have a code generation engine g_s which follows the mapping strategy s to generate code from diagrams, and a reverse-engineering engine e_s which follows the same

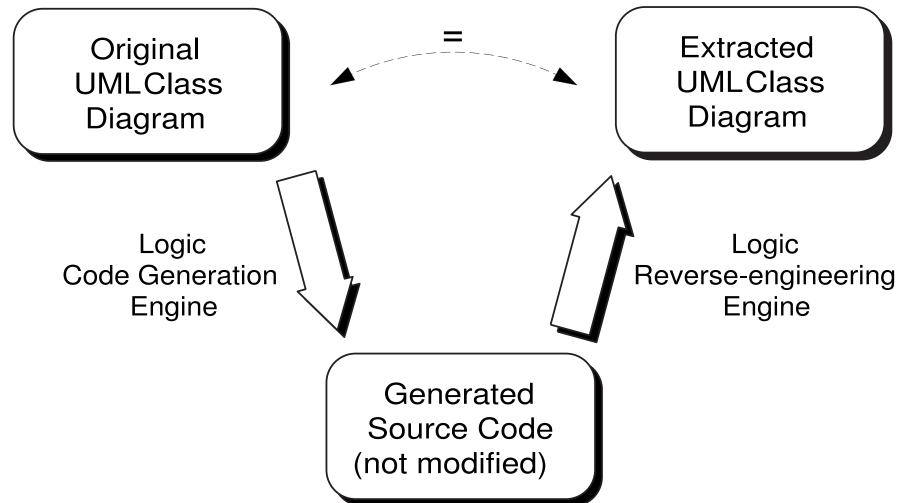

Figure 5.1: Code generation and diagram extraction without manual changes.

strategy s for extracting diagrams from code. If we use g_s to generate code from a diagram D and we use e_s to extract a diagram D_e from this code, then $D = D_e$. Equation 5.1 formalizes this.

$$e_s(g_s(D)) = D \tag{5.1}$$

This means that if the generated source code is not modified, it still has a correspondence to the diagram from which it was generated, and such diagram can be extracted from the code again using the appropriate reverse-engineering engine. Figure 5.1 illustrates this.

It is important to notice that, in this case, the generation engine does not take any already existing code into account. All the code that is generated was derived from the diagrams using the diagram-code mapping strategies. In the same way, the reverse-engineering engine does not have any knowledge about the original diagram. It only extracts information from the source code.

This mechanism of generation and extraction is possible given that the source code *ideally* contains all the information which is present in the diagram. Every little arrow the designer draws in the diagram is intended to express some structure or behavior that should be present in the software system, and the source code is the way it is actually implemented. Given that, it is “just” a matter of knowing how to extract the diagram from the code.

Knowing how to extract diagrams from any existing code — i.e., reverse-engineering techniques —, is a complex task which is still unsolved, as we discussed in chapter 2. However, we are not facing that problem here. In figure 5.1, we *choose* a given mapping strategy for generating code,

ensuring the way in which the constructs from the UML class diagram are generated in the code. In other words, we know how the UML constructs are being represented in the code because we assumed a given convention. Having this, we can assume the same strategy for extracting the diagram from the code.

5.1.1 A small example

Let us show a simple example to demonstrate how logic rules are used to declare the mapping strategies which generate code and extract diagrams. This example generates Java code from a class diagram with classes and attributes.

First of all, we need to declare the UML class diagram using logic predicates. In this example, we use a simplified definition style. The complete set of rules used in our approach are defined later on in chapter 6.

```
umlClass(?ClassName).
umlAttribute(?ClassName, ?Type, ?Name, ?Visibility).
```

The `umlClass` predicate declares a class named `?ClassName`. The predicate `umlAttribute` declares an attribute of type `?Type` in the class named `?ClassName`. This attribute is named `?Name` and has the visibility `?Visibility` (`public` or `private`). [1]

Some concepts in this diagram representation are easily mapped to code: a class with a certain name in the diagram is a class with the same name in the code; an attribute with a certain name in a diagram class is an instance variable with the same name in a code class.

But there is one feature of UML class attributes that is not straightforward to be represented in the code: `?Visibility`. This happens because, accordingly to the official coding conventions for Java ([26], section 10.1), instance variables of classes are supposed to have *private* visibility, except for extra-ordinary cases. Public access to them is usually provided by accessor methods (get- and set-).

To generate code accordingly to this convention, in this example, every public UML attribute is an instance variable with an accessor, and a private attribute is an instance variable without accessor.

We have now stated the conceptual mapping strategy between source code and UML class diagram that we use in this example. We use the same mapping strategy both for generating the code and for extracting the class diagram. However, one important thing to notice here

[1] For clarity, other features of UML class attributes such as type and changeability were omitted in this example.

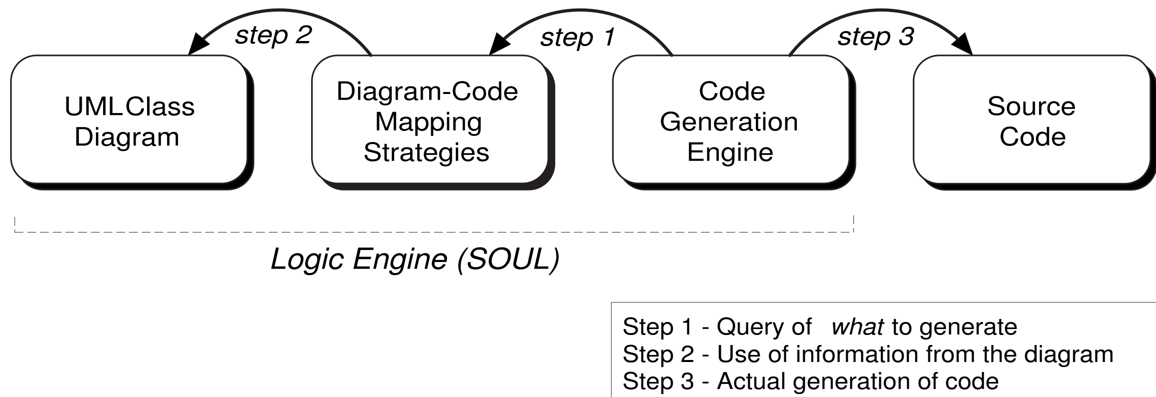

Figure 5.2: Code generation queries.

is that, in the implementation of the logic predicates for the mapping, this strategy is split in two: code generation mapping strategies and reverse-engineering mapping strategies. Both correspond to the same conceptual mapping, but are declared in separately in the code.

Code Generation

The code generation engine is responsible for performing queries to discover what code should be generated and for actually generating it. This is illustrated by figure 5.2. (step 1) The generation engine queries the mapping strategies to know which classes, methods and instance variables should be generated. (step 2) The mapping strategies inspect the class diagram to build the code that has to be generated. (step 3) The logic engine generates the code.

The queries performed by the logic engine (step 1) are quite simple. They ask for every class, instance variable or method that should be generated. The mapping strategies must declare logic predicates with the same headers as those of the query from the generation engine. These headers are the following:

```
classToGenerate(<?ClassName, ?SuperClassName, ?NamespaceName>)
instanceVariableToGenerate(<?ClassName, ?Type, ?VariableName>)
methodToGenerate(<?ClassName, ?MethodCode>)
```

The mapping strategies of this example are also simple. Our diagram has no generalizations — i.e., every class is subclass of `Object` — and we are going to generate a class in the code for every class in the diagram. These classes are generated in a pre-defined package called `umlgenerated`. The rule that declares this generation is the following:

```
classToGenerate(<?ClassName, Object, umlgenerated>) if
  umlClass(?ClassName).
```

Every attribute from the UML diagram is represented by an instance variable in the code:

```
instanceVariableToGenerate(<?ClassName, ?Type, ?AttributeName>)
  umlAttribute(?ClassName, ?Type, ?AttributeName, ?).
```

Notice that we discard the third parameter (`?Visibility`) of the predicate `umlAttribute` when generating instance variables. As mentioned, in this example, we represent the visibility of class attributes through accessors. Accessors are methods, and are therefore treated in the `methodToGenerate` predicate:

```
methodToGenerate(<?ClassName, ?MethodCode>)
  umlAttribute(?ClassName, ?AttributeName, public),
  stringCapitalizedOf(?CapitalizedName, ?AttributeName),
  equals(?Code, {public ?TypeName get?CapitalizedName() {
    return ?AttributeName;
  }}).
```

This logic rule states that an accessor method should be generated for every UML class attribute. First, the name of the accessor is created according to Java conventions, then a quoted code template is used to format the code of the accessor.

The three rules that we just described form the mapping strategies that we chose for generating code from our class diagram. They are queried by the code generation engine (`classToGenerate`, `methodToGenerate` etc), which corresponds to step 1 of figure 5.2, and they perform queries on the UML class diagram predicates (`umlClass`, `umlAttribute`), which correspond to step 2.

Step 3 of figure 5.2 is the actual code generation, where the generation engine takes the information returned by the query on the mapping strategies and transforms it into code, as described in chapter 4.

So, if we have a diagram described by the following facts:

```
umlClass(Person).
umlAttribute(Person, name, String, public).
umlAttribute(Person, age, int, private).
```

The code generation engine using the mapping we just stated would generate:

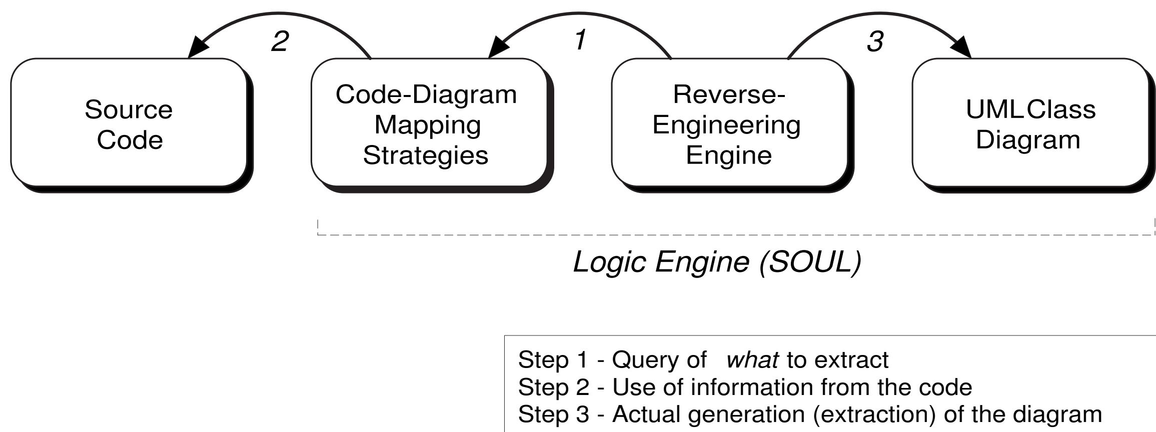

Figure 5.3: Reverse engineering queries.

```
public class Person {
  private String name;
  private String age;
  public String getName() {
    return name;
  }
}
```

UML class diagram reverse-engineering

The mechanism for extraction of a UML class diagram from code is similar to the one for code generation. It also uses the mapping strategies for querying what has to be generated. The difference is that, now, a class diagram is generated instead of code.

As illustrated by figure 5.3, the reverse engineering engine queries the mapping strategies, which, taking the source code into account, determine which classes and attributes have to be created in the diagram. Notice that the original diagram is *not* used in this process.

In the previous section, the generation engine was interested in which classes, instance variables and methods needed to be generated in the code, so it made queries like `classToGenerate` etc. Now, when doing reverse-engineering, we are interested in which entities need to be created in the diagram. In our example, since we are working on a reduced representation of class diagrams, these entities are classes and attributes. The headers of the queries done by the reverse-engineering engine are:

```
classToExtract(?ClassName).
```

```
attributeToExtract(?ClassName, ?Type, ?Name, ?Visibility).
```

As in code generation, the mapping strategies have to define rules with these headers.

To extract classes, we have to be careful to not extract a UML class diagram that contains all the classes of the system (which, for example, are thousands in case of Visual Works 7). We can restrict this using, for example, the names of the classes or a specific namespace. We choose the second option because it is more generic:

```
classToExtract(?ClassName) if
  generatedClass(?Class),
  classWithName(?Class, ?ClassName).
generatedClass(?Class) if
  namespaceWithName(?Namespace, umlgenerated),
  classInNamespace(?Class, ?Namespace),
```

These predicates state that a UML class should be extracted for every class that exists in the namespace (package) called `umlgenerated`. This is the namespace where the classes where generated in the code generation phase.

The extraction of the attributes requires the declaration of auxiliary predicates for the determination of the visibility.

```
attributeToExtract(?ClassName, ?Type, ?Name, ?Visibility) if
  generatedClass(?Class),
  instanceVariableInClass(?Name, ?Class),
  instanceVariableWithType(?Name, ?Type),
  attributeInClassWithVisibility(?Type, ?Name, ?Class, ?Visibility),
  classWithName(?Class, ?ClassName).
attributeInClassWithVisibility(?Type, ?Name, ?Class, public) if
  accessorForInstanceVariableInClass(?, ?Type, ?Name, ?Class).
attributeInClassWithVisibility(?Type, ?Name, ?Class, private) if
  accessorForInstanceVariableInClass(?, ?Type, ?Name, ?Class).
```

The predicate `attributeToExtract` returns one result for every instance variable present in each of the generated classes. The name of the attribute is the name of the instance variable. The visibility of the attribute is returned by the predicate `attributeInClassWithVisibility`, which verifies the presence of an accessor for the instance variable that represents the attribute in the code. The attribute is public in case this accessor exists, and false otherwise.

To verify the presence of an accessor for an instance variable, a method is searched which is named as an accessor. In case this method is found, its parse tree is matched with the pattern

of an accessor method, i.e., a method whose body consists only of a return statement for the instance variable. The following rule does this verification using the predicates `methodInClass` and `parseTreeOfMethod` from LiCoR:

```
accessorForInstanceVariableInClass(?Accessor, ?Type, ?Name, ?Class) if
  stringCapitalizedOf(?Capitalized, ?Name),
  methodInClass(?Accessor, ?Class),
  parseTreeOfMethod(?ParseTree, ?Accessor),
  equals(?ParseTree,
         method(?Class,
                signature(?, ?, ?Type, [ 'get' , ?Capitalized ], <>),
                arguments(<>),
                ?,
                statements(<return(variable(?Type, ?VariableName))>))).
```

Using these rules for reverse-engineering, we can extract exactly the same diagram we used for code generation. More details about how these rules work will be given in the next chapter.

5.1.2 Discussion

This section showed how mapping strategies between UML class diagrams and source code can be clearly stated using logic rules. Starting from a logic representation of a simplified UML class diagram, we used such strategies to generate source code. And using the LiCoR library, which allows us to reason about source code in terms of logic programming, we described simple logic rules which enable the extraction of the same UML class diagram from the generated source code.

Using an example, we also showed that if code is generated from a diagram accordingly to a *conceptual* mapping strategy, the use of the same conceptual strategy in the reverse-engineering phase will produce a diagram which is equal to the original one.

The mapping strategies that we used here could have been implemented in any other programming language. One could, for example, write a Java tool which takes an ad hoc representation of a UML class diagram and generates code from it. As we did in the example, this tool would also follow a mapping strategy. The same applies for a reverse-engineering tool.

The problem for developing such tools in an imperative language like Java is that the declaration of the mapping strategies is not clearly separated from the code of the transformation engine (code generator or diagram extractor). This kind of program usually assumes a specific mapping strategy and this is implemented is a hard-wired way into the code of the transformation engine, providing very few possibilities of parametrization or substitution.

As we brought out in chapters 1 and 2, the ability to substitute diagram-code and code-diagram mapping strategies must be key feature in a code generation, reverse engineering or round-trip engineering tool.

This section has shown that a good way to achieve this goal is to state this mapping strategies using a logic language. Like this, the strategies can be well separated from the transformation engines, thus becoming more easily interchangeable. For example, if we wanted to change the strategy we followed and state that all the attributes in the diagram should be *public* and mapped to *private* instance variables in the code, we would only have to eliminate the rule `methodToGenerate` and change the rule `umlAttributeToExtract` to:

```
umlAttributeToExtract(?ClassName, ?Type, ?Name, public) if
  generatedClass(?Class),
  instanceVariableInClass(?Name, ?Class),
  instanceVariableWithType(?Name, ?Type),
  classWithName(?Class, ?ClassName).
```

5.2 Achieving round-trip engineering

Round-trip engineering is about supporting changes in both class diagram and source code, and to have such changes reflected in the other side. The problem is: we have two views of the same software system (source code and class diagram) and we want to keep them consistent. When the code changes, the tool performs an update in the diagram. And when the diagram changes, the tool performs an update in the code.

Up to now, we are not dealing with code or diagram updates. Our code generation engine takes information only from the UML class diagram and generates code, and the reverse-engineering engine takes information from the source code and extracts a class diagram. When the code is being generated, code which is already present in the repository in not taken into account, that is, this is a code creation process and not a code update process.

Code or diagram update is naturally more complicated than simple generation or extraction, given that, in the former, what is present in the existing part has also to be taken into account. A possible solution to avoid this complication is to store all the information about the system in one view at a time. For example, we can use a diagram view which contains all the information about the system (design, implementation etc). Then we transform this into a source code view, which contains all the information too, so that we do not need the diagrams anymore. When desired, we transform this source code view again into a diagram view which also contains all the description of the system, so that we can discard the source code view, and so on. That is, at every moment during the development of the system, only one view represents it *completely*. This view can be either the source code or UML diagrams, but not both at the same time.

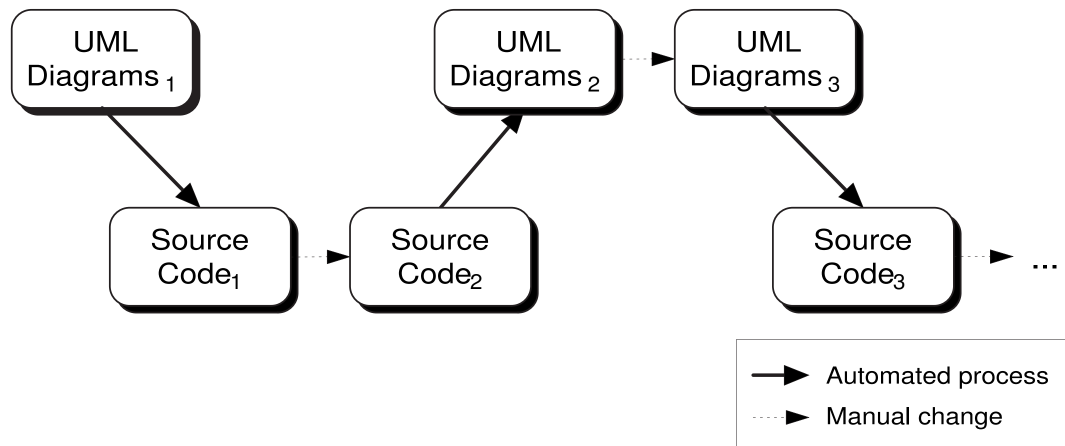

Figure 5.4: Ideal round-trip engineering.

Figure 5.4 illustrates this.

Let us follow the example of this figure. We start from UML diagrams and generate source code from them. Then the code is manually changed by the developer. Now, when the developer wants to change something at diagram level, UML diagrams are extracted from the source code that was manually changed. Now, the developer can make the desired modifications in these diagrams. Afterwards, source code is generated from the modified diagram, and so on.

We can notice that the first diagrams (UML Diagrams_1 in figure 5.4) were not taken into account when UML Diagrams_2 were extracted from Source Code_2 (the source code that was modified by the developer). To do this, we assume that, when manually changing the code, the developer follows the same mapping strategies that the round-trip engine does. This should be no problem since we believe that, using LMP, mapping strategies can be easily implemented to meet the practices of the developer.

This described scenario of round-trip engineering seems to be ideal. One just has to worry about the conception of mapping strategies for code generation and reverse-engineering, that is, how to transform the diagram view into a source code view which is equivalent, and vice versa.

But, unfortunately, this is *not* the real context.

5.2.1 Round-trip engineering with updates

Even if we used the whole set of diagrams from the Unified Modelling Language [32] — like we suggested with “UML Diagrams” instead of “UML class diagram” in figure 5.4 —, we would not be able to express all the information which is present in the source code. There are parts

of the source code which are very particular to it. The main example of this are method bodies. Method bodies are the core part of the behavior of a system. Even though there are UML diagrams for representing this (e.g., Statechart or Sequence diagrams), they are not enough powerful to express all the information present in the method bodies. [2]

On the other side, source code is not capable to represent information about the whole system either. Design-level information is sometimes mapped to code constructs which do not unambiguously identify their presence. For example, association classes (figure 5.5) are really difficult to be determined in the code [21]. Implementation of association classes is usually achieved by the means of a *map* (key-value dictionary) in the source class with instances of the target class as keys, and the association class as values. The problem is that not all the maps that follow this pattern necessarily represent association classes. Reverse-engineering mappings which assume this kind of pattern could incorrectly extract association classes. A developer which starts to deal with the code at this stage might think that such constraint exists while it does not.

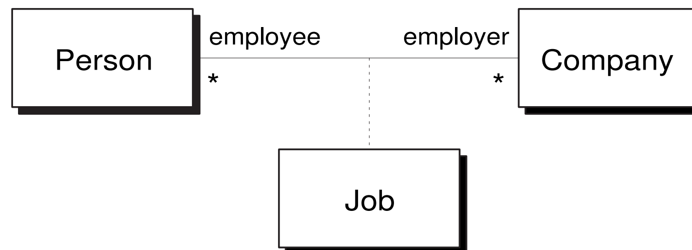

Figure 5.5: `Job` is an association class.

In fact, the observations we made in the two previous paragraphs are the motivation for the management of co-evolution between source code and higher-level models. It is not possible to extract all the higher-level design information from the code on demand. Models need to be kept updated along the lifetime of a software system. They must not be discarded after the code starts to be manually changed.

To have our approach for round-trip engineering following this idea, we need to keep both class diagram and source code. When code is generated, the code that possibly already exists is not discarded as well as the existing diagram is not discarded when the source code is being reverse-engineered.

It is important to remember that our approach does not handle parallel changes in both diagram and source code at the same time. This would be the case when the source code is changed and the diagram is changed too. When, for example, the code generation engine would be generating code again, it would have to check the whole consistency between the changes in the code and in the diagram.

[2]We do not claim that this is a shortcoming of UML diagrams. Their goal is to describe design-level information and not every detail of implementation.

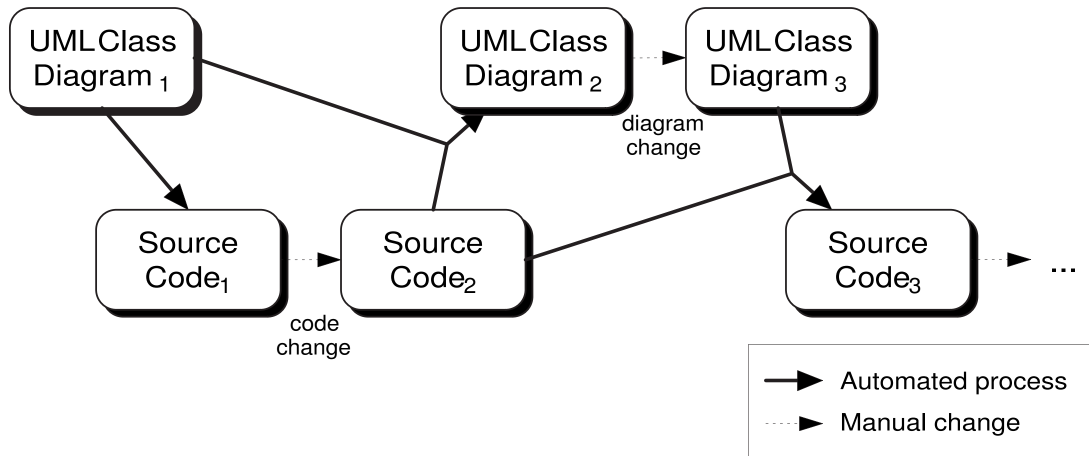

Figure 5.6: Round-trip engineering without discarding existing information.

In other words, we assume that the developer changes either the diagram or the code. When he is going to make changes in the other side, the appropriate transformation engine (code generation or reverse-engineering) has to be used to obtain the version which is appropriate for modifications.

In the example illustrated by figure 5.6, the developer builds a class diagram and the generation engine generates code from it. After the generation, the developer manually changes the code (e.g., to add some method bodies). Now, when the developer decides that he wants to make changes at design-level again, the reverse-engineering engine is used to extract the appropriate diagram which he can modify. The same thing would happen, now with code generation, when the developer wants to manually change the code.

Regarding the way in which transformations are performed, figure 5.6 shows that the existing diagram (*UML Class Diagram*$_1$) is now taken into account when reverse-engineering the a new class diagram (*UML Class Diagram*$_2$) from the modified code (*Source Code*$_2$). We do not completely rely on the mapping strategies to extract the diagram anymore, since we know that there may be ambiguities or missing information in the code and this problems can be solved using the diagram that already exists. The same goes for code generation, which also takes the existing code (*Source Code*$_2$) into account to generate new code (*Source Code*$_3$) from the changed diagram (*UML Class Diagram*$_3$).

Still on figure 5.6, one can notice that it contains only class diagrams, in contrast to figure 5.4, which referred to a set of UML diagrams. Class diagrams are the most popular among UML diagrams and are very suitable for representing the static structure of a software system. In this thesis, as a reduction of scope, we restrict ourselves to UML class diagrams. [3].

[3]Generalization of the approach to other UML diagrams is mentioned in section 8.2 as future work.

5.2.2 Refining the updates

In order to clarify the way in which existing code and class diagram are used in the transformation process, we divide source code and diagram into two parts: a unambiguously mappable and a ambiguously mappable. These parts help us to organize the transformation and tell from where each of the transformed constructs was brought. Let us describe this divisions of class diagrams:

Unambiguously mappable This part represents the diagram constructs which are easily mappable to source code and can also be extracted from it only using the information from the code.

A common example of these constructs are classes. The presence of a class in the diagram is simply inferred from the presence of that class in the code. That is, the previous existing diagram does not have to be consulted when doing the reverse-engineering in this case.

Ambiguously mappable These are the constructs in the diagram which are not easily recovered when doing reverse-engineering of source code. To be able to completely identify one of these constructs, information from the previously existing diagram has to be used.

Associations are example of this. Associations are not directly represented in the code. What is actually in the code are the association ends, which are normally mapped to instance variables, and not the associations themselves. So, when recovering a class diagram, we can only find the association ends. These ends then have to be merged to form an association. However some information about the association, such as its name, cannot be found in the code. In this case, the previous existing diagram has to be consulted to maintain the original name of the association.

With the addition of the concept of the distinction of ambiguous and unambiguous parts of diagram and code, figure 5.7 shows how these parts are used in the transformations, illustrating the final view of our approach.

The transformations are made mainly using the mapping strategies and, when needed, the already existing code or diagram is used. For example, in the case of reverse-engineering, the class diagram is extracted from the source code using the mapping strategies for reverse-engineering. In case of constructs which are not unambiguously extractable from the code, the previously existing diagram is used.

Let us take the example of a developer which starts to develop a software system by building a class diagram. This diagram (*modified class diagram*, in figure 5.7) contains ambiguous and unambiguously mappable constructs. All of these constructs are used by the mapping strategies and the code generation engine to generate the initial code (*generated source code*). Then the developer modifies the code not regarding which part is ambiguously or unambiguously mappable — he does not have to care about that, this is a task for the development environment.

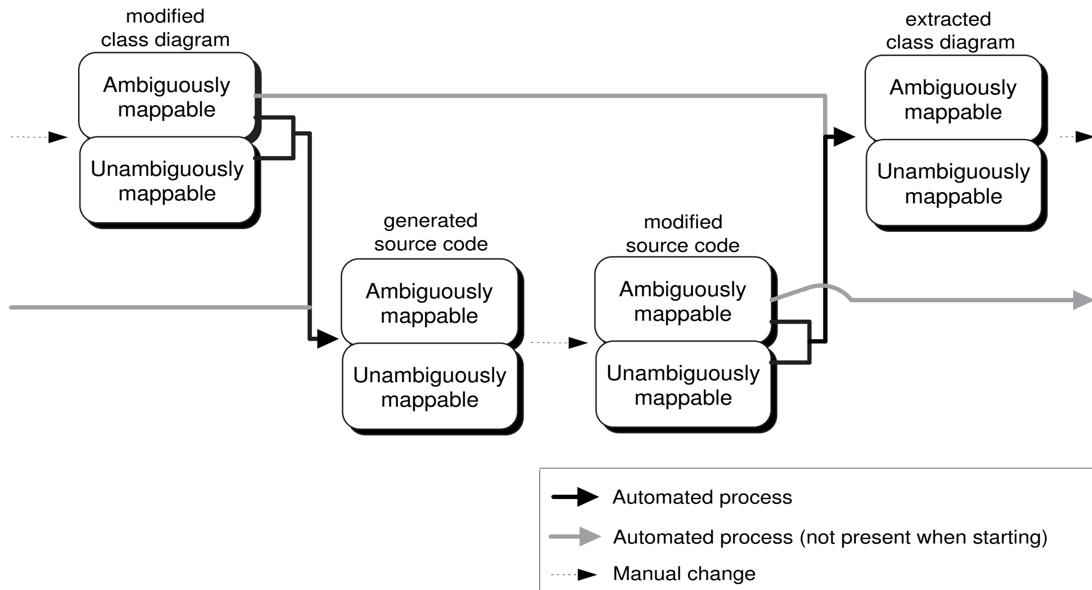

Figure 5.7: Development cycle using our approach for round-trip engineering.

Now, the reverse-engineering engine, together with the mapping strategies, extracts a new diagram (*extracted class diagram*) from the *modified source code*, solving possible ambiguities using the previously existing diagram (*modified class diagram*). Notice that the main extraction mechanism is the black arrow between *modified source code* and *extracted class diagram*, and the gray arrow represents the queries done to solve ambiguity conflicts when doing reverse-engineering.

The next step is the manual change of the diagram by the developer. Then the cycle begins again. The diagram resulting from this change (*modified class diagram*) is used by the mapping strategies to generate new code (*generated source code*). Ambiguities which appear in this generation process (e.g., method bodies) are solved by querying the previously existing code (modified source code).

Conclusion

A naïve approach to round-trip engineering could be to try to have only one representation of the system at a time. For example, to start from a diagram D_1 to and generate code C_1 from it. To change this code and then extract a diagram D_2 from it. Then generate code from D_2 and so on. We discussed that this approach is not feasible because there are constructs which are particular to code and constructs which are particular to diagrams.

Instead of that, we proposed a kind of “co-evolution” approach, in which the environment

maintains both diagram and code up-to-date, without discarding any of them. When the code is being generated, the previously existing code is taken into account, as well as the existing diagram is taken into account when doing reverse-engineering.

To implement this approach, we introduced the use of logic metaprogramming (LMP). LMP is a natural technique for stating the kind of transformation rules we need. As we will see in the next chapter, even quite complex mapping strategies can be clearly declared using LMP, achieving an interesting separation between the transformation engine and the transformation rules themselves.

Chapter 6

Mapping strategies

This chapter gives a detailed view of how the code-diagram and diagram-code mapping strategies are declared using SOUL. After describing our logic representation of UML class diagrams, we show a complete example of how a UML class diagram is mapped to Java source code and how a UML class diagram is reverse-engineered from this generated code. We also describe a simple method for verifying if this reverse-engineering process was correct. Finally, we discuss the language-independence of our approach, illustrating how it can be applied to other languages.

To achieve round-trip engineering, our goal is to maintain the source code updated and consistent with its class diagram. On one hand, we have the class diagram itself, which is, as we discussed in the previous chapter, represented by logic predicates. On the other hand, we have the source code, which, with the help of the Library for Code Reasoning (LiCoR), can also be reasoned about by means of logic predicates.

Having both views — code and diagram — expressed in a logic language provides us a good starting point to also state our mapping between them also by means of logic predicates. As such, we can use normal logic programming to map code to diagram and vice versa.

6.1 UML Class Diagram representation

> "Class diagrams show the static structure of the model, in particular, the things that exist (such as classes and types), their internal structure, and their relationships to other things." [32]

Let us start by formalizing the way in which we represent a UML class diagram using logic predicates.

UML class diagrams have a simple structure. The elements that can be contained in a class diagram are defined in the Unified Modelling Language specification [32]. The most important are classes, operations, attributes, associations and generalizations. Some features of these constructs are very specific and rarely used (e.g., stereotypes of associations or powertypes of generalizations). In this thesis, we assumed that a reduced set of possible class diagram constructs is sufficient to validate our approach. The addition of other constructs can be regarded as future work.

We decided to represent the main constructs of class diagrams only: classes, attributes, operations, generalizations and associations. [1] The most important features of each of these constructs are also represented: visibility, changeability, multiplicity etc. The following list describes each of these constructs.

Classes A class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics [32]. A class is represented by predicates following this pattern:

```
umlClass(?Diagram, ?Name, ?Concreteness)
```

- `?Diagram` is the diagram that contains this class. This argument exists because there may be more than one diagram declared in the logic system.
- `?Name` is the name of the class.
- `?Concreteness` indicates whether this class is concrete or abstract. The possible values are `concrete` and `abstract`.

Here is an example of the declaration of a class called `Communication` in a diagram called GSMCase.

```
umlClass(GSMCase, Communication, abstract).
```

Attributes An attribute is a named slot within a class that describes a range of values that instances of the class may hold [32]. Attributes are represented by predicates following this pattern:

```
umlAttribute(?Diagram, ?ClassName, ?Name, ?Type, ?Visibility,
             ?Changeability)
```

- `?Diagram` is the diagram that contains this attribute.
- `?ClassName` is the name of the class which contains this attribute.

[1] Since we are working in the context of both Smalltalk and Java, we decided not to use *interfaces*, since they are very specific to Java.

- `?Name` is the name of the attribute.
- `?Type` is the name of the type of this attribute.
- `?Visibility` indicates the visibility of this attribute. It can have the following values: `public`, `protected` or `private`.
- `?Changeability` indicates the changeability of this attribute. The possible values are: `frozen`, for attributes that have constant value, and `changeable`, for attributes that can have their value changed.

There is also another predicate, optional for each attribute, for specifying its initial value. The term `?Value` can be any valid expression in the language for which the code will be generated:

```
umlAttributeInitialValue(?Diagram, ?ClassName, ?Name, ?Value).
```

This is the example of an attribute and its initial value:

```
umlAttribute(GSMCase, Caller, active, Boolean, public, changeable).
umlAttributeInitialValue(GSMCase, Caller, active, true).
```

Operations An operation is a service that can be requested from an object to effect behavior [32]. In our approach, an operation is described by:

```
umlOperation(?Diagram, ?ClassName, ?Name, ?Parameters, ?ReturnType,
             ?Concreteness, ?Visibility)
```

- `?Diagram` is the diagram that contains this operation.
- `?ClassName` is the name of the class that contains this operation.
- `?Name` is the name of the operation.
- `?Parameters` are the parameters for the operation. These parameters are described in a list which contains sublists of two elements stating the type and name of each parameter.
- `?ReturnType` is the name of the return type of this operation.
- `?Concreteness` indicates whether this operation is concrete or abstract. The possible values are `concrete` and `abstract`.
- `?Visibility` indicates the visibility of this operation. Possible values are `public`, `protected` or `private`.

Here is an example of a predicate that represents a UML operation:

```
umlOperation(GSMCase, Main, printInvoices, <<Integer, aMonth>>, void,
             concrete, private).
```

Generalizations A generalization is a taxonomic relationship between a more general element and a more specific element, i.e., a subclass [32]. In our approach, the generalizations of a UML class diagram are represented by the following predicate:

```
umlClassGeneralizes(?Diagram, ?Superclass, ?Superclass)
```

- `?Diagram` is the diagram that contains this generalization.
- `?Superclass` is the name of the more general class.
- `?Subclass` is the name of the more specific class.

An example of a class generalization follows:

```
umlClassGeneralizes(GSMCase, Communication, SMS).
```

In order to help the representation of the concepts of subclass and superclass in the diagram, we declare three more logic rules. Using these rules, we can assume that a certain class has always a superclass, either from a generalization or simply by being a subclass of the root class, `Object`.

```
umlSuperclassOf(?Diagram, ?SuperclassName, ?SubclassName) if
  umlClassGeneralizes(?Diagram, ?SuperclassName, ?SubclassName).

umlSuperclassOf(?Diagram, Object, ?SubclassName) if
  umlClass(?Diagram, ?SubclassName, ?),
  not(umlClassGeneralizes(?Diagram, ?, ?SubclassName)).

umlSubclassOf(?Diagram, ?SubclassName, ?SuperclassName) if
  umlSuperclassOf(?Diagram, ?SuperclassName, ?SubclassName).
```

Associations An association defines a semantic relationship between class [32]. Associations are the most complex structure of a class diagram, given the large number of annotated attributes (e.g., navigability, changeability, ordering etc). In our representation, associations are represented by a predicate with many arguments. Notice that we treat only binary associations.

```
umlAssociation(?Diagram,
               ?ClassName1, ?ClassName2,
               ?AssociationName,
               ?Role1, ?Role2,
```

```
                ?Navigability1, ?Navigability2,
                ?Changeability1, ?Changeability2,
                ?Visibility1, ?Visibility2,
                ?Lower1, ?Upper1, ?Lower2, ?Upper2,
                ?Ordering1, ?Ordering2).
```

The following list describes each of these items. The suffixes “1” and “2” refer to the two distinct association ends.

- `?Diagram` is the diagram that contains this association.
- `?ClassName` is the name of the class in this end of the association.
- `?Role` is the role name in this end of the association.
- `?Navigability` indicates whether this end (target end) is accessible from the other end (source end). It can be `inaccessible` or `navigable`.
- `?Changeability` indicates the changeability of this association end. The possible values are: `frozen`, for ends that have fixed value; `addOnly`, for associations with upper multiplicity n to which elements can only be added but not removed or set; and `changeable`, for association ends that can have the value freely changed.
- `?Visibility` indicates the visibility of this end of the association. Possible values are `public`, `protected` or `private`.
- `?Lower` and `?Upper` represent, respectively, the lower and upper multiplicities of this end of the association. `?Lower` can assume the values `0` or `1`, while upper can assume `1` or `n`.
- `?Ordering` indicates whether this association end is ordered. This applies only for ends with upper multiplicity `n`. The possible values are ordered, `unordered` and `notapplicable` (if upper multiplicity equals to 1).

This fact has a simple constraint: the two ends have to be put in alphabetical order of the name of the classes. This is to avoid ambiguity in the reverse-engineering, as we will see later on. An example of a predicate for an association end follows. In this example, an operator has one or more pricing policies:

```
umlAssociation(GSMCase,
               Operator, PricingPolicy,
               defines,
               operator, pricingPolicy,
               inaccessible, navigable,
               changeable, changeable,
               public, public,
               1, 1, 1, n,
               notapplicable, ordered).
```

As we show later on in this chapter, we are often more concerned about association ends rather than the complete association. So, to be able to easily refer to all the information for one of the association ends, we declared two rules which are based on the previous one. We show here only one of them, since the other is the same but in the opposite direction.

```
umlAssociationEnd(?Diagram, ?ClassName1, ?ClassName2, ?Role2,
               ?Navigability2, ?Changeability2, ?Visibility2,
               ?Lower2, ?Upper2, ?Ordering2) if
  umlAssociation(?Diagram, ?ClassName1, ?ClassName2, ?, ?, ?Role2,
                 ?, ?Navigability2, ?, ?Changeability2, ?,
                 ?Visibility2, ?, ?, ?Lower2, ?Upper2, ?, ?Ordering2).
```

6.2 A mapping strategy

This section finally brings us to a concrete example of something we have been discussing since the very first chapter: a strategy for mapping between code and class diagram. We describe conceptually how we are going to map class diagram constructs to source code and, after that, we show how this conceptual mapping is declared in logic predicates to get information about the diagram and transform it into source code, and vice versa.

But, first, let us remember an important point about our approach to round-trip engineering. To make round-trip engineering feasible by means of code generation and reverse-engineering, we have to be sure that the *same* mapping strategies are used both when generating code and when extracting the class diagram. Unfortunately, although the strategies for generating and extracting are conceptually the same, in the logic predicates they are stated separately. One of them states what are the code entities to be generated given the existing UML entities, and the other states what are the UML entities to be generated given the existing code entities. Therefore, to conceive a mapping strategy, we have to follow define the following mappings:

Conceptual mapping This is the first task to do when constructing a mapping strategy. It consists of stating conceptually what is the mapping that we are going to use in our example. This is a very important step, since all our mapping strategy predicates will be based on what is stated in this conceptual mapping.

Code generation mapping This mapping is stated by logic predicates that describe what are the UML patterns which imply the generation of code constructs. The code generation engine performs queries on the predicates of this mapping, such as `classToGenerate` or `methodToGenerate`.

Reverse-engineering mapping This mapping is stated by logic predicates that describe what are the code patterns which imply the extraction of UML constructs. The reverse-

engineering engine performs queries on the predicates defined in this mapping, such as `associationToExtract` or `operationToExtract`.

It is important to remember that, besides the fact that the code generation and the reverse-engineering mapping have to be coherent to the conceptual mapping, the developer must use the conceptual mapping as a coding convention. It is no use to have an environment which propagates, for example, changes in the code to the diagram accordingly to a mapping strategy if the developer does not follow this strategy while manually editing the code.

6.2.1 Conceptual mapping

The conceptual mapping we have chosen is described in table 6.1. All the mapping strategies in this section generate code for Java. A discussion about how our approach can be applied to other languages, like Smalltalk, is presented on section 6.3.

Looking at this table, we notice that declaring code diagram mapping strategies is not easy, even in a natural language. To make it clearer, we chose to organize this table as it would be read by someone who wants to perform *reverse*-engineering. So it states how class diagram constructs are represented in the code. The other way around, i.e., how code constructs can be mapped to code, can be deduced from this.

Some of the mappings are quite straightforward, like classes and generalizations, which are one-to-one mapped to source code constructs. But others are more complex, such as attributes and, in particular, association ends. More details about this mapping will be given when we show its declaration using logic rules.

6.2.2 Code generation mappings

Both code generation and reverse-engineering, the two pillars on which our approach is founded, use a pull approach to perform the transformations. The pull approach, as it was described in chapter 4, has two phases: one, called the *query phase*, to discover what has to be generated; and another one, called the *generation phase*, for actually generating the entities. The mapping strategies correspond to the query phase and are consulted by the transformation engines.

In the case of code generation, we need to implement the queries of the code generation engine. These queries ask for the constructs that can be generated in the code: classes, methods and instance variables. The headers of the queries are these:

```
classToGenerate(?Description)
```

UML	Source code
class	class
concreteness	concreteness
generalization	inheritance relationship
attribute	instance variable (primitive type)
changeability	`changeable` if the variable is written somewhere outside the initializer method, `frozen` otherwise
visibility	The visibility of the accessor method. If the variable does not have a corresponding accessor, then the visibility of the variable itself
initial value	If such assignment exists, the literal assigned to the instance variable inside the constructor
operation	method in the code which is *not* an accessor, a mutator, an initializer or an add method
visibility	visibility
concreteness	concreteness
association end	instance variable with: non-primitive type (upper multiplicity=1) container type (upper multiplicity=n)
role	instance variable name
navigability	every association end present in the code is navigable
changeability	upper multiplicity=1: `changeable` if variable is written somewhere in the class outside the initializer `frozen` otherwise. upper multiplicity=n: `changeable` if either the variable is written somewhere in the class outside the initializer, or there are calls to its `remove` or `set` methods. If none of the conditions to be `changeable` is satisfied but there are calls to its `add` methods, then `addOnly`. Otherwise, `frozen`.
visibility	The visibility of the accessor method. If the variable does not have a correspondent accessor, then the visibility of the variable itself
association name	not mapped to the code

Table 6.1: UML conceptual mapping of the example.

```
instanceVariableToGenerate(?Description)
methodToGenerate(?Description)
```

The term `?Description` should be bound to a list which contains all the information about the construct which has to be generated. In each case, `?Description` has a different meaning. The description of classes is a logic list of the following format:

```
<?ClassName, ?SuperClassName, ?ClassNamespaceName, ?Concreteness>
```

It generates a class with a given superclass in a given namespace. The `?Concreteness` term distinguishes abstract classes from the concrete ones.

For instance variables, the description list is as follows:

```
<?ClassName, ?TypeName, ?VariableName, ?Visibility>
```

This generates an instance variable named `?VariableName` with type `?TypeName` and a given `?Visibility` in the class `?ClassName`.

For methods, the format of the `?Description` list is:

```
<?ClassName, ?MethodCode, ?KindOfMethod>
```

The first two elements of this description are intuitive: the name of the class where the method should be generated and the actual code of the method. The third element of the list is a variable which specifies the kind of the method which is being compiled. It can be a normal method (`InstanceMethod`) or a constructor (`Constructor`).

Now, let us describe the actual logic predicates for the code generation mapping strategies.

Classes

The classes that need to be generated are exactly the ones that are in the diagram. We have also to take the generalizations into account, but this is also simple since we have declared the auxiliary rules to make their use easier. So this is the predicate for the generation of classes:

```
classToGenerate(<?Name, ?SuperName, ?NamespaceName, ?Concreteness>) if
  namespaceForDiagram(?Namespace, ?Diagram),
  namespaceWithName(?Namespace, ?NamespaceName),
  umlClass(?Diagram, ?Name, ?Concreteness),
  umlSuperclassOf(?Diagram, ?SuperName, ?Name)
```

The predicate `namespaceForDiagram` is used to retrieve the namespace in which all the classes of the diagram should be generated. [2] After obtaining the name of the namespace with the `namespaceWithName` predicate of LiCoR, the representation of the diagram predicates `umlClass` and `umlSuperclassOf` are used to built the remainder of the class description list.

Instance variables

As we can verify in table 6.1, there are two kinds of UML constructs which are mapped to instance variables: attributes and association ends.

For attributes One instance variable is generated for every attribute. For the name of the type of the attribute, we use an auxiliary predicate, `implementationTypeNameOf`, to be able to translate a data type of the class diagram to a data type of the implementation language. For example, we can translate `Double` to `java.lang.Double`. The rule for the mapping of instance variables is:

```
instanceVariableToGenerate(<?ClassName, ?TypeName, ?Name, private>) if
  umlAttribute(?, ?ClassName, ?Name, ?UmlTypeName, ?, ?),
  implementationTypeNameOf(?TypeName, ?UmlTypeName).
```

Notice that, for both attributes and association ends, we generate instance variables with private visibility. Public access to them is provided by accessors and mutators, as we describe later on.

For association ends One instance variable is generated also for each *navigable* association end, so we need to declare another rule for the predicate `instanceVariableToGenerate`:

```
instanceVariableToGenerate(<?ClassName, ?TypeName, ?Name, ?>) if
  umlAssociationEndInstanceVariable(?Diagram, ?ClassName, ?, ?, ?TypeName,
                                    ?Name, ?).
```

We use the predicate `umlAssociationEndInstanceVariable` to find all the instance variables that have to be generated for the association ends of the diagram. This was factored out from the above rule because, as we will see, this auxiliary predicate is useful in other situations too. Its code follows:

[2] We assume that all the classes of the diagram are in the same namespace.

```
umlAssociationEndInstanceVariable(?Diagram, ?Class1, ?Class2, ?Role,
                                  ?TypeName, ?VariableName, ?Visibility) if
  umlAssociationEnd(?Diagram, ?Class1, ?Class2, ?Role, navigable, ?,
                    ?Visibility, ?, ?Upper, ?Ordering),
  associationEndInstanceVariableName(?Role, ?Upper, ?VariableName),
  associationEndInstanceVariableType(?Class2, ?Upper, ?Ordering, ?TypeName).
```

An instance variable is generated for every association end. There are two other predicates which are used here. The first, `associationEndInstanceVariableName`, does a simple transformation in the name of the role to make it plural in case of association ends with multiplicity `n`. The other predicate, `associationEndInstanceVariableType`, is used to obtain the type of the instance variable for this association end. In cases when the upper multiplicity is `1`, the type is the class in the target side of the association. When the upper multiplicity is `n`, then the type is a container type (collection). This container type is `java.util.Vector` for ordered association ends, and `java.util.Set` for unordered ones.

Methods

In order to fulfill the requirements of the conceptual mapping strategy that we chose, in addition to methods corresponding to the operations of the diagram, we have to generate accessor methods, mutators and constructors. We also generate add and remove methods for the associations with upper multiplicity `n`.

For operations Operations are the first construct we encounter for which we must use information from the existing code when generating. Now, we do not have to think about the first code generation, when there is only the class diagram and no code. We have to think about the case where the developer has already changed the code, probably including working code in the method bodies. Taking that into account, we generate one method for each operation in the diagram, however, when a method with that name already exists in the code, the body of the method to be generated is used instead of a template one.

The main logic rule for the generation of methods for operations is:

```
operationToGenerate(<?ClassName, ?Code, InstanceMethod>) if
  umlOperation(?, ?ClassName, ?Name, ?UmlParameters, ?UmlReturnType,
               ?Abstract, ?Visibility),
  implementationTypeNameOf(?ReturnType, ?UmlReturnType),
  findall(<?Type, ?ParName>,
          and(member(<?UmlType, ?ParName>, ?UmlParameters),
              implementationTypeNameOf(?Type, ?UmlType)),
```

```
        ?Parameters),
    generateOperationCode(?ClassName, ?Name, ?Parameters, ?ReturnType, ?Abstract,
                          ?Visibility, ?Code).
```

For each UML operation, the return and parameter types are mapped to types of the implementation language. After that, the predicate `generateOperationCode` generates the appropriate code for the method. When the method is abstract, the `generateOperationCode` simply generates its signature with the abstract keyword. When the operation is concrete, there are two cases: when the method already exists in the code and when it does not. There is one rule for each of these cases. The first case is:

```
generateOperationCode(?ClassName, ?Name, ?Parameters, ?ReturnType, concrete,
                      ?Visibility, ?MethodCode) if
  generateMethodName(?Visibility, concrete, ?ReturnType, ?Name, ?Parameters,
                     ?MethodName),
  classWithName(?Class, ?ClassName),
  methodWithNameInClass(?Method, ?MethodName, ?Class),
  methodWithSourceCode(?Method, ?MethodCode).
```

The auxiliary predicate `generateMethodName` generates the name (signature) of a method. Then, it is verified if the class named `?ClassName` already contains a method with this signature. In case it contains, then the source code of this method is returned as the source code of the operation to be generated.

Another rule is declared for the case when the operation is not yet represented as a method in the code.

```
generateOperationCode(?ClassName, ?Name, ?Parameters, ?ReturnType, concrete,
                      ?Visibility, ?MethodCode) if
  generateMethodName(?Visibility, concrete, ?ReturnType, ?Name, ?Parameters,
                     ?MethodName),
  not(and(classWithName(?Class, ?ClassName),
          methodWithNameInClass(?Method, ?MethodName, ?Class))),
  generateEmptyMethodBody(?ReturnType,?MethodBody),
  equals(?MethodCode, {?MethodName { ?MethodBody }}).
```

This rule ensures that, in the given class, there is not yet a method with the same name as the one which is being generated. After that, the predicate `generateEmptyMethodBody` is used to generate an empty method body. This predicate returns and empty body when the return type is `void`, otherwise it returns `null`, `0`, `false` etc, depending on the return type. This body is then combined with the method name to form the complete code of the method.

For accessors and mutators Accessors and mutators are a quite important part of our conceptual mapping strategy. They are used to map the visibility and changeability of associations and attributes. The generation of all the accessor and mutator methods is organized by the following rules:

```
methodToGenerate(?MethodDescription) if
  accessorToGenerate(?MethodDescription).
methodToGenerate(?MethodDescription) if
  mutatorToGenerate(?MethodDescription).

accessorToGenerate(?MethodDescription) if
  accessorForAttributeToGenerate(?MethodDescription).
accessorToGenerate(?MethodDescription) if
  accessorForAssociationToGenerate(?MethodDescription).

mutatorToGenerate(?MethodDescription) if
  mutatorForAttributeToGenerate(?MethodDescription).
mutatorToGenerate(?MethodDescription) if
  mutatorForAssociationToGenerate(?MethodDescription).
```

The rules which consult the class diagram to check which accessors and mutators have to be generated are very simple. They all just get information about the variables and format it as an accessor or mutator.

The code for the generation of accessors for attributes follows:

```
accessorForAttributeToGenerate(<?ClassName, ?Code, InstanceMethod>) if
  umlAttribute(?, ?ClassName, ?VariableName, ?UmlType, ?Visibility, ?),
  implementationTypeNameOf(?TypeName, ?UmlType),
  generateAccessorCode(<?Visibility, ?TypeName, ?VariableName>, ?Code).
```

For each UML attribute, the code of the accessor is generated with its visibility, the implementation type of its type and its name. The predicate `generateAccessorCode` puts this information in the appropriate form of an accessor:

```
generateAccessorCode(<?Visibility, ?TypeName, ?VariableName>, ?Code) if
  stringCapitalizedOf(?CapitalizedInstVar, ?VariableName),
  equals(?Code, {?Visibility ?TypeName get?CapitalizedInstVar()
     { return ?VariableName; }}).
```

Accessors for association ends are very similar. They use a predicate which we have already discussed, `umlAssociationEndInstanceVariable`, to check which are the association ends from

the diagram which need to be represented in the code. Then, the same predicate as for attribute accessors, `generateAccessorCode`, is also used to format this information:

```
accessorForAssociationToGenerate(<?ClassName, ?Code, InstanceMethod>) if
  umlAssociationEndInstanceVariable(?, ?ClassName, ?, ?, ?TypeName,
                                    ?VariableName, ?Visibility),
  generateAccessorCode(<?Visibility, ?TypeName, ?VariableName>, ?Code).
```

The generation of mutators is analogous to the generation of accessors. The only difference is that we only generate mutators for attributes and association ends which are changeable. For example, for attributes, we enforce the `?Changeability` term of the predicate `umlAttribute` to have the value `changeable`.

For *add* methods Association ends with upper multiplicity `n` have a changeability attribute which can assume the values `frozen`, `changeable` and `addOnly`. We do not generate mutators for neither `addOnly` nor frozen association ends. The only difference between these two changeability values in the code is that instance variables representing ends with `addOnly` changeability have calls to its *add* methods. These add methods are specific to the container type which is being used. In our case, we are using `java.util.Vector` and `java.util.Set`, and we consider the method `add(Object)` as the add method for these types.

To express in the code this distinction between the changeabilities, we generate an add method for association ends which have the `addOnly` changeability. The following code is an example of an add method for the association end which has the role named `activeSession` of the type `HttpSession`:

```
public void addActiveSession(HttpSession activeSession) {
  activeSessions.add(activeSession);
}
```

For generating add methods, we declare another rule for the predicate `methodToGenerate`, which calls `addForAssociationToGenerate`.

```
methodToGenerate(?MethodDescription) if
  addForAssociationToGenerate(?MethodDescription).
```

The rule `addForAssociationToGenerate` finds every association end which changeability `addOnly` and upper multiplicity `n`. For each of them, it generates an add method using the predicate `generateAddCode`.

```
addForAssociationToGenerate(<?Class1, ?Code, InstanceMethod>) if
  umlAssociationEnd(?Diagram, ?Class1, ?Class2, ?Role, ?, addOnly, ?, ?, n, ?),
  umlAssociationEndInstanceVariable(?Diagram, ?Class1, ?Class2, ?Role,
                                    ?TypeName, ?VariableName, ?Visibility),
  generateAddCode(<?Visibility, ?Class2, ?Role, ?VariableName>, ?Code).
```

The formatting predicate is much similar to that of accessors and mutators. It capitalizes the role name of the association and inserts it, together with the original role name, visibility, target class name and variable name, into the template to form the code of the method.

```
generateAddCodeForJava(<?Visibility, ?Class2, ?Role, ?VariableName>,
                       ?Code) if
  stringCapitalizedOf(?Capitalized, ?Role),
  equals(?Code, {?Visibility void add?Capitalized(?Class2 ?Role)
    { this.?VariableName.add(?Role); }})
```

For constructor and initializers The last methods which we need to generate in the code are constructors and initializers. Initializers are private instance methods which are responsible for doing the initializations of the object, such as default initial values, initialization of associations etc. They are called by the constructor. The following code is the example of a constructor and its corresponding initialize method:

```
public Student(String name, String registration) {
  super(name);                      //call the constructor of the superclass
  initialize(registration);        //call the initialize method
}
private void initialize(String registration) {
  this.registration = registration;       //initialization of an attribute
  this.age = 0;                           //a default value for an attribute
  this.courses = new java.util.Vector();  //initialize an N association
}
```

We generate one constructor for each class. This constructor contains a call to the constructor of the superclass, and a call to the initialize method of the same class. The predicate for generating the code is the following:

```
constructorToGenerate(<?ClassName, ?Code, Constructor>) if
  umlClass(?Diagram, ?ClassName, ?),
  diagramExtractInitializeData(?Diagram, ?ClassName, ?InitializePairs, ?),
  diagramExtractConstructorParameters(?Diagram, ?ClassName, ?ConstructorPairs),
```

```
    umlSuperclassOf(?Diagram, ?SuperClassName, ?ClassName),
    diagramExtractConstructorParameters(?Diagram, ?SuperClassName, ?SuperPairs),
    generateConstructorCode(<?ClassName, public, ?ConstructorPairs, ?SuperPairs,
                            ?InitializePairs>, ?Code).
```

Besides the size of this predicate, its code is fairly simple. The main idea is that the generation of a constructor needs three lists with information about parameters:

- The parameters of the initialize method of the class, to build the call to it. These parameters are: (1) the association ends with *lower* multiplicity equal to `1` or changeability equal to `frozen`, because this enforces the user to pass a value for them in the creation of an instance of this class; (2) all the attributes which are frozen, for the same reason.
- The parameters of the constructor itself, to build its signature. The parameters of the constructor of a class are the parameters of the initialize method of the class plus the parameters of the constructor of the super class.
- The parameters of the constructor of the superclass, to build the call to it. The parameters of the constructor of the superclass are built recursively using the name of the super class. This recursion stops when the super class is `Object`, since its constructor already exists and has no parameters.

These parameters are obtained as the logic lists `?InitializePairs`, `?ConstructorPairs` and `?SuperPairs`, using auxiliary predicates. The first of them is for finding the parameters of the initialize method of a class:

```
  diagramExtractInitializeParameters(?Diagram, ?ClassName, ?ParameterPairs) if
    findall(<?Type, ?Name>,
            and(umlAssociationEnd(?Diagram, ?ClassName, ?Class2, ?Role, navigable,
                                  ?Changeability, ?, ?Lower, ?Upper, ?),
                or(equals(?Lower, 1), equals(?Changeability, frozen)),
                associationEndInstanceVariableType(?Class2, ?Upper, ?Ordering, Type),
                associationEndInstanceVariableName(?Role, ?Upper, ?Name)),
            ?AssociationParameterPairs),
    findall(<?Type, ?Name>,
            and(umlAttribute(?Diagram, ?ClassName, ?Name, ?UmlType, ?, frozen),
                implementationTypeNameOf(?Type, ?UmlType)),
            ?AttributeParameterPairs),
    append(?AssociationParameterPairs, ?AttributeParameterPairs, ?BrutePairs),
    noDups(?BrutePairs, ?PairsAsSet),
    sorted(?PairsAsSet, ?ParameterPairs).
```

This predicate has two `findall` predicates. The first one collects all the types and names of the instance variables which represent associations with lower multiplicities equal to `1` or changeability equal to `frozen`. The second `findall` finds all the instance variables representing attributes which have the changeability `frozen`. Each of the `findall` predicates accumulates lists with type and name of the instance variables. The two resulting lists are them appended and the possible duplicated elements are removed. Before returning the pairs of parameters, the list is sorted in order to create some deterministic order in the list, because we cannot ensure the sequence in which the rules will be evaluated and collected by the `findall` predicates.

For obtaining the parameters of the constructor of a class, we use another auxiliary predicate, called `diagramExtractConstructorParameters`. This predicate is based on a recursion on the hierarchy of classes:

```
diagramExtractConstructorParameters(?Diagram, ?ClassName, ?ParameterPairs) if
  umlSuperclassOf(?Diagram, ?SuperClassName, ?ClassName),
  diagramExtractInitializeParameters(?Diagram, ?ClassName, ?MyInitializePairs),
  diagramExtractConstructorParameters(?Diagram, ?SuperClassName, ?SuperPairs),
  append(?MyInitializePairs, ?SuperPairs, ?ParameterPairs).
```

First of all, it gets the parameters of the initialize method of the class, using the predicate diagramExtractInitializeParameters, which we have just described. After that, it obtains the superclass of the class and the parameters of its constructor. To get these, the predicate `diagramExtractConstructorParameters` itself is used, causing a recursion which ends when the `?ClassName` variable is `Object`. The rule for this stop condition is the following:

```
diagramExtractConstructorParameters(?Diagram, Object, <>).
```

Notice that this rule does not overlap with the previous one, because when the previous one is called with `Object` as `?ClassName` the predicate `umlSuperclassOf` fails, since `Object` does not have a superclass.

Having all this information about the parameters, we can use it to format the code of the constructor:

```
generateConstructorCode(<?ClassName, public, ?ConstructorPairs, ?SuperPairs,
                         ?InitializePairs>, ?Code) if
  generateMethodArgumentList(?ConstructorPairs, ?ConstructorArgs),
  generateMethodCallArgumentList(?SuperParameterPairs, ?SuperArgs),
  generateMethodCallArgumentList(?InitializeParameterPairs, ?InitializeArgs),
  equals(?Code, {?Visibility ?ClassName(?ConstructorArgs) {
          super(?SuperArgs);
          initialize(?InitializeArgs);}})
```

This rule does mainly formatting of the receiving lists and puts it in a quoted code template for constructors. The predicate `generateMethodArgumentList` generates a string with the declaration of the parameters of a method from a logic list of pairs. For example, from the list `<<Client,buyer>,<LineItem,item>>`, it generates:

```
Client buyer, LineItem item
```

The predicate `generateMethodCallArgumentList` generates a string with the names of the parameters separated by commas.

Now that we have generated the constructors calling the initialize methods, we need to generate also the initialize methods themselves. Initialize methods are private instance methods which are responsible for the initialization of instance variables of the class. We generate one initialize method for each class. The rule for doing that is the following:

```
initializeToGenerate(<?ClassName, ?Code>) if
  umlClass(?Diagram, ?ClassName, ?),
  diagramExtractInitializeData(?Diagram, ?ClassName, ?ParameterPairs,
                               ?AssignPairs),
  generateInitializeCode(<?ClassName, private, ?ParameterPairs, ?AssignPairs>,
                         ?Code)
```

This rule takes each class from the diagram, obtains the information needed to build its initialize method and formats the code for the method using the rule `generateInitializeCode`. The predicate `diagramExtractInitializeData` returns all the assignments which have to be made in the initializer as a list of variable-value pairs, and also a list with the parameters of the initializer to allow the construction of its signature. The code of this predicate is the following:

```
diagramExtractInitializeData(?Diagram, ?ClassName, ?ParameterPairs,
                             ?AssignPairs) if
  umlClass(?Diagram, ?ClassName, ?),
  findall(<?Name, ?AssociationInitialValue>,
          and(umlAssociationEnd(?Diagram, ?ClassName, ?, ?Role2, navigable,
                                ?, ?, ?, n, ?Ordering),
              associationInitialValue(?Ordering, ?AssociationInitialValue),
              associationEndInstanceVariableName(?Role2, n, ?Name)),
          ?AssociationAssignmentPairs),
  findall(<?AttributeName, ?InitialValue>,
          and(umlAttributeInitialValue(?Diagram, ?ClassName, ?AttributeName,
                                       ?UmlInitialValue),
              implementationExpressionOf(?InitialValue, ?UmlInitialValue)),
```

```
        ?AttributeAssignmentPairs),
    append(?AttributeAssignmentPairs, ?AssociationAssignmentPairs, ?AllPairs),
    diagramExtractInitializeParameters(?Diagram, ?ClassName, ?ParameterPairs),
    findall(<?Name, ?Value>,
           and(member(<?Name, ?Value>, ?AllPairs),
               not(member(<?, ?Name>, ?ParameterPairs))),
           ?AssignmentPairs).
```

This is the largest among the predicates for code generation mapping strategies, however, its explanation is quite simple. The algorithm is the following: for each class in the diagram, for all of its navigable association ends with upper multiplicity equal to `n` of the class, the name of its representing variable and an initialization for it are stored in a list. This initialization value, in the case of Java, can be `new java.util.Vector()`, for ordered association ends, and `new java.util.Set()` for unordered ones. After that, variable-value pairs are also stored for each attribute with an initial value defined in the diagram. Notice that this initial value is translate to an implementation language expression by the predicate `implementationExpressionOf`. This can be used, for example, to translate UML expressions in higher-level languages such as OCL [17] to the actual implementation language. Having the assignment pairs, we need to remove some of them which can become superfluous since they are also being received as parameters of the initialize method. For example, an association end which has the changeability frozen and it is going to be received as parameter of the initializer cannot be assigned with a new instance of a container type.

Having now all the information to generate the initialize method, we only need to call the predicate `generateInitializeCode` to format its code. This predicate has two rules, and these rules have following code:

```
  generateInitializeCode(<?ClassName, ?Visibility, ?ParameterPairs, ?AssignPairs>,
                        ?Code) if
    generateMethodName(?Visibility, concrete, void, initialize, ?ParameterPairs,
                      ?MethodName),
    not(and(classWithName(?Class, ?ClassName),
           methodWithNameInClass(?, ?MethodName, Class))),
    findall(?Assign,
           and(member(<?Type, ?VariableName>,?ParameterPairs),
               equals(?Assign, { this.?VariableName  = ?VariableName;})),
           ?ParameterAssignmentList),
    findall(?Assign,
           and(member(<?Left, ?Right>, ?AssignPairs),
               equals(?Assign, {this.?Left = ?Right; })),
           ?ValueAssignmentList),
    append(?ParameterAssignmentList, ?ValueAssignmentList, ?AssignmentList),
    listAsSeparatedString(?AssignmentList, [Character cr asString], ?Assignments),
```

```
    equals(?Code, {?MethodName { ?Assignments }}).

  generateInitializeCode(<?ClassName, ?Visibility, ?ParameterPairs, ?AssignPairs>,
                         ?Code) if
    generateMethodName(?Visibility, concrete, void, initialize, ?ParameterPairs,
                       ?MethodName),
    classWithName(?Class, ?ClassName),
    methodWithNameInClass(?Method, ?MethodName, ?Class),
    methodWithSourceCode(?Method, ?Code).
```

The first of these rules verifies and succeeds only if there is no initialize method yet in the code. In this case, it collects of the assignments of parameters and all the assignments that were passed by bound to `?AssignPairs`. These assignments are then concatenated together separated by a line break. After that the name of the method and the assignments are inserted into a quoted code template to form the code of the initialize method.

In the second rule, we see another point in which we take the possible existing code into account for generation. This rule succeeds if there is already a method with this name in the code. In this case, the code of the generated initialize method is the code of the method which already exists, providing the developer with the ability to modify and adapt the initializations at his/her convenience.

6.2.3 Reverse-engineering mappings

In this section, we describe in detail the rules for the mapping strategies from code to class diagram.

As for the diagram-code mapping strategies, the code-diagram mapping strategies are queried by the transformation engine (reverse-engineering engine, in this case). As we saw in code generation, the queries from the transformation engine represent our goals, i.e., what we want to generate. In that case, we wanted to generate classes, instance variables and methods. Now, in reverse-engineering, we want to extract UML class diagram constructs from the code. For this, we have predicate headers which are similar to those of code generation, but related to the constructs we want to extract from the code. As our conceptual sense here is *extraction* of diagram rather than generation, the suffixes are `toExtract` instead of `toGenerate`:

```
classToExtract(?Description)
attributeToExtract(?Description)
attributeInitialValueToExtract(?Description)
operationToExtract(?Description)
generalizationToExtract(?Description)
associationToExtract(?Description)
```

The description term in all these predicates is a list with the same elements as the predicates for logic description of UML class diagrams seen in section 6.1. For example, the `classToExtract` predicate has a list with the same form as the predicate for the description of a UML class, `umlClass`:

```
classToExtract(<?Diagram, ?Name, ?Concreteness>)
umlClass(?Diagram, ?Name, ?Concreteness)
```

This is natural, since we want to extract all the characteristics representable in our UML class diagram logic description.

Library for UML Code Reasoning

We have organized the predicates of this mapping strategy as a library, called *Library for UML Code Reasoning*. This library is built on top of LiCoR and, apart from using it to do reverse-engineering, we can use it to make ad hoc queries about UML constructs which are present in the code. For example, we can perform a query for finding out what are all the associations present in the code of a diagram named GSMCase:

```
if dcAssociationEnd(GSMCase, ?AssociationEnd).
```

For the sake of organization, all the predicates of the Library for UML Code Reasoning have the name prefixed with “dc” (historical acronym for “diagram-code”).

The Library for UML Code Reasoning represents the conceptual mapping that we have chosen for this example and contains all of the mapping predicates for reverse-engineering.

Classes

The extraction of UML classes from the code is straightforward: one UML class for each source code class. This is done by the predicate `classToExtract`:

```
classToExtract(<?Diagram, ?Name, ?Concreteness>) if
  dcClass(?Diagram, ?Class),
  classWithName(?Class, ?Name),
  dcClassWithConcreteness(?Diagram, ?Class, ?Concreteness).
```

This rule uses the predicate `dcClass` to extract all the code classes which belong to a diagram. After that, the predicate `classWithName` of LiCoR is used to get the name of each class, and the predicate `dcClassWithConcreteness` gets its concreteness (abstract or concrete).

The predicate `dcClass` is very important since it delimits the diagram in the code. It does this using a namespace constraint. The classes of a diagram are the classes related in a determined namespace, which is related to the diagram through the rule `namespaceForDiagram`. [3] The predicate `dcClass` has the following code:

```
dcClass(?Diagram, ?Class) if
  dcNamespace(?Diagram, ?Namespace),
  classInNamespace(?Class, ?Namespace).
```

The predicate for determining the concreteness of a class, `dcClassWithConcreteness`, has two very simple rules:

```
dcClassWithConcreteness(?Diagram, ?Class, abstract) if
  dcClass(?Diagram, ?Class),
  abstractClass(?Class).
dcClassWithConcreteness(?Diagram, ?Class, concrete) if
  dcClass(?Diagram, ?Class),
  not(abstractClass(?Class)).
```

Both of these rules use the predicate `dcClass` to ensure that the variable `?Class` is bound to a class of this diagram. After that, they use the predicate `abstractClass` of LiCoR to verify if the class is abstract or not.

Attributes

According to our conceptual strategy, the extraction of attributes uses the convention that attributes represented in the code as instance variables of primitive types. The main rule for their extraction is the following:

```
attributeToExtract(<?Diagram, ?ClassName, ?Name, ?TypeName, ?Visibility,
                    ?Changeability>) if
  dcInstanceVariableForAttribute(?Diagram, ?, ?Class, ?Attribute),
  classWithName(?Class, ?ClassName),
  dcAttributeWithName(?Diagram, ?Attribute, ?Name),
```

[3] In Java, namespaces are packages.

```
    dcAttributeWithTypeName(?Diagram, ?Attribute, ?TypeName),
    dcAttributeWithVisibility(?Diagram, ?Attribute, ?Visibility),
    dcAttributeWithChangeability(?Diagram, ?Attribute, ?Changeability).
```

This rule uses the `dcInstanceVariableForAttribute` to obtain all the attributes of every class of the diagram. This attribute is a functor which contains information to identify it in the code. This functor is then used to recover all the characteristics of this attribute.

The predicate which really identifies an instance variable as the representation of an attribute is `dcInstanceVariableForAttribute`. It obtains each instance variable in the diagram which has a primitive type, and constructs a functor with its information:

```
dcInstanceVariableForAttribute(?Diagram, ?InstanceVariable, ?Class, ?Attribute) if
  dcClass(?Diagram, ?Class),
  instanceVariableWithTypeInClass(?InstanceVariable, ?Type, ?Class),
  dcPrimitiveType(?Diagram, ?Type),
  equals(?Attribute, attribute(?Class, ?InstanceVariable, ?Type)).
```

The predicates `dcAttributeWithName` and `dcAttributeWithTypeName` are very simple. They only analyze the functor to retrieve the respective information about the attribute. The extraction of visibilities and changeabilities, however, uses the advanced reasoning features of LiCoR and SOULJava, such as parse tree traversal.

For extracting the visibility of an attribute, we need to verify if there is an accessor for it. Given that this accessor exists, the visibility of the attribute is the visibility of the accessor method. If the accessor does not exist, then the visibility of the instance variable. The code follows:

```
dcAttributeWithVisibility(?Diagram, ?Attribute, ?Visibility) if
  dcInstanceVariableForAttribute(?Diagram, ?InstanceVariable, ?Class, ?Attribute),
  and(dcAccessorForInstanceVariable(?Diagram, ?Method, ?InstanceVariable, ?Class),
      methodWithVisibility(?Method, ?Visibility))).
dcAttributeWithVisibility(?Diagram, ?Attribute, ?Visibility) if
  dcInstanceVariableForAttribute(?Diagram, ?InstanceVariable, ?Class, ?Attribute),
  not(dcAccessorForInstanceVariable(?Diagram, ?, ?InstanceVariable, ?Class)),
  instanceVariableWithVisibility(?InstanceVariable, ?Visibility).
```

The first rule succeeds when there is an accessor method for the instance variable that represents the attribute, returning the visibility of this method as the visibility of the attribute. The second rule covers the case when there is no accessor method for the instance variable, returning the visibility of the instance variable as the visibility of the attribute.

Now a question can be raised: what is the definition of an accessor method?

An accessor is a method which is used to access the value of an instance variable. In Java, they usually have the form:

```
public VariableType getVariableName() {
  return VariableName;
}
```

However, different developers or companies can opt to choose another convention, for example, use only the name of the variable as the name of the method, or maybe to enforce some verification before returning the value.

Using logic metaprogramming, we have a very declarative and clear way to express the form of accessors, which can be easily changed, according to appropriate conventions. In our convention, we chose to use accessors in the way they are normally used in Java. Therefore, to extract and accessor, we analyze its name and its form to see if it conforms to our conventions.

To analyze the structure of a method, we use its logic parse tree. This parse tree is a logic functor which contains all the information about the structure of a method, including its return type, name, parameters, modifiers and body.

In the original version of SOUL, which reasons about Smalltalk, the parse tree of method is specific to this language. The syntax of Smalltalk is very simple making it very easy to express the body of a method through a logical functor. In Java, using SOULJava, we have a language-specific parse tree representation also but it is quite more complex than the one of Smalltalk, given the complexity of the syntax of Java itself (static typing, arrays, operators etc). The parse tree of a Java method has the following general format: [4]

```
method(?Class,
       signature(?Visibility, ?Modifiers, ?ReturnType, ?Name, ?Parameters),
       arguments(?ArgumentDeclarations),
       temporaries(?TemporaryVariables),
       statements(?Statements of method))
```

To illustrate this better, let us show the source code of a method and its corresponding logic parse tree. This is the code of the accessor for the instance variable `mainOperator` of a class `Main`:

```
private void setMainOperator(Operator mainOperator) {
  this.mainOperator = mainOperator;
}
```

And this is the corresponding logic parse tree:

[4] For clarity, we omitted some parts of the parse tree which are not relevant in this context.

```
method([Main],
       signature(private, <>, void, setMainOperator, <>),
       arguments(<variableDecl(Operator, mainOperator)>),
       temporaries(<>),
       statements(<assign(fieldAccess(variable(Main, this), mainOperator),
                          variable(Operator, mainOperator))>))
```

In fact, this is a normal data structure containing the representation of the abstract syntax tree returned by the Java compiler. The important point here is that the statements of the method are arranged in a recursive structure (e.g., nested method calls or control structures) which can be traversed by the backtracking engine in search for specific constructs. The determination of accessor methods and changeability of instance variables are an example of this.

Back in our mapping strategy predicates, we were looking for a way to find accessors for a given instance variable in the code. We do analyzing the parse three of every method in the class:

```
dcAccessorForInstanceVariable(?Diagram, ?Method, ?InstanceVariable, ?Class)
  dcClass(?Class),
  instanceVariableWithName(?InstanceVariable, ?VariableName),
  instanceVariableWithType(?InstanceVariable, ?Type),
  classWithName(?Type, ?TypeName),
  methodInClass(?Method, ?Class),
  stringCapitalizedOf(?Capitalized, ?VariableName),
  parseTreeOfMethod(?ParseTree, ?Method),
  equals(?ParseTree,
         method(?Class,
                signature(?, ?, ?TypeName, [ 'get' , ?Capitalized ], <>),
                arguments(<>),
                ?,
                statements(<return(variable(?TypeName, ?VariableName))>))).
```

This rule uses the predicate `dcClass` together with some predicates of LiCoR to obtain all the information about the possible accessor. The predicate parseTreeOfMethod is a predicate of LiCoR which returns the parse tree representation of a method. This parse three is unified through the predicate `equals` with our template representation of an accessor. This only succeeds when the parse tree has the same structure as the one defined in our template.

The determination of mutator methods follows the same idea, the only difference is the template for the parse tree.

Another characteristic needed for the extraction of attributes is the changeability. As we described in our conceptual mapping (table 6.1), the changeability of an attribute is `changeable`

if the instance variable which represents it in the code is written somewhere outside the initializer method, and frozen otherwise. Therefore, to extract the changeability of an attribute, we analyze all the methods of its class to verify if its instance variable is assigned with some value. [5] The main rule for this is `dcAttributeWithChangeability`:

```
dcAttributeWithChangeability(?Diagram, ?Attribute, changeable) if
  dcInstanceVariableForAttribute(?Diagram, ?InstanceVariable, ?Class, ?Attribute),
  one(and(methodInClass(?Method,?Class),
      not(dcInitializeInClass(?Diagram,?Method,?Class)),
      dcInstanceVariableAssignedWithValueInMethod(?Diagram, ?InstanceVariable, ?,
                                                  ?Method))).

dcAttributeWithChangeability(?Diagram, ?Attribute, frozen) if
  dcAttribute(?Diagram, ?Attribute),
  not(dcAttributeWithChangeability(?Diagram, ?Attribute, changeable)).
```

The first rule looks up for all the instance variables which represent attributes in a class. For each of these, it verifies if they are assigned with a value in some method of the class, except the initialize method. If so, then the changeability of the attribute is changeable.

The second rule just negates the first one, expressing that all the attributes whose instance variable is assigned with some value have the changeability set to `frozen`.

To find all the instance variables which are assigned with a value in a given method, we traverse the parse tree of this method searching for all assignments. The following predicate needs to receive a method bound to the variable `?Method`. It returns a solution for every different value assigned to that instance variable inside this method. This is the code of the predicate:

```
dcInstanceVariableAssignedWithValueInMethod(?InstanceVariable, ?Value, ?Method) if
  instanceVariableWithName(?InstanceVariable, ?VarName),
  findall(<?VarName, ?Value>,
          traverseJavaMethodParseTree(?Method, <?VarName, ?Value>,
                                      processAssignmentInMethod),
          ?BruteResults),
  noDuplicates(?BruteResults, ?Results),
  member(<?VarName, ?Value>, ?Results).

processAssignmentInMethod(assign(fieldAccess(variable(?, this), ?FieldName),
                                 ?Value),
                          <?FieldName, ?Value>).
```

[5] Notice that the good programming practice of generating all the instance variables as private in the code keeps us from having to verify assignments for it outside its class.

This rule finds all the values which are assigned to a variable. To do that, it makes use of the predicate `traverseJavaMethodParseTree` of LiCoR. This predicate traverses the parse tree of a method trying to satisfy a given predicate and returns one solution for each succeeding of it. The form of the `traverseJavaMethodParseTree` is:

```
traverseJavaMethodParseTree(?Method, ?Result, ?ProcessPredicateName)
```

The term `?ProcessPredicateName` is the name of the predicate which is use applied in each iteration of the tree traversal process. It must be a predicate with two terms, the first for trying to unify with the current parse tree element, and the other for formatting the result, in case of success of this unification.

Given so, we defined a predicate called `processAssignmentInMethod` which succeeds for assignments to instance variables and returns a pair with its name and the assigned value. In our rule, the name of the instance variable is then unified with `?VarName`, which contains the name of the instance variable for which we are searching assignments.

Attribute initial values

We have also to extract the predicates for the attributes which have an initial value. When we generated the code, we have mapped initial values of attributes to assignments inside the initialize method. This is exactly the strategy that we use here to extract initial values from the code. The main predicate follows:

```
attributeInitialValueToExtract(<?Diagram, ?ClassName, ?Name, ?InitialValue>) if
  dcAttributeWithInitialValue(?Diagram, ?Attribute, ?InitialValue),
  dcClassWithName(?Diagram, ?Class, ?ClassName),
  dcAttributeWithName(?Diagram, ?Attribute, ?Name).
```

This rule mainly manipulates the information from the attribute returned by the predicate `dcAttributeWithInitialValue`. This predicate searches for the initialize method of the class which contains the attribute and then, inside this method, tries to find an assignment of a literal to the instance variable which represents the attribute. For example, all the assignments inside the following method represent attributes with initial values:

```
private void initialize() {
  this.consumedEnergy = 0;
  this.available = false;
  this.invoiceType = "perMonth";
}
```

This is the code of the predicate `dcAttributeWithInitialValue`:

```
dcAttributeWithInitialValue(?Diagram, ?Attribute, ?InitialValue) if
  dcAttribute(?Diagram, ?Attribute),
  dcInstanceVariableForAttribute(?Diagram, ?InstanceVariable, ?Class, ?Attribute),
  dcInitializeInClass(?Diagram, ?Method, ?Class),
  dcInstanceVariableAssignedWithLiteralInMethod(?Diagram, ?InstanceVariable,
                                                ?ImplValue, ?Method)
  implementationExpressionOf(?ImplValue, ?InitialValue).
```

Conversely to what we have done when generating code, we translate the initial value expression from implementation language to a possibly higher-level one. To find the initializer of a class, the parse tree of the method is again used. In the predicate `dcInitializeInClass`, we search for a method in the class which has the name (not the signature) equal to "initialize". The parse tree template for this is:

```
method(?, signature(?, ?, ?, initialize, ?), ?, ?, ?))
```

Having the initialize method, we now need to search for assignment of literals to the instance variable representing our attribute. This process also requires logic analysis of the parse tree and is done by the predicate `dcInstanceVariableAssignedWithLiteralInMethod`:

```
dcInstanceVariableAssignedWithLiteralInMethod(?Diagram, ?InstanceVariable,
                                              ?Literal, ?Method) if
  dcInstanceVariableAssignedWithValueInMethod(?Diagram, ?InstanceVariable,
                                              literal(?, ?Literal), ?Method).
```

This rule calls the predicate `dcInstanceVariableAssignedWithValueInMethod`, which we have just discussed, with the value `literal(?, ?Literal)`. This functor is the representation used for literal assignments in the logic parse tree of methods. It has the general format:

```
literal(?Type, ?Value)
```

This searches for assignments of literals to the instance variable bound to `?InstanceVariable`.

Operations

In our conceptual mapping, UML operations are directly mapped to methods when generating code. But when doing reverse-engineering, the mapping is not simply bidirectional. There are methods in the code which we do not want to represent as operations in the diagram: accessors, mutators, add methods, initializers and constructors. This is our predicate for extracting operations from the code:

```
operationToExtract(<?Diagram, ?ClassName, ?OperationName, ?Parameters,
                   ?ReturnType, ?Concreteness, ?Visibility>) if
  dcOperationInClass(?Diagram, ?Operation, ?Class),
  classWithName(?Diagram, ?Class, ?ClassName),
  dcMethodWithName(?Diagram, ?Operation, ?OperationName),
  dcParametersOfMethod(?Diagram, ?Parameters, ?Operation),
  dcMethodWithReturnType(?Diagram, ?Operation, ?ReturnType),
  dcOperationWithConcreteness(?Diagram, ?Operation, ?Concreteness),
  dcOperationWithVisibility(?Diagram,? Operation, ?Visibility).
```

The main idea of this predicate is that the `dcOperationInClass` finds each method which represents an operation, and the other predicates simply do queries to LiCoR to obtain information about it, such as name and parameters. This information is represented in the method itself and does not need any advanced reasoning. The predicate `dcOperationInClass` is quite simple too:

```
dcOperationInClass(?Diagram, ?Operation, ?Class) if
  dcClass(?Diagram, ?Class),
  methodInClass(?Method, ?Class),
  not(dcAccessorForInstanceVariable(?Diagram, ?Method, ?, ?Class)),
  not(dcMutatorForInstanceVariable(?Diagram, ?Method, ?, ?Class)),
  not(dcInitialize(?Diagram, ?Method, ?Class)),
  not(dcAddForInstanceVariable(?Diagram,?Method, ?, ?Class)).
```

It finds all methods in all classes, then verifies if it is not an accessor, mutator, initializer or add method. Notice that constructors are not verified because they are not returned by the `methodInClass` predicate. From the negated predicates, we have already discussed the ones about accessors, mutators and initialize methods. The predicate `dcAddForInstanceVariable`, used for determination of add methods, is very similar to those for determination of accessors and mutators, only differing in the parse tree template.

Generalizations

As they are easily for being generated, generalizations are also easy to be extracted from the code. Every generalization in the source code (class-superclass relationship) is mapped to a UML generalization. The predicate that does that is the following:

```
generalizationToExtract(<?Diagram, ?SuperclassName, ?SubclassName>) if
  dcClass(?Diagram, ?Superclass),
  dcClass(?Diagram, ?Subclass),
  superclassOf(?Superclass, ?Subclass),
  classWithName(?Superclass, ?SuperclassName),
  classWithName(?Subclass, ?SubclassName).
```

One solution is returned for every pair of classes from the diagram which have a generalization relationship between them. Notice that generalizations which have one of the classes outside the diagram are not returned.

Associations

Before stating to discuss the extraction of associations, let us distinguish the associations from association ends. From the UML specification [32]:

> "An association end is an endpoint of an association, which connects the association to a classifier. Each association end is part of one association."

In our mapping strategy, we are only treating binary associations, so each association has two ends. This ends, however, are not necessarily both represented in the code. As we saw in section 6.2.2, only *navigable* association ends are represented in the code.

When extracting the diagram, though, we want the associations to treated as whole again and not as two separated ends. To do that, we extract all the association ends from the code and, afterwards, we merge them to create the associations. Let us discuss first how to extract the association ends.

As for code generation, the predicates for doing reverse-engineering of association ends are also quite lengthy, given the large amount of features an end has. The main predicate for extracting associations has the following code:

```
associationEndToExtract(<?Diagram, ?ClassName1, ?ClassName2, ?Role, ?Navigability,
                         ?Changeability, ?Visibility, ?Lower, ?Upper, ?Ordering>) if
  dcAssociationEndBetween(?Diagram, ?AssociationEnd, ?Class1, ?Class2),
  classWithName(?Diagram, ?Class1, ?ClassName1),
  classWithName(?Diagram, ?Class2, ?ClassName2),
  dcAssociationEndWithRoleName(?Diagram, ?AssociationEnd, ?Role),
  dcAssociationEndWithNavigability(?Diagram, ?AssociationEnd, ?Navigability),
  dcAssociationEndWithChangeability(?Diagram, ?AssociationEnd, ?Changeability),
  dcAssociationEndWithVisibility(?Diagram, ?AssociationEnd, ?Visibility),
  dcAssociationEndWithMultiplicity(?Diagram, ?AssociationEnd, ?Lower, ?Upper),
  dcAssociationEndWithOrdering(?Diagram, ?AssociationEnd, ?Ordering).
```

This rule looks up for all the association ends present in the code of the diagram using the predicate `dcAssociationEndBetween`. This predicate returns a functor in the variable `?AssociationEnd` which is then used to extract each feature of the association end, such as role name, multiplicity and ordering. Let us describe some of these predicates.

The most important one is `dcAssociationEndBetween`, which does the actual identification of an association end between a source class and a target class:

```
dcAssociationEndBetween(?Diagram, ?AssociationEnd, ?Class1, ?Class2) if
  dcInstanceVariableForAssociationEndInClass(?Diagram, ?, ?Class1,
                                             ?AssociationEnd),
  dcAssociationEndWithTarget(?Diagram, ?AssociationEnd, ?Class2).
```

An association end in the code is represented by an instance variable with a non-primitive type. These non-primitive typed instance variables split in two cases: container-typed and non-container-typed. For instance variables with types that are not container types, the target class is the type of the instance variable itself. The predicate for this is quite similar to the one we used for extracting attributes:

```
dcInstanceVariableForAssociationEndInClass(?Diagram, ?InstanceVariable,
                                           ?Class, ?AssociationEnd) if
  dcClass(?Diagram, ?Class),
  instanceVariableWithTypeInClass(?InstanceVariable, ?Type, ?Class),
  not(dcPrimitiveType(?Diagram, ?Type)),
  not(dcContainerTypeForOrdering(?Type, ?)),
  equals(?AssociationEnd, associationend(?Class, ?InstanceVariable, ?Type)).
```

The functor returned has three elements: the source class of the association end, the instance variable representing it and the target class. For instance variables with container types, there is another rule, which returns a functor with four elements: the source class, the instance variable, the container type and the target class:

```
dcInstanceVariableForAssociationEndInClass(?Diagram, ?InstanceVariable,
                                           ?Class, ?AssociationEnd) if
  dcInstanceVariableWithContainerTypeInClass(?Diagram,?InstanceVariable,
                                             ?ContainerType, ?Type, ?Class),
  equals(?AssociationEnd,
         associationend(?Class, ?InstanceVariable, ?ContainerType, ?Type)).
```

But discovering the type contained in the container type is not straightforward, since collections can accept any type of object. We need to infer the type contained inside this collection. To infer this type, we inspect all the methods that are called in (messages that are sent to) the instance variable that represents the association end. Methods like `add(Object)` or `remove(Object)` serve for manipulating the objects inside the collection. Using information from the parse tree of the method which calls the method in the collection, we can discover the type which is being passed to the collection.

Here, we can notice the advantage of using LMP. This kind of type inference would probably be very difficult to be implemented in a non-declarative language. Using the declarative nature of SOUL, we see that, beside being long, the rule for finding the type of the collection is quite intuitive:

```
dcInstanceVariableWithContainerTypeInClass(?Diagram, ?InstanceVariable,
                                           ?ContainerType, ?Type, ?Class) if
  instanceVariableWithTypeInClass(?InstanceVariable, ?ContainerType, ?Class),
  dcContainerTypeForOrdering(?ContainerType, ?),
  instanceVariableWithName(?InstanceVariable, ?VariableName),
  dcUnaryHandlingMessagesForContainerType(?Messages, ?ContainerType),
  findall(?AType,
        and(methodInClass(?M, ?Class),
            traverseJavaMethodParseTree(?M, send(?, fieldAccess(variable(?, this),
                                                                ?VariableName),
                                                                ?Message,
                                                  <variable(?AType, ?)>),
                                        isMessageSend),
            member(?Message, ?Messages)),
        ?RawCandidateList),
  noDuplicates(?RawCandidateList, ?CandidateList),
  chooseOneWithUser(?CandidateList,
                {Please choose the type for ?InstanceVariableName of ?Class: }]
                ?Type).
```

First, we obtain the type of the instance variable and verifies if it is a container type. If so, we obtain its name and the names of the messages which manipulate collections of that type. Having this, we traverse the parse tree of all methods of the class, looking for all the messages

sent to this instance variable. For that, we use the predicate `traverseJavaMethodParseTree` passing `isMessageSend` as the process predicate. We are only interested in the type that is passed as argument for the method call, and we bind it to the variable `?AType`. After the `findall`, `?RawCandidateList` is a list with all the types that have been found. After eliminating the duplicated types using `noDuplicates`, it may happen that there is more than one option of type in the list. For this, we use the predicate `chooseOneWithUser`. In cases where there is only one element in the list, this element is bound to `?Type`, otherwise, a list with the options appears to the user, who has to choose one among them.

Now that we have the functor with the association end, we have to extract its features. The predicate for extracting the role name (`dcAssociationEndWithRoleName`) uses the same predicate we used in code generation for mapping the role name to the name of the instance variable, `associationEndInstanceVariableName`. It maps plurals back to singular in case of associations with upper multiplicity `n`.

The navigability is an easy one: all the association ends in the code are navigable.

For the changeability of association ends, as established in table 6.1, we have five possible cases, depending on the upper multiplicity. For ends with upper multiplicity equal to `1`, the determination of changeability is mostly the same as for attributes: if assignments for the variable exist outside the initializer method, then it it `changeable`; `frozen`, otherwise. For multiplicities `n`, we have three possibilities: `changeable`, `addOnly` and `frozen` which are covered by the following a predicates:

```
dcAssociationEndWithChangeability(?Diagram, ?End, changeable) if
  dcAssociationEndWithUpperMultiplicity(?Diagram, ?End, n),
  one(or(dcAssociationEndChangeableByAssignment(?Diagram, ?End),
         dcAssociationEndChangeableBySend(?Diagram, ?End))).

dcAssociationEndWithChangeability(?Diagram, ?End, addOnly) if
  dcAssociationEndWithUpperMultiplicity(?Diagram, ?End, n),
  not(dcAssociationEndWithChangeability(?Diagram, ?End, changeable)),
  dcInstanceVariableForAssociationEndInClass(?Diagram, ?InstVar, ?Class, ?End),
  one(and(methodInClass(?Method, ?Class),
          not(dcInitializeInClass(?Diagram, ?Method, ?Class)),
          dcContainerAddingSend(?SendName),
          dcInstanceVariableWithSendInMethod(?Diagram, ?InstVar, ?SendName, ?,
                                             ?Method))).

dcAssociationEndWithChangeability(?Diagram, ?End, frozen) if
  dcAssociationEndWithUpperMultiplicity(?Diagram, ?End, n),
  not(dcAssociationEndWithChangeability(?Diagram, ?End, changeable)),
  not(dcAssociationEndWithChangeability(?Diagram, ?End, addOnly)).
```

The first of these predicates covers changeable association ends. According to table 6.1, an association ends is changeable if the variable which represents it is either written somewhere in the class outside the initializer, or there are calls to its `remove` or `set` methods. The predicate `dcAssociationEndChangeableByAssignment` searches for assignments just in the same way it is done for attributes or association ends with upper multiplicity `1`. To verify if the `remove` and `set` methods are called in the instance variable representing the association, the predicate `dcAssociationEndChangeableBySend` is used:

```
dcAssociationEndChangeableBySend(?Diagram, ?End) if
  dcInstanceVariableForAssociationEndInClass(?Diagram, ?InstVar, ?Class, ?End),
  one(and(methodInClass(?Method, ?Class),
      not(dcInitializeInClass(?Diagram, ?Method, ?Class)),
      dcContainerChangingSend(?Send),
      dcInstanceVariableWithSendInMethod(?Diagram, ?InstVar, ?Send, ?, ?Method))).
```

First, this rule obtains the instance variable which represents the instance variable. Then, it succeeds if at least one message is sent to the instance variable such that the message satisfies the predicate `dcContainerChangingSend`. These messages are `remove` and `set`:

```
dcContainerChangingSend(remove).
dcContainerChangingSend(set).
```

The predicate `dcInstanceVariableWithSendInMethod` does a parse tree traversal looking for message sent in a method:

```
dcInstanceVariableWithSendInMethod(?Diagram, ?InstanceVariable, ?MethodName,
                                    ?Arguments, ?Method) if
  traverseJavaMethodParseTree(?M, send(?, fieldAccess(variable(?, this),
                                           ?FieldName),
                                           ?MethodName,
                                     ?Arguments)).
```

The second rule for the changeability of association ends with multiplicity `n` covers the changeability `addOnly`. According to table 6.1, ends with `addOnly` changeability do not satisfy any condition to be `changeable` but have a call to the method. It first checks if the end is not `changeable`, then performs the same verification as `dcAssociationEndChangeableBySend`, but now looking for sends of the `add` message:

```
dcContainerAddingSend(add).
```

The rule for `frozen` n-multiplicity ends, as a typical "otherwise" rule, just denies the other two cases.

Now, to complete the description of the main predicate for extracting association ends which we presented in page 82, `associationEndToExtract`, we are only missing three features: visibility, multiplicity and ordering. For extracting the visibility, we look for the visibility of the accessors just like we do for attributes (page 6.2.3). The extractions of the upper and lower multiplicities are split in two by the following rule:

```
dcAssociationEndWithMultiplicity(?Diagram, ?End, ?Lower, ?Upper) if
  dcAssociationEnd(?Diagram, ?End),
  dcAssociationEndWithUpperMultiplicity(?Diagram, ?End, ?Upper),
  dcAssociationEndWithLowerMultiplicity(?Diagram, ?End, ?Lower).
```

To obtain the upper multiplicity is easy: if the instance variable representing the association end has a container type, then the multiplicity is `n`. Otherwise, it is `1`. This is done by the following rule:

```
dcAssociationEndWithUpperMultiplicity(?Diagram, ?End, n) if
  dcInstanceVariableForAssociationEndInClass(?Diagram, ?InstVar, ?Class, ?End),
  dcInstanceVariableWithContainerTypeInClass(?Diagram, ?InstVar, ?, ?, ?Class).

dcAssociationEndWithUpperMultiplicity(?Diagram, ?End, 1) if
  not(dcAssociationEndWithUpperMultiplicity(?Diagram, ?End, n)).
```

During the generation of code, we used the information about lower multiplicity of associations to generate the initialize methods. So we now search for this information in the initialize methods. An instance variable representing an association end with lower multiplicity equal to `1` is initialized with a parameter of the initialize method. The rules which state this is the following:

```
dcAssociationEndWithLowerMultiplicity(?Diagram, ?End, 1) if
  dcInstanceVariableForAssociationEndInClass(?Diagram, ?InstVar, ?Class, ?End),
  dcInitializeInClass(?Diagram, ?Method, ?Class),
  parametersOfMethod(?Parameters, ?Method),
  member(<?, ?Name>), ?Parameters),
  dcInstanceVariableAssignedWithValueInMethod(?, ?InstVar, variable(?, ?Name),
                                              ?Method).

dcAssociationEndWithLowerMultiplicity(?Diagram, ?End, 0) if
  not(dcAssociationEndWithLowerMultiplicity(?Diagram, ?End, 1)).
```

The first rule obtains the instance variable which represents the association end and verifies if, inside the initialize method of the class of this variable, there is an assignment of a parameter value to the variable. This is done by the predicate `dcInstanceVariableAssignedWithValueInMethod`, which we have already discussed. If such assignment exists, then the lower multiplicity is `1`. Otherwise, as the second rules states, it is `0`.

Finally, we extract the ordering of the association ends. For association ends with upper multiplicity `n`, we verify the type of the collection: if it is `java.util.Set`, then it is an `unordered` end; if it is `java.util.Vector`, then it is an `ordered` end. For ends with upper multiplicity `1`, the ordering is `notapplicable`. The rules for this are the following:

```
dcAssociationEndWithOrdering(?Diagram, ?End, ?Ordering) if
  dcInstanceVariableForAssociationEndInClass(?Diagram, ?InstVar, ?Class, ?End),
  dcInstanceVariableWithTypeInClass(?Diagram, ?InstVar, ?Type, ?Class),
  dcContainerTypeForOrdering(?Type, ?Ordering).

dcAssociationEndWithOrdering(?Diagram, ?End, notapplicable) if
  dcInstanceVariableForAssociationEndInClass(?Diagram, ?InstVar, ?Class, ?End),
  dcInstanceVariableWithTypeInClass(?Diagram, ?InstVar, ?Type, ?Class),
  not(dcContainerTypeForOrdering(?Type, ?Ordering)).

dcContainerTypeForOrdering([#'java.util.Set'], ordered).
dcContainerTypeForOrdering([#'java.util.Vector'], unordered).
```

The first rule covers ends with upper multiplicity equals to `n` and discovers the ordering based on the type of the instance variable, using the predicate `dcContainerTypeForOrdering`. The second rule is used for association ends that are represented by an instance variable which does not have container type. These have upper multiplicity equal to `1` and the ordering is always `notapplicable`.

Merging associations ends

Now that we have extracted the association ends, it is time to merge them into complete associations. To do this, we find all the pairs of association ends which are between the same classes and group their features together in one list. These are associations which have two navigable ends. Notice that, as we have already mentioned, we support only one association per pair of classes. Associations with one navigable end are created for the ends which do not have corresponding inverse end. The predicate for this is the following:

```
extractCodeAssociation(<?Diagram, ?ClassName1, ?ClassName2, ?Name, ?R1, ?R2, ?N1,
                        ?N2, ?C1, ?C2, ?V1, ?V2, ?L1, ?U1, ?L2, ?U2, ?O1, ?O2>) if
```

```
    dcAssociationEndBetween(?Diagram, ?End1, ?Class1, ?Class2),
    dcAssociationEndBetween(?Diagram, ?End2, ?Class2, ?Class1),
    classWithName(?Class1, ?ClassName1),
    classWithName(?Class2, ?ClassName2),
    smallerThan(?ClassName1, ?ClassName2),
    dcAssociationWithName(?Diagram, ?End1, ?Name),
    associationEndToExtract(<?Diagram, ?ClassName1, ?ClassName2, ?R1, ?N1,
                            ?C1, ?V1, ?L1, ?U1, ?O1>),
    associationEndToExtract(<?Diagram, ?ClassName2, ?ClassName1, ?R2, ?N2,
                            ?C2, ?V2, ?L2, ?U2, ?O2>),

  extractCodeAssociation(<?Diagram, ?ClassName1, ?ClassName2, ?Name, ?R1, ?R2, ?N1,
                         ?N2, ?C1, ?C2, ?V1, ?V2, ?L1, ?U1, ?L2, ?U2, ?O1, ?O2>) if
    dcAssociationEndBetween(?Diagram, ?End1, ?Class1, ?Class2),
    not(dcAssociationEndBetween(?Diagram, ?End2, ?Class2, ?Class1)),
    classWithName(?Class1, ?ClassName1),
    classWithName(?Class2, ?ClassName2),
    smallerThan(?ClassName1, ?ClassName2),
    dcAssociationWithName(?Diagram, ?End1, ?Name),
    associationEndToExtract(<?Diagram, ?ClassName1, ?ClassName2, ?R1, ?N1,
                            ?C1, ?V1, ?L1, ?U1, ?O1>),
    equals(<?R2,          ?N2, ?C2, ?V2, ?L2, ?U2, ?O2>,
           <nil, inaccessible, nil, nil, nil, nil, nil>).
```

The first of the rules for this predicate covers the cases when there are two inverted association ends between two classes. It finds the first association end and then tries to find another one between the same classes but with inverted direction. If we just extracted the association ends without giving an order to the class names, we would have all of them in double. This would happen because, for example, if we had two inverse ends $A \rightarrow B$ and $B \rightarrow A$, `?Class1` would be bound to A and `?Class2` would be bound to B in one solution. But there would be also another solution where `?Class1` would be bound to B and `?Class2` to A. To avoid this, we allow only one of these solutions to succeed by enforcing that the first class name must be alphabetically smaller (be first in the dictionary) than the second one. After this, we obtain the name of the association by means of the predicate `dcAssociationWithName` and the rest of the characteristics of the ends by means of the predicate `associationEndToExtract`.

The second rule is for the case when there is only one association end between two classes. In this case, the features of the inaccessible association end are left as `nil`. There is also another rule which is just the same as this second one but with the names `?Class1` and `?Class2` inverted in the first and second clauses.

The last diagram extraction predicate that we are going to discuss is `dcAssociationWithName`. As the reader may remember, we did not use the name of the association when generating code. Thus, we do not have from where to extract it in the code. Now is when the original diagram

takes place. We are going to consult it to obtain the information about the name of a given association. In case the given association did not exist before, we ask the user for a new name. The rules are:

```
dcAssociationWithName(?Diagram, ?End, ?Name) if
  dcAssociationEndBetween(?Diagram, ?End, ?Source, ?Target),
  classWithName(?Source, ?SourceName),
  classWithName(?Target, ?TargetName),
  umlAssociationAll(?Diagram, ?SourceName, ?TargetName, ?Name,
                    ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?).

dcAssociationWithName(?Diagram, ?End, ?Name) if
  dcAssociationEndBetween(?Diagram, ?End, ?Source, ?Target),
  classWithName(?Source, ?SourceName),
  classWithName(?Target, ?TargetName),
  not(umlAssociation(?Diagram, ?SourceName, ?TargetName, ?Name,
                    ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)),
  promptUser({Please enter a name for the association ?End:}, ?Name).
```

When we generated code, we consulted the original code in order not to override the code of existing methods. Is is analogous to what we are doing now, consulting the existing diagram to get the correct information about the association name.

6.2.4 Verifying the extraction

Our approach for round-trip engineering claims that when the same mapping strategies are used for code generation and reverse-engineering, we are able to generate code from a class diagram and extract the same class diagram from this generated code. For example, given a diagram D, if we generate code C from it using the conceptual mapping strategy S, and, without changing the code, if we extract a diagram D_{new} from C using the same strategy S, then D_{new} is equal to D.

We do not have yet a formal method to verify this process. There is no formalism for verifying if a *code generation* mapping strategy and a *reverse-engineering* mapping strategy follow the same *conceptual* mapping strategy.

But we can at least do stress tests to verify if our mapping strategies are consistent. For this purpose, we have developed a simple test tool to perform automated tests on the results returned by the round-trip engine. This tool consists of a set of logic predicates which compare the original diagram with the extracted one.

Our original diagram is declared by logic facts prefixed with "uml". In order not to mix the extracted predicates with the original ones, our round-trip tool, LURE, provides a parametrization of this prefix. Therefore we have a set of facts prefixed with "uml" for the original diagram, and a set of facts prefixed with "euml" for the extracted diagram.

Having this, we can define our set of predicates which are going to verify:

- What was extracted and should not have been.
- What was not extracted and should have been.
- What was extracted incorrectly.

All the predicates are split in two parts: identification and features. Identification is the minimal information needed for identifying that UML construct uniquely. Features are all the characteristics of the construct. Let us take the predicate class as an example. It is stated by the following rule:

```
umlClass(?Diagram, ?Name, ?Concreteness)
```

The two first parameters of this rule, `?Diagram` and `?Name`, serve for identification of the class while the third, `?Concreteness` is a feature of it.

All the constructs whose generation and extraction process are going to be verified must be declared in a `toVerify` fact, which has the form:

```
toVerify(?PredicateSuffix, ?IdentificationList, ?FeaturesList1, ?FeaturesList2)
```

The term `?FeaturesList2` is a repetition of `?Features1` but with different variable names. This is used to test incorrectly extracted facts. The verification of class facts, for example, is stated by the following fact:

```
toVerify(Class, <?Diagram, ?ClassName>, <?Concreteness1>, <?Concreteness2>).
```

After the declaration of the `toVerify` facts for all the UML constructs whose code generation and diagram extraction process need to be verified, we just have to run the query:

```
if verifyGenerationExtraction(?Error).
```

This query will return one result for each inconsistency (error) that it finds in the registered facts. Example of this inconsistencies are:

```
<missing, eumlAttribute, <GSMCase, Call, duration>>
<incorrect, eumlAttribute, <GSMCase, Call, startTime>, <Date, public, frozen>,
                                                        <Date, public, changeable>>
<inexcess, eumlAttribute, <GSMCase, Call, caller>>
```

The first list represents an inconsistency where an attribute is missing in the extracted diagram. In the original diagram, there is an attribute called `duration` in the class `Call`, but it was not extracted from the code. This can mean that there is a problem in the generation of attributes or in the extraction itself. The second inconsistency is an attribute which was extracted but incorrectly: the changeability, which should be `frozen`, was extracted as `changeable`. The third inconsistency is an attribute which was extracted from the code but does not exist in the original diagram. This could have been caused, for example, by the misrecognition of an association end as an attribute.

Implementation of the verification

The implementation of this verification illustrates how clear, well-organized and elegant the use of logic programming to reason about software systems is. By writing down four simple rules, we can built a very useful verification tool to check the correctness of the development using our tool. Let us take a quick look at these rules.

The first one is just a utility rule to concatenate the prefixes to the predicate name which is determined in the predicate `toVerify` with four terms:

```
toVerify(?Predicate1, ?Predicate2, ?Identification, ?Features1, ?Features2) if
  toVerify(?Predicate, ?Identification, ?Features1, ?Features2),
  originalUmlPrefix(?OriginalPrefix),
  equals(?Predicate1, [ (?OriginalPrefix , ?Predicate) ] ),
  extractedUmlPrefix(?ExtractedPrefix),
  equals(?Predicate2, [ (?ExtractedPrefix , ?Predicate) ] ).
```

Now, we define our three other rules which verify each of the inconsistency items that we mentioned in the previous section: missing fact, fact in excess and incorrectly extracted fact. This is the one which checks for missing facts:

```
verifyUml(?Error) if
```

```
    toVerify(?Predicate1, ?Predicate2, ?Identification, ?Features1, ?Features2),
    append(?Identification, ?Features1, ?All1),
    apply(?Predicate1, ?All1),
    append(?Identification, ?Features2, ?All2),
    not(apply(?Predicate2, ?All2)),
    equals(?Error, <missing, ?Predicate2, ?Identification>).
```

After getting the names and data about the predicates to be verified, this rule appends the lists of `?Identification` and `?Features1` (original diagram) to form the whole description of the class. Then `?Predicate1` is applied to this list to find all the classes that existed in the original diagram. After that, the same concatenation is done for `?Identification` and `?Features2`, resulting in `?All2`. Now, descriptions with the same identification of each class that was found in the original diagram are verified to exist in the extracted diagram, by applying `?Predicate2` to `?All2`. This is negated, that is, it succeeds only if the class that exists in the original diagram does not exist in the extracted diagram. If so, this rule `verifyUml` succeeds returning the information about the inconsistency.

The predicate for finding classes that should not have been extracted but were has the same form. It swaps `?Predicate1` and `?Predicate2`, and changes the way information is given about the error.

For the verification of UML constructs that were extracted with incorrect features, we declare another `verifyUml` rule:

```
  verifyUml(?Error) if
    toVerify(?Predicate1, ?Predicate2, ?Identification, ?Features1, ?Features2),
    append(?Identification, ?Features1, ?All1),
    apply(?Predicate1, ?All1),
    not(apply(?Predicate2, ?All1)),
    append(?Identification, ?Features2, ?All2),
    apply(?Predicate2, ?All2),
    equals(?Error,
           <incorrect, ?Predicate2, ?Identification, ?Features1, ?Features2>).
```

This beginning of this rule is the same as the one for verification of missing extractions, verifying all the classes that exist in the original diagram. After that, it ensures that each class was not *correctly* extracted by negating the application of `?Predicate2` to the results of `?Predicate1`. Having succeeded this step, the predicate generates a new list of terms (`?All2`) with the same identification but with different feature variables. It then finds the features that were extracted for that class by applying `?Predicate2` to this `?All2`. After confirming the inconsistency, information about it is returned as result of this rule: the name of the incorrectly extracted predicate, the identification of the class, the original features and the extracted features.

This simple verification predicates provide an interesting tool for developing and verifying mapping strategies. Using it, generation/extraction tests can be instantly and automatically verified.

6.3 Language independence

Up till now, all the examples we have shown are in Java. This was the main language we used for our experiments. However, one of our goals while conceiving our approach was not to make it language-specific, we wished to make it as language-independent as possible.

Evidently, we knew from the beginning that this was a very difficult goal, if possible at all. Language independence often has to do with language meta-models, their relations and transformations between them (e.g., [11]). Instead, we tried to reduce the set of language-specific constructs that we treated. We developed our approach having a generic object-oriented language in mind, with the usual constructs they contain: classes, instance variables and methods.

From this, we developed a code generation engine and a reverse-engineering engine that are language-independent. Language-independent in the sense that they do not refer to language specific constructs and can be used to do code generation or reverse-engineering of general object-oriented languages, given that (1) the language provides the constructs cited above and (2) there is an implementation of the MLI of LiCoR for this language.

For this thesis, we had two implementations of the MLI available for use: Java and Smalltalk. For most of the experiments, we used Java, essentially because it is a statically typed language, which makes it easier to extract information from the code (e.g., types of attributes or parameters). We also used Smalltalk for some experiments, mainly to test if what would be the issues encountered when doing a language-independent round-trip engineering tool.

Most of the problems are overcome with the architecture of LiCoR, which allows transparent reasoning about a base object-oriented language. Since the mapping strategies are built on top of LiCoR, they take advantage of this fact. The rule `classToExtract` listed below, for example, can be used both for Java and for Smalltalk:

```
classToExtract(<?Name, ?Concreteness>) if
  classWithName(?Class, ?Name),
  classWithConcreteness(?Class, ?Concreteness).
classWithConcreteness(?Class, abstract) if
  abstractClass(?Class).
classWithConcreteness(?Class, concrete) if
  not(abstractClass(?Class)).
```

It only uses predicates from LiCoR. Like this, when the MLI being used is for Smalltalk, Smalltalk classes are returned. If the MLI is for Java, i.e., we are using SOULJava, Java classes

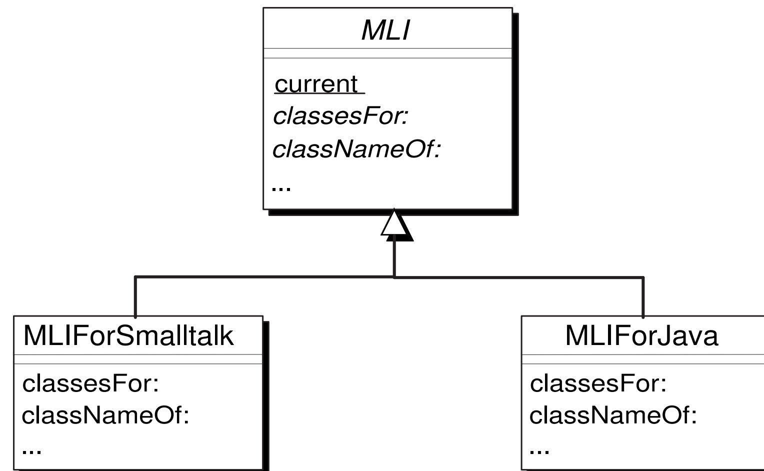

Figure 6.1: The hierarchy of the MLI with the abstract MLI and two concrete implementations: one for Java and one for Smalltalk.

are returned. The MLI is a Smalltalk façade which is responsible for the meta-level manipulation of the entities from the base language. To better understand how this mechanism for language independence works, let us take a look at the implementation of the predicate `classWithName`:

```
classWithName(?Class, ?ClassName) if
  class(?Class),
  equals(?ClassName, [ Soul.MLI current classNameOf: ?Class ])
```

The first rule called ensures that a class is bound to the variable `?Class`. This can be by just verifying is the bound atom is a class or by looking up for all the classes of the system, in case `?Class` is an open variable. Afterwards, a Smalltalk invocation to the MLI is made to obtain the name of the class. This name is then unified with the variable `?ClassName`.

The class `Soul.MLI` is a singleton which contains an instance of the active MLI, which can be `MLIForJava` or `MLIForSmalltalk`. The hierarchy of the MLI is illustrated by figure 6.1. The method `current` returns the current implementation, which is used to obtain the name of `?Class` in the previous example.

However, in some points of the mapping strategies, some differences between the languages appear that are above the level of the MLI. One major example of this are the templates of quoted code. An accessor for Java has the same essential structure of an accessor for Smalltalk, but, down in code, the syntax is very different.

To overcome this problem, we use the dynamic ability of SOUL to use variables as predicate names. The name of the active language is used to build the name of the predicate which has to be called. The example of the accessor becomes:

```
generateAccessorCode(<?Visibility, ?Type, ?VariableName>, ?Code) if
  currentLanguage(?Language),
  {generateAccessorCodeFor?Language}(<?Visibility, ?Type, ?VariableName>, ?Code).

generateAccessorCodeForSmalltalk(<?Visibility, ?Type, ?VariableName>, ?Code) if
  equals(?Code, {?VariableName
    ^ ?VariableName}).

generateAccessorCodeForJava(<?Visibility, ?Type, ?VariableName>, ?Code) if
  stringCapitalizedOf(?CapitalizedInstVar, ?VariableName),
  equals(?Code, {?Visibility ?Type get?CapitalizedInstVar() {
    return ?VariableName; }}).
```

Like this, the predicate `generateAccessorCode` in a language independent way. But the way the accessor is generated depends on the language which is being used. The active language is obtained through the predicate `currentLanguage`, which consults the MLI and returns a string with the name of the language.

6.3.1 Language independence issues

Let us discuss some of the problems we faced while implementing our tool for both Java and Smalltalk.

Most of the problems we encountered were the related to the absence of static types in Smalltalk. It is quite a difficult task to extract a class diagram with types from a language without static typing. This problem occurs in the when we need to know the types of parameters, instance variables and for inferring the types contained in collections.

In the case of instance variables, we made use of the Smalltalk type inference mechanism of LiCoR. This mechanism is an algorithm which traverses the parse tree of the methods of the class looking for the messages that are sent to the instance variable and the expression types which are assigned to it. The first possible types for the instance variable are obtained by finding the classes which understand all the messages sent to the instance variable. The rest are the expression types which are assigned to the instance variable. These expression types result, for example, from message sent for instance creation (e.g., `Contract new`).

All these possible values are returned by the predicate `classWithInstvarOfType`, which is specific to Smalltalk. Our rule stores these results in a list and, in case there is more than one option, the options are shown to the user, who has to choose one among them.

Having this, to make a language-independent predicate for determining the type of an instance variable, we implemented the three following rules:

```
dcInstanceVariableWithType(?InstanceVariable, ?Type, ?Class) if
  currentLanguage(?Language),
  {dcInstanceVariableWithTypeFor?Language}(?InstanceVariable, ?Type, ?Class).

dcInstanceVariableWithTypeForJava(?InstanceVariable, ?Type, ?Class) if
  instanceVariableWithType(?InstanceVariable, ?Type, ?Class).

dcInstanceVariableWithTypeForSmalltalk(?InstanceVariable, ?Type, ?Class) if
  findall(?OneType,
          classWithInstvarOfType(?Class, ?InstanceVariable, ?Type),
          ?ListOfTypes),
  chooseOneWithUser(?ListOfTypes,
                    {Choose a type for ?InstanceVariable of class ?Class},
                    ?Type).
```

The first rule just chooses the appropriate predicate to call, based on the current language. If we are using Java, the regular predicate `instanceVariableWithType` is called and immediately returns the correct type of the instance variable. If we are using Smalltalk, the rule `dcInstanceVariableWithTypeForSmalltalk` uses the predicate `classWithInstvarOfType` of LiCoR to collect all the possible types for the instance variable. After collecting them, the predicate `chooseOneWithUser` is used to show the user the list of possible types. If there is only one option, it is chosen automatically.

For parameters, return types and types contained in collections, we did not use type inferencing because it was not available in LiCoR. So to workaround this problem, we used a simple tagging mechanism by means of logic facts. So, if the developer adds a method or an instance variable with a collection type, he must also add a fact stating the type information about it. This way, the mapping strategies can consult these facts to correctly reverse-engineer the code. Example of these facts are:

```
typingTag(operation, GSMCase, Call, addToInvoice, void, <Double, Integer>).
typingTag(container, GSMCase, Contract, clients, <Client>).
```

The first facts declares the typing information about the parameters and return type of the operation `addToInvoice` of class `Call`. The second one declares that the type contained in the collection `clients` of the class `Contract` is `Client`.

These were the artifices we implemented in order to recover type information in Smalltalk. Another option would be to work with non-typed diagrams, i.e., diagrams without explicit types for the attributes and parameters. But we have not done experiments with that.

Apart from this, there are some other points in which we need separation between predicates for Smalltalk and for Java. Here are some of them:

Parse tree traversal The parse trees of methods are very different between Smalltalk and Java. The cause for that is, again, that Java has static types and Smalltalk does not. The parse tree of Smalltalk is quite simple, reflecting the simplicity of the syntax of this language. In contract, the parse tree of a Java method is far more complex since it contains all type information about every expression, assignment, parameter or variable. Therefore, every time a parse tree has to be analyzed, two different predicates has to be created, one for Java and one for Smalltalk.

Code generation formatting This is an inherent consequence from the fact the we are working with different *languages*: the code is different. Therefore, the predicates for formatting code have to be split in two rules as well. One example of that is the predicate `generateAccessorCode`, shown in page 94.

Code conventions Smalltalk standard coding conventions are different than the ones of Java. The usual example is that accessors in Smalltalk have the name of the instance variable and, in Java, they follow the JavaBeans naming convention [37] (get*VariableName*). This also implies the creation of separate rules for Java and Smalltalk.

6.4 Lure — our tool for round-trip engineering

LURE[6] is the prototype tool which resulted from our experiments on doing round-trip engineering. It allows the development of Java applications and can be extended to support Smalltalk. The default mapping strategies are the ones we showed in chapter 6, but other strategies can be implemented and installed according to the preferences of the developer. This section serves as a brief demonstration and user instructions.

LURE is essentially based on SOUL, LiCoR and SOULJava. Like these, LURE is built on top of a Smalltalk IDE, Cincom VisualWorks [6]. Therefore, to use LURE, the user has to start VisualWorks and load the following bundles of classes (preferably in this order): SOUL, LiCoR, SOULJava and Roundtrip. After loading the last bundle, all the rules of LURE are automatically loaded into the logic repository of SOUL and the tool can already be used.

6.4.1 Starting to use Lure

There are two ways to start using LURE: from a UML class diagram or from Java source code. The instructions for doing this follow.

From a UML class diagram A design description of the system (in this case, a class diagram) is a usual way to start developing a software system. Our class diagram, as we discussed,

[6] The name LURE derives from Logic for UML Round-trip Engineering.

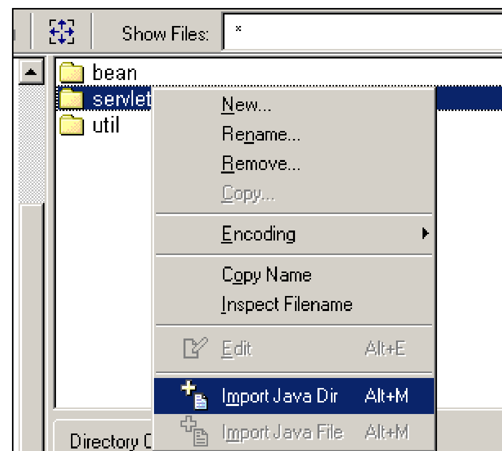

Figure 6.2: Importing a directory with Java source files.

is represented by logic facts. The developer can manually create these facts to declare his diagram. This can be a quite annoying process for diagrams with more than half a dozen classes, so another way to do it is to create the diagrams using a CASE tool, and import them using the LURE XMI interface. This process is detailed in section 6.4.3.

From source code Instead of starting from a design description, the developer can start by using existing code. To do this, the import facility of SOULJava is used. The user must open the file browser of VisualWorks and browse to the files to be imported. These can be single Java source files or a directory containing them. Figure 6.2 shows the pop-up menu containing the option "Import Java Dir" which appears when a folder is right-clicked with the mouse. Choosing this option causes all the `*.java` files inside that directory to be imported into the Java code repository.

After the import of the source code or of the diagram, one more small configuration has to be made. The user has to declare a logic fact called `namespaceNameForDiagram` which states the name of the namespace (package, in case of java) in which are the classes of the diagram. This is namespace where the classes of the diagram are generated. It servers also as a restriction of scope for reverse-engineering, making only the classes from this namespace to be extracted. Therefore, all the classes of a diagram must be in the same namespace. One example of declaration of this logic fact is:

```
namespaceNameForDiagram(gsmcase, GSMCase).
```

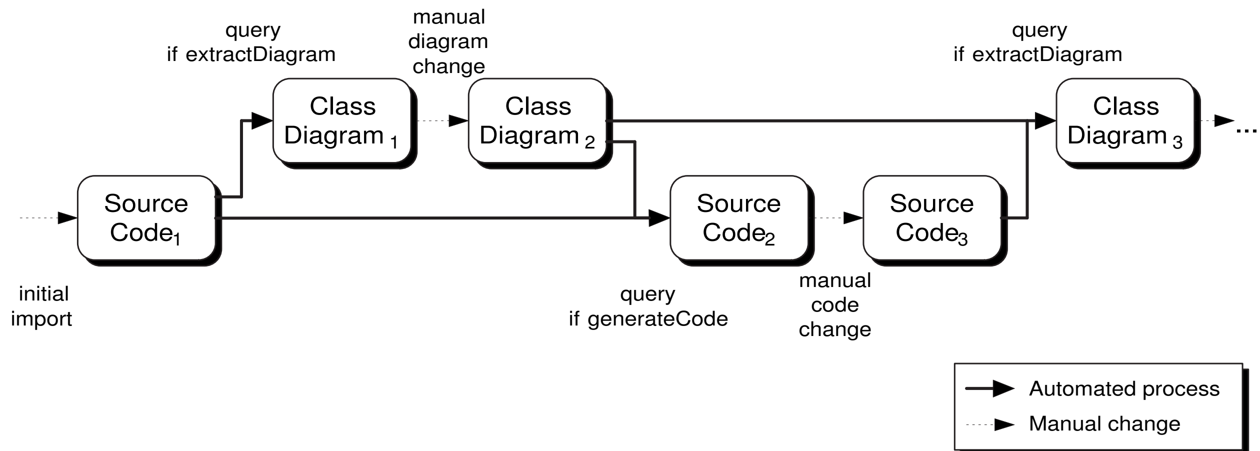

Figure 6.3: The process of development using LURE.

6.4.2 Round-trip engineering with Lure

The process of development using *Lure* is simple and it is illustrated in figure 6.3. Let us say that the developer imports a set of Java files with the source code of an application into the environment ($SourceCode_1$). At the time he wants to change the design of this application, he has to execute the following query:

```
if extractDiagramAndOverride
```

This extracts a UML class diagram from the code and declares it with facts in the logic repository of SOUL ($ClassDiagram_1$). The predicate `extractDiagramAndOverride` is similar to `extractDiagram`, which we have seen in chapter 6, with the difference that, after extracting, it removes the previously existing diagram and saves the extracted one with the prefix `uml`. The predicate `extractDiagram` does not do this in order to allow the verification of extraction (section 6.2.4). The changes the developer performs in this diagram result in a modified diagram, represented by $ClassDiagram_2$ in figure 6.3. Now, when the developer wants to edit the application at source code level, he needs to update the existing code ($SourceCode_1$) using the information from the modified diagram ($ClassDiagram_2$). To do this, the following query is executed:

```
if generateCode
```

Having done that, the code is up-to-date with respect to the diagram and can be freely modified by the developer. When he wants to change the application at design-level again, the query for extracting code must be executed again. The only difference is that now a diagram already

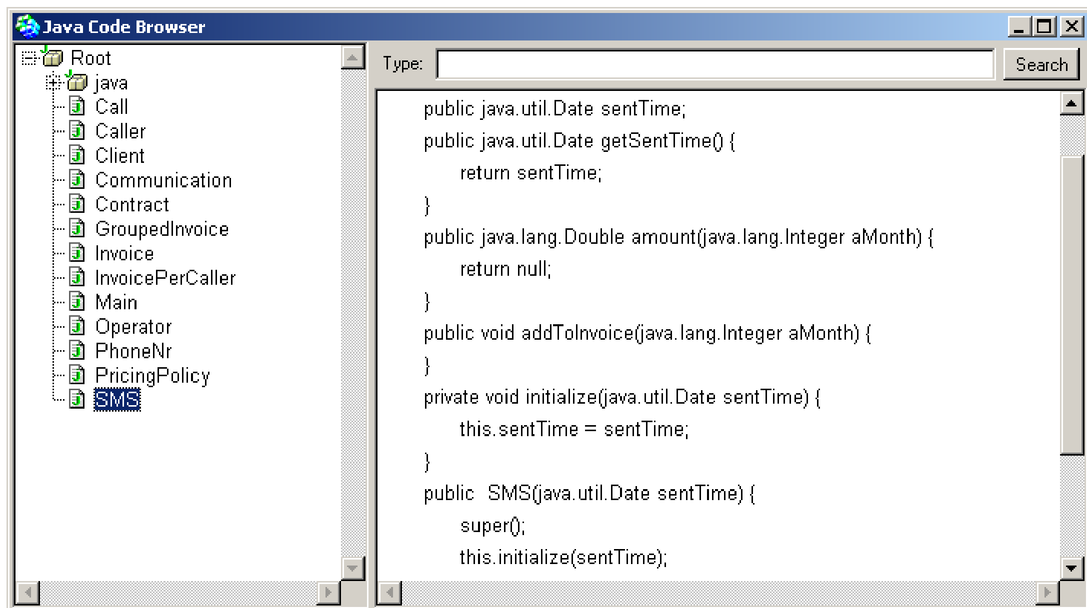

Figure 6.4: The Java Code Browser of SOUL Java.

exists ($ClassDiagram_2$) and is used together with the source code ($SourceCode_3$) to generate a new diagram ($ClassDiagram_3$). From this point on, the cycle starts again, alternating between code generations and diagram extractions.

6.4.3 Editing code and diagram

The Java code repository where the source code is stored can be browser through the SOULJava Code Browser. This tool, illustrated in figure 6.4, provides basic functionalities for editing the code from the repository.

If the developer wants to use its preferred IDE to edit the Java code, it can be exported using the predicate `exportPackageWithName`. This rule exports all the classes in a package to a directory with a given name. An example of use of this predicate is:

```
if exportPackageWithName(gsmcase, ['/home/joao/workspace/GSMCase/src/']).
```

This exports the classes of the package `gsmcase` to the indicated directory. To import the modified code again into LURE, the developer can use the code import facility of SOULJava, as already described.

The environment of LURE itself does not provide a possibility for graphic manipulation of class diagrams. Inside the tool, the only way to modify a class diagram is to edit its logic facts.

However, LURE provides an XMI [31] interface for exchanging diagrams with other tools. The developer can export the class diagram to an XMI document. And this document can be imported by a CASE tool that supports graphic visualization and editing of class diagrams. To update the code with the diagram again, the diagram can be imported back into LURE also using the XMI interface.

To export a diagram, the query exportDiagramWithName has to be executed:

```
if exportDiagramWithName(gsmcase, ['/home/joao/gsmcase.xmi']).
```

This example saves the diagram gsmcase as an XMI file with the indicated name.

For our experiments, we used Gentleware Poseidon version 1.6.1_02, a CASE tool for UML modelling. It imports XMI documents and allows graphic visualization and editing of the diagrams. It also exports the diagrams as XMI documents. A diagram exported from Poseidon can be imported into LURE using the predicate `exportDiagramWithName`. One example of that is:

```
if importDiagramWithName(gsmcase, ['/home/joao/gsmcase.xmi']).
```

This query creates the logic facts for the diagram `gsmcase`, importing them from the indicated XMI document into LURE.

Conclusion

Declarative programming — of which logic programming is a particular kind —, in the strong sense, means that programs are theories and all control information is supplied automatically by the system. [22].

In this chapter, indeed, we have not seen any declaration of control information, nothing stating how to generate code from a class diagram or how to extract a diagram from the code. Every logic rule or fact we showed here stated either code patterns for UML constructs or UML patterns for code constructs. In the terminology we assumed, we just stated *mapping strategies*.

In fact, this term turns out to be very appropriate. We have a set of strategies available, all of them declared in the logic repository, representing our knowledge (or, why not, our theory) about how code is mapped to diagram and vice versa. The code generation and reverse-engineering engines, which represent the automated (mechanic) part of the system, query these strategies in order to perform their tasks.

Using this approach, we achieve a true separation between the mechanic part, with its imperative nature, and the knowledge part, with its declarative nature. The main benefit of this separation is ease to modify the strategies without having to touch the transformation engine. Moreover, this modification — and the actual declaration — of the mapping strategies is done through a logic programming language, a kind of programming that is well-known for its ease and power for declaring and reasoning about knowledge bases.

As we showed in section 6.2.3, the backtracking engine of SOUL together with LiCoR can be very handy to perform quite advanced reasoning about source code structure. We can, for example, discover all the messages that are sent to all the instance variables of all classes of the system by means of just one query:

```
if instanceVariableWithNameInClass(?InstVar, ?VarName, ?Class),
   methodInClass(?Method, ?Class),
   traverseJavaMethodParseTree(?Method, send(?, fieldAccess(variable(?, this),
                                     ?VarName), ?Message, ?), isMessageSend).
```

We have also shown that the structure of the meta-level interface (MLI) of LiCoR, the languages facilities of SOUL and a good design of the mapping strategies predicates can lead to an interesting solution to language-independence. We demonstrated this by showing examples of predicates for transparent multi-language reasoning about Java and Smalltalk code.

Finally, we described Lure, the concrete implementation of our approach for round-trip engineering using LMP.

Chapter 7

Case Study — GSMCase

In this chapter, we describe some experiments that we have made to validate our approach. They were mainly based on a case study called *GSMCase*. First, we give a short description of the GSMCase application and then we demonstrate how LURE is used to do round-trip engineering with this application. We show how GSMCase evolves from a simple class diagram to a fully-working application.

7.1 Case defintion

7.1.1 Requirements

The first thing to do is to define our case. GSMCase is a small application for simulating the billing of GSM telecommunication systems. It does not have a graphic user interface and the only use case is the following:

1. Generate simulation data about contracts, clients, calls etc.

2. Compute and print the invoices for all the generated data.

When the application is started, the main method should be responsible for performing these three operations in order.

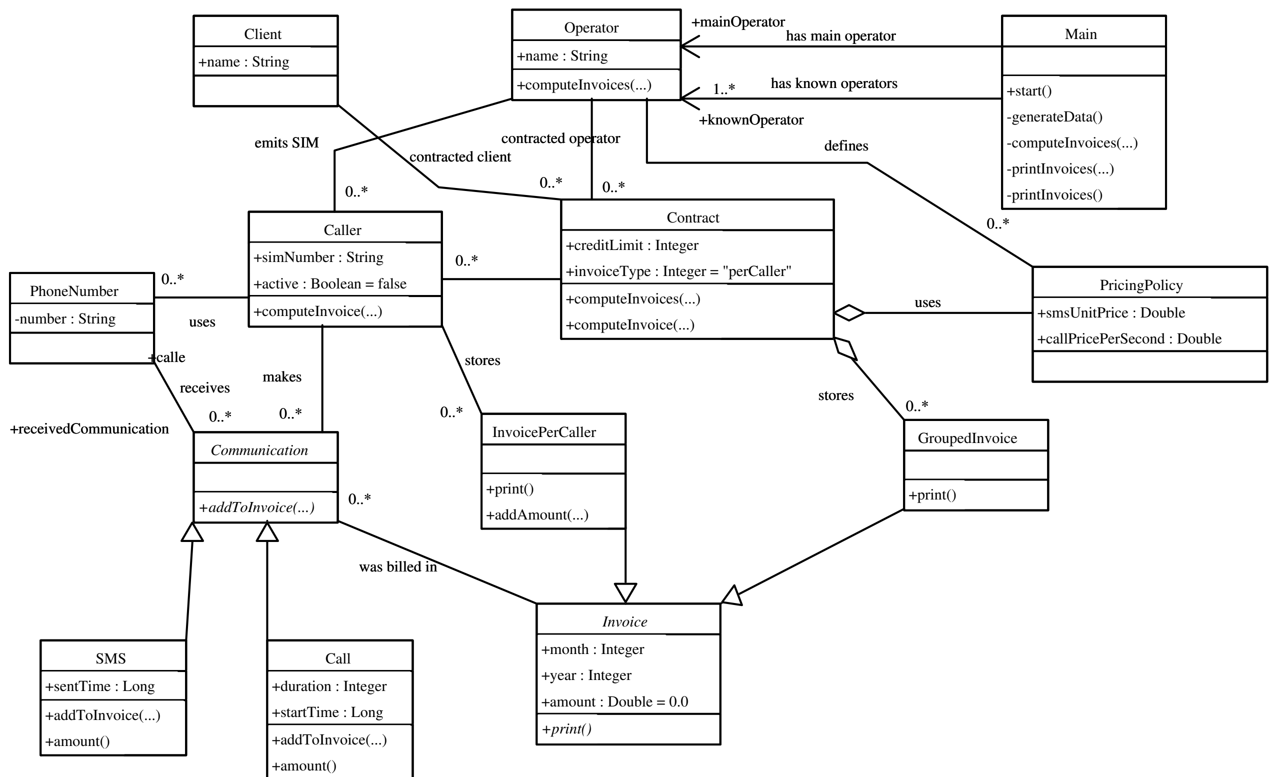

Figure 7.1: The initial class diagram of our case study, GSMCase.

7.1.2 Initial diagram

We start the development of our system by defining the initial design, described as a UML class diagram. This diagram, illustrated in figure 7.1, contains thirteen classes and fourteen associations.

Unfortunately, some of the features in this diagram are not yet supported by our tool. These constructs are aggregations (`Contract`–`GroupedInvoice` and `Contract`–`PrincingPolicy`) and more than one association between two classes (`Main`–`Operator`). In the case of the aggregations, we declare them as regular associations. In the case where there is more than one association between one class, we leave them both declared in the code.

The class diagram of GSMCase is defined using logic facts. They follow the same representation which we have described in section 6.1. Some examples of the facts that describe the class diagram of GSMCase are the following:

```
umlClass(GSMCase, Main, concrete).
```

```
...
umlAttribute(GSMCase, Caller, simNumber, String, public, frozen).
...
umlAttributeInitialValue(GSMCase, Contract, invoiceType, [ 'perCaller' ]).
...
umlOperation(GSMCase, Caller, computeInvoice, <<Integer, aMonth>>,
             void, concrete, public).
...
umlClassGeneralizes(GSMCase, Communication, SMS).
...
umlAssociation(GSMCase, Communication, Invoice, [#'was billed in'],
               communication, invoice, navigable, navigable, changeable,
               changeable, public, public, 1, 1, 0, n, ordered, ordered).
...
```

7.2 Round-trip engineering

To describe the experiment of developing GSMCase using Lure, this section tells a summarized history about how it was started, the changes we have chosen to make and the how we have reached the final product, a working application for simulation of billing for GSM companies.

7.2.1 First code generation

Having declared the class diagram, we can now start the process of round-trip engineering. The first task is to generate code from the initial class diagram. As described in chapter , this is done by the query:

```
if generateCode
```

This first code generation process takes approximately 14 seconds to be executed [1]. It generates the whole skeleton for the system: all the classes, instance variables for attributes and association ends, constructors, initializers, add methods, accessors, mutators and empty methods for operations. Table 7.1 lists the number of UML constructs in the initial diagram and the number of code constructs in the generated code. As expected, the number of classes in the code is the same as in the diagram. The number of instance variables generated is the sum of navigable association ends and attributes. The large amount of methods is related to the number of accessors, mutators, add and initialize methods.

[1] The time measurements were carried out in a PC with a Pentium III processor, 650 MHz, 128 MB of RAM, running Windows 2000.

UML constructs in GSMCase	
Class	13
Generalization	4
Operation	18
Attribute	15
Associations	14 (28 ends)

(a)

Code constructs in GSMCase	
Class	13
Method	118
Constructor	13
Instance variable	39

(b)

Table 7.1: (a) Constructs in the initial UML class diagram of GSMCase. (b) The constructs that exist in the code after the first code generation.

Let us pick one of the generated classes to discuss about the generated constructs. The class `PhoneNr` represents a GSM phone number. A phone number is related to its owner (`Caller`) and accumulates all the communications (`Communication`) that have been made to it. The code generated for the class `PhoneNr` can be seen in figure 7.2.

Since the class `PhoneNr` is not the child class of any generalization in the diagram, in the code it is generated as a subclass of `Object`. The first instance variable, `receivedCommunications`, represents the association end of `PhoneNr` in the `association` PhoneNr-`Communication`. Since a `PhoneNr` can receive zero or more communications (short text messages (`SMS`) or calls (`Call`)), this association end is represented by a collection type. Since the `Communication` end is ordered, this collection type is `java.util.Vector`. The end on the `Communication` side of the association has the changeability `addOnly`, so no mutator was generated for this instance variable. The generated methods were one accessor, `getReceivedCommunications`, and one add method, `addReceivedCommunication`.

The second instance variable is `caller` and represents the association between `PhoneNr` and `Caller`. In the class diagram, the association end in the `Caller` side has the changeability `changeable`, so, besides the acessor, a mutator was also generated for the `caller` instance variable.

The third instance variable, `phoneNumber`, is the actual text representation of the phone number and represents the UML attribute with the same name. Since the attribute has changeability `frozen`, only an accessor was generated for it, and not a mutator. Another fact to notice is the type of this variable: `java.lang.String`. The diagram we have built did not contain any language-specific type like this one. The type `String` of the diagram was translated into `java.util.String` during the generation of the instance variable.

As for all classes, a constructor was generated for this class which calls the constructor of the super class and the initialize method of this class. The initialize method receives the parameter `phoneNumber`, which corresponds to the value which must be assigned to the instance variable `phoneNumber`, which represents a `frozen` attribute. The initialize method also assigns a new

instance of `java.util.Vector` to the variable `receivedCommunications`, in order to correctly initialize the association end.

7.2.2 Implementing the simulation data

Now, we are going to start to insert behavior in our system, that is fill in the method bodies and get the code working. The next functionality that we are going to implement now is the generation of the simulation data. This consists of creating some instances of the classes of GSMCase with hypothetic values for attributes, and connecting these instances through the class associations.

Initially, we had planned that this should be in the class `Main`, but then we changed our minds and decided to create another separate class to take care of this data creation. We called this class `DataGenerator` and it has a one-to-one association with the class `Main`. Another change we wanted to perform in the code was addition of a method called `getMainPhoneNr` in the class `Caller`, to make it easier to get the first phone number of that caller (notice in the diagram that a caller can have `n` phone numbers).

7.2.3 Verifying the changes

Our next step now is to take a look at how the diagram stayed after this change in the code. Before that, we are going to make use of our verification tool (section 6.2.4) to verify all the exact changes between the original diagram and the one that is going to be extracted now. As we described in section 6.4.2, we can extract the diagram without overriding the previous one by running the query:

```
if extractDiagram
```

In this experiment, the extraction query took about 63 seconds to extract a class diagram from the code and stored it in the logic repository of SOUL. Using the notation we defined in section 6.2.4, some of the differences we found were:

```
<inexcess, eumlClass, <GSMCase, DataGenerator>>
<inexcess, eumlAssociationEnd, <GSMCase, DataGenerator, Main>>
<inexcess, eumlOperation, <GSMCase, DataGenerator, generateData,<>>>
<inexcess, eumlOperation, <GSMCase, DataGenerator, generateDuration,<>>>
...
<inexcess, eumlOperation, <GSMCase, Caller, getMainPhoneNr,<>>>
```

```
package gsmcase;

public class PhoneNr extends Object {
  public java.util.Vector receivedCommunications;
  public Caller caller;
  public java.lang.String phoneNumber;
  public java.util.Vector getReceivedCommunications() {
    return receivedCommunications;
  }
  public Caller getCaller() {
    return caller;
  }
  public java.lang.String getPhoneNumber() {
    return phoneNumber;
  }
  private void initialize(java.lang.String phoneNumber) {
    this.phoneNumber = phoneNumber;
    this.receivedCommunications = new java.util.Vector();
  }
  public void addReceivedCommunication(Communication receivedCommunication) {
    (this.receivedCommunications).add(receivedCommunication);
  }
  public PhoneNr(java.lang.String phoneNumber) {
    super();
    this.initialize(phoneNumber);
  }
  public void setCaller(Caller caller) {
    this.caller = caller;
  }
}
```

Figure 7.2: Code generated for the class `PhoneNr`.

7.2.4 Changing the design

Our next task is to implement the algorithm to compute the invoices and print them in the console. However, by taking a closer look at how we have initially designed the system, we have just realized that an interesting improvement can be done in the class structure. Instead of implementing a huge algorithm with nested loops iterating through all the operators, contracts etc, we can implement a *visitor design pattern* [13] to take care of the invoice computations and printings.

To adapt our classes to work with the visitor, we create a class `Visited` and make the classes `Operator`, `Contract` and `Caller` be a subclass of it. We create also a `Visitor` class and a specialization of it called `InvoicerVisitor`. The class `InvoicerVisitor` is our concrete visitor, which "visits" all the classes calculating and printing the invoices.

When doing this, we have also realized that the class GroupedInvoice is not needed anymore and, from now on, all the invoices are calculated per caller.

To perform these changes, some new facts and rules had to be added to the diagram:

```
umlClass(GSMCase, Visited, abstract).
umlClass(GSMCase, Visitor, abstract).
umlClass(GSMCase, InvoicerVisitor, concrete).
umlClassGeneralizes(GSMCase, Visitor, InvoicerVisitor).
umlClassGeneralizes(GSMCase, Visited, ?Sub) if
  member(?Sub, <Operator, Contract, Caller>)
umlOperation(GSMCase, Visitor, visitCaller, <<Caller, caller>>,
             void, concrete, public).
umlOperation(GSMCase, Visitor, visitContract, <<Contract, contract>>,
             void, concrete, public).
umlOperation(GSMCase, Visitor, visitOperator, <<Operator,operator>>,
             void, concrete, public).
umlOperation(GSMCase, Visited, accept, <<Visitor,visitor>>,
             void, abstract, public).
```

Notice that we generate `visitXXX` methods as concrete methods. Like this, they will be generated with empty bodies, leaving the overriding as optional for the subclass. In the case of the `accept` method, we declare it as abstract, because each class will have to implement it in its own way.

7.2.5 Finishing the implementation

After properly updating the diagram, we executed the code generation query one more time. This time, code was generated in 34 seconds. Now that we have generated code for all the visited

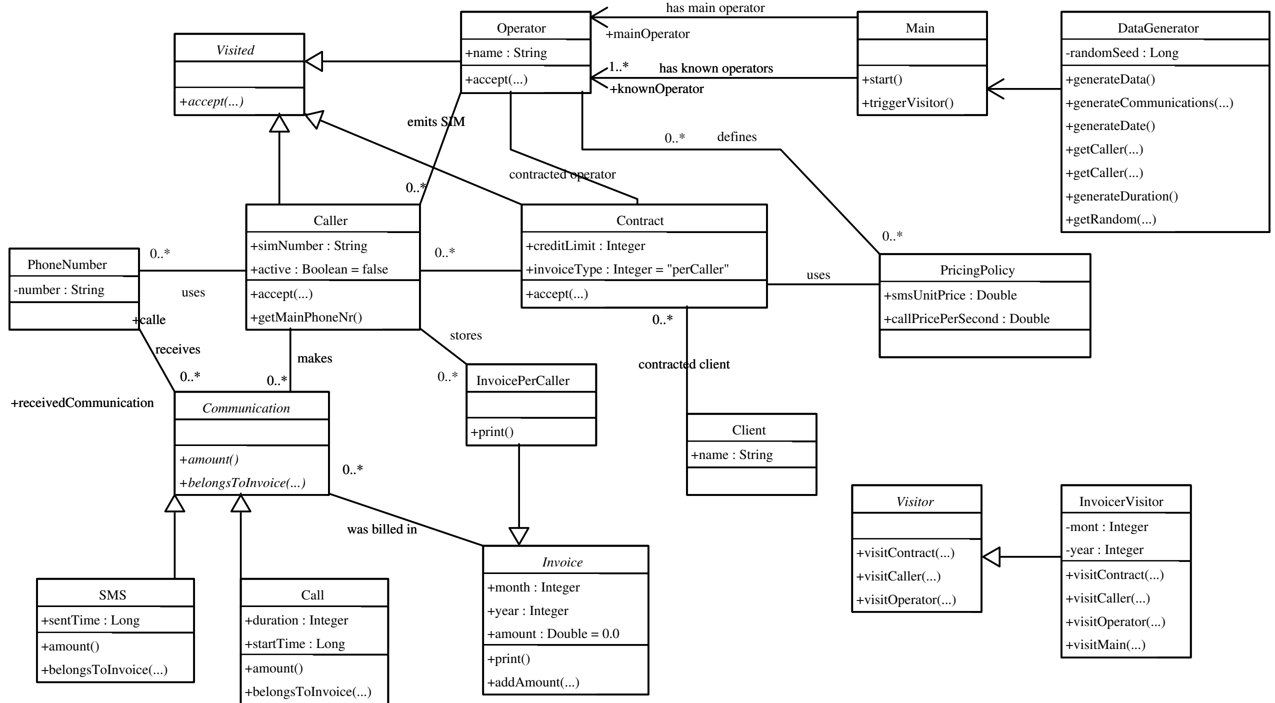

Figure 7.3: The final class diagram GSMCase.

and visitor classes, we have to insert working code into them in order to calculate and print the invoices. In addition to implementing the "visit" methods, we also implement the methods which calculate the prices for each communication and the methods that print them, and we also add some utility methods.

Unfortunately, one of the features that are not yet supported by LURE are static methods. To have a working application in Java, a static method called `main` is required. For that, we implement a simple method which creates an instance of the class `Main` and calls the `start` method. The main method is not extracted to the class diagram.

After implementing the code of the visitor, our application is finished and working. When started, it creates the simulation data (contracts, clients, calls etc), calculates the invoices for each of the clients, then prints these invoices and exits. From the working code, we extracted a class diagram to analyze the final design of GSMCase. This extraction took 78 seconds. The graphic representation of this diagram can be seen in figure 7.3.

Conclusion

The complete source for this application, which can be found in appendix A, has more than 600 lines of code. We started from a high-level design, with abstractions for all the classes we thought we would need. From this design, we generated a large amount of source code, as we mentioned in table 7.1 (a). At a certain point, we realized that the design of the application should be changed, that a visitor design pattern would be a more appropriate approach for calculating and printing the invoices. Thanks to the round-trip engineering tool, we did not need to perform this kind of design change at code level, editing superclass references, modifying constructors, initializations etc. Using `Lure`, we could extract the class diagram from the system, modify it accordingly to our needs, and then re-generate the code.

We can also notice the importance of having consistence between code generation and reverse-engineering. For example, it is no use to generate the large amount of helper methods that we have generated in the beginning (accessors, mutators, initializers etc) if, when doing the reverse-engineering of the code, we do not recognize them as helper methods. This would create one operation for each of these methods in the diagram, making it more polluted. The same for association ends: if we did not recognized the instance variables that we generated for association ends with multiplicity `n` as representing association ends, we would extract many attributes of type, for example, `java.util.Vector`, that would only make the diagram more confusing.

With respect to the use of Lure, even being only a research prototype, it has shown to be quite handy in this case study. We carried out the development of this application with 600 lines of code in one day. Its performance is also acceptable, since the transformations usually took not more than one minute. Of course, it cannot be compared to an industry-level tool, like Together Control Center, in the aspects of ease of use and performance, and neither have we tested its scalability to bigger applications. But, assuming the goal we proposed to ourselves when constructing it — managing round-trip engineering semi-automatically using logic metaprogramming —, its overall results are quite good.

Chapter 8

Conclusion and future work

8.1 Conclusion

Round-trip engineering is currently a popular technique for developing software systems. In fact, it results from an old need in the software engineering community: the ability to maintain design views of a system while it evolves. Round-trip engineering is the maintenance of design views specifically while the source code is being developed.

Since UML is the most popular modelling language these days, it is very common to use it as the way to express the design views while doing round-trip engineering. As we discussed in chapter 2, UML round-trip engineering is being considered as a mandatory feature in industry-level integrated development environments (IDEs). However, there is no standardized way neither of representing the source code as UML diagrams, nor to generate source code from UML diagrams. This is a significant problem, given that different tools use different approaches, enforcing the developer to change its practices when he changes from one tool to other. Moreover, as we saw also in chapter 2, existing round-trip approaches provide very limited support for customizing the way in which code generation and reverse-engineering are performed. The developer cannot adapt the tool to his needs and is forced to adapt himself to the tool.

In this thesis, we presented logic metaprogramming (LMP) as a good alternative to do round-trip engineering of software systems. LMP, as described by us in chapter 3, is a technique which uses a logic programming language at meta-level to reason about object-oriented source code. Our claim that LMP is suitable for doing round-trip engineering is based on two of its characteristics:

Reasoning power LMP has proven to be an efficient paradigm to reason about object-oriented source code, and extract design views from it. Since UML diagrams are a particular kind of design views, LMP can be used for creating them by extracting their patterns from

source code. As we extensively discussed in section 6.2.3, the characteristics of LMP, such as unification, pattern-matching and backtracking, provide a very powerful tool to infer UML constructs from the code. An example of that is the predicate in which we deduced the changeability of associations, by traversing the parse trees of all methods in a class, searching for assignments and method sends.

This reasoning power can be used not only to increase the level of abstraction but also to generate detailed source code from design descriptions. Section 6.2.2 described how we applied LMP to the generation of Java source code from UML class diagrams.

Clearness If we want to provide a customizable way to map from UML diagrams to source code and vice-versa, an interesting first step is to define an explicit way to declare this mapping. This mapping cannot be tangled in the code of the code generator or the reverse-engineering engine. The declarative nature of LMP provides a clear way to define these strategies. They are declared by means of logic rules which are totally separated from the transformation engines.

Joining these two characteristics, we have a technique which, one, has enough power to extract diagrams from code and generate code from diagrams, and, two, is naturally suited for clearly defining these kind of mappings. This seems to be a correct path towards the conception of powerful and customizable round-trip engineering tools.

Our main experiments were focused on class diagrams and Java, and resulted in a prototype tool called Lure. This tool is used for doing UML round-trip engineering of Java code. It can also be used for separately as a Java code generator or a UML reverse-engineering tool.

In our experiments with another language, namely Smalltalk, we have seen that using LMP — in particular, SOUL and LiCoR — can also cope with some aspects of language independence. In section 6.3, we described how to implement language-independent predicates for transparent reasoning about (code generation for) Java and Smalltalk.

8.2 Future work

In this section, we mention some possible and interesting topics for future research related to this thesis.

Only one mapping strategy As we saw in section 6.2, we have one conceptual mapping strategy for mapping UML constructs for code constructs. When we implemented this strategy, we split it in two parts. One which has logic predicates for representing the mapping strategies from class diagram to source code, and other for representing the mapping strategies from source code to class diagram.

We do not see this as an ideal solution, since the responsibility of maintaining both implementation strategies consistent is of the developer of the strategies. The verification tool which we presented in section 6.2.4 serves as an auxiliary tool for verifying examples of generation and extraction, but it is not a formal approach for checking whether the strategy used for code generation is consistent with the one being used for diagram extraction.

Therefore, an important extension of our approach would be the ability to unify the mapping strategies of code generation and reverse-engineering into only one mapping. In this way, we would be able to ensure that code is being generated and reverse-engineered using the same mapping. However, this does not seem to be a trivial task, if possible at all.

Round-trip engineering with change consistency check In this thesis, we have studied round-trip engineering focused on the mapping strategies for code generation and reverse-engineering. What we did was to directly propagate the changes from diagram to code and vice versa. We did not verify if these changes were consistent, if they would damage the structure of the existing constructs in the resulting part. For example, if the developer removes an operation in the design view, when this change is propagated to the code, it may cause errors, since there may be methods whose code is calling that operation. Same thing when a change is made in the code which can influence in a high-level design decision (e.g., breaking a design pattern at code level).

Therefore, consistency checks before change propagation can be regarded as an interesting subject when doing round-trip engineering. One of the possibilities for implementing this could be to insert an additional step before every change propagation (code generation or reverse-engineering) to verify if the changes made in one view are valid in the other view.

Filtering design information One of the subjects which we would like to have studied, but did not because of time constraints was the variable abstraction level of design diagrams. Round-trip engineering (and round-trip engineering) tools, in general, extract all the constructs they find in the code, and this often results in a diagram which is not very adequate for higher-level design analysis by the user. Diagrams with too many constructs and too much information look graphically polluted and do not provide the developer with the desired global coarse-grained view of the system.

An interesting area of study is how to define a general mechanism for filtering the information which is shown to the user in the diagram. This should derive from pre-defined levels of abstraction and from developer-defined rules. These could be application-independent (e.g., "hide all the navigabilities of associations") or specific to the needs of the application (e.g., "hide all the classes with the prefix 'Impl'").

Generalization for other UML diagrams This is a natural direction for continuation of our work, whose focus was exclusively on class diagrams. The Unified Modelling Language defines a set of nine diagrams, and some of them would surely help to illustrate the design view when doing round-trip engineering.

Supporting simultaneous changes Our approach is based on sequential updates. That is, when the developer wants to change the code, he generates code from the diagram and when he wants to change the diagram, he extracts this diagram from the code. We do not support the case when the developer changes the design and the code simultaneously. In our case study, we showed that round-trip engineering can be normally performed using this approach. However, there are other scenarios when simultaneous changes might be interesting. A typical case is team development. When many people are developing the same software, an environment which does not allow simultaneous changes can cause a decrease in productivity, since the developers have to do the checkout/modify/commit cycle one at a time.

For these kind of scenarios, it would be interesting to study how the support for simultaneous changes could be added to our approach. This would probably have to do with the consistency check that we have just mentioned, which would have to be performed before generating the code or extracting the diagram in order to verify if the changes made in both sides do not have conflicts.

Language independence In our experiments, we have identified some characteristics of LMP that can help on the conception of a language-independent tool for round-trip engineering. However, since we had chosen to focus on one language, we did not studied this issue in deep. More formal definitions of language independence issues could be subject to future research. This would probably be accompanied with a re-organization of LiCoR in terms of a language-independent meta-model.

Bibliography

[1] ArgoUML. http://argouml.tigris.org/.

[2] Gentleware AG. Poseidon for UML. http://www.gentleware.com/.

[3] Uwe Aßmann. Automatic Roundtrip Engineering. In Uwe Assmann, Elke Pulvermueller, Isabelle Borne, Noury Bouraqadi, and Pierre Cointe, editors, *Electronic Notes in Theoretical Computer Science*, volume 82. Elsevier, 2003.

[4] Krzysztof Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.

[5] Kalle Burbeck and Henrik Larsson. Automatic Model View Controller Engineering. Master's thesis, University of Linköping, 2002.

[6] Cincom. Smalltalk VisualWorks. http://www.cincom.com/smalltalk.

[7] Borland Software Corporation. Together Control Center. http://www.borland.com/together/controlcenter.

[8] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[9] 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)*, Kaiserslautern, Germany, 1999. Springer-Verlag.

[10] Theo D'Hondt, Kris De Volder, Kim Mens, and Roel Wuyts. Co-evolution of object-oriented software design and implementation. In *Proceedings of the international symposium on Software Architectures and Component Technology*, 2000.

[11] Stéphane Ducasse. Reengineering object-oriented applications. Technical report, 2001.

[12] Johan Fabry. Supporting development of enterprise javabeans through declarative meta programming. In *Object-Oriented Information Systems*. Springer, 2002.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] Leif Geiger and Albert Zündorf. Graph based debugging with fujaba. In Tom Mens, Andy Schürr, and Gabriele Taentzer, editors, *Electronic Notes in Theoretical Computer Science*, volume 72. Elsevier, 2002.

[15] Martin Gogolla and Ralf Kollmann. Re-documentation of java with uml class diagrams. In Eliot Chikofsky, editor, *Proc. 7th Reengineering Forum, Reengineering Week 2000*, 2000.

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

[17] Object Managment Group. Object Constraint Language Specification. In *Unified Modelling Language Specification Version 1.5*, chapter 6. March 2003. http://www.uml.org.

[18] Anders Henriksson and Henrik Larsson. A definition of round-trip engineering. Technical report, University of Linköping, 2003.

[19] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[20] Ralf Kollman, Petri Selonen, Eleni Stroulia, Tarja Systä, and Albert Zündorf. A study on the current state of the art in tool-supported uml-based static reverse engineering. In Elizabeth Burd and Arie van Deursen, editors, *Proceedings 9th Working Conference on Reverse Engineering*. IEEE, 2002.

[21] Ralf Kollmann and Martin Gogolla. Application of uml associations and their adornments in design recovery. In Peter Aiken and Elizabeth Burd, editors, *Proc. 8th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2001.

[22] John W. Lloyd. *Declarative Programming in Escher*. University of Bristol, June 1995.

[23] Kim Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, October 2000.

[24] Kim Mens, Johan Fabry, and João Del Valle. Multi-language code generation with logic metaprogramming — an experience report. Submitted to The 10th Working Conference on Reverse Engineering, but not published.

[25] 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.

[26] Sun Microsystems. Code Conventions for the Java Programming Language. http://java.sun.com/docs/codeconv/.

[27] U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The fujaba environment. In *Proc. of the 22nd International Conference on Software Engineering (ICSE), Limerick, Irland*, pages 742–745. ACM Press, 2000.

[28] U.A. Nickel, J. Niere, J.P. Wadsack, and A. Zündorf. Roundtrip Engineering with FUJABA. In J. Ebert, B. Kullbach, and F. Lehner, editors, *Proc of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany*. Fachberichte Informatik, Universität Koblenz-Landau, August 2000.

[29] Ulrich Nickel, Jorg Niere, and Albert Zundorf. The FUJABA environment. In *International Conference on Software Engineering*, pages 742–745, 2000.

[30] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348, May 2002.

[31] Object Managment Group. *XML Metadata Interchange (XMI) Specification 1.2*, January 2002. http://www.omg.org/technology/documents/formal/xmi.htm.

[32] Object Managment Group. *Unified Modelling Language Specification Version 1.5*, March 2003. http://www.uml.org.

[33] 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, 1999.

[34] IBM Rational Software. Rational Rose Family. http://www.rational.com/products/rose/.

[35] IBM Rational Software. Rational XDE. http://www.rational.com/products/xde/.

[36] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, 1994.

[37] Sun Microsystems. *The JavaBeans 1.01 Specification*, August 1997.

[38] Tarja Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, University of Tampere, 2000.

[39] Tom Tourwé, Tom Mens, and Francisca Muñoz. Detecting bad smells and refactoring opportunities with logic meta programming. Submitted to CSMR'03 and Journal of Software Maintenance and Evolution, 2003.

[40] Roel Wuyts. Declarative reasoning about the structure object-oriented systems. In *Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.

[41] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, 2001.

Appendix A

Final code for GSMCase

This appendix has the code that resulted from our case study, GSMCase. The major part of this code was automatically generated.

```
public class Call extends Communication {
  private java.lang.Integer duration;
  private java.util.Date startTime;
  public java.lang.Integer getDuration() {
    return duration;
  }
  public java.util.Date getStartTime() {
    return startTime;
  }
  public java.lang.Double amount() {
    return new Double(this.getCaller().getContract().getPricingPolicy()
      .getCallPricePerSecond().doubleValue() * duration.doubleValue());
  }
  public java.lang.Boolean belongsToInvoice(InvoicePerCaller inv) {
    return new java.lang.Boolean(true);
  }
  private void initialize(java.lang.Integer duration, java.util.Date startTime) {
    this.duration = duration;
    this.startTime = startTime;
  }
  public Call(java.lang.Integer duration, java.util.Date startTime, Caller caller,
    PhoneNr callee) {
    super(caller, callee);
    this.initialize(duration, startTime);
  }
}
```

```
public class Caller extends Visited {
  private Operator operator;
  private Contract contract;
  private java.util.Vector phoneNrs;
  private java.util.Vector invoices;
  private java.util.Vector communications;
  private java.lang.String simNumber;
  private java.lang.Boolean active;
```

```
public Operator getOperator() {
  return operator;
}
public Contract getContract() {
  return contract;
}
public java.util.Vector getPhoneNrs() {
  return phoneNrs;
}
public java.util.Vector getInvoices() {
  return invoices;
}
public java.util.Vector getCommunications() {
  return communications;
}
public java.lang.String getSimNumber() {
  return simNumber;
}
public java.lang.Boolean getActive() {
  return active;
}
public PhoneNr getMainPhoneNr() {
  return (PhoneNr) phoneNrs.firstElement();
}
private void initialize(Operator operator, Contract contract,
  java.lang.String simNumber) {
  this.operator = operator;
  this.contract = contract;
  this.simNumber = simNumber;
  this.active = new Boolean(true);
  this.phoneNrs = new java.util.Vector();
  this.invoices = new java.util.Vector();
  this.communications = new java.util.Vector();
}
public void addPhoneNr(PhoneNr phoneNr) {
  (this.phoneNrs).add(phoneNr);
}
public void addInvoice(InvoicePerCaller invoice) {
  (this.invoices).add(invoice);
}
public void addCommunication(Communication communication) {
  (this.communications).add(communication);
}
public void removePhoneNr(PhoneNr phoneNr) {
  (this.phoneNrs).remove(phoneNr);
}
public void removeInvoice(InvoicePerCaller invoice) {
  (this.invoices).remove(invoice);
}
public void removeCommunication(Communication communication) {
  (this.communications).remove(communication);
}
public  Caller(Operator operator, Contract contract, java.lang.String simNumber) {
  super();
  this.initialize(operator, contract, simNumber);
}
public void setOperator(Operator operator) {
  this.operator = operator;
}
public void setContract(Contract contract) {
  this.contract = contract;
}
public void setPhoneNrs(java.util.Vector phoneNrs) {
  this.phoneNrs = phoneNrs;
```

```
  }
  public void setInvoices(java.util.Vector invoices) {
    this.invoices = invoices;
  }
  public void setCommunications(java.util.Vector communications) {
    this.communications = communications;
  }
  public void setActive(java.lang.Boolean active) {
    this.active = active;
  }
  public void accept(Visitor visitor) {
    visitor.visitCaller(this);
  }
}


public class Client extends Object {
  private java.util.Vector operatorContracts;
  private java.lang.String name;
  public java.util.Vector getOperatorContracts() {
    return operatorContracts;
  }
  public java.lang.String getName() {
    return name;
  }
  private void initialize() {
    this.operatorContracts = new java.util.Vector();
  }
  public void addOperatorContract(Contract operatorContract) {
    (this.operatorContracts).add(operatorContract);
  }
  public void removeOperatorContract(Contract operatorContract) {
    (this.operatorContracts).remove(operatorContract);
  }
  public  Client() {
    super();
    this.initialize();
  }
  public void setOperatorContracts(java.util.Vector operatorContracts) {
    this.operatorContracts = operatorContracts;
  }
  public void setName(java.lang.String name) {
    this.name = name;
  }
}


public abstract class Communication extends Object {
  private Caller caller;
  private PhoneNr callee;
  private Invoice invoice;
  public Caller getCaller() {
    return caller;
  }
  public PhoneNr getCallee() {
    return callee;
  }
  public Invoice getInvoice() {
    return invoice;
  }
  public abstract java.lang.Double amount();
  public abstract java.lang.Boolean belongsToInvoice(InvoicePerCaller invoice);
  private void initialize(Caller caller, PhoneNr callee) {
    this.caller = caller;
```

```
    this.callee = callee;
  }
  public  Communication(Caller caller, PhoneNr callee) {
    super();
    this.initialize(caller, callee);
  }
  public void setInvoice(Invoice invoice) {
    this.invoice = invoice;
  }
}
```

```
public class Contract extends Visited {
  private Operator operator;
  private Client client;
  private PricingPolicy pricingPolicy;
  private java.util.Set callers;
  private java.lang.Integer creditLimit;
  private java.lang.String invoiceType;
  public Operator getOperator() {
    return operator;
  }
  public Client getClient() {
    return client;
  }
  public PricingPolicy getPricingPolicy() {
    return pricingPolicy;
  }
  public java.util.Set getCallers() {
    return callers;
  }
  public java.lang.Integer getCreditLimit() {
    return creditLimit;
  }
  public java.lang.String getInvoiceType() {
    return invoiceType;
  }
  private void initialize(Operator operator, Client client, PricingPolicy pricingPolicy,
    java.lang.Integer creditLimit) {
    this.operator = operator;
    this.client = client;
    this.pricingPolicy = pricingPolicy;
    this.creditLimit = creditLimit;
    this.invoiceType = "perCaller";
    this.callers = new java.util.HashSet();
  }
  public void addCaller(Caller caller) {
    (this.callers).add(caller);
  }
  public void removeCaller(Caller caller) {
    (this.callers).remove(caller);
  }
  public  Contract(Operator operator, Client client, PricingPolicy pricingPolicy,
    java.lang.Integer creditLimit) {
    super();
    this.initialize(operator, client, pricingPolicy, creditLimit);
  }
  public void setPricingPolicy(PricingPolicy pricingPolicy) {
    this.pricingPolicy = pricingPolicy;
  }
  public void setCallers(java.util.Set callers) {
    this.callers = callers;
  }
  public void setInvoiceType(java.lang.String invoiceType) {
```

```
    this.invoiceType = invoiceType;
  }
  public void accept(Visitor visitor) {
    visitor.visitContract(this);
  }
}


public class DataGenerator extends Object {
  private Main main;
  private java.lang.Long randomSeed;
  public void setMain(Main main) {
    this.main = main;
  }
  public Main getMain() {
    return main;
  }
  public void generateData() {
    Operator op1 = new Operator("BASE");
    Operator op2 = new Operator("TIM");
    PricingPolicy pp1 = new PricingPolicy(op1);
    PricingPolicy pp2 = new PricingPolicy(op2);
    pp1.setCallPricePerSecond(new Double(0.01d));
    pp1.setSmsUnitPrice(new Double(0.12d));
    pp2.setCallPricePerSecond(new Double(0.005d));
    pp2.setSmsUnitPrice(new Double(0.1d));
    Client client1 = new Client();
    Client client2 = new Client();
    client1.setName("VUB");
    client2.setName("PUCPR");
    Contract con1 = new Contract(op1, client1, pp1, new Integer(100));
    Contract con2 = new Contract(op2, client2, pp2, new Integer(50));
    op1.addClientContract(con1);
    op2.addClientContract(con2);
    Caller caller1 = new Caller(op1, con1, "Kim");
    Caller caller2 = new Caller(op1, con1, "Johan");
    Caller caller3 = new Caller(op2, con2, "Joao");
    con1.addCaller(caller1);
    con1.addCaller(caller2);
    con2.addCaller(caller3);
    PhoneNr pn1 = new PhoneNr(caller1, "91077351");
    PhoneNr pn2 = new PhoneNr(caller2, "91072935");
    PhoneNr pn3 = new PhoneNr(caller3, "91149517");
    caller1.addPhoneNr(pn1);
    caller2.addPhoneNr(pn2);
    caller3.addPhoneNr(pn3);
    this.generateCommunications(new Caller[] {caller1, caller2, caller3});
    main.addKnownOperator(op1);
    main.addKnownOperator(op2);
  }
  private void generateCommunications(Caller [] callers) {
    Communication [] coms = new Communication[50];
    for (int i = 0; i < coms.length / 2; i++) {
      Caller c = this.getCaller(callers);
      coms[i] = new Call(this.generateDuration(), this.generateDate(), c,
        this.getCaller(callers, c).getMainPhoneNr());
      c.addCommunication(coms[i]);
    }
    for (int i = coms.length / 2; i < coms.length; i++) {
      Caller c = this.getCaller(callers);
      coms[i] = new SMS(this.generateDate(), c,
        this.getCaller(callers, c).getMainPhoneNr());
      c.addCommunication(coms[i]);
    }
```

```
  }
  private java.util.Date generateDate() {
    return new java.util.Date();
  }
  private Caller getCaller(Caller [] callers) {
    return callers[this.getRandom(callers.length)];
  }
  private Caller getCaller(Caller [] callers, Caller c) {
    Caller ca;
    do ca = callers[this.getRandom(callers.length)]; while (ca == c);
    return ca;
  }
  private java.lang.Integer generateDuration() {
    return new java.lang.Integer(this.getRandom(10000));
  }
  private int getRandom(int n) {
    java.util.Random r = randomSeed == null ? new java.util.Random()
      : new java.util.Random(randomSeed.longValue());
    int i = r.nextInt(n);
    this.randomSeed = new Long(r.nextLong());
    return i;
  }
  private void initialize() {
  }
  public  DataGenerator() {
    super();
    this.initialize();
  }
}


public abstract class Invoice extends Object {
  private java.util.Vector communications;
  private java.lang.Integer month;
  private java.lang.Integer year;
  private java.lang.Double amount;
  public java.util.Vector getCommunications() {
    return communications;
  }
  public java.lang.Integer getMonth() {
    return month;
  }
  public java.lang.Integer getYear() {
    return year;
  }
  public java.lang.Double getAmount() {
    return amount;
  }
  public void addAmount(java.lang.Double anAmount) {
    if (this.amount == null) this.amount = new Double(0);;
    this.amount = new Double((this.amount).doubleValue() + anAmount.doubleValue());
  }
  public void print() {
    System.out.println("Total: R$ " + this.getAmount());
    for (java.util.Iterator iter = communications.iterator(); iter.hasNext(); ) {
      Communication element = (Communication) iter.next();
      System.out.print(element.getClass().getName() + " \t");
      System.out.print(element.getCaller().getMainPhoneNr().getPhoneNumber() + "\t");
      System.out.print(element.getCallee().getPhoneNumber() + "\t");
      System.out.println(element.amount());
    }
  }
  private void initialize(java.lang.Integer month, java.lang.Integer year) {
    this.month = month;
```

```
    this.year = year;
    this.amount = new Double(0.0d);
    this.communications = new java.util.Vector();
  }
  public void addCommunication(Communication communication) {
    (this.communications).add(communication);
  }
  public void removeCommunication(Communication communication) {
    (this.communications).remove(communication);
  }
  public  Invoice(java.lang.Integer month, java.lang.Integer year) {
    super();
    this.initialize(month, year);
  }
  public void setCommunications(java.util.Vector communications) {
    this.communications = communications;
  }
  public void setAmount(java.lang.Double amount) {
    this.amount = amount;
  }
}


public class InvoicePerCaller extends Invoice {
  private Caller caller;
  public Caller getCaller() {
    return caller;
  }
  public void print() {
    System.out.println("Caller " + caller.getSimNumber());
    super.print();
  }
  private void initialize(Caller caller) {
    this.caller = caller;
  }
  public  InvoicePerCaller(Caller caller, java.lang.Integer month,
    java.lang.Integer year) {
    super(month, year);
    this.initialize(caller);
  }
  public void setCaller(Caller caller) {
    this.caller = caller;
  }
}


public class InvoicerVisitor extends Visitor {
  private java.lang.Integer month;
  private java.lang.Integer year;
  private void initialize() {
  }
  public  InvoicerVisitor() {
    super();
    this.initialize();
  }
  public void visitMain(Main main) {
    System.out.println("Printing invoices for " + month + "/" + year);
    for (java.util.Iterator iter = main.getKnownOperators().iterator(); iter.hasNext(); ) {
      ((Operator) iter.next()).accept(this);
    }
  }
  public void visitCaller(Caller caller) {
    InvoicePerCaller inv = new InvoicePerCaller(caller, month, year);
    for (java.util.Iterator iter = caller.getCommunications().iterator(); iter.hasNext(); ) {
```

```
      Communication com = (Communication) iter.next();
      inv.addCommunication(com);
      inv.addAmount(com.amount());
    }
    inv.print();
  }
  public void visitContract(Contract contract) {
    System.out.println("--------------------------------");
    System.out.println("Contract of client " + contract.getClient().getName());
    for (java.util.Iterator iter = contract.getCallers().iterator(); iter.hasNext(); ) {
      ((Caller) iter.next()).accept(this);
    }
  }
  public void visitOperator(Operator operator) {
    System.out.println("================================");
    System.out.println("Operator " + operator.getName());
    for (java.util.Iterator iter = operator.getClientContracts().iterator(); iter.hasNext(); ) {
      ((Contract) iter.next()).accept(this);
    }
  }
  public java.lang.Integer getMonth() {
    return month;
  }
  public java.lang.Integer getYear() {
    return year;
  }
  public void setMonth(java.lang.Integer integer) {
    this.month = integer;
  }
  public void setYear(java.lang.Integer integer) {
    this.year = integer;
  }
}


public class Main extends Object {
  private java.util.Vector knownOperators;
  private Operator mainOperator;
  public java.util.Vector getKnownOperators() {
    return knownOperators;
  }
  public Operator getMainOperator() {
    return mainOperator;
  }
  public static void main(String [] args) {
    (new Main(new java.util.Vector(), null)).start();
  }
  public void start() {
    DataGenerator gen = new DataGenerator();
    gen.setMain(this);
    gen.generateData();
    this.triggerVisitor();
  }
  private void triggerVisitor() {
    InvoicerVisitor visitor = new InvoicerVisitor();
    visitor.setMonth(new Integer(8));
    visitor.setYear(new Integer(2003));
    visitor.visitMain(this);
  }
  private void initialize(java.util.Vector knownOperators, Operator mainOperator) {
    this.knownOperators = knownOperators;
    this.mainOperator = mainOperator;
  }
  public void addKnownOperator(Operator knownOperator) {
```

```
    (this.knownOperators).add(knownOperator);
  }
  public void removeKnownOperator(Operator knownOperator) {
    (this.knownOperators).remove(knownOperator);
  }
  public  Main(java.util.Vector knownOperators, Operator mainOperator) {
    super();
    this.initialize(knownOperators, mainOperator);
  }
  public void setKnownOperators(java.util.Vector knownOperators) {
    this.knownOperators = knownOperators;
  }
  public void setMainOperator(Operator mainOperator) {
    this.mainOperator = mainOperator;
  }
}
```

```
public class Operator extends Visited {
  private java.util.Vector clientContracts;
  private java.util.Vector pricingPolicys;
  private java.util.Vector callers;
  private java.lang.String name;
  public java.util.Vector getClientContracts() {
    return clientContracts;
  }
  public java.util.Vector getPricingPolicys() {
    return pricingPolicys;
  }
  public java.util.Vector getCallers() {
    return callers;
  }
  public java.lang.String getName() {
    return name;
  }
  private void initialize(java.lang.String name) {
    this.name = name;
    this.clientContracts = new java.util.Vector();
    this.pricingPolicys = new java.util.Vector();
    this.callers = new java.util.Vector();
  }
  public void addClientContract(Contract clientContract) {
    (this.clientContracts).add(clientContract);
  }
  public void addPricingPolicy(PricingPolicy pricingPolicy) {
    (this.pricingPolicys).add(pricingPolicy);
  }
  public void addCaller(Caller caller) {
    (this.callers).add(caller);
  }
  public void removeClientContract(Contract clientContract) {
    (this.clientContracts).remove(clientContract);
  }
  public void removePricingPolicy(PricingPolicy pricingPolicy) {
    (this.pricingPolicys).remove(pricingPolicy);
  }
  public void removeCaller(Caller caller) {
    (this.callers).remove(caller);
  }
  public  Operator(java.lang.String name) {
    super();
    this.initialize(name);
  }
  public void setClientContracts(java.util.Vector clientContracts) {
```

```
    this.clientContracts = clientContracts;
  }
  public void setPricingPolicys(java.util.Vector pricingPolicys) {
    this.pricingPolicys = pricingPolicys;
  }
  public void setCallers(java.util.Vector callers) {
    this.callers = callers;
  }
  public void accept(Visitor visitor) {
    visitor.visitOperator(this);
  }
}


public class PhoneNr extends Object {
  private java.util.Vector receivedCommunications;
  private Caller caller;
  private java.lang.String phoneNumber;
  public java.util.Vector getReceivedCommunications() {
    return receivedCommunications;
  }
  public Caller getCaller() {
    return caller;
  }
  public java.lang.String getPhoneNumber() {
    return phoneNumber;
  }
  private void initialize(Caller caller, java.lang.String phoneNumber) {
    this.caller = caller;
    this.phoneNumber = phoneNumber;
    this.receivedCommunications = new java.util.Vector();
  }
  public void addReceivedCommunication(Communication receivedCommunication) {
    (this.receivedCommunications).add(receivedCommunication);
  }
  public  PhoneNr(Caller caller, java.lang.String phoneNumber) {
    super();
    this.initialize(caller, phoneNumber);
  }
  public void setCaller(Caller caller) {
    this.caller = caller;
  }
}


public class PricingPolicy extends Object {
  private Operator operator;
  private java.lang.Double smsUnitPrice;
  private java.lang.Double callPricePerSecond;
  public Operator getOperator() {
    return operator;
  }
  public java.lang.Double getSmsUnitPrice() {
    return smsUnitPrice;
  }
  public java.lang.Double getCallPricePerSecond() {
    return callPricePerSecond;
  }
  private void initialize(Operator operator) {
    this.operator = operator;
  }
  public  PricingPolicy(Operator operator) {
    super();
    this.initialize(operator);
```

```
  }
  public void setOperator(Operator operator) {
    this.operator = operator;
  }
  public void setSmsUnitPrice(java.lang.Double smsUnitPrice) {
    this.smsUnitPrice = smsUnitPrice;
  }
  public void setCallPricePerSecond(java.lang.Double callPricePerSecond) {
    this.callPricePerSecond = callPricePerSecond;
  }
}


public class SMS extends Communication {
  private java.util.Date sentTime;
  public java.util.Date getSentTime() {
    return sentTime;
  }
  public java.lang.Double amount() {
    return this.getCaller().getContract().getPricingPolicy().getSmsUnitPrice();
  }
  public java.lang.Boolean belongsToInvoice(InvoicePerCaller inv) {
    return new java.lang.Boolean(true);
  }
  private void initialize(java.util.Date sentTime) {
    this.sentTime = sentTime;
  }
  public  SMS(java.util.Date sentTime, Caller caller, PhoneNr callee) {
    super(caller, callee);
    this.initialize(sentTime);
  }
}


public abstract class Visited extends Object {
  public abstract void accept(Visitor visitor);
  private void initialize() {
  }
  public  Visited() {
    super();
    this.initialize();
  }
}


public abstract class Visitor extends Object {
  public void visitOperator(Operator operator) {
  }
  public void visitContract(Contract contract) {
  }
  public void visitCaller(Caller caller) {
  }
  private void initialize() {
  }
  public  Visitor() {
    super();
    this.initialize();
  }
}
```