UNIVERSITY OF OTTAWA

Reverse Engineering Object-Oriented Systems into Umple: An Incremental and Rule-Based Approach

by

Miguel A. Garzón Torres

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science
Under the auspices of the Ottawa-Carleton Institute for Computer Science

in the
Faculty of Graduate and Postdoctoral Studies
Computer Science

March 2015

"Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better."

Edsger W. Dijkstra

UNIVERSITY OF OTTAWA

Abstract

Faculty of Graduate and Postdoctoral Studies
Computer Science

Doctor of Philosophy

by Miguel A. Garzón Torres

This thesis investigates a novel approach to reverse engineering, in which modeling information such as UML associations, state machines and attributes is incrementally added to code written in Java or C++, while maintaining the system in a textual format. Umple is a textual representation that blends modeling in UML with programming language code. The approach, called umplification, produces a program with behavior identical to the original one, but written in Umple and enhanced with model-level abstractions. As the resulting program is Umple code, our approach eliminates the distinction between code and model. In this paper we discuss the principles of Umple, the umplification approach and a rule-driven tool called the Umplificator, which implements and validates the depicted approach. The present thesis consists of three main parts. The first part (Chapter 1 and 2) present the research questions and research methodology and introduces Umple and the Umplification concept. The core of our research is presented in Chapters 3 and 5. The last part, Chapter 6, presents details of a case study conducted on an open source software application, JHotDraw. Finally, the expected contributions are listed in Chapter 7.

Acknowledgements

I would like to thank my dear supervisor, Professor Timothy Lethbridge, for guiding me ...

Contents

Abstract ii

Acknowledgements iii

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Research Questions . . . 2
- 1.2 Hypothesized Solutions . . . 3
- 1.3 Research Activities . . . 3
- 1.4 Thesis Contributions . . . 3
- 1.5 Outline . . . 3

2 Background 5
- 2.1 Umple Modeling Language . . . 5
  - 2.1.1 Umple Language Definition . . . 7
    - 2.1.1.1 Grammar . . . 7
      - Non-Terminals . . . 7
      - Terminals . . . 8
      - Native code blocks . . . 8
    - 2.1.1.2 Metamodel . . . 10
  - 2.1.2 Umple Attributes . . . 11
    - 2.1.2.1 Basic Attributes . . . 11
    - 2.1.2.2 Immutable Attributes . . . 11
    - 2.1.2.3 Lazy and Immutable Attributes . . . 12
    - 2.1.2.4 Defaulted attributes . . . 12
    - 2.1.2.5 Unique attributes . . . 13
    - 2.1.2.6 Autounique attributes . . . 13
    - 2.1.2.7 Constant attributes . . . 13
    - 2.1.2.8 Array attributes . . . 14
  - 2.1.3 Umple Associations . . . 14
  - 2.1.4 Code Injections . . . 16
  - 2.1.5 Umple Architecture and Tools . . . 18
- 2.2 Transformations . . . 19

2.2.1 Forward Engineering . . . 21
2.2.2 Reverse Engineering . . . 22
2.2.3 Model Transformations . . . 23
2.2.3.1 Model-To-Text Approaches . . . 26
2.2.3.2 Model-To-Model Approaches . . . 26
2.2.3.3 Model Transformations Languages and Tools . . . 27
2.2.4 Refactorings . . . 29
2.2.5 Re-engineering . . . 31

3 Reverse Engineering of Object Oriented Systems into Umple 33
3.1 Umplification Process . . . 33
3.1.1 Description . . . 33
3.1.2 Properties . . . 34
3.1.3 Overview of Transformations cases . . . 35
3.1.3.1 More Details of the Initial Transformation . . . 37
3.1.3.2 Details of the Transformations to Create Attributes . . . 38
3.1.3.3 Transformations to Create Associations . . . 39
3.2 Motivations . . . 39
3.2.1 Model-code duality . . . 39
3.2.2 Improving Program Comprehension . . . 40
3.3 School System - Manual Umplification Example . . . 41
3.3.1 School system: Initial Transformation . . . 43
3.3.2 School system: Transformations to Create Attributes . . . 45
3.3.3 School system: Transformations to Create Associations . . . 48
3.4 ATM system - Manual Umplification Example . . . 50
3.4.1 ATM system: Initial Transformation . . . 54
3.4.2 ATM system: Transformations to Create Attributes . . . 54
3.4.3 ATM system: Transformations to Create Associations . . . 54
3.5 Summary . . . 54

4 Detection Mechanisms for UML/Umple Constructs 56
4.1 A Notation for Transformation Rules . . . 57
4.2 Transformation Rules for the Initial Transformation Step . . . 59
4.3 Member Variables Analysis . . . 62
4.3.1 Refactoring to Create Attributes . . . 62
4.3.2 Refactoring to Create Associations . . . 63
4.3.3 Refactoring to Create State Machines . . . 64

5 The Umplificator Technologies 65
5.1 The Umplification tool support goals . . . 65
5.2 Alternative Approaches Studied . . . 67
5.2.1 TXL . . . 67
5.2.1.1 Java to Umple Implementation . . . 69
5.2.1.2 Design process of the TXL Program . . . 69
5.2.1.3 Transformation 1: Transforming the Class Header . . . 71
5.2.1.4 Transformation 2: Transforming the package . . . 72
5.2.1.5 Transformation 3: Transforming the imports . . . 72

5.2.1.6 Final transformation: The main program . . . 73
5.2.2 ATL . . . 74
5.2.2.1 The basics of ATL . . . 75
5.2.2.2 ATL Tool Support —Eclipse M2M . . . 76
5.2.2.3 Transformations Examples with ATL . . . 76
5.3 Discussion . . . 79
5.4 The Umplificator . . . 79
5.4.1 Architecture . . . 80
5.4.2 Parser and Model Extractor . . . 83
5.4.3 Transformer . . . 87
5.4.3.1 Drool's Rule Engine . . . 88
5.4.3.2 The Rule Language . . . 90
5.4.4 Generator . . . 93
5.5 Automated Umplification Example . . . 94
5.5.1 Initial transformation . . . 94
5.5.2 Automated Umplification of Attributes . . . 99
5.6 Umplificator Tooling . . . 104

6 Evaluation **106**
6.1 Testing Phase . . . 107
6.1.1 Testing the Base Language Code Parsers . . . 107
6.1.2 Testing the Model Extractor . . . 108
6.1.3 Testing the Transformer . . . 110
6.1.4 Testing the Umple Code Generator . . . 112
6.2 Pre-Validation Phase . . . 113
6.3 Initial Phase of Validation . . . 117
6.4 Second Phase of Validation . . . 119
6.5 Results . . . 119
6.5.1 JHotDraw . . . 119
6.5.2 Weka . . . 119

7 Related Work **123**
7.1 Literature Review Methodology . . . 123
7.1.1 Research Questions . . . 123
7.1.2 Search process . . . 124
7.1.3 First Phase Queries . . . 124
7.1.4 Second Phase Queries . . . 124
7.1.5 Inclusion and exclusion criteria . . . 124

8 Conclusions and Contributions **125**

A Appendix **127**
A.1 Source Code of the ATM software system . . . 127

Bibliography **128**

List of Figures

2.1 Umple Metamodel (Partial) . . . 10
2.2 Umple Architecture . . . 18
2.3 Umple Online . . . 19
2.4 Transformations across the different software life-cycle phases . . . 20

3.1 The Umplification process generalized . . . 34
3.2 UML Class Diagram of the Mentor-Student example - Level 1 . . . 45
3.3 UML Class Diagram of the Mentor-Student example - Level 2 . . . 47
3.4 UML Class Diagram of the Mentor-Student example - Level 3 . . . 50

4.1 Input-Output relationships for rule TypeToUmpleClass . . . 60

5.1 TXL Program for transforming Java to Umple . . . 69
5.2 Structure of the JavaToUmple program . . . 70
5.3 The JavaToUmple ATL program . . . 75
5.4 A simplified version of the Java metamodel . . . 77
5.5 A simplified version of the Umple metamodel . . . 78
5.6 The umplification process flow . . . 81
5.7 The Umplificator components . . . 82
5.8 The Parser and Model Extractor components . . . 88
5.9 The Transformer component inputs and outputs . . . 88
5.10 High Level View of the Drools Rule Engine . . . 89
5.11 Forward vs Backward Chaining . . . 90
5.12 The Generator component inputs and outputs . . . 93
5.13 Pattern Matching and creation of an UmpleClass . . . 96
5.14 Rule Engine snapshot after initial transformation . . . 100
5.15 Pattern Matching and creation of an UmpleClass . . . 102
5.16 The Umplificator online - A PHP Web application . . . 105

6.1 Umplificator Testing Infrastructure . . . 107
6.2 The Pre-Validation Phase: Comparing UmpleModel and UmpleModel' . . . 114
6.3 UML Class diagram of the Access Control system . . . 115

List of Tables

2.1 API generated methods from Umple attributes - Accessor methods [1] . . 14
2.2 API generated methods from Umple attributes - Mutator methods [1] . . 15
2.3 API generated from Umple Associations - Accessor Methods [1] 16
2.4 API generated from Umple Associations - Mutator Methods [1] 16
2.5 Summary of Reverse engineering approaches 23
2.6 Summary of Model transformation technologies 30

3.1 Refactorings to methods required for each transformation 37
3.2 Analysis of Member variables of class Student 45

4.1 Analyzing instance variables for presence in the constructor and getter/setters 62
4.2 Umple Primitive Data Types 63
4.3 Accessor Methods parsed and analyzed 64
4.4 Mutator methods parsed and analyzed 64

5.1 Third Party Technologies employed in the Umplificator tool 82
5.2 Eclipse projects used in the Umplificator 83
5.3 Sample Uses of an AST for Code Analysis 85
5.4 Rule attributes 91
5.5 The input Java Model elements 95
5.6 Artifacts deployed during the building process of the Umplificator 104

6.1 Small examples used for first phase of validation 121
6.2 Open-source systems umplified 122

Chapter 1

Introduction

Many software systems experience growth and change for an extended period of time. Maintaining consistency between documentation and the corresponding code becomes challenging. This situation has long been recognized by researchers, and significant effort has been made to tackle it. Reverse engineering is one of the fruits of this effort and has been defined as the process of creating a representation of the system at a higher level of abstraction [2].

Reverse engineering, in general, recovers documentation from code of software systems. When such documentation follows a well-defined syntax it is often now referred to as a model. Such models are often represented using UML (Unified Modeling Language), which visually represents the static and dynamic characteristics of a system.

There is a long and rich literature in reverse engineering [3]. Most existing techniques result in the generation of documentation that can be consulted separately from the code. Other techniques generate models in the form of UML diagrams that are intended to be used for code generation of a new version of the system. The technique discussed in this paper goes one step further: It modifies the source code to add model constructs that are represented textually, but can also be viewed and edited as diagrams. The target language of our reverse engineering process is Umple [4], which adds UML and other constructs textually to Java, C++ and PHP.

We call our approach to reverse engineering a software system umplification. This is a play on words with the concept of 'amplification' and also the notion of converting into Umple. In our previous work [5], we have found that umplifying code is reasonably

straightforward for someone familiar with Umple, and with knowledge of UML modeling pragmatics. Moreover, we have per-formed manual umplification of several systems, including Umple itself.

The present thesis focuses on how the umplification process can be performed automatically by a reverse engineering technology. In Section 1.1, we state the research problem addressed by this proposal and list the research questions. In Section 1.2, we present the methodology that we will follow in order to answer our research questions. We conclude the present chapter by

1.1 Research Questions

The problem to be addressed in this research is as follows:

Developers currently often work with large volumes of legacy code. Tools exist to allow them to extract models or transform their code in a variety of ways. However doing so tends to result in a system that is quite different in syntax and structure. They are thus inhibited from using reverse engineering tools except to generate documentation. The Umple technology partly solves this problem by allowing incremental addition of modeling constructs into familiar programming language code. This allows developers to maintain the essential 'familiarity' with their code as they gradually transform it. Converting to Umple (Umplification) has been done manually indeed it was applied to the Umple compiler itself [5] but it ought to have tool support so it can be done in a more automatic, systematic and error-free manner on large systems.

1. What transformation technology, transformations and refactoring patterns will work best for umplification?
2. What percentage of code reduction and complexity reduction can we achieve by umplification, and how can we measure the complexity reduction?
3. Overall, what are the benefits of automated or semi-automated umplification as compared to manual umplification or the use of other reverse-engineering or transformation approaches?
4. What should be the architecture, implementation and user interface of an umplification tool?

1.2 Hypothesized Solutions

1.3 Research Activities

The major steps in the methodology are the following:

1. Manually perform umplification to gain an understanding of what will be needed
2. Iteratively develop The Umplificator tool, exploring the effectiveness of various reusable components and transformation approaches. This includes selection or creation of an easy-to-use tool to express transformations from the base language to Umple. We want to avoid complex XML-based solutions since usability will be key.
3. Start with a major case study (JHotDraw), iteratively umplifying it and improving the Umplificator until the Umple version of the case study compiles and a significant number of constructs have been umplified successfully
4. Iteratively develop more and more transformations to convert additional Java code into Umple. Introduce additional case studies until the Umplificator works well on 10-15 reasonably large open-source systems.
5. Compare the work to alternative approaches.

1.4 Thesis Contributions

1.5 Outline

This thesis proposal is organized as follows.

Chapter 2

Chapter 2 presents background research, a brief introduction to Umple and its modeling constructs. Covered in this chapter are existing technologies in reverse engineering into UML.

Chapter 3

Chapter 2 presents umplification in detail, the core of this thesis.

Chapter 4

Chapter 4 presents the mechanisms allowing us to detect Umple constructs.

Chapter 5

Chapter 5 presents three different technologies that were explored as part of our research activities. We evaluate ATL and TXL to see to which extent they could fulfill our needs. ATL and TXL are two famous model-to-model transformation technologies. We discuss all the design decisions and propose a set of tools and technologies that our reverse engineering tool uses.

Chapter 6

Chapter 6 presents the case study conducted to evaluate the feasibility and efficiency of our approach. The case study shows the results of the umplification performed to the JHotDraw framework, we measure lines of code to compare the original system and the umplified version of the system.

Chapter 7

Chapter 7 presents selected on-going research activities that bear similarity to our research. We focus on highlighting aspects of the existing research that influenced our direction, and position our research with respect to existing work.

Chapter 8

Chapter 7 summarizes our research activities and gives an outline of future research directions.

Chapter 2

Background

This chapter presents the background knowledge required for readers to fully understand the subsequent chapters. The *umplification* approach presented in this thesis is an incremental reverse engineering technique performing model transformations to transform base language program into Umple. In the following sections, we introduce the Umple language and we present the most important concepts about reverse engineering.

2.1 Umple Modeling Language

Umple [4] is an open-source textual modeling and programming language that adds UML abstractions to base programming languages including Java, PHP, C++ and Ruby.

Umple has been designed to be general purpose and has UML class diagrams and UML state diagrams as its central abstractions. It has state-of-the art code generation and can be used incrementally, meaning that it is easy for developers to gradually switch over to modeling from pure programming. Umple was designed for modeling and developing large systems and for teaching modeling [6]. Umple is written in itself – the original Java version was manually umplified many years ago. That experience was one of the motivations for the current work. In addition to classes, interfaces and generalizations available in object oriented languages, Umple allows software developers to specify:

1. *Associations*: As in UML, these specify the links between objects that will exist at run time. Umple supports enforcement of multiplicity constraints and manages referential integrity ensuring that bidirectional references are consistently maintained in both directions [7].

2. *Attributes*: These abstract the concept of instance variables. They can have properties such as immutability, and can be subject to constraints, tracing, and hooks that take actions before or after they are changed [8].

3. *State Machines*: These also follow UML semantics, and can be considered to be a special type of attributes, subject to events that cause transitions from one value to another. States can have entry or exit actions, nested and parallel substates, and activities that operate in concurrent threads [9].

4. *Traits*: A trait is a partial description of a class (containing elements such as methods, attributes and state machines) that can be reused in several different classes, with optional renaming of elements. They can be used to describe reusable patterns.

5. *Patterns*: Umple currently supports the singleton and immutable patterns, as well as keys that allow generation of consistent code for hashing and equality testing.

6. *Aspect Oriented Code Injection*: This allows injection of code that can be run before or after methods, including Umple-defined actions on attributes, associations and the elements of state machines. Such code can be used as preconditions and post-conditions or for various other purposes. Code can be injected into the API methods (those methods generated by Umple) as well as into user-defined methods.

7. *Tracing*: A sublanguage of Umple called MOTL (Model-oriented tracing language) allows developers to specify tracing at the model level, for example to enable understanding of the behavior of a complex set of state machines operating in multiple threads and class instances [10].

8. *Constraints*: Invariants, preconditions and postconditions can be specified.

9. *Concurrency*: Umple provides several mechanisms to allow concurrency to be specified easily, including active objects, queuing in state machines, ports, and the aforementioned state activities.

The Umple compiler supports code generation for Java, PHP, Ruby and C++, as well as export to XMI and other UML formats. The compiler generates various types of methods including mutator, accessor, and event methods from the various Umple features. A mutator (e.g. set(), add()) method is a method used to control changes to a variable and an accessor (e.g. get()) method is the one used to return values of the variable. An event method triggers state change. An extended summary of the API generated by Umple from attributes, associations, state machines and other features can be found at [1]. Umple can also generate diagrams, metrics, and various other self-documentation artifacts. Umple models can be created or edited using the *UmpleOnline* Web tool [11], the command line compiler or an Eclipse plugin.

The umplification method discussed in this thesis currently focuses on associations, attributes and state machines with some generation of code injections. The next sub-sections introduce these Umple constructs in greater detail.

2.1.1 Umple Language Definition

In this sub-section we present the grammar and metamodel defining the syntax and semantics of the Umple language.

2.1.1.1 Grammar

Umple's language description is written in a slightly non-standard EBNF syntax. In standard EBNF grammars all tokens have to be strictly defined. Umple, however, supports blocks of code written in the different programming languages that don't need to be parsed. This means that an Umple developer/user can choose to embed a wide variety of blocks of native code within Umple code. The following are the main elements of the Umple grammar notation:

Non-Terminals A *rule-based* non-terminal uses double square brackets and represents a references to a another rule. In the example in Listing 2.1, *classContent* is a rule-based non terminal referring to the ClassContent rule. This allows reuse of such rules. If the rule is declared with a minus sign following it, then the rule name is not

added to the resulting tokenization string (abstract syntax tree) for simplicity; an example of this is found in 2.2. The definition of rule *classContent* is shown in Listing 2.3.

LISTING 2.1: Grammar for Umple classes

```
1  classDefinition : class [name] { [[classContent]]* }
```

Terminals Terminals come in two types. Those shown in single square brackets match, by default, any alphanumeric string. In Listing 2.1, *name* is a terminal that matches an arbitrary alphanumeric string, and is expected to be followed by a curly bracket in this case. The second type of terminal is shown as actual text and guides the parsing. So for example in 2.1 'class' is a terminal that must match exactly, as as are the open and close curly brackets. Note that the regular parentheses are metacharacters used for grouping, and the asterisk means means zero-or-more matches.

Native code blocks As we already discussed, blocks of native code are skipped. The rule in Listing 2.2 defines the body of a method. The ***code* will match everything until the ending curly bracket is reached, while properly dealing with nested pairs of curly brackets found in the code. This allows the grammar to stay unchanged as new languages are added.

In the following grammar examples, rules are shown in blue, terminal symbols are in red, identifiers are in green and arbitrary input is in **black** and surrounded by [**].

LISTING 2.2: Grammar for Umple classes

```
1  methodBody- : ( [[codeLangs]] { ( [[precondition]] | [[postcondition]] )*
       [**code] } )+
```

The grammar to parse classes (declarations), attributes, , and associations is presented in Listings 2.3-2.5 respectively.

A class definition starts with a name, followed by a curly bracket and any of the items in Lines 3-14 of Listing 2.3 such as attributes, state machines or inline associations. Note that for conciseness reasons, some rules have been omitted from Listing 2.3. Additional details about the grammar metalanguage can be obtained in the online Umple user manual at http://cruise.eecs.uottawa.ca/umple/UmpleGrammar.html. The entire Umple grammar definition is located at http://grammar.umple.org.

LISTING 2.3: Umple Grammar for classes

```
classDefinition : class [name] { [[classContent]]* }

classContent- : [[comment]] | [[abstract]] | [[keyDefinition]] | [[
    softwarePattern]] | [[depend]] | [[symmetricReflexiveAssociation]] |
    [[attribute]] | [[stateMachine]] | [[inlineAssociation]] |[[
    concreteMethodDeclaration]]| [[constantDeclaration]] [[invariant]] |
    [[exception]] |[[extraCode]]
```

LISTING 2.4: Umple Grammar for attributes

```
attribute : [[simpleAttribute]] | [[autouniqueAttribute]] | [[
    derivedAttribute]] | [[complexAttribute]]

simpleAttribute- : [=gpIdentifier:%]? [~name] ;
autouniqueAttribute- : [=autounique] [~name] ;
derivedAttribute- : [=modifier:immutable  |settable  |internal |defaulted
     |const |fixml]? [[typedName]] = ([[moreCode]] )+
complexAttribute- : [=unique]? [=lazy]? [=modifier:immutable |settable |
    internal |defaulted  |const |fixml]?[[typedName]] (= [**value])? ;
```

LISTING 2.5: Umple Grammar for associations

```
association : [=modifier:immutable]? [[associationEnd]] [=arrow:-- |->
    |<- |>< |<@>- |-<@>] [[associationEnd]] ;
symmetricReflexiveAssociation : [[multiplicity]] self [roleName] ;
inlineAssociation : [=modifier:immutable]? [[inlineAssociationEnd]] [=
    arrow:-- |-> |<- |>< |<@>-  |-<@>] [[associationEnd]] ;
inlineAssociationEnd : [[multiplicity]] [~roleName]? [[isSorted]]?
singleAssociationEnd : [[multiplicity]] [type] [~roleName]? ;
associationEnd : [[multiplicity]] [=gpIdentifier:%]? [type] [~roleName]?
    [[isSorted]]?
multiplicity- : [!lowerBound:\d+|[**]] .. [!upperBound:\d+|[**]] | [!
    bound:\d+|[**]]
isSorted- : sorted { [priority] } : [=modifier:immutable]? [[
    inlineAssociationEnd]] [=arrow:--
  |->
  |<-
  |><
  |<@>-
  |-<@>] [[associationEnd]] ;
inlineAssociationEnd : [[multiplicity]] [~roleName]? [[isSorted]]?
multiplicity- : [!lowerBound:\d+|[**]] .. [!upperBound:\d+|[**]] | [!
    bound:\d+|[**]]
```

The complete set of grammar files, which are part of the source code of Umple [12], can be found in the following directory: `cruise.umple/src/*.*.grammar`

The intent of discussing the underlying grammar of Umple is to help provide context. To obtain a deeper appreciation for the capabilities of Umple one needs to understand the semantics, which we will outline in the following subsections.

2.1.1.2 Metamodel

Umple is represented internally using a metamodel that describes all the elements and their relationships. The Umple metamodel is developed in Umple itself; figure 2.1 gives a sample from of the core of it. This class diagram was generated using Umple's internal diagram-drawing mechanism. As shown in the metamodel, an UmpleClass can be associated with many attributes, association variables and code injections. For a complete view of the Umple metamodel refer to [13].

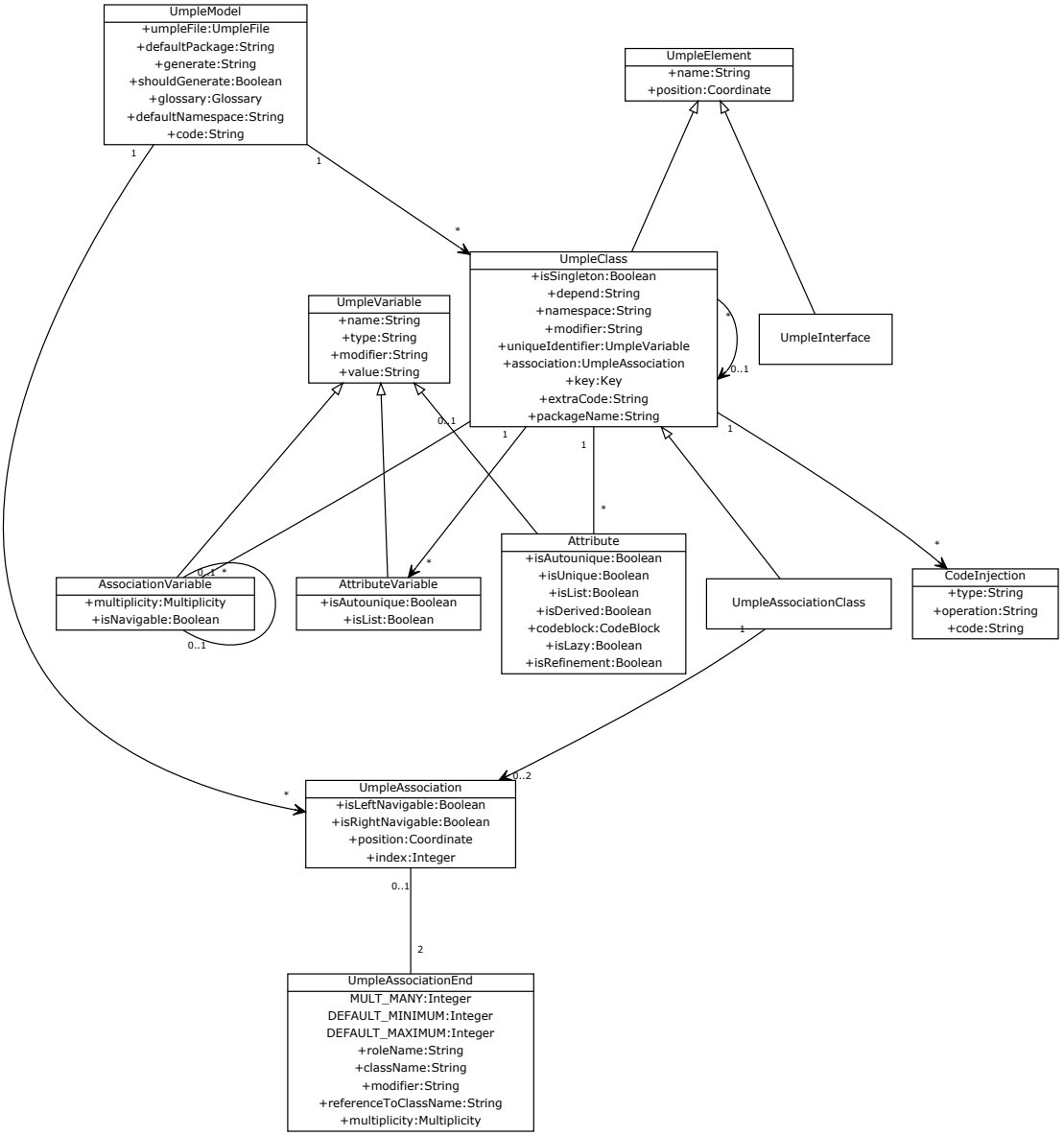

FIGURE 2.1: Umple Metamodel (Partial)

2.1.2 Umple Attributes

An Umple attribute is a property of an object. For instance, a Person object might have a *name* and an *address*. An attribute can have various properties described in the following subsections.

2.1.2.1 Basic Attributes

A basic attribute in Umple represents simple data and is composed of one of the Umple data types and the name of the attribute. As shown in Tables 2.1 and 2.2, the implications on code generation include a parameter in the constructor and a simple set and get methods to manage access to the attribute. The String datatype in Umple is the default type, when no type is specified. The example in in Listing 2.6 shows multiple attributes having different (Umple) datatypes.

LISTING 2.6: Basic Umple attribute

```
class Demo
{
  name; // String type
  Integer i;
  Float flt;
  String str;
  Double dbl;
  Boolean bln;
  Date dte;
  Time tme;
}
```

2.1.2.2 Immutable Attributes

The value of an immutable attribute can not change during the lifetime of an instance of the class. The resulting base language code (e.g. Java) for an immutable attribute would be the same as the basic attribute implementation except that there would be not setter method generated. A constructor argument is required so the value can be set at construction time but cannot be changed afterwards since no setter is generated. The syntax for an immutable attribute is shown in Listing 2.7. In this example, the *studentId* must be initialized during construction and cannot changed after it.

LISTING 2.7: Immutable Umple attribute

```
class Student
```

```
{
  immutable Integer studentId;
}
```

2.1.2.3 Lazy and Immutable Attributes

In cases where the attribute should be immutable, but the value is not available at the time of construction, the attribute can be declared as *lazy immutable*. The use of the lazy syntax means that the attribute is not initialized in the constructor (i.e. it is not part of the constructor's signature). The generated code will contain a flag to track whether the object has been set yet, allowing only a single set to occur. In Listing 2.8, attribute x is declare as a lazy immutable attribute. The setter method of this attribute can be call once, since it will return false if we try to set it again (setter returns a boolean). Lazy immutable attributes are useful in architectures where the developer doesn't possess any control over the creation of the objects and therefore he can't specify constructor arguments.

LISTING 2.8: Lazy immutable Umple attribute

```
class A
{
  lazy immutable x;
}
```

2.1.2.4 Defaulted attributes

A defaulted attribute is set in the constructor to the default value, and can be reset to the default any time by calling a reset method (in this example *resetName()*). It can be also set to any other value using its setter method. In Listing 2.9, the attribute 'name' is initializated to the default value 'UOttawa'. This default value can be queried by calling *getDefaultName()*.

LISTING 2.9: Defaulted Umple attribute

```
class School
{
  String name="UOttawa";
}
```

2.1.2.5 Unique attributes

The unique attribute guarantees its uniqueness within a particular class. For instance, in the example in Listing 2.10, in the set method of attribute 'name', prior to setting its value , we will check for uniqueness.

Listing 2.10: Unique Umple attribute

```
class Student
{
  unique String name;
}
```

2.1.2.6 Autounique attributes

The implementation of autounique attributes is very similar to the implementation of unique attributes presented in the previous sub-section. The main difference is that the autounique attribute is set in the constructor to the next available value. Autounique attributes must be of type Integer as shown in Listing 2.11.

Listing 2.11: Autonique umple attributes

```
class Student
{
  autounique Integer studentId;
}
```

2.1.2.7 Constant attributes

A constant (class level) attribute is identified using the *const* keyword as illustrated below. A constant is associated with the type itself, rather than an *instance* of the type (i.e. it would be generated as a static variable in Java). Listing 2.12 declares a constant of type Integer.

Listing 2.12: Constants in Umple

```
class Student
{
  const Integer MAX_COURSES = 10;
}
```

2.1.2.8 Array attributes

Umple supports attributes that might contain multiple values. The square brackets notation '[]' is used as illustrated in Listing 2.13.

LISTING 2.13: Array attributes

```
class Student
{
  String[] nickname;
}
```

In translating Umple attributes into object-oriented programming languages such as Java it is common to generate mutator and accessor methods. Tables 2.1 and 2.2 present the list of accessor and mutator methods generated from Umple attributes. In Tables 2.1 and 2.2, T is the type of the attribute (String if omitted) and z is the attribute name.

TABLE 2.1: API generated methods from Umple attributes - Accessor methods [1]

	T getZ()	**boolean isZ()**	**boolean equals(Object)**
	returns the value	returns the value	tests for reference equality
Basic	Yes	Yes; if T is boolean	No
Initialized	Yes	Yes; if T is boolean	No
Lazy	Yes	Yes; if T is boolean	No
Defaulted	Yes	Yes; if T is boolean	No
Immutable	Yes	Yes; if T is boolean	No
Lazy immutable	Yes	Yes; if T is boolean	No
Autounique	Yes; T always int.	No	No
Constant	No	No	No
Internal	No	No	No
Key	Yes	Yes	Yes

2.1.3 Umple Associations

In Umple, as in UML, an association defines a relationship from a class to another class. Furthermore, it specifies which links such as references or pointers may exist at run time between the different instances of the classes. More specifically, an Umple association is composed of the following information:

TABLE 2.2: API generated methods from Umple attributes - Mutator methods [1]

	boolean setZ(T)	**boolean resetZ()**
Description	mutates the attribute	restores original default
Basic	Yes	No
Initialized	Yes	No
Lazy	Yes	No
Defaulted	Yes	Yes;
Immutable	No	No
Lazy immutable	Yes; only once.	No
Autounique	No	No
Constant	No	No
Internal	No	No
Key	Yes	No

- *Association Ends*: These are the classes involved in the relationship.
- *Navigability:* The navigability determines whether or not the association can be accessed from the opposite end. The notation '–' is used when each class can access the linked objects of the other class and '->' or '<-' to indicate that the navigation is possible in only one direction.
- *Multiplicity:* These are the restrictions on the numbers of objects allowed in the relationship.
- *Role names:* These are used to clarify the relationship and avoid name collisions if two classes are associated in multiple ways. Role names are optional except in reflexive associations or in other situations when name collisions might exist. The Umple compiler will give an error message if a role name is needed.

The code segment in Listing 2.14 illustrates an association between instances of classes *School* and *Person*. In this example, an instance of class School can be associated to zero or more instances of class Student. The 'isA' notation is used to denote an inheritance relationship between the classes (Student is a subclass of Person). It should be noted that 'isA' is also used for other generalization relationships in Umple, such as implementation of Interfaces.

LISTING 2.14: An example of an inline Umple Association

```
class School {
 0..1 -- * Student student; //inline association
}
class Student {
```

```
  isA Person;
}
class Person { }
```

Alternatively, in addition to defining an association embedded in one of the associated classes, it is also possible to specify an association independently as shown in Listing 2.15.

Listing 2.15: An example of an independent Umple Association

```
class School {
}
class Student {
  isA Person;
}
class Person { }

association {
 0..1 School -- * Student student;
}
```

In Tables 2.3 and 2.4, we show the generated API methods for accessing and mutating the links in the association. X is the name of the current class, W is the name of the class at the other association end and r is a role name used when referring to W.

Table 2.3: API generated from Umple Associations - Accessor Methods [1]

Method Signature	Description
W getW()	Return the W;
W getW(index)	Picks a specific linked W;
List <W>getWs()	Gets immutable list of links;
boolean hasWs()	Returns true if cardinality is >0;
int indexOfW(W)	Return index of W in the list;
int numberOfWs()	Return the cardinality;;

Table 2.4: API generated from Umple Associations - Mutator Methods [1]

Method Signature	Description
boolean setW(W)	Adds a link to existing W
W addW(args)	Constructs a new W and adds link
boolean addW(W)	Adds a link to existing W
boolean setWs(W)	Adds a set of links
boolean removeW(W)	Removes link to W if possible

2.1.4 Code Injections

Code injections are used to insert certain code statements **before** or **after** various Umple-defined actions on attributes, associations and (components of) state machines.

Using **before** statements allows the developer to enforce preconditions and **after** statements to enforce postconditions. Code injections (after and before statements) can be added into the constructor, any user defined methods and into the API generated methods such as *getX, setX, addX, removeX, getXs, numberOfXs, indexOfX*, where X is the name of the attribute or association.

LISTING 2.16: A code injection into the constructor

```
1  class Operation {
2    const Boolean DEBUG=true;
3    query;
4    before constructor {
5      if (aQuery == null)
6      {
7        throw new RuntimeException("Please provide a valid query");
8      }
9    }
10   after constructor {
11     if (DEBUG) { System.out.println("Created " + query); }
12   }
13 }
```

The following gives details of the above:

- Line 2. Declares a constant (static final in Java).
- Line 3. Declares a simple (String) attribute.
- Line 4-9. Declares a code injection to be inserted at the beginning of the constructor.
- Line 10-12. Declares a code injection to be inserted at the end of the constructor.

The code in Listing 2.16 generates the following (Java) constructor:

LISTING 2.17: Generated constructor after code injection

```
1 public Operation(String aQuery)
2 {
3   if (aQuery == null)
4   {
5     throw new RuntimeException("Please provide a valid query");
6   }
7   query = aQuery;
8   if (DEBUG) { System.out.println("Created " + query); }
9 }
```

2.1.5 Umple Architecture and Tools

The Umple compiler, which was originally written in Java, was fully rewritten in Umple in 2008, and has been developed and maintained in Umple since then.

The compiler has a layered and pipelined architecture as shown in Figure 2.2. The components of Umple includes: a parser, an analyzer as well as several code generators and model-to-model transformation engines. These are described below:

1. **Parser**: The parser receives an input model, written in Umple language, tokenizes it and passes it to the next component in the pipeline.

2. **Analyzer**: This component processes the tokens previously obtained and converts them into an internal representation consistent with Umple's metamodel. Errors and warnings are produced at this stage. Warnings mean that generation can occur, but there may be issues with the generated result. Errors mean that no code generation will be possible.

3. **Code-Generator(s)**: The internal representation is then translated into other artifacts; either additional models like Papyrus XMI, EMF, Yuml, Xuml; various diagrams, or source code such as Java, C++, PHP, Ruby or SQL. The compiler generates various types of methods including mutators (to control changes to a variable) and accessors (to return the value of a variable) from the various Umple features. Sophisticated code for managing state machines, tracing [10], generation templates, patterns, aspects and concurrency can also be generated from models.

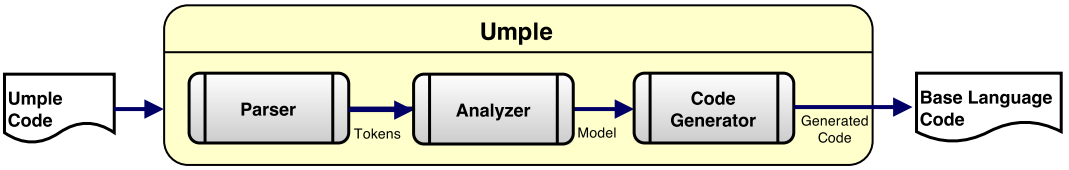

FIGURE 2.2: Umple Architecture

Each component is tested independently to ensure that the input is processed correctly and the output produced is valid. Testing the Umple parser is centered on tokenization of Umple code. Testing the metamodel classes ensures that the analyzer component produces valid metamodel instances. Testing of generated systems is also performed [14].

In addition to the Eclipse plugin and command-line based compiler, UmpleOnline [11], a web-based application shown in Figure 2.3, allows to instantly experiment with Umple on the Web.

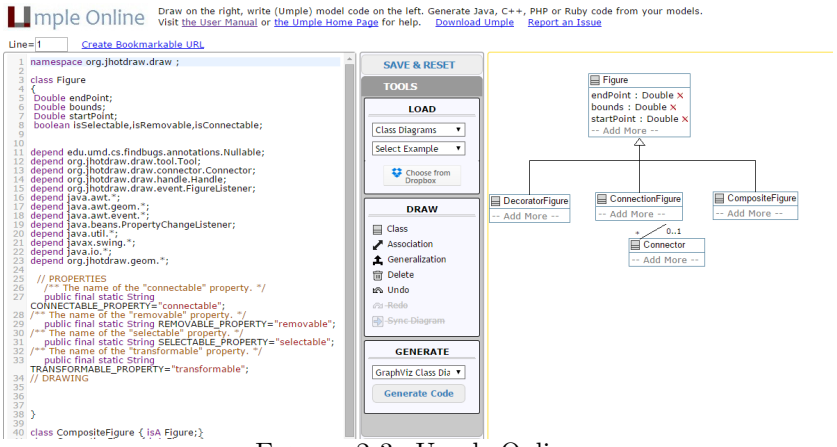

FIGURE 2.3: Umple Online

2.2 Transformations

In this section we will describe the different transformations that can occur during software design and implementation activities that are relevant to our research. In particular, we will present transformations that can occur during the design phase, during the implementation phase and when moving from one phase to another. Forward engineering, reverse engineering, model transformations and refactorings will be introduced in the following sections. The different transformations enumerated above are relevant to our work for the following reasons:

- The umplification approach presented in this thesis is a *reverse engineering* technique performing *model transformations* to transform a base language model into an Umple model.
- The output of umplification is an Umple model. Umple models can be used to generate high quality code (*Forward engineering*).
- *Refactorings* by means of umple code injections are required to adapt the different methods of an input class to conform to the Umple generated methods.

- The umplification approach can be used in some measure to re-engineer an existing software system. A case study presenting a modernized software system is studied in the last section of Chapter 6.

To better conceptualize the different transformations, it is necessary to place them in the larger context of the software system lifecycle. As described in [2] and illustrated in Figure 2.4, we assume a software life cycle with three main stages: Requirements, Design and Implementation. In the requirements phase, the problem being solved is specified, in the design phase the solution is specified using a well-defined model and in the implementation phase the code is produced from the previously obtained model. The work presented in this thesis as well as the related work studied focuses exclusively on the transformation that occur in the two last (abstract) stages of the life-cycle. Figure has been taken from [2] but has been simplified and modified for our purposes. As illustrated in Figure 2.4 the level of abstraction is higher in early stages and lower in later stages. Five different types of transformations can be distinguished:

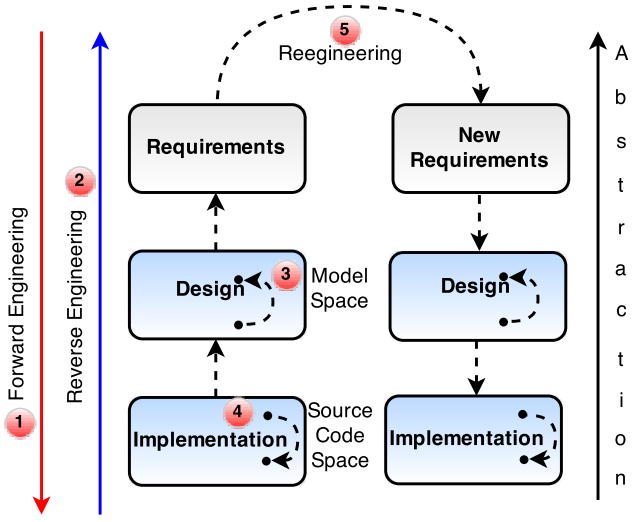

FIGURE 2.4: Transformations across the different software life-cycle phases

1. *Forward engineering* produces source code corresponding to an object model. The modeling constructs in the model such as attributes, associations and high level constraints, are automatically mapped to source code constructs supported by the selected programming language.

2. *Reverse engineering* produces a model corresponding to source code. This transformation is often applied when the design of the system gas been lost and must be recovered. Reverse engineering can be seen as going backwards through the development cycle. Furthermore, reverse engineering does not involve changing aspects of the subject system.

3. *Model transformations* involve a conversion of an object model into another object model.

4. *Refactorings* involve transformations that operate on source code elements. Their goal is to improve aspects of the system without changing its functionality. They operate in the source code space.

5. *Re-engineering* produces a new form of the subject system.

2.2.1 Forward Engineering

In Forward Engineering the target system is created by moving down from high-level abstractions to logic design and proceeds to the physical design of the system. As the of abstraction level decreases, information is collected about the system. The main goal of forward engineering is to maintain a strong correspondence between the object design model and the code and to reduce the development effort.

> "Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system." [2]

Examples of forward engineering transformations include:

Mapping UML classes to Base language Classes Classes in the well-defined model (UML, Umple) are taken one by one and mapped into classes in the target programming language.

Mapping attributes to member variables Attributes in the model are mapped into member variables according to their characteristics. Implications on code generation in a object-oriented language such as Java, include a parameter in the constructor and mutator and accessor methods to manage access to the attribute.

Mapping associations to Collections Associations, model concepts representing links between two or more objects are mapped into member variables and methods. Object-oriented programming languages do not support directly the concept of associations, they provide references to objects that can be stored. Associations are then realized in terms of references, taking into account the different parts of an association such as the role name, multiplicities and navigability.

Mapping Contracts to Exceptions Model constraints expressed in the model in *OCL* or in natural language are mapped to code that checks the preconditions, postconditions and invariant in the target programming language.

Mapping Object models to a persistent storage schema Object models are mapped to a storage schema. Attributes of the object models are mapped into columns and associations between the objects are mapped using a combination of primary and foreign keys.

2.2.2 Reverse Engineering

Reverse engineering which involves extracting design artifacts and recovering models that are less implementation-dependent has been defined by Chikofsky and Cross [2] as:

> Reverse engineering is the process of analyzing a subject system to
>
> - identify the system's components and their interrelationships and
> - create representation of the system in another form or at a higher level of abstraction

Reverse-engineering are generally used to:

- Cope with complexity: understand large an complex systems.
- Generate alternative views: automatically generate different ways to view systems
- Recover lost information: extract what changes have been made and the reasons.
- Detect side effects: help understand ramifications of changes.

- Synthesize higher abstractions: create alternative views that transcend to higher abstracts of software.
- Facilitate reuse: detect candidate reusable artifacts and components.

Table 2.5 summarizes the different reverse engineering approaches[2]:

TABLE 2.5: Summary of Reverse engineering approaches

Approach	**Description**	**Related Techniques**
Re-documentation	Is the creation or revision of a semantically equivalent representation within the same relative abstraction level.	Pretty Printers, Diagram generators, reference listing generators.
Design Recovery	Extracts design abstractions from a combination of code and existing documentation of the system.	Software metrics generators, static analyzers, dynamic tracers, visualization tools.
Restructuring	Is the transformation from one representation form to another at the same relative representation form to another at the same relative abstraction level, while preserving the system's external behavior	Source code analyzers, source code translators.
Data re-engineering	Is the process of analyzing and is the process of analyzing and reorganizing the data structures (and sometimes the data values) in a system to make it more understandable	Data model analyzers
Refactorings	Is restructuring within an object-oriented context	Refactoring APIs

2.2.3 Model Transformations

Model transformation is a method that allows the automation of many activities like reverse engineering, refactoring, integration, analysis and simulation [15], which are used extensively in software development, maintenance and modernization. Model transformations are employed in tools such as code generators and parsers as well.

Kleppe et al. [16] provide a more formal definition of a model transformation:

> A *transformation* is the automatic generation of a target model from a source model, according to a transformation definition.
>
> A transformation *definition* is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language.
>
> A transformation *rule* is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.

Model transformations provide a mechanism for automatically creating or updating target models based on information contained in a source model, e.g. the generation of code from a UML model, the translation of a UML Class Diagram into an ER diagram or the translation of base language code into UML or any other textual or visual modeling language, such as Umple. In order to perform a transformation between models, the models need to be expressed in a software language. This language can be specified by a *grammar* or a *metamodel*. As stated by Kleppe [17], grammars focus on the concrete syntax of the language while metamodels focus on the abstract syntax. Grammars are useful to describe the structure of the words in a language, metamodels are better in describing the language's concepts and its relations. Compared to metamodels, grammars have a strong mathematical basis (induction can be used to prove its correctness) and is tree based (e.g. parse trees are generated from grammar). Also, there are a great variety of tools advanced tools to produce, parse and validate grammars. On the other hand, metamodels are graph based in which relations between language elements are better perceived. Moreover, metamodels are more suitable when defining object-oriented languages but do not contain information on how the concepts in the metamodel are to be represented to the language user[17]. Nevertheless, as studied by Alanen et al. [18] it is possible to transform a grammar definition into a metamodel definition and vice-versa. In fact, the relation between a metamodel and a BNF grammar can in practise be defined using two mappings, one transforming a BNF grammar to a MOF metamodel, and one transforming a MOF metamodel to a BNF grammar.

Previous work [15, 19] model transformations allow us conclude that no model transformation tool or technique exists is *absolutely* better than another one. Instead we can

search for a model transformation approach that is suitable for a specific transformation problem. The following are the main properties of model transformation problems [15]:

- Change of abstraction: Model transformations can change the level of abstraction between the source and the target model. That is, the transformation increase or decrease the level of details or leave it as unchanged.
- Change of Metamodel:
  - In a *endogenous* [20] transformation both the source and target metamodels are the same.
  - In a *exogenous* [20] transformation both the source and target metamodels are different.
- Supported technical spaces: Model transformations can operate between the metametamodel, metamodel and model levels [21].
- Supported number of models: A transformations can involved one (same source model resulting into a modified target model), two models (source and target models are different) and multiple models (several source models that produce a single target model).
- Supported target type: The target model can be either text or another model. A Model-to-Text transformation create its target as a set of strings while a Model-to-Model transformation create its target as an instance of the target metamodel.
- Preservation of properties: Transformations can be performed in such a way that the source and target model have a common property that is not transformed by the transformation [15]. The intent of that common property can be to preserve the semantics, syntax or behavior of the source and target models.

Based on the target type supported, we categorized the model transformations approaches into two major categories[19]: Model-to-Model and Model-to-Text approaches.

2.2.3.1 Model-To-Text Approaches

Visitor-Based Approaches An approach based on the notion of traversing the internal representation of the model. The output is written to a text stream. The approach is based on the Visitor software pattern [22].

Template-Based Approaches Template-based approaches are used in the implementation of code generators. The basic idea is to refine and transforms models into code. The templates contain fragments of the target text and pieces of code that are replaced with information derived from the source model (called meta-programs). This approach can be combine with the visitor-based approach for model traversing.

2.2.3.2 Model-To-Model Approaches

Direct-Manipulation Approaches The approach typically consist of an internal representation of the model (e.g. AST) and some API's to manipulate and query it. Modisco [23] technology falls under this category.

Structure-Driven Approaches The basic idea behind this approach is to copy model elements from the source to the target, which can then be adapted to achieve the transformations goals. The approach is structure-driven because this approach first creates the hierarchical structure of the target model. *OptimaJ* frameworks is one of the representative technologies falling under this category.

Operational Approaches Operational approaches extend the metamodeling formalism with facilities for expressing mapping rules. Imperative approaches focus on how the transformation itself needs to be performed. Operational transformation languages provides support to describe how the transformation language is supposed to be executed. The constructs and concepts of an imperative language are similar to those of general purpose programming languages such as Java. The model trans-formation in this case is described as an ordered sequence of actions. *Kermeta* and *QVT* presented later in this chapter are examples of technologies in this category.

Declarative Approaches Declarative approaches do not offer explicit control flow. They don't describe how the trans-formation should be executed but instead what

should be mapped by the transformation. In other words, they describe the relationship between the source and the target metamodels. The transformation descriptions (or mapping rules) for these languages are, in general, short and concise.

Hybrid Approaches Hybrid transformation languages are a mix of declarative and operational approaches and offer the possibility of declaring how and what elements of the metamodels are going to be mapped [24].

Graph-Transformation-Based Approaches Graph-based approaches can be considered as a special subcategory of declarative languages. Models are interpreted as graphs, and the transformation manipulates these graphs [25]. For instance, the Triple Graph Grammar (TCG) [25] is a way of describing graph transformations. Their rules are specified using three graphs, the left-hand side graph corresponding to the source graph, the right-hand side graph corresponding to the target graph and a correspondence graph describing the mapping between elements of the left-hand side and elements of the right-hand side.

XML Approaches In this approach models are serialized as XML and then traversed and transform using XSLT. However, the use of XMI and XSLT has scalability limitations [26].

2.2.3.3 Model Transformations Languages and Tools

We now introduce some of the most popular existing model transformation technologies.

ATL The ATLAS Transformation Language [27] is a hybrid model-to-model transformation language supporting both declarative and imperative constructs. ATL is integrated in the Eclipse development environment and can handle models based on EMF (Ecore). The ATL code (.atl files) is compiled and then executed by its own transformation engine. Examples of Java-to-Umple ATL transformations will be presented in Chapter 5.4.

QVT The QUERY/VIEW/Transformation [28] is a standardized language for model transformation established by the Object Management Group (OMG). QVT defines three syntaxes for model-to-model transformations: a textual concrete syntax

a XMI based metamodel and visual syntax for matching element between metamodels. Listing 2.18 shows a partial transformation from an UML class model to an Umple model.

LISTING 2.18: A basic QVT transformation

```
1  transformation uml2Umple(
2   in uml : SimpleUML,
3   out umple : SimpleUmple
4  );
5
6  main() {
7    uml.objectsOfType(Class)->map UMLClassToUmpleClass();
8  }
9
10 mapping Class::classToUmpleClass () : UmpleClass
11 {
12   name := self.name;
13   attributes : self.attributes->map attributeToUAttribute();
14 }
15
16 mapping Attributes::attributeToUAttribute () : Attribute {
17  ... omitted
18 }
```

The following gives details of the above:

- Lines 1-4. The transformation declaration specifies the parameter models. The transformation is unidirectional from UML to Umple.
- Line 6. The entry point for the execution is the function *main()*, which invokes the *UMLClassToUmpleClass* mapping on all UML classes.
- Lines 10 and 16. The mappings are defined using the OCL notation. The attributes of each UML class are traversed and converted to Umple attributes (code is omitted). The body of the mapping populates the properties of the return object, while self refers to the object on which the mapping is invoked.

JET Java Emitter Templates (JET) is a generic template engine that can be used to generate SQL, XML, Java source code and other output from templates. The templates uses a JSP-like syntax. For instance, the code in Listing 2.19 will print the words "Hello, Thesis Reader!" to the standard console output. The JET Builder translates the template to a class named BasicTemplate. The template file receives a string argument.

LISTING 2.19: A basic JET Template

```
1 <%@ jet package="hello" class="BasicTemplate" %>
2  Hello, <%=argument%>!
```

To pass arguments to the template method we use the generate *method* as shown in Listing 2.20. Note that is possible to pass a reference (a model element) type as argument.

LISTING 2.20: Instantiating the BasicTemplate class

```
1 BasicTemplate sayHello = new BasicTemplate();
2 String result = sayHello.generate("Thesis Reader");
3 System.out.println(result);
```

Kermeta Kermeta [29] is an imperative programming language used to perform model transformations and for other more general purposes. It offers EMF meta-modeling, checks and behavior support. Incremental model transformations are supported. The Kermeta language uses metamodel-based actions to manipulate elements from different metamodels (in XMI format) and transform them.

ETL ETL [30] is a hybrid model-to-model transformation language. It can handle several source and several target models. It offers support for query/navigate/modify both source and target models. It works at the metamodel level and support EMF models.

TXL TXL [31] is a programming language designed for a variety of analysis and source transformation tasks. TXL is a hybrid rule-based language and it is best at tasks involving source-to-source transformations. Examples of Java-to-Umple TXL transformations will be presented in Chapter 5.4.

Table 2.6 summarizes the most representative tools for each model transformation approach discussed.

2.2.4 Refactorings

A *refactoring* is a transformation of the *source code* aiming at improving its readability and/or design of the code without changing the behavior of the system [32]. To ensure that the refactoring does not change the behavior of the system, the refactoring is done in small incremental steps that are interleaved with tests. For example, a sequence of three refactorings are performed to the source code in Listing 2.21. The resulting source code after the transformations is presented in Listing 2.22. The refactorings are performed

TABLE 2.6: Summary of Model transformation technologies

Tranformation Approach	**Technologies**
M2T- Visitor-Based Approaches	Jamada, CodeWirters
M2T- Template-Based Approaches	JET, Velocity, XDoclet, Codagen
Direct-Manipulation Approaches	JML
Structure-Driven Approaches	QVT, OptimalJ
Operational Approaches	XMF-Mosaic, QVT-Relational, Kermeta
Declarative Approaches	ATL, ETL
Hybrid Approaches	CSCWMDA
Graph-Based Approaches	AGG, AToM3,VIATRA, GReAT, UMLX, BOTL, MOLA, and Fujaba
XML Approaches	XSLT

one by one and ensuring that the refactoring do not change the intended behavior of the system.For a complete catalog of refactorings refer at [32].

1. Pull Up Field: Common field in subclasses is moved to the superclass. In our example, field *name* is moved to superclass.

2. Pull up Constructor: Common code in constructor bodies of subclasses is moved to the superclass constructor.

3. Pull up Method: Methods with identical results on subclasses are moved to superclass. In our example, the methods accessing the *name* field are moved from the subclasses to the superclass.

LISTING 2.21: Before refactorings

```
1  public class Student {
2    private String name;
3    public Student(String name) {
4     this.name = name;
5    }
6    public String getName() {
7   return name;
8    }
9  }
10 ///----- Class Mentor ------
11 public class Mentor {
12   private String name;
13   public Mentor(String name) {
14    this.name = name;
15   }
16   public String getName() {
17  return name;
18   }
19 }
```

LISTING 2.22: After refactorings

```
1  ///----- SuperClass ------
2  public class Person {
3    private String name;
4    public Person(String name) {
5     this.name = name;
6    }
7    public String getName() {
8   return name;
9    }
10 }
11 ///----- Class Student ------
12 public class Student extends
       Person {
13   private String name;
14   public Student(String name) {
15    super(name);
16   }
17 }
18 ///----- Class Mentor ------
19 public class Mentor extends
       Person {
20   private String name;
21   public Mentor(String name) {
22    super(name);
23   }
24 }
```

2.2.5 Re-engineering

Re-engineering is the examination and alteration of a subject system to reconstitute it in a new form [2]. Re-engineering transformations are usually concerned with reimplementing a system (or parts of it) to make it more maintainable. Re-engineering involves redocumenting the system, organizing and restructuring the system or translating the system to a more modern programming language, know as modernization. Modernization is performed to extract the main components of the system, written in old (legacy) code, and to reproduce the original system using a more recent programming language or using modern frameworks and libraries.

The main activities in a typical re-engineering process are:

Source code Translation The most simple form of re-engineering is source code translation and involves the automatic translation of source code written in one programming language to source code in another (i.e., C to C++). The translation should not modify the structure and organization of the system. Source code translation can be done to the same but more modern version of the language (i.e.,Java

1.4 to Java 1.8). A source code translation is very desirable when the language compiler or support is discontinued or when the organization policies impose a change on the language (i.e., Microsoft technologies to Open source technologies). Furthermore, systems written in modern languages are often easier to understand, test and maintain than legacy systems [33].

Reverse Engineering Reverse Engineering can be used as part of the Reengineering process to recover the original program design. The design can then help developers understand the program internals before attempting to improve it. As originally explained by Chikofsky [2], reverse engineering and reengineering differ in their purpose. The main goal of reverse engineering is to derive the specification or design of a system from its source code, while the purpose of reengineering is to produce a new but more maintainable system.

Program Improvements This part of the reengineering process involves improving the structure of the program to optimize memory use or to simplify the logic structure of the system.

In the following chapter, we will describe the core concept of this thesis, the umplification technique, a *reverse engineering* technique that employs *model-to-model transformations* to incrementally transform base language code into Umple code.

Chapter 3

Reverse Engineering of Object Oriented Systems into Umple

In this chapter we provide an overview of our reverse-engineering technique, called *umplification*. Then, we discuss our motivations for developing the umplification technique and present a comprehensive example.

3.1 Umplification Process

Umplification is a play on words with the concept of 'amplification' and also the notion of converting into Umple. The techniques produces a program with behavior identical to the original one, but written in Umple. The umplification process is incrementally performed until the desired level of abstraction is achieved.

3.1.1 Description

Umplification, as illustrated in Figure 3.1, involves recursively modifying the Umple model/code or the base language code to incorporate additional abstractions, while maintaining the semantics of the program, and also maintaining, to the greatest extent possible, such elements as layout. The end product of umplification is an Umple program/model that can be edited and viewed textually just like the original program, and also diagrammatically, using Umple's tools.

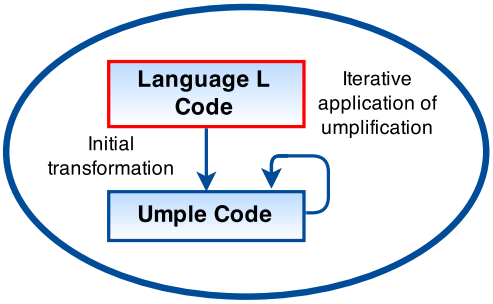

FIGURE 3.1: The Umplification process generalized

3.1.2 Properties

The umplification process has several properties. It is:

1. **incremental**,
2. **transformational**,
3. **interactive**,
4. **extensible**, and
5. **implicit-knowledge** conserving.

The approach is **incremental** because it can be performed in multiple small steps that produce (quickly) a new version of the system that has a small amount of additional modeling information, such as the presence of one new type of UML/Umple construct. At each step, the system remains compilable. The approach proceeds incrementally performing additional transformations until the desired level of abstraction is achieved. These incremental transformations allow for user interaction to provide needed information that may be missing or hard to automatically obtain because the input (the source code) does not follow any of the idioms the automatic umplification tool is yet able to recognize. This characteristic of umplification allows developers, if they wish, to repeatedly re-introspect the transformed program and manually validate each change with an understanding of the incremental purpose of the change.

The approach is **transformational** because it modifies the original source rather than generating something completely new. It first translates the original language (Java, C++ etc.) to an initial Umple version that looks very much like the original, and

then translates step-by-step as more and more modeling constructs are added, replacing original code.

The approach is **interactive** because the user's feedback may be used to enhance the transformations.

The approach is **extensible** because it uses the set of transformation rules can be readily extended to refine the transformation mechanism.

Finally the approach is **implicit-knowledge** conserving because it preserves code comments, and, where possible, the layout of whatever code is not (yet) umplified. The latter includes as the bodies of algorithmic methods known as *action code* in UML.

Taken together, the above properties allow developers to confidently umplify their systems without worrying about losing their mental model of the source code. Developers gain by having systems with a smaller body of source code that is intrinsically self-documented in UML.

3.1.3 Overview of Transformations cases

The following gives a summary of the abstract transformations currently implemented.

Transformation 0: Initial transformation Source files written in language L (e.g. Java, C++) code are initially renamed as Umple files, with extension .ump. File, package and data type dependencies are translated into Umple dependencies by using the Umple depend construct.

Transformation 1: Transformation of generalization/specialization, and namespace declarations The notation in the base language code for subclassing is transformed into the Umple 'isA' notation. Umple now recognizes the class hierarchy. Notations for namespaces or packages are transformed into the Umple 'namespace' directives. At this stage, an Umple program, when compiled should generate essentially identical code to the original program.

Transformation 2: Analysis and conversion of many instance variables, along with the methods that use the variables This transformation step is further decomposed into sub-steps depending on the abstract use of the variables. The sub-steps are defined as follows.

Transformation 2a: Transformation of variables to UML/Umple attributes If variable *a* is declared in class A and the type of *a* is one of the primitive types in the base language, then *a* is transformed into an Umple attribute. Any accessor (e.g. getA()) and mutator (e.g. setA()) methods of variable a are transformed as needed to maintain a functioning system. In particular, any getter and setter methods in the original system must be adapted to conform to or call the Umple-generated equivalents.

Transformation 2b: Transformation of variables in one or more classes to UML/Umple associations If variable *a* is declared in Class A and the type of *a* is a reference type B, then a is transformed into an Umple Association with ends {a, b}. At the same time, if a variable b in class B is detected that represents the inverse relationship then the association becomes bidirectional. The accessor and mutator methods of variable *a* (and *b*) are adapted to conform to the Umple-generated methods. Multiplicities and role names are recovered by inspecting both types A and B;

Transformation 2c: Transformation of variables to UML/Umple state machines If *a* is declared in Class A, has not been classified previously as an attribute or association, has a fixed set of values, and changes in the values are triggered by events, and not by a set method, then a is transformed to a state machine.

As mentioned before, as part of each transformation step, the accessor, mutator, iterator and event methods are adapted (refactored) to conform to the Umple generated methods. Table 3.1 summarizes these additional required refactorings.

In the following section, we provide a more detailed view of the transformation cases and an example to summarize the main points of the umplification process. To help distinguish between Umple and Java code presented in this thesis, the Umple examples appear in solid borders with blue shading, pure Java examples have solid borders with green shading.

TABLE 3.1: Refactorings to methods required for each transformation

Transformation case	Method Transformations
(0) Classes	None
(1) Inheritance	None
2a) Attributes	Accessor (getter) and mutator (setter) methods are removed from the original code if they are simple since Umple-generated code replaces them. Custom accessors and mutators are refactored so Umple generates code that maintains the original semantics.
(2b) Associations	Accessor and mutator methods are removed or correctly injected into the Umple code.
(2c) State Machines	Methods triggering state change are removed if they are simple (just change state) or modified to call Umple-generated event methods.

3.1.3.1 More Details of the Initial Transformation

As mentioned, the first step in umplification (Transformation 0) is to rename the Java/C++ files as .ump files.

After this, various syntactic changes are made (Transformation 1) to adapt the code to Umple's notations for various features that are expressed differently in Java and C++. Umple maintains its own syntax for these features so as to be language-independent. First the base language notation for inheritance (e.g. 'extends' in Java) or interface implementation (e.g. 'implements') is changed into the Umple notation 'isA'. This Umple keyword is used uniformly to represent the generalization relation-ship for classes, interfaces and traits. The same notation is used for all three for flexibility so that, for example, an interface can be converted to a class with no change to its specializations, or a trait can be generated as a superclass in languages such as C++ where multiple inheritance is allowed.

After this, the dependency notation in the native language (e.g. 'import' in Java) is changed to the 'depend' notation in Umple. Finally 'package' declarations are transformed into Umple namespace declarations. Transformations made as part of these first refactoring steps, are one-to-one direct and simple mappings between constructs in the base language and Umple. No methods need changing. The final output after execution of the above transformations, is an Umple model/program that can be compiled in the

same manner as the original base language code. At this point, any available test cases may be run to ensure that the program's semantics are preserved.

3.1.3.2 Details of the Transformations to Create Attributes

The goal of this step is to transform member variables meeting certain conditions into Umple attributes (Transformation 2a). An Umple attribute, as discussed in Chapter 2, it is more than just a plain private variable: It is designed to be exclusively operated on by mutator methods, accessed by accessor methods and (depending on its properties) automatically initialized in the constructor. These methods, in turn can have semantics such as preconditions and tracing injected into them.

We start by analyzing all instance variables for their presence in constructor and get/set methods and decide whether the member variable is a good candidate to become an Umple attribute. In addition to the previous conditions, if the candidate attribute has as its type either:

1. a simple data type
2. a class that only itself contains instance variables meeting conditions in a and b (for attributes with 'many' multiplicity)

Then, the member variable is transformed into an Umple Attribute. If it is not possible to draw a conclusion regarding whether or not the member variable corresponds to an Umple Attribute, the member variable is left to be later transformed into an association or a state machine. If the member variable does not meet any of the criteria required to perform the transformations, the member variable (and its accessors/mutators) is not processed. Further details on this decision making process will be provided in the next chapter of this thesis.

We culminate this refactoring step by removing or refactoring getters and setters of the previously identified attributes. More specifically, the getters and setters need to be refactored if they are custom. Simple getters/setters are those that only return/update the attribute value. Custom getters/setters are those that provide behavior apart from getting and setting the variable such as validating constraints, managing a cache or filtering the input.

3.1.3.3 Transformations to Create Associations

As discussed earlier, in the various cases of the refactoring steps, analyses are applied to the input variables to determine whether each variable can be transformed into an Umple association. An association specifies a semantic relationship that occurs between typed instances. A variable represents an association if all of the following conditions apply:

- Its declared type is a Reference type (generally a class in the current system).
- The variable field is simple, or the variable field is a container (also known as a collection).
- The class in which the variable is declared, stores, access and/or manipulates instances of the variable type.

The sort of refactoring discussed above for attributes is also performed when associations are found, although the actual code logic is more sophisticated.

A complete analysis on the different transformations cases is presented in Chapter 4. From this analysis, a set of transformations rules are derived. The actual implementation of the transformations rules is discussed in Chapter 5.4.

3.2 Motivations

Our desire to develop our reverse-engineering approach arose for two main reasons. We address each of these in the following sections.

3.2.1 Model-code duality

Developers often work with large volumes of legacy code. Reverse engineering tools allow them to extract models in a variety of ways [34], often with UML as the resulting formalism.

The extracted models can be temporary, just-in-time aids to understanding, to be discarded after being viewed. Such a mode of use can be useful, but is limited in several

ways: Developers still need to know where to start exploring the system, and they need to remember how to use the reverse engineering tool every time they perform an exploration task.

Developers generally therefore would benefit from choosing reverse engineering tools that create a more permanent form of documentation that can be annotated or embedded in larger documents, and serve as the definitive description of the system.

However by making the latter choice, the developer then needs to maintain two different artifacts, the original code and the output model. The recovered models become obsolete quickly, unless they are continuously updated or are used for 'roundtrip engineering'. The complexity of this inhibits developers from using reverse engineering tools for permanent documentation.

The umplification technique we present in this thesis overcomes the problems with either mode of reverse engineering described above. It results in a system with a model that can be explored as easily as with just-in-time tools. But there is also no issue with maintaining the model, because model and code become the same thing.

In other words, the key difference compared to existing reverse engineering techniques and the main motivation for this work is that the end-product of umplification is not a separate model, but a single artifact seen as both the model and the code. In the Umple world, modeling is programming and vice versa. More specifically, for a programmer, Umple looks like a programming language and the Umple code can be viewed as a traditional UML diagram. This allows developers to maintain the essential 'familiarity' with their code as they gradually transform it into Umple [35].

3.2.2 Improving Program Comprehension

In addition to solving the problem of having two different software artifacts to maintain, umplification can be used to simplify a system. The resulting Umple code base tends to be simpler to understand [4] as the abstraction level of the program has been 'amplified'.

With a system written in Umple, large amounts of boilerplate code are avoided. The benefits of *umplifying* a system not only include recovering a textual model but also eliminating that repetitive code from the programs. For instance, when an association

is umplified, all the methods for adding, removing and setting links of the association, are removed (or refactored under certain conditions that will be explained later). This promotes code readability and reduces code volume and code density. Kiczales provides in his work [36] some evidence that reducing the code volume can help to improve program comprehension.

The Umplification technique improves program comprehension by:

- Allowing developers to describe and develop a system at a more abstract level and
- Removing boilerplate code when incorporating a new abstraction
- Reducing the complexity when an Umple association is incorporated. An Umple association consists of a single line of Umple code. To implement the association in a language like Java, we need to include member variables in both classes, methods to add, delete, query an iterate through links, as well as some code in the constructors.
- Reducing the complexity when an Umple attribute is incorporated. First, attributes can remain untyped (defaulted to a String implementation) and generate both set/get accessor methods, which further reduces the code volume and code density. Complexity is also significantly reduced since the mechanism to manage a list attribute does not have to be coded. This API includes methods like adding and removing entities, retrieving one or all entities as well as asking how many entities belonging to the instance.

3.3 School System - Manual Umplification Example

We illustrate the umplification process with a small example. The original Java is presented through Listings 3.1 - 3.3.

Listing 3.1: Student.java

```
20 package university;
21
22 public class Student extends
       Person{
23
24  public static final int
       MAX_PER_GROUP = 10;
25  private int id;
26  private String name;
27  public Mentor mentor;
28
29  public Student(int id,String
       name){
30   id = id; name = name;
31  }
32  public String getName(){
33   String aName = name;
34    if (name == null) {
35     throw new RuntimeException("
       Error");
36    }
37    return aName;
38  }
39  public Integer getId() {
40   return id;
41  }
42  public void setId(Integer id) {
43   this.id = id;
44  }
45  public boolean getIsActive() {
46   return isActive;
47  }
48  public void setIsActive(boolean
       aIsActive) {
49   isActive = aIsActive;}
50  }
51  public Mentor getMentor() {
52   return mentor;
53  }
54  public void setMentor(Mentor
       mentor) {
55   this.mentor = mentor;
56  }
57 }
```

Listing 3.2: Mentor.java

```
1  package university;
2  import java.util.Set;
3
4  public class Mentor extends
       Person{
5
6   Mentor() {}
7   public Set<Student> students;
8   public Set<Student> getStudents
       () {
9    return students;
10  }
11  public void setStudents (Set<
       Student>students) {
12   this.students = students;
13  }
14  public void addStudent( Student
       aStudent){
15   students.add(aStudent);
16  }
17  public void removeStudent(
       Student aStudent) {
18   students.remove(aStudent);
19  }
20  public String toString() {
21       return(
22          (name==null ? " " : name
       ) + " " +
23          students.size()+ "
       students"
24       );
25  }
26 }
```

Listing 3.3: Person.java

```
1 package university;
2 public class Person {
3  public String getName() {return this.name;}
4  public void setName(String name){
5   this.name= name;
6  }
7 }
```

3.3.1 School system: Initial Transformation

We create the .ump files, one Umple file per input class. Three Umple files, in Listings 3.4, 3.5,3.6 ,are created as result of this initial transformation.

As discussed earlier, the dependency, package and generalization notation is changed to their respective Umple notation. For instance, the Java code of class Mentor shown in Listing 3.2 would result in Umple implementation shown in Listing 3.4 (in file Mentor.ump). The following give details of the initial transformation:

- Package declaration in 3.2 becomes an Umple namespace declaration in Listing 3.4.
- Import declaration in 3.2 becomes an Umple depend declaration in Listing 3.4.
- Generalization notation in 3.2 is transformed to the 'isA' notation (Line 5) in Listing 3.4.
- Code in Lines 6-26 in Listing 3.2 is not touched by the initial trasnformation; The same exact code is found in the Umple file in Listing 3.4 (Lines 7-27).

LISTING 3.4: Mentor.ump

```
1  namespace university;
2  class Mentor {
3
4   depend java.util.Set;
5   isA Person;
6
7   Mentor() {}
8   public Set<Student> students;
9   public Set<Student> getStudents() {
10   return students;
11  }
12  public void setStudents (Set<Student>students) {
13   this.students = students;
14  }
15  public void addStudent( Student aStudent){
16   students.add(aStudent);
17  }
18  public void removeStudent(Student aStudent) {
19   students.remove(aStudent);}
20  }
21  public String toString() {
22       return(
23          (name==null ? " " : name) + " " +
24          students.size()+ " students"
25       );
26  }
27 }
```

The umple code for classes Student and Person after completion of the first refactoring step is presented in Listings 3.5 and 3.6.

Listing 3.5: Student.ump

```
namespace university;
class Student {
 isA Person;
 public static final int MAX_PER_GROUP = 10;
 private int id;
 private String name;
 public Mentor mentor;

 public Student(int id,String name){
  id = id; name = name;
 }
 public String getName(){
  String aName = name;
   if (name == null) {
    throw new RuntimeException("Error");
   }
   return aName;
 }
 public Integer getId() {
  return id;
 }
 public void setId(Integer id) {
  this.id = id;
 }
 public boolean getIsActive() {
  return isActive;
 }
 public void setIsActive(boolean aIsActive) {
  isActive = aIsActive;}
 }
 public Mentor getMentor() {
  return mentor;
 }
 public void setMentor(Mentor mentor) {
  this.mentor = mentor;
 }
}
```

Listing 3.6: Person.ump

```
namespace university;
class Person {
 public String getName() {return this.name;}
 public void setName(String name){
  this.name= name;
 }
}
```

The visual representation of the Umple model at the end of this transformation step is shown in Figure 3.2. At this point, we have gained knowledge about the hierarchical structure of the system. Attributes and associations are not shown in the diagram since they have not been yet reverse-engineered.

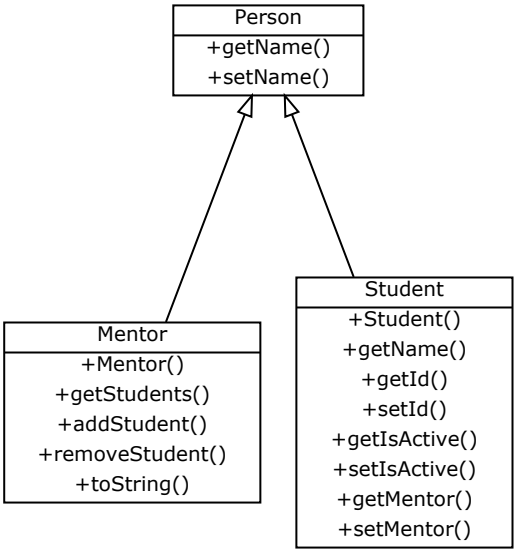

FIGURE 3.2: UML Class Diagram of the Mentor-Student example - Level 1

3.3.2 School system: Transformations to Create Attributes

Assuming that we have successfully performed the initial transformation on all the input files, at this point the input for the second transformation step are three Umple files. We first analyze the variables (they are still variables even if they are inside an Umple class) to determine certain characteristics such as the following:

1. Is the field present in the parameters of the constructor?
2. Does the field possess a getter?
3. Does the field possess a setter?
4. Is the field's type, a primitive type?

For instance, if we analyze the member variables in class Student, we obtain the results in Table 3.2.

TABLE 3.2: Analysis of Member variables of class Student

Member Variable	1	2	3	4
id	Yes	Yes	Yes	Yes
isActive	No	Yes	Yes	Yes
name	Yes	Yes	No	Yes
MAX_PER_GROUP	No	No	No	Yes

The results of this analysis allow us to generate Umple code with the required types and stereotypes. For example the stereotype 'lazy' is added to 'isActive' because it should

not appear in the constructor, and the stereotype 'immutable' is added to *name* since there is no setter. The transformed Umple code after completion of this refactoring step (transformation 2a) is shown in Listing 3.7. Note that this continues to generate a program that is semantically identical to the pre-transformation version.

LISTING 3.7: Student.ump

```
1  namespace university;
2
3  class Student {
4   Integer id;
5   lazy Boolean isActive;
6   immutable name;
7   const Integer MAX_PER_GROUP = 10;
8   after getName {
9    if (name == null) {
10    throw new RuntimeException("Error");
11   }
12  }
13   public Mentor mentor;
14   public Mentor getMentor() {
15    return mentor;
16   }
17   public void setMentor(Mentor mentor) {
18    this.mentor = mentor;
19   }
20 }
```

The following gives details of Listing 3.7:

- Line 4: Field id becomes an Umple attribute. Getter getId() and setter setId() are removed.
- Line 5: Field isActive becomes an Umple attribute of Boolean type. As the field is not required in the constructor we marked as 'lazy' so the Umple compiler does not generate a constructor argument for this attribute.
- Line 6: Field name becomes an Umple attribute and is marked as 'immutable'. Immutable attributes must be specified in the constructor, and no setter is provided. In this particular example, the input java code (Listing 3.1) does not contain code that sets this variable anywhere, therefore it is transformed into an immutable umple attribute.
- Line 7: Field MAX_PER_GROUP becomes a constant (special type of Umple attribute). We have drawn this conclusion because of the field modifiers (e.g. static final) and because of the ALL_CAPS convention.

- Line 8-12: As the getter for field name was custom in Listing 3.1, we have adapted it so it conforms to the one that can be generated by the Umple compiler. A code injection, code that is injected before and/or after statements, have been used for this purpose.

- Code in Lines 32-37 in Listing 3.1 is not touched by this transformation; The same exact code is found in the Umple file in Listing 3.7 (Lines 13-19).

In the same manner, we umplify the attributes of classes Mentor and Person. The Umple code for class Mentor remains identical as in Listing 3.4, since we could not find any member variables in this class meeting the conditions to become an Umple attribute. On the other hand, the member variable 'name' in class Mentor has been transformed into an attribute of String type. The resulting Umple code for class Person, after this transformation is shown below in Listing 3.8.

LISTING 3.8: Person.ump

```
1 namespace university;
2 class Person {
3  String name;
4 }
```

The visual representation of the Umple model at the end of this transformation step is shown in Figure 3.3. At this point, we have gained knowledge about the hierarchical structure of the system and the attributes of each class. Associations are not shown in the diagram since they have not been yet reverse-engineered.

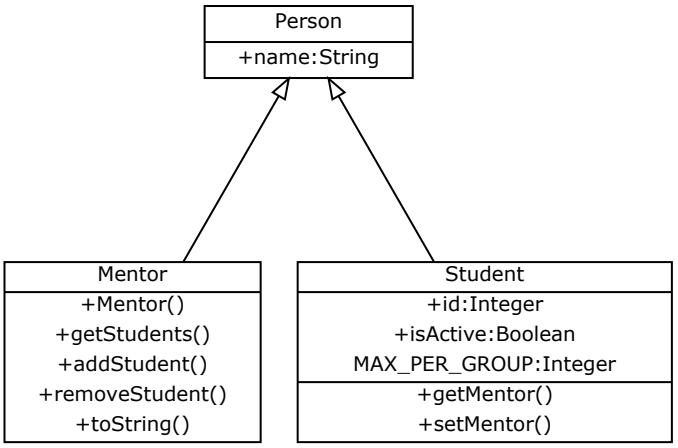

FIGURE 3.3: UML Class Diagram of the Mentor-Student example - Level 2

3.3.3 School system: Transformations to Create Associations

Assume again that the source code has already passed through the two first refactoring steps and the input at this point is the Umple code found in Listings 3.8, 3.4 and 3.7.

The resulting Umple code for class Mentor after completion of this refactoring step (transformation 2b) is shown below in Listing 3.9. Line 2 contains the association derived from the Java code that can be read as: a mentor can have many students associated but a student can only be associated to at most one mentor.

LISTING 3.9: Mentor.ump

```
1  namespace university;
2  class Mentor {
3   0..1 -- 0..* Student;
4
5   public String toString() {
6    return(
7    (name==null ? " " : name) + " " +
8     students.size()+ " students");
9   }
10 }
```

LISTING 3.10: Student.ump

```
1  namespace university;
2  class Student {
3   Integer id;
4   lazy Boolean isActive;
5   immutable name;
6   const Integer MAX_PER_GROUP = 10;
7
8   after getName {
9    if (name == null) {
10    throw new RuntimeException("Error");
11   }
12  }
13 }
```

The following particularities have been taken into consideration during the extraction of the association:

In class *Mentor*:

1. The students variable in class Mentor is of a reference type and possesses a getter and a setter.

2. We inferred the multiplicity of the association end "0..*" by a) inspecting the cardinality of the member, and b) by analyzing the getter/setter of the member variable.

3. We inferred the navigability of the association "–" by inspecting the two classes involved. In this case, each class can access the linked objects of the other class. The notation "->" would otherwise have been used to represent a unidirectional association.

4. The association end is optional-many because the member is not present as a parameter in the constructor (not required upon construction) of an instance of the class Mentor and because the member represents a collection of elements.

In class *Student*:

1. The mentor in class Student is of a Reference type and it possesses a getter and a setter.

2. We inferred the multiplicity of the association end "0..1" by inspecting the constructor of the class Student. It is optional-one because it is not required upon construction of class Student.

3. Methods *setMentor()* and *getMentor()* are no longer needed in class Student and therefore removed.

Consider again the previous example. If we inject now the constructor of Listing 3.11 into the Student class, the multiplicity for the association end would become "1" instead of "0..1".

Listing 3.11: A new constructor added to Student class

```
1 public Student(Mentor aMentor){
2   mentor = aMentor;
3 }
```

Note that in the examples, the Java input made use of generics (templates using '<>' syntax) for the specification of a collection of elements. For those cases in which the type of the member variable cannot be directly inferred (in older Java code), we analyze the add/remove methods to determine the type of the element that is added to the

collection. We will explore all possible detection mechanisms for associations in Chapter 4. Ultimately, each refactoring step should involve testing (running the test suites) to check that the program's semantics are preserved.

The visual representation of the Umple model at the end of this transformation step is shown in Figure 3.4. As a result,we have gained knowledge about the hierarchical structure of the system, attributes and associations of each class.

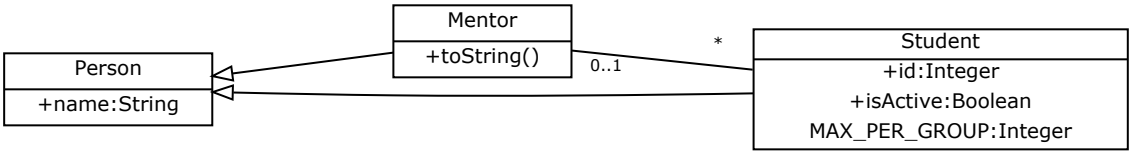

FIGURE 3.4: UML Class Diagram of the Mentor-Student example - Level 3

Finally, for this small system, the Umple code extracted (33 LOC) contains substantially fewer lines of code the original system written in Java (77 LOC). Despite being a simple metric, the number of lines of code is a fair indicator of complexity [37].

3.4 ATM system - Manual Umplification Example

We will now illustrate the transformations steps through an example. This will show how umplification is performed manually. In Chapter 5 we will show how to perform automated umplification.

In this section, we will umplify a moderately-sized Java system comprised of:

- 24 files, 2470 Lines of Code.
- Two top packages *atm* and *banking*.
  - atm: ATM, Session
  - atm.physical: CardReader, CashDispenser, CustomerConsole, EnvelopeAcceptor, Log, NetworkToBank, OperatorPanel, ReceiptPrinter
  - atm.transaction: Transaction, Withdrawal, Deposit, Transfer, Inquiry
  - banking: AccountInformation, Balances, Card, Message, Money, Receipt , Status.

- Two 'top-level' classes: ATMMain and ATMApplet allowing the system to be run as an application or as an applet.

The original Java source code has been taken from [38]. Although this can be considered as a small application, by going through the source code of the ATM program it is not easy to understand how the classes are organized, how they interact with each other and how the responsibilities are distributed among the classes. A programmer aiming to use, document or extend this ATM application may want to know the impact of a change by looking at the dependencies, generalizations and associations connecting the entities or to obtain a high level view of the system for program understanding. The reader can take any versions of the code, at each step of umplification (obtained from [39]), and paste it into UmpleOnline [11] or our Eclipse-based Umple environment.

In the following sub-sections, we present the transformation details for the classes in package 'atm.banking' and the main program class for the system 'ATMMain.java'. Note that the comments have been ignored and code in some methods have been omitted to save space. Listings 3.12 - 3.18 present each of the classes in this package, the input code. The Umple code resulting from each step will be shown as well as the UML class diagram (visual representation of the model).

Listing 3.12: Card.java

```
58 package banking;
59
60 public class Card
61 {
62  private int number;
63
64  public Card(int number)
65  {
66   this.number = number;
67  }
68
69  public int getNumber()
70  {
71   return number;
72  }
73 }
```

Listing 3.13: AccountInfo.java

```
27 public class AccountInformation
28 {
29  public static final String []
       ACCOUNT_NAMES =
30     { "Checking", "Savings", "
       Money Market" };
31
32  public static final String []
       ACCOUNT_ABBREVIATIONS =
33     { "CHKG", "SVGS", "MMKT" };
34 }
```

Listing 3.14: Status.java

```
74 package banking;
75
76 public abstract class Status
77 {
78  public String toString()
79  {
80   if (isSuccess())
81    return "SUCCESS";
82   else if (isInvalidPIN())
83    return "INVALID PIN";
84   else
85    return "FAILURE " + getMessage
       ();
86  }
87
88  public abstract boolean
       isSuccess();
89  public abstract boolean
       isInvalidPIN();
90  public abstract String
       getMessage();
91 }
```

Listing 3.15: Receipt.java

```
35 package banking;
36
37 import atm.ATM;
38 import atm.transaction.
       Transaction;
39 import java.util.Date;
40 import java.util.Enumeration;
41
42 public abstract class Receipt
43 {
44
45  private String [] headingPortion
       ;
46  protected String []
       detailsPortion;
47  private String []
       balancesPortion;
48
49  protected Receipt(ATM atm, Card
       card, Transaction transaction
       , Balances   balances)
50  {
51   // Code omitted
52  }
53
54  public Enumeration getLines()
55  {
56   // Code omitted
57  }
58 }
```

Listing 3.16: Balances.java

```
92 package banking;
93
94 public class Balances
95 {
96  private Money total;
97  private Money available;
98
99  public Balances(){}
100
101  public void setBalances(Money
       total, Money available)
102  {
103    this.total = total;
104    this.available = available;
105  }
106
107  public Money getTotal()
108  {
109   return total;
110  }
111
112  public Money getAvailable()
113  {
114   return available;
115   }
116 }
```

Listing 3.17: Money.java

```
59 package banking;
60
61 public class Money
62 {
63  private long cents;
64
65  public Money(int dollars)
66  {
67   this(dollars, 0);
68  }
69
70  public Money(int dollars, int
       cents)
71  {
72   this.cents = 100L * dollars +
       cents;
73  }
74
75  public Money(Money toCopy)
76  {
77   this.cents = toCopy.cents;
78  }
79
80  public String toString()
81  {
82   return "$" + cents/100 +
83   (cents %100 >= 10  ? "." +
       cents % 100 : ".0" + cents %
       100);
84  }
85
86  public void add(Money
       amountToAdd)
87  {
88   this.cents += amountToAdd.cents
       ;
89  }
90
91  public void subtract(Money
       amountToSubtract)
92  {
93   this.cents -= amountToSubtract.
       cents;
94  }
95
96  public boolean lessEqual(Money
       compareTo)
97  {
98   return this.cents <= compareTo.
       cents;
99  }
100 }
```

3.4.1 ATM system: Initial Transformation

We create the .ump files, one Umple file per input class. 24 Umple files in total (7 for the package 'atm.banking') are created as result of this initial transformation.

The dependency, package and generalization notation is changed to their respective Umple notation.

For instance, the Java code (in file ATMMain.java) shown in Listing 3.18 would result in Umple implementation shown in Listing 3.19 (in file ATMMain.ump). Only the code transformed has been shown in Listings 3.18 – 3.18. 'The rest of the code' corresponds to the code that has not been touched during the transformation.

LISTING 3.18: ATMMain.java

```
117 import java.awt.*;
118 import java.awt.event.*;
119 import atm.ATM;
120 import simulation.Simulation;
121
122 // Main program
123 public class ATMMain
124 {
125    // The rest of the code
126 }
```

LISTING 3.19: ATMMain.ump

```
101 // Main program
102 class ATMMain
103 {
104   depend simulation.Simulation;
105   depend atm.ATM;
106   depend java.awt.event.*;
107   depend java.awt.*;
108   // The rest of the code
109 }
```

At the end of this transformation step, the visual representation of the umplified model is shown in Figure **??**.

3.4.2 ATM system: Transformations to Create Attributes

3.4.3 ATM system: Transformations to Create Associations

3.5 Summary

Umplification is a process for converting a base language program into an Umple program, involving a set of transformation steps. Umplification responds to two needs: The first concerns models that become obsolete very quickly. The second is the desire to simplify existing systems.

In this chapter we have discussed the different transformations steps of the Umplification process. In the next Chapter, we will present the mapping rules derived from each of

the transformations steps. These transformation rules will be necessary to automate our reverse engineering process. Implementation of the mapping rules are then presented in Chapter 5.4.

Chapter 4

Detection Mechanisms for UML/Umple Constructs

In this chapter we will present the different mechanisms to detect UML/Umple attributes, associations and state machines from source code written in an object-oriented programming language. We started to build our detection rules based on the following::

1. Documented implementations of attributes, associations and state machines in high level programming languages mentioned in the literature. In particular, we have considered research that discusses how to generate code from this modeling concepts (Forward engineering).

2. Documented techniques in the literature for discovering modeling constructs in object-oriented source code (Reverse Engineering).

3. Our own analysis of open source systems written in object-oriented programming languages.

To present the set of transformation rules we developed from the above, we will employ a semi-formal notation. The rules define the transformation from input to output. It must be pointed out that these transformations rules can be described using a metamodel-based language or grammar-based language. Both types of descriptions allow us to specify mappings between source and target models in the form of conditions (patterns) and conversions (replacements) executed from input to output when those conditions

hold. As the intent of our work is to transform object-oriented language programs into umple programs, we will employ a metamodel-based approach to better express the input/output relationships. The metamodel-based language will be enhanced with OCL [40] expressions.

4.1 A Notation for Transformation Rules

??

Transformation rules presented in this chapter will contain the following information:

1. A name for each transformation rule used for reference purposes.
2. The source language reference
3. Constants used in the generation of the target
4. Helper methods used in the extraction of information from target or input models.
5. A set of named source language model elements from the source language metamodel that we call B.
6. A set of named target language model elements from the target language metamodel that we call U.
7. The source language conditions: invariants that state the conditions that must hold in the source model for this transformation to be applied.
8. The target language conditions: invariants that state the conditions that must hold in the target model for this transformation to be applied.

Listing 4.1 presents a template definition for the transformations rules.

Listing 4.1: Template definition for Transformation rules

```
1 Transformation Name  (InputModel, OutputModel){
2  order n
3  params
4   ...
5  in
6    sourceElement: ...
7  out
8    targetElement: ...
```

```
9  in conditions
10    ...
11 out conditions
12  ...
13 mappings
14    sourceElement.propertyA -> targetElement.propertyA;
15    sourceElement.propertyB -> targetElement.propertyB;
16 }
```

The various parts of a transformation rule have their own specific notation:

- Line 1. Every transformation possess a name. The source and target languages are referenced by stating both language names after the transformation name.
- Line 2. The order of application of the rule is determined by the integer number following the keyword *order*. Rules with lower order numbers are executed first. Order of application of the rules doesn't matter when two or more rules possess the same order number.
- Line 3. Local and global variables to be used in the transformation rule are specified following the keyword *params*.
- Lines 5 and 7. The source model and target elements are written as variable declarations following the keywords *in* and *out* respectively.
- Lines 9 and 11. The conditions that must hold on the source and target model elements are specified following the keywords *in conditions* and *out conditions* respectively. The mappings are performed only if both the in and out conditions are true. Conditions can be specified using OCL syntax.
- Line 13. The mapping rules come after the keyword *mappings*. The symbol ->is used as an infix operator with two operands. The symbol specifies a transformation of the left hand side operand to the right side operand (rules are unidirectional).

We will now introduce the transformation rules for each of the transformation steps presented in Chapter 3. The set of rules for each transformation case constitutes a transformation case definition. Rules in the set are to be executed in sequence.

4.2 Transformation Rules for the Initial Transformation Step

Transformations rules for this step are straightforward and aim at transforming the package declaration, namespace declarations, import declaration and the generalization notation into the corresponding Umple notation.

The mapping rule in Listing 4.4 specifies the transformation of a Class into an Umple class using the language defined in the previous section. A type declaration in the base language model represents an object-oriented type (i.e., a class, interface, struct, etc). The only required condition in the rule shown in Listing 4.4 (Line 7) is that the *typeDeclaration* must declare a class and not an interface or abstract class. A very similar rule is then required for the transformation of a *typeDeclaration* into an Umple Interface. The relationship between the source model, the target model and the mapping rule are illustrated in Figure 4.1. Transformation rules defined in a general but formal language can then be adapted to specific languages such as Java or C++. For instance, the name of a typeDeclaration if the input language is Java can be extracted using the *getName().getFullyQualifiedName()* call sequence. Actual implementation of the mapping rules will be presented in Chapter 5.4. For our first example, we show the BNF grammar for a type declaration in Java and an Umple class declaration in Listings 4.2 and 4.3 respectively. It must be pointed out that the transformation rule can be understood either by looking at the metamodel or the grammar of input/output models.

Listing 4.2: Grammar for Java Types

```
1  TypeDeclaration:
2                  ClassDeclaration
3                  InterfaceDeclaration
4   ClassDeclaration:
5        [ Javadoc ] { Modifier } class Name
6                           [ extends Type]
7                           [ implements Type { , Type } ]
8                           { { ClassBodyDeclaration | ; } }
9   InterfaceDeclaration:
10       [ Javadoc ] { Modifier } interface Identifier
11                          [ extends Type { , Type } ]
12                          { { InterfaceBodyDeclaration | ; } }
```

Listing 4.3: Grammar for Umple Classes

```
1  umpleClass : class Name { [[classContent]]* }
```

LISTING 4.4: Rule JavaTypeToUmpleClass

```
Transformation JavaTypeToUmpleClass (BaseLanguageMetamodel,
    UmpleMetamodel){
 in
   typeDeclaration : BaseLanguageModel::TypeDeclaration;
 out
   UmpleClass: UmpleMetamodel::UmpleClass;
 in conditions
   oclIsTypeOf(ClassDeclaration);
 mappings
   typeDeclaration.name -> UmpleClass.name;
}
```

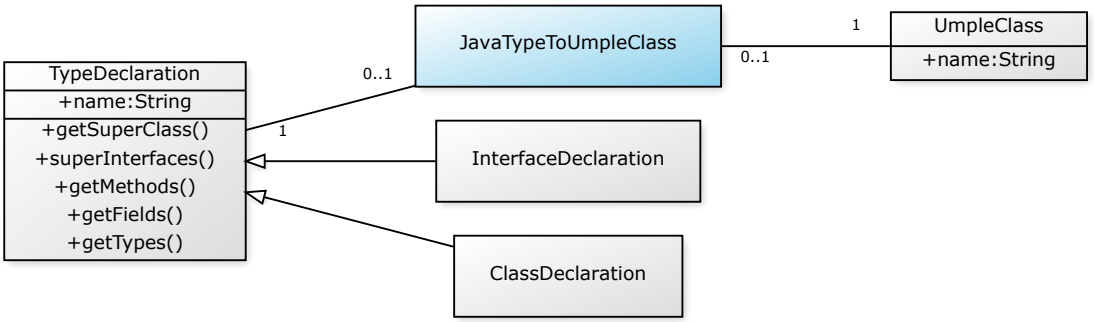

FIGURE 4.1: Input-Output relationships for rule TypeToUmpleClass

Similarly, rule *JavaImportToUmpleDepend* in Listing 4.5 specifies the transformation of a import declaration into a depend declaration and rule *JavaPackageToUmpleNamespace* in Listing 4.6 specifies the transformation between a package declaration into a namespace declaration.

LISTING 4.5: Rule JavaImportToUmpleDepend

```
Transformation JavaImportToUmpleDepend (BaseLanguageMetamodel,
    UmpleMetamodel){
 in
   importDeclaration : BaseLanguageModel::ImportDeclaration;
 out
   depend: UmpleMetamodel::Depend;
 in-out conditions
   UmpleClass:name = typeDeclaration.name ;
 mappings
   importDeclaration.name -> depend.name;
   fieldDeclaration.importDeclaration -> UmpleClass.depend;
}
```

LISTING 4.6: Rule PackageToUmpleNamespace

```
Transformation PackageToUmpleNamespace (BaseLanguageMetamodel,
    UmpleMetamodel){
 in
   packageDeclaration : BaseLanguageModel::PackageDeclaration;
   typeDeclaration : BaseLanguageModel::TypeDeclaration;
 out
   UmpleClass: UmpleMetamodel::UmpleClass;
 in-out conditions
```

```
8     UmpleClass:name = typeDeclaration.name;
9   mappings
10    packageDeclaration.name -> UmpleClass.namespace;
11  }
```

Furthermore, the generalization in the base language notation is transformed into the Umple notation *isA* as shown in Listing 4.7.

LISTING 4.7: Rule GeneralizationToUmpleIsA

```
1  Transformation GeneralizationToUmpleIsA (BaseLanguageMetamodel,
       UmpleMetamodel){
2   in
3     fieldDeclaration : BaseLanguageModel::FieldDeclaration;
4   out
5     UmpleClass: UmpleMetamodel::UmpleClass;
6   in conditions
7     (typeDeclaration.isInterface() = false);
8   mappings
9     typeDeclaration.superInterfaceTypes -> UmpleClass.parentInterfaces;
10    typeDeclaration.superClassType -> UmpleClass.extendsClass;
11 }
```

Finally, since at this transformation step, we do not attempt to transform any variable into an Umple attribute, association end or state machine. Therefore, the field declarations and related method declarations are simply appended to the body of the Umple class; some of these will be transformed later. The rule *ClassBodyToUmpleClassExtracode* in Listing 4.8 performs the desired operation. We employ the OCL feature *iterate* to traverse the collection of methods and fields belonging to the field declaration.

LISTING 4.8: Rule ClassBodyToUmpleClassExtracode

```
1  Transformation ClassBodyToUmpleClassExtracode (BaseLanguageMetamodel,
       UmpleMetamodel){
2   in
3     fieldDeclaration : BaseLanguageModel::FieldDeclaration;
4     methodDeclaration : BaseLanguageModel::MethodDeclaration;
5     typeDeclaration : BaseLanguageModel::TypeDeclaration;
6   out
7     UmpleClass: UmpleMetamodel::UmpleClass;
8   in conditions
9     (typeDeclaration.isInterface() = false);
10  mappings
11    typeDeclaration->iterate( f: FieldDeclaration |
12      fieldDeclaration.toString -> UmpleClass.extraCode);
13    methodDeclaration->iterate( f: FieldDeclaration |
14      fieldDeclaration.toString -> UmpleClass.extraCode);
15 }
```

Rules aiming at transforming field declarations into Umple interfaces are very similar to those presented above except for the inclusion of the condition '*typeDeclaration.isInterface()*'.

4.3 Member Variables Analysis

Member variables can represent not only attributes, but also associations, state machine variables, and internal data such as counters, caching, or sharing of local data. In this section, we analyze the characteristics of member variables and present the mapping rules guiding the transformation of these member variables into attributes, associations or state machines variables. Furthermore, we analyze the different patterns supported by existing reverse engineering tools when it comes to the detection of these UML/Umple constructs. We demonstrate our reverse engineering patterns for attributes, associations and state machines variables using Java as the input language.

4.3.1 Refactoring to Create Attributes

We start by analyzing all instance variables for their presence in constructor and get/set methods and decide whether the member variable is a good candidate to become an Umple attribute [12]. In Table 4.1, we present the developed (programmable) heuristics used for the partial analysis of member variables. The instance variables with a low or very low probability of being attributes are ignored for now. Those with high and medium probability are further analyzed.

TABLE 4.1: Analyzing instance variables for presence in the constructor and getter/setters

Constructor	**Setter**	**Getter**	**Attribute (Probability)**
Yes	Yes	Yes	High
Yes	Yes	No	Low
Yes	No	Yes	High
Yes	No	No	Low
No	Yes	Yes	High
No	Yes	No	Low
No	No	Yes	Medium
No	No	No	Very Low

Let us now illustrate this refactoring through an example. Assume that we have already trans-formed the Java class into an Umple class, so the input at this point is an Umple file containing Java. In this example code we first analyze the member variables to determine the following: Is the field present in the parameters of the constructor?

1. Is the field present in the parameters of the constructor?

TABLE 4.2: Umple Primitive Data Types

Type	Description
Integer	Includes signed and unsigned integers.
String	All string and string builder types
Boolean	true/false types
Double	All decimal object types
Date/Time	All date, time and calendar object types.

2. Does the field possess a getter?

3. Does the field possess a setter?

4. Is the field's type, a primitive type?

The results of this analysis allow us to generate Umple code with the required types and stereotypes. For example the stereotype 'lazy'.

4.3.2 Refactoring to Create Associations

In this sub-section, we discuss how the umplification technique infers associations from source code (Transformation 2b). More specifically, we discuss how our technique infers all the fields that represent associations including the role name, association ends, multiplicities and directionality.

In the Umplificator, the tool we will describe in the next section, these conditions are expressed as rules. The transformation of variables into associations involves a considerable number of transformations and code manipulations. In order to guarantee the correct extraction of an association and to avoid false-negative cases, we consider not only the getter and setter of the fields but also the iteration call sequences (iterators). Table 4.3 and Table 4.4 present the list of methods considered (parsed and analyzed) in order to infer associations. These methods can be categorized as mutator and accessor methods. In the tables, W is the name of the class at the other end of the association and '' refers to a collection of elements. We have considered those collections of elements defined using Map, Set, List and Hash classes (from the Java collections framework or the Standard Template Library in C++).

A simple example is presented now to summarize the main idea behind this transformation step. Assume that Umple code shown below has already passed through the two

TABLE 4.3: Accessor Methods parsed and analyzed

Method Signature	**Description**
W getW()	Returns the W
W getW(index)	Picks a specific linked W
List<W>getWs()	Returns immutable list of links

TABLE 4.4: Mutator methods parsed and analyzed

Method Signature	**Description**
boolean setW(W)	Adds a link to existing W
W addW(args)	Constructs a new W and adds link
boolean addW(W)	Adds a link to existing W
boolean setWs(W)	Adds a set of links
boolean removeW(W)	Removes link to W if possible

first refactoring steps. As a result, classes, dependencies, and attributes (if any) have been properly extracted.

4.3.3 Refactoring to Create State Machines

Chapter 5

The Umplificator Technologies

In this chapter, we provide an overview of the tool we have developed to support umplification; as well as discuss some of its technical details including its architecture and a detailed description of the Rule-Engine component. We also present the various design decisions we made as well as the alternatives implementations we attempted during the initial stages of our work.

5.1 The Umplification tool support goals

In this section, we state what are the desirable aspects for a tool supporting the Umplification process.

Our objective is to create an accurate tool that can enable developers to efficiently recover the model from existing software systems written in an object-oriented programming language. The Umplificator should provide extensible mechanisms to create and define transformation rules. In fact, the most important goal for a successful reverse engineering environment is that it must provide an extensible toolset [41]. The extensibility should be present in all the different operations of the tool such as parsing the input source code, transforming the source code and presenting the information. The end-user should be able to provide their own tools for these activities or to extend the ones already provided. The high-level **general** and **specific** requirements for the tool are presented below. General requirements are the ones that every reverse engineering

tool should possess and the specific requirements are the ones additionally required to implement the Umplification process (which may differ from other approaches).

General Requirements

A reverse engineering tool generally performs operations to gather information from a software system, organizes the information and presents it in manner such that software engineers can better understand the system. In the literature explored in Chapter 7 most of of the tools exhibit a layered architecture with a parser, analyzer and (XMI, XML) code generator as common components.

The general requirements for our specific tool are presented below with an emphasis on the component involved.

- The tool must be able to **parse** any of the most popular Object-oriented programming languages.
- The tool must be able to handle of the different idioms and programming conventions of those programming languages (parser and analyzer).
- The tool should be able to **export** the output in Umple.
- The tool must offer both GUI and command-line capabilities. Command line capabilities are needed for automated testing, and scripting and for back-ends that permit deployment of the tool on the Web.
- The tool should support incremental updates of the target model. This is required for large models as the target model does not need to be regenerated completely after each transformation.

From the developer's perspective:

- The tool should be easy to debug. We should be able to quickly identify the location of an error and fix it.
- The mapping rules should be as general and extensible as possible.

5.2 Alternative Approaches Studied

We have explored two different and famous model transformation technologies with the purpose of umplifying a software system: TXL [31] and ATL [27]. In the following two sub-sections we present the mapping rules, grammar and program directives that allowed us to transform a Java Program into Umple.

5.2.1 TXL

TXL is "a programming and rule-based language and rapid prototype system designed for implementing source transformation tasks" [31].

The TXL paradigm consists of parsing the input text into a tree according to a specified grammar, transforming the tree to create a new output parse tree and processing the new tree to finally produce the output text. In TXL, grammars and transformation rules are specified in the TXL programming language. The TXL processor is responsible for interpreting both the grammar and mapping rules by using an internal tree-structured bytecode. TXL programs depend on no other tools or technologies and can run on any platform directly from the command line.

TXL programs are composed of a *base grammar*, which specifies the syntactic forms of the input structure, a set of *grammar overrides*, which extend the grammar to be used and a set of *transformation rules and functions*, that specify how the input structure will be transformed to produce the desire output structure.

The *grammar* in TXL is a set of recursive rewriting rules used to generate patterns of strings. A grammar in TXL is used to specify how the input is partitioned into tokens of the input language and how the sequences of input tokens are grouped into structured types of the program.

The*mapping rules and functions* specify how to transform the input text into the desired output. The mapping rules are specified using pattern and replacement pairs:

```
LeftHSPattern -> RightHSPattern IF Condition
```

Where *LeftHSPattern* and *RightHSPattern* are term patterns. The result of a mapping rule is the instantiation of the *RightHSPattern* and is produced when the term matches the *LeftHS_Pattern* and the condition is true. Rules are applied recursively until they fail. Functions are similar to Rules but they are applied once on the entire function input.

TXL has been used widely in software engineering tasks and other areas including database migrations and artificial intelligence. We present our experiment in building a *Java-to-Umple* transformer using TXL. We first studied the similarities and differences between Java and Umple and classified the necessary transformations for converting Java programs to Umple into three categories.

The first category represents the direct transformations where one-to-one mapping between the two languages exists and some rules for minor adaptations are required. For instance, a Java class declaration can be written as:

```
ClassModifier class Identifier TypeParameter Super Interfaces ClassBody
```

In this, the *ClassModifiers* are used to control the access to members of a class, the Identifier specifies the name of a class, the optional *TypeParameter* are used when the class is generic and declares one or more type variables, the Super clause specifies the direct superclasses of the current class, and the Interfaces clause specifies the name of the interfaces that are direct super-interfaces of the class being declared.

Very similarly, an Umple class is defined as: class Identifier *ClassBody*. In this case we will need a mapping rule matching the Identifier and class keyword in the Java program to produce the desired output, the Umple class.

The second category corresponds to the *indirect transformations* where some special functions are needed to map a Java construct to an Umple one. For example, a Java instance variable can be mapped to an Umple attribute, an Umple Association or an Umple state machine. This kind of transformations requires helper and additional functions in the TXL program.

5.2.1.1 Java to Umple Implementation

In this section, we describe the design process. Next, we describe the implementation of the *JavaToUmple* program that partially converts Java code to Umple. Lastly, we provide examples of transformations rules in the TXL language. Figure 5.1 presents the components of the TXL *JavaToUmple* program.

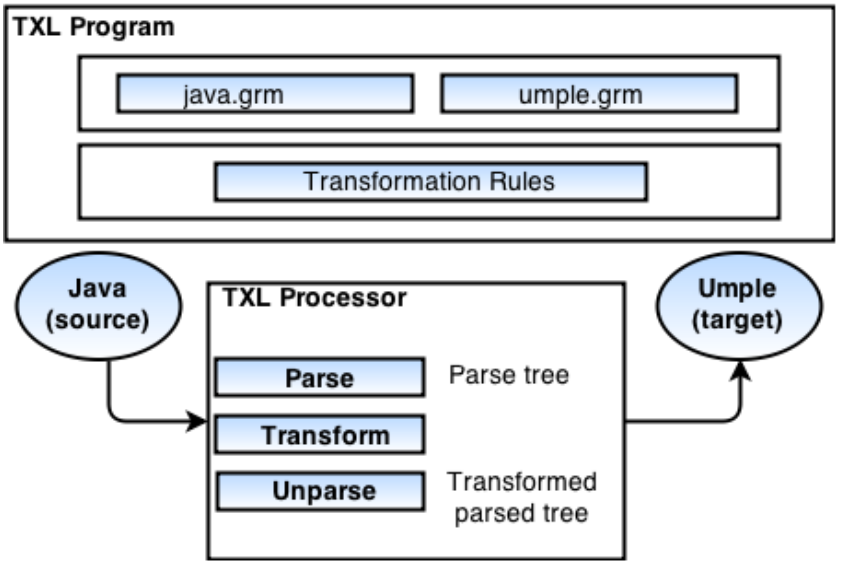

FIGURE 5.1: TXL Program for transforming Java to Umple

5.2.1.2 Design process of the TXL Program

The first step in writing a source transformer is writing working grammars for both the target and the source language and then writing a union grammar that accepts constructs for both languages. A grammar for Java 1.5 is available from the TXL website [42]. We wrote the grammar for Umple in EBNF format required by the transformation engine. We then built the TXL rules and functions grouped in modules. Each module targets conversion of one specific language construct of Java to the equivalent in Umple and is stored in a separate file. The overall structure of the transformer is shown in Figure 5.2. It contains the modules for the different language constructs and the main program that starts the program. Below, we briefly describe the different modules:

- JavaToUmple.Txl: This is the main program. It is used by TXL to match an input Java program against the Java Grammar and to call the transformation rules.

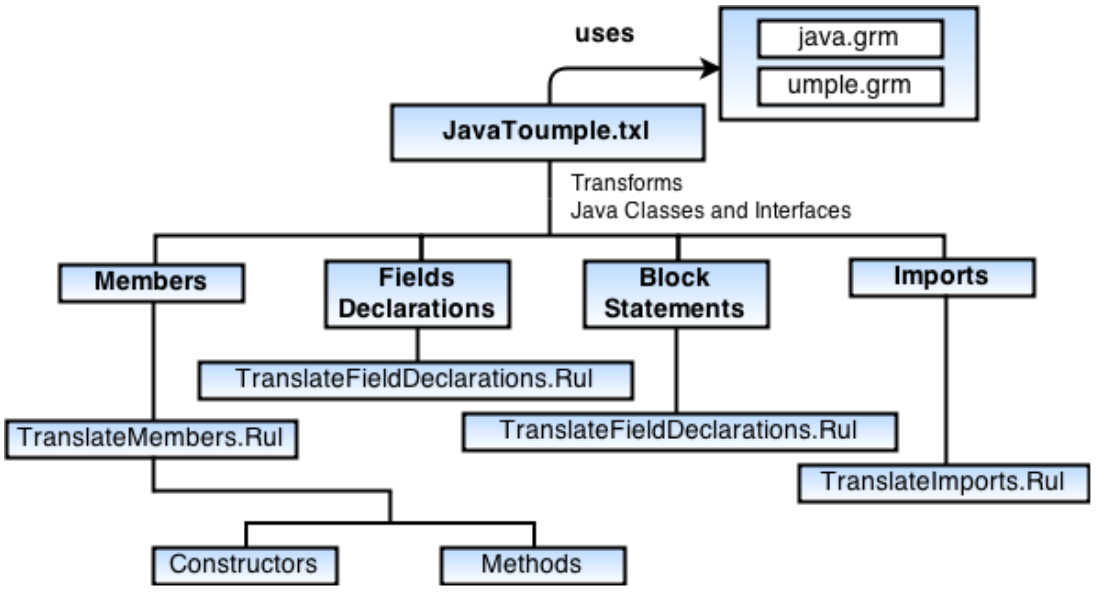

FIGURE 5.2: Structure of the JavaToUmple program

- TranslateMembers.Rul: Contains rules and functions to transform nested declarations.
- TranslateFieldDeclarations.Rul: Contains rules and functions to transform field declarations.
- TranslateBlockStatements.Rul: Contains rules and functions for matching bodies of code belonging to constructors and methods.
- TranslateImports.Rul: Contains rules for matching Java imports.
- TranslateConstructors.Rul: transforms the Java constructors.
- TranslateMethods.Rul: transforms Java Methods.

The original Java source code remains untouched after applying the transformation. A set of one or more Umple files is produced as a result of the transformation. The **JavaToUmple** program can be invoked using the command:

```
< txl    o  outputFileName.ump inputFileName.Java JavaToUmple.txl >
```

In the following sub-section we provide some transformation examples. We first show the Java and Umple grammar for the single constructs we transform as well as the TXL transformations rules that guide the transformation.

5.2.1.3 Transformation 1: Transforming the Class Header

In order to transform a Java class into an Umple class, we need to first transform the class header. The code excerpt in Listing 5.1 below shows the EBNF grammar for class definitions in both Java and Umple languages. An example of class definitions is also provided in Listing 5.2.

Listing 5.1: "Class definition grammar in BNF form"

```
JavaClassDeclaration:
   ClassModifiers? class Identifier Super? Interfaces? ClassBody


UmpleClassDeclaration:
   class Identifier ClassBody  ClassBody: '{' ClassContents '}'
```

Listing 5.2: Class definitions in Java and Umple

```
// In Java:
public class A extends X implements Z {
    //  some contentW
}
// In Umple:
class A
{
 //.. some content
}
```

The mapping rule called '*changeClassHeader*' in file *TranslateMembers.Rul* that transforms class headers of a Java class is presented below in Listing 5.3. In order to transform the class header from Java to Umple, we need to deconstruct the class header (Line 4) of a Java class and take only what is required in an Umple header, the identifier of the class. The modifiers of the class are discarded and the extends and implements clauses are ignored at this moment, they are analyzed and transformed in subsequent steps of the program transformation.

Listing 5.3: TXL Mapping rule for transforming the class headers

```
rule  changeClassHeader
   replace $[class_header]
        ClassHead[class_header]
        deconstruct ClassHead
        modifiers[repeat modifier] 'class Name[class_name]
        ExtendClause[opt extends_clause]
        ImplmntClause[opt implements_clause]
    by   'class Name
end rule
```

5.2.1.4 Transformation 2: Transforming the package

A **package** in Java can be defined as a grouping of related classes (and types). In Umple a **namespace** allows to group Umple classes. Listings 5.4 and 5.5 show the EBNF grammar of package definition in both languages and an example.

Listing 5.4: Java Package

```
PackageDeclaration:
   package PackageName;

package aPackageName;
```

Listing 5.5: Umple Namespace

```
PackageDeclaration:
   namespace NamespaceName;

namespace aNamespaceName;
```

The mapping rule called '*changePackageToNamespace*' that transforms package declarations is presented below:

Listing 5.6: TXL mapping rule for the transformation of the package declaration

```
rule changePackageToNamespace
    replace [opt package_header]
            'package Name [package_name] ';
    by
            'namespace Name ';
end rule
```

5.2.1.5 Transformation 3: Transforming the imports

An import declaration in Java allows a named type or a group of named types to be referred to. The '*Depends*' construct in Umple is similar to this.

Listing 5.7: Java Import

```
ImportDeclaration:
     import  QualifiedName;


import java.io.StreamReader;
public class A {
 //
}
```

Listing 5.8: Umple Depend

```
DependDeclaration:
   depend QualifiedName;


class A {
  depend java.io.StreamReader;
}
```

The mapping rule called '*changeImportToDepend*' in file *TranslateImports.Rul* that transforms import declarations is presented below:

Listing 5.9: TXL mapping rule for the transformation of the import declaration

```
changeImportToDepend
    replace [repeat import_declaration]
           'import Name [imported_name] ';
```

```
    by      'depend Name ';
end rule
```

As seen in the example, the depend declarations appear inside the Umple class, so we need additional rules to remove them from the top of the Java class and place them in the right place prior the generation of the Umple code. The rule below removes all the import declarations. The main program, presented next, illustrates how the program executes the mapping rules in order to produce the output. Note that in TXL the input program is not modified since the transformation only occurs on the parse tree of the input program.

LISTING 5.10: Helper Function used to remove the imports declarations

```
function removeImports
    replace * [package_declaration]
        PkgHead [opt package_header]
        ImpDecl [repeat import_declaration]
        TypeDecl [repeat type_declaration]
    by
        PkgHead      TypeDecl
end function
```

5.2.1.6 Final transformation: The main program

The main program in Listing 5.11 is used to execute the three mapping rules presented in the examples above; it calls one by one the rules and the functions and generates the output. Additionally, the main program links, via inclusion constructs, the grammars from the target and source languages (Line 1-2). In the **JavaToUmple** program we use two grammar files to map Java and Umple constructs: *Java.GRM* and *Umple.GRM*.

LISTING 5.11: The ATL main program - JavaToUmple.Txl

```
include "java.Grm"
include "Umple.Grm"

function main
    replace [program]
        P [program]
    by  P [javaToUmple]
end function

function javaToUmple
    replace [program]
         P [program]
    by
        P
        [changePackageToNamespace]
        [changeImportToDepend]
```

```
        [removeImports]
        [changeClassHeader]
end function
% ****  MAPPING RULES HERE   ****
```

The transformation program above uses the two grammar files to map Java and Umple constructs: java.GRM and Umple.GRM. The program rules have been modularized for a better understanding as has been shown in Figure 5.2.

5.2.2 ATL

ATL (ATL Transformation Language) [27] is a model transformation language that provides ways to produce a set of target models from a set of source models and allows users to define model-to-model transformations in both a declarative and imperative way.

ATL has been developed in Eclipse as a set of plug-ins by the Institut National de Recherche en Informatique et en Automatique (INRIA) as an answer to the Object Management Group's QVT language request for proposals [43]. The ATL environment in Eclipse offers an ATL editor with syntax highlighting and code completion capabilities, a debugger and a profiler that aims to ease the development and testing of model transformations.

In this section, we describe how queries, views and transformations are handled in ATL. Additionally, we explore the ATL transformations required to umplify a Java system. Figure 5.3 presents the necessary components to implement and ATL transformation between Java and Umple. An ATL program (*JavaToUmple.atl* in the Figure) takes model *Java.xmi* as input and produces model *Umple.xmi* as output. Both models need to be expressed in the OMG XMI standard [44]. The Java model conforms to metamodel *Java.ecore* and the Umple model to metamodel *Umple.ecore*. The ecore [45] notation is a simple metamodel specification language. The ATL program *JavaToUmple.atl* is also a model, so it conforms to a metamodel (the ATL metamodel). As we will see in Section 5.2.2.3, the program is composed of a header, a set of helper functions and a set of (transformation) rules.

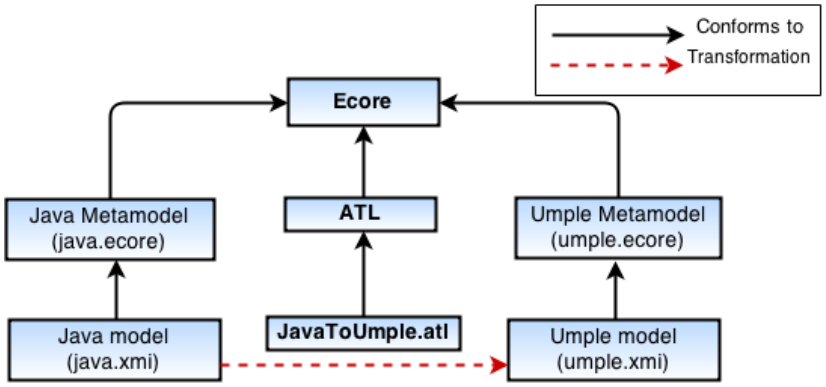

FIGURE 5.3: The JavaToUmple ATL program

5.2.2.1 The basics of ATL

The ATL language is composed of expressions to query model elements (queries), views to handle incremental transformations and transformation rules to direct the transformations of a set of source models to a set of target models.

Queries

A query in ATL is an expression allowing one to search and return model elements from a model defined in an OMG-compliant format. A query is an OCL expression that can return primitive values, model elements or a combination of these. A query can not alter the source model. It is possible to navigate across model elements and call query operations on these. For instance, when the following query is executed on a Java model, it first gets the set of all existing JavaElement classes in the model and gets the size of the computed set. The computed integer value is cast into a string before being written into the file 'metrics.txt'.

```
query JavaElementNb  =
  JavaModel!JavaElement.allInstances()->size().toString()
      .writeTo('metrics.txt')
```

View Views in the ATL world are a special case of transformation. Views offer support for incremental transformations. The user can query a model; perform a transformation on a subset of the source model and save results on a view. Then, she can update the view from its source without executing the whole transformation again.

Transformation Rules There are different kinds of rules in ATL based on the way they are called and how they specify the results: matched rules, lazy rules and called rules [46].

- **Matched Rules**: This kind of rule specifies which source element is to be matched, along with the target element that is to be produced.
- **Lazy Rules**: This kind of rule is similar to a matched rule, but it is not executed when matched; they rely on being called by other rules.
- **Called Rules:** This kind of rule can have parameters and can be called only from blocks of imperative code. Assignments, 'for' and 'if' statements are the only three types of (imperative) statements supported in ATL.

5.2.2.2 ATL Tool Support —Eclipse M2M

The ATL project is composed of four parts (or four different plug-ins in Eclipse). The Core, Compiler, Parser and the Virtual Machine (VM) [43], which are described below:

- **Core** - Contains the classes used to internally represent a model, to allow the creation of models and metamodels, to save and load models and to supply ways to launch the model transformations.
- **Compiler** - Uses the ACG (ATL VM code generator) domain-specific language to compile and generate code.
- **Parser** -Contains all classes to parse an ATL transformation input and to generate an output model compliant with the target metamodel.
- **VM** - A byte-code interpreter.

5.2.2.3 Transformations Examples with ATL

In this section we provide some transformation examples in ATL. We present parts of the metamodels, models, mapping rules and the final results (Umple code) of some Java to Umple ATL model transformations. The JavaModel to UmpleModel examples describe a transformation from a simplified Java Model to an Umple model.

Metamodels

The source metamodel of Java in Figure 5.4 consists principally of *JavaElements* which all have a name. A *JavaClass* has Methods and Fields and belongs to a package. *Methods*, *Fields* and *JavaClasses* are subclasses of the class Modifier and indicate whether they are public, static or final. Java classes and methods declare with the isAbstract attribute whether they are abstract or not. Fields and methods have also a Type. The Java metamodel in Figure 6 has been fully described by the Java Specification [47] and has been simplified for the purpose of this transformation example.

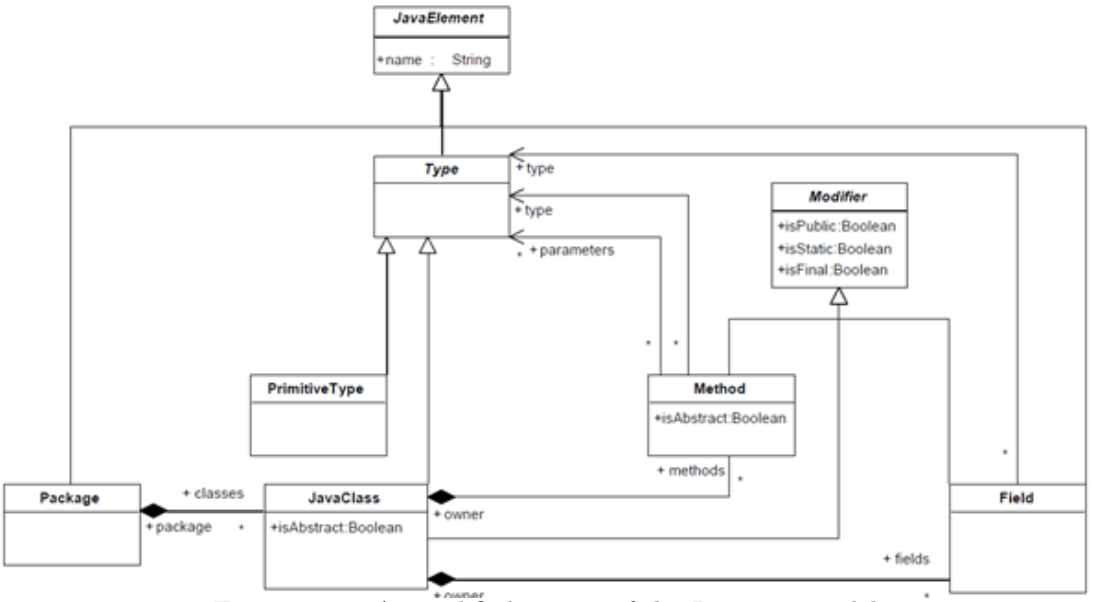

FIGURE 5.4: A simplified version of the Java metamodel

A simplified version of the Umple metamodel (target metamodel) is presented in Figure 5.5. The complete metamodel for Umple can be found at [13].

Both metamodels have been defined in XMI format, as required by the ATL metamodel loader.

Transformation rules

These are the rules to transform a Java Model to an Umple model. The ATL code for the transformation, shown in Listing 5.12 consists of several functions and rules. Among the functions, we can mention the *getExtendedName* in Lines 4-8 which recursively explores the namespace to concatenate a full path name.

LISTING 5.12: ATL Transformations rules

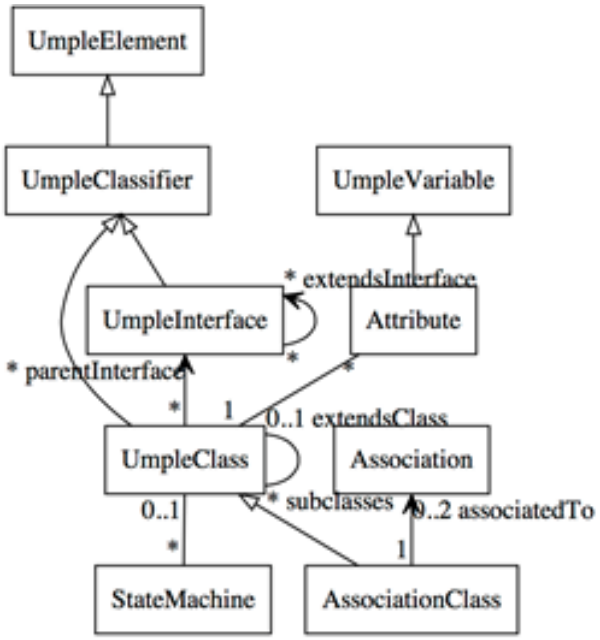

FIGURE 5.5: A simplified version of the Umple metamodel

```
module JavaToUmple;
create OUT: Umple from IN: Java;

helper context Java!Namespace def: getExtendedName() : String =
 if self.namespace.oclIsUndefined() then ''
 else if self.namespace.oclIsKindOf(UML!JavaModel) then  ''
 else self.namespace.getExtendedName() + '.'
endif endif + self.name;

rule P2P {
 from j :Java!Package (e.oclIsTypeOf(Java!Package))
 to out : Umple!Namespace (
     name <- j.getExtendedName())
}

rule C2C {
 from j : Java!JavaClass
 to out : Umple!UmpleClass (
   name <- j.name,
   isAbstract <- j.isAbstract,
   // .. parts ignored
 )
}

rule F2A {
 from j : Java!Field to out : Umple!UmpleAttribute (
 name <- j.name,
 value <-  FieldHelper.getValue(j)
 isConstant <- FieldHelper.isContant(j),
 isImmutable <- FieldHelper.isImmutable(j),
 isLazy <- FieldHelper.isLazy(j),
 )
}
```

The three rules presented above are part of the set of rules required to transform a Java Model to an Umple model. The first rule '*P2P*' in Lines 10-14 specifies how to map a Java package to an Umple namespace. The second rule '*C2C*' in Lines 16-23 declares how we can match a Java Class to an Umple Class. The last rule '*F2A*' aims at transforming a Java Field to an Umple Attribute. This rule is a *called rule* as it is just called whenever a Java Field matches an Umple Attribute. Remember that a Java Field can match an Attribute, Association or State Machine in Umple.

The *FieldHelper* (Lines 29-33) used in rule *F2A* is an utility class used to determine certain properties of a Java field that can derive into properties of a Umple attribute. For instance the *FieldHelper.isLazy(aJavaField)* returns *true* if the java field passed as parameter is not one of the constructor parameters of its parent class. This helper class is also used to compute components of a Java Field not having a one-to-one match to an Umple class. The (static) method *FieldHelper.getValue(aJavaField)* extracts the value of a field (if any).

5.3 Discussion

5.4 The Umplificator

In this section, we provide a detailed description of the tool we have developed to support umplification; as well as discuss some of its technical details.

Our tool called, Umplificator, takes as input a set of files containing classes written in base language code (Java, C++ etc.), Umple files, source code directories or software projects (source code containers as represented in many popular IDEs such as Eclipse). The output is an Umple textual model containing base language code with modeling abstractions.

At its core, the Umplificator is a language interpreter and static analyzer that parses base language and Umple code/models, populates a concrete syntax graph of the code/-model in memory (*JavaModel*, *CPPModel*), performs model transformation on the base language representation in memory and then outputs Umple textual models.

The Umplificator relies on initial parsing by tools such as the Java Development Tool (JDT) for Java, CDT for C++, and PDT for PHP. These extract the input model from base language code. The use of JDT and its siblings reduces the need to write an intermediate parser for the base language.

The base language model is then transformed in a series of steps into an Umple model. To do this, the Umplificator uses a predefined set of refactoring rules written in the Drools rule language [48]. Drools is a rule management system with a forward- and backward-chaining rules engine.

The Umplificator includes other subsidiary and internal tools such as:

- Language validators A set of base language validators allowing validation of the base language code that is generated after compilation of the recovered Umple models.
- Umplificator statistics A metrics-gathering tool to analyze certain aspects of a software system such as the number of classes and interfaces, the number of variables present in the code, the cyclomatic complexity, the number of lines of code [49].
- Umplificator Workflow A tool that guides the umplification process within Eclipse.

The development of the Umplificator follows a test-driven approach to provide confidence that future enhancements will not regress previously functioning and tested aspect of the system. Test-driven testing for the Umplificator is discussed in section 6.1.

5.4.1 Architecture

The Umplificator has a layered and pipelined software architecture. The pipelines (components) in this architectural style are arranged so that the output of each element is the input of the next. Figure 5.7 presents the architecture which is comprised of four components. The parser, model extractor, transformer and generator components are explained in the following sub-sections.

The process of umplifying am object-oriented software system in this architecture is described below an illustrated in Figure 5.6.

1. The input is a set of source code files in the base language and/or Umple.
2. (Parser) The source code is parsed.
3. (Model Transformer) The source code is transformed into base-a model of the base language and Umple constructs.
4. (Transformer) The model previously obtained is entered into the next stage of the pipeline. The input model is transformed a model with additional Umple features using pre-defined mapping rules.
5. The target Umple model, is then validated.
6. (Generator)Finally, Umple code (.ump files) are generated from the Umple model.

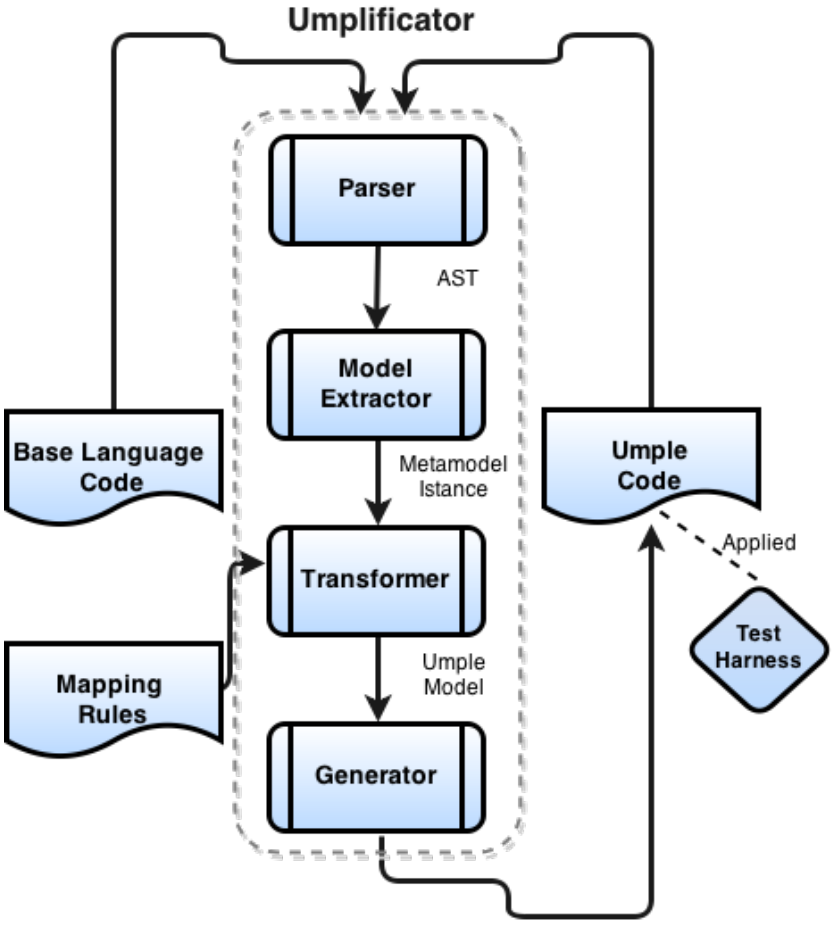

FIGURE 5.6: The umplification process flow

The Umplificator employs the libraries and technologies summarized in Table 5.1 to implement its reverse engineering capabilities. The dependencies between the external and internal components of the Umplificator is shown in Figure 5.7, where our *Parser*

and *ModelExtractor* components uses the JDT/CDT/PDT projects and the *Transformer* the Drools Rule Engine.

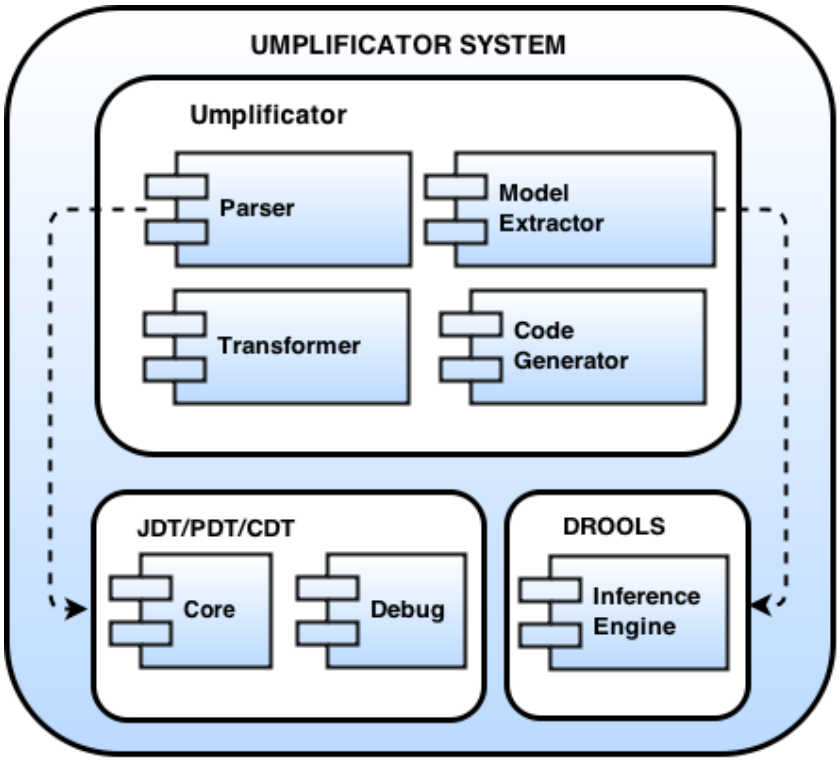

FIGURE 5.7: The Umplificator components

The Table also shows the Umplificator component using the technology. Note that if the technology is used in more than one component, we mark it as 'General'.

TABLE 5.1: Third Party Technologies employed in the Umplificator tool

Technology	**Targeted component(s)**	**Description**
JDT/CDT/PDT	Parser and Model Extractor	APIs for parsing object-oriented source code.
Drools Rule Engine	Transformer	A Rule Engine for creating and managing the mapping rules used in the Umplificator.
JOpt Simple	General	Library for parsing command line options
Log4j	General	A logging library used to collect (reverse-engineering) process data.
Perf4j	General	Set of utilities for calculating and displaying performance statistics in the Umplificator code.

The different components of the Umplificator as well as the third-party technologies employed are discussed next.

5.4.2 Parser and Model Extractor

The parser component receives a set of source code files in the base language and/or Umple and creates an abstract syntax tree (AST) as representation of the code. Umple code is allowed as input to allow repeated application to refine the model. To implement its parsing and base language model extraction capabilities, the Umplificator uses various Eclipse Projects, as summarized in the following table. These projects provide APIs to access and manipulate object-oriented source code. They also provide access to the source code via two different means: a base language model within the Eclipse Workspace and an Abstract Syntax Tree (AST) for a standalone usage (outside the Eclipse IDE). Table 5.2 summarizes the Eclipse projects used in the Umplificator for **parsing** purposes. We then provide some details about the capabilities and usage of each project.

TABLE 5.2: Eclipse projects used in the Umplificator

Project	**Targeted programming language**	**Components used (plug–ins)**
Java Development Tooling	Java	org.eclipse.jdt.core , org.eclipse.jdt.core.dom
C++ Development Tooling	C++	org.eclipse.cdt.core
PHP Development Tools	PHP	org.eclipse.pdt.core

Eclipse is not simply a programming language IDE. In fact, Eclipse is an extensible platform for building IDEs. Eclipse functionality is wrapped into pluggable components called *plug-ins*. These plug-ins allow developers to extend the basic functionality offered by Eclipse. The projects mentioned in the above table, are plug-ins that can be used in other projects inside Eclipse or as a standalone component, as in our case.

Architecturally, the JDT/CDT/PDT projects are divided into two domains: the model (core) and the user interface. The model is a representation of the Base language elements; the user interface is a set of views, actions, perspectives and menus that work together. The user interface domain can be extended but only works inside Eclipse (not intended for standalone usage). The Umplificator uses the model component of these projects to *parse* and *extract* a base language model from source code.

Java Development Tooling (JDT)

Eclipse Java Development Tooling (JDT) [50] offers a comprehensive Java development environment. JDT also provides APIs for analyzing Java source code. It provides several levels of source code analysis that can be reused. The level of source code analysis used in the Umplificator is the Abstract Syntax Tree (AST) framework. We use the AST to analyze the Java source code as a tree of nodes, where each node represents a part of the source code (for instance a variable declaration, a method body, a contructor and so on). The AST framework defines over 60 *ASTNode* [51] subclasses representing the different elements of the Java language.

The AST framework includes also interfaces that help retrieve specific source information beyond what is indicated by the ASTNode source pointers. To traverse the nodes returned by the parser (ASTParser) and collect the desired information about the source code, we employ multiple visitor classes that follows the Visitor software design pattern [22]. The visitor pattern is a standard way to decouple the data from the operations that process the data. For each different AST node type T, two methods are offered:

- *public boolean visit(T node)* – Visits the given node to perform some arbitrary operation. If true is returned, the given node's child nodes will be visited next;
- *public void endVisit(T node)* – This method is called after all of the given node's children have been visited (or immediately, if visit returned false). The default implementation provided by this class does nothing;

Generally, the AST visitor can be used to **transform** AST nodes or to **derive** information. A derivation collects information and stores result along the way. For instance, if our intention is just to collect the import declarations of a Java class, we could write a visitor as in Listing 5.13. In the method visit(...) we return false to stop the visitor from visiting child nodes of the import declaration. The variable *importDeclarations* is an array containing the (visited) import declarations.

Listing 5.13: A visitor for Import declarations in Java source code

```
1
2 public class SimpleVisitor extends ASTVisitor{
3
4  private List<ImportDeclaration> importDeclarations;
5
```

```
6  public boolean visit(ImportDeclaration node) {
7      importDeclarations.add(node);
8      return false;
9  }
10 }
```

As an example, consider the code of class '*Test*' in Listing 5.14. Once the code is parsed, we used a visitor to collect the desired information. Table 5.3 presents the resulting AST node types, the corresponding source fragment and the visitor employed to collect the information. This Table recapitulates the entire process of parsing and extracting the model for our sample code.

LISTING 5.14: Test.java

```
1  package umplificatorTest;
2
3  import java.util.Date;
4
5  public class Test {
6   public int number;
7
8   public int  getNumber() {
9    return number;
10  }
11 }
```

TABLE 5.3: Sample Uses of an AST for Code Analysis

ASTNode Type	Source Fragment	Visitor Code
CompilationUnit	Entire source code	visit(CompilationUnit cu)
PackageDeclaration	"package umplificatorTest"	visit(PackageDeclaration pd)
TypeDeclaration	"public class Test"	visit(TypeDeclaration td)
FieldDeclaration PrimitiveType("int") SimpleName("number")	"public String name"	visit(FieldDeclaration fd) td.getType() td.getSimpleName()
MethodDeclaration PrimitiveType("int") SimpleName("getNumber")	"public int getNumber()"	visit(MethodDeclaration md) td.getReturnType() td.getName()
Block **ReturnStatement**	".." "return number;"	bl= md.getBody() stmt = b1.getStatements(0);

Note that the AST node type CompilationUnit is the type root of an AST (first row of above table) and the object returned by the ASTParser after completion of the parsing a Java file. The source range for the CompilationUnit type node is the entire source

file, including leading and trailing whitespace and comments. In Java 1.4 to 1.7, a CompilationUnit is composed of a *PackageDeclaration*, *ImportDeclaration*, and one or more of these types: *TypeDeclaration*, *EnumDeclaration*, *AnnotationTypeDeclaration*.

The code on the right of Table 5.3 shows several examples of what can be done inside a visitor method:

- Extracting the name of the package: Code a *visit(PackageDeclaration)* and get its name as an instance of *SimpleName* (e.g. package test) or *QualifiedName* (e.g. package cruise.compiler.*).
- Getting the list of types referenced in a compilation unit? Code a *visit(TypeDeclaration)* and get theirs names as instances of *Simple* or *QualifiedName*.
- Finding all literal integers referenced only within methods and not fields? Code a 'sub' *visit(IntegerLiteral)* inside the visit method for *MethodDeclaration*.

Java Development Tooling (CDT)

In the same manner as the JDT technology and using the same concepts for parsing an model extraction, the CDT provides powerful features to analyze code. CDT contains two parsers, for C and C++, that generate an AST representation from source code. The CDT project, as we have explained for JDT, is a set of plug-ins that adds full support for parsing, analyzing and developing C/C++ applications. The Umplificator uses the core component of CDT to implement its reverse engineering capabilities (org.eclipse.cdt.core). The following are some of the CDT core features that are used in the Umplificator:

- **Preprocessor**: Converts source code text into a token stream and evaluates inclusion directives and macros. The preprocessor phase runs before the parser.
- **Parser**: Converts the token stream into an AST
- **AST**: Used to traverse and collect information about the source code (CPPModel). A visitor API is also provided.
- **AST Rewrite API**: Used to implement refactoring (method refactoring mostly).

The CDT supports the different C++ language constructs such as multiple inheritance, templates, header files, etc. The AST represents the structure of source code, as it was the case for JDT. One of the main differences of CDT and JDT is that the root object returned by the ASTParser is the '*TranslationUnit*' and not a '*CompilationUnit*'. A TranslationUnit (CDT) is assembled from multiple source files, a CompilationUnit (JDT) represents a unique Java file. For instance, the very simple in C++ printing a string in the console, when compiled produces more than 1000 lines due to inclusion of header file 'stdio.h' (<gcc -E test.c — wc -l >returns 1052).

LISTING 5.15: Simple Example in C++ - test.c

```
1 #include <stdio.h>
2 int main() {
3  printf("Hello World\n");
4 }
```

Comments are preserved in the AST and can be accessed as comment nodes.

PHP Development Tooling (PDT)

The PHP Development Tools (PDT) is a toolset intended to encompass all tools necessary to develop PHP based software. It provides the primary modules: the core, the debug and the user interface. The *core* component is, as in the previous cases, used in the Umplificator to parser and analyze PHP source code. We will not provide further detail on the PDT, since it follows the same architecture and model extraction concepts, that we have already covered in the discussion of JDT and CDT eclipse technologies.

To recapitulate this sub-section, the *parser* component of the Umplificator, leveraging various parsing technologies, parses source code, creates a AST representation of the code that is traversed by the *model extractor* to finally obtain a base language model. The base language model is then traverse using a series of visitors. The input/output relationship of the parser and model extractor components is illustrated in Figure 5.8.

5.4.3 Transformer

The core of the tool suite is the Transformer. The Transformer receives a base language model (e.g. JavaModel, CPPModel OR PHPModel) from the extractor and an empty Umple model which is then populated. In fact, the base language model is decomposed into a series of objects representing each particular piece of the source code (a package, an

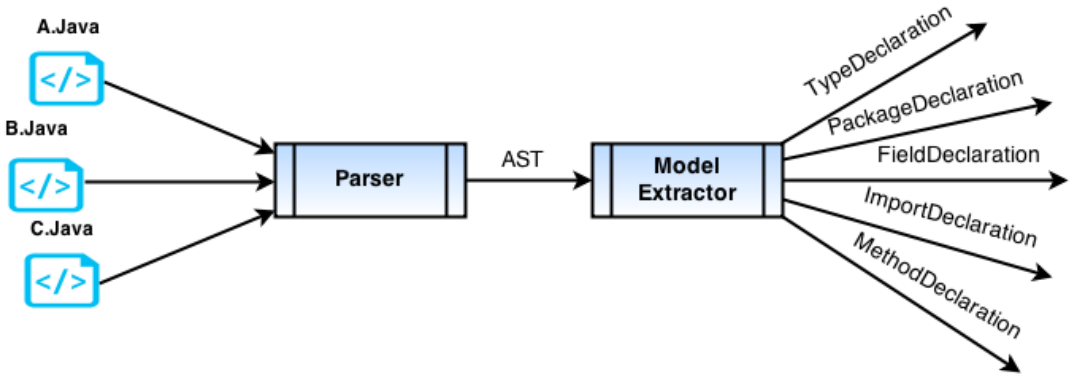

FIGURE 5.8: The Parser and Model Extractor components

import, a field and so on). Furthermore, the base-language model is transformed using a predefined set of mapping rules. If the input model is Umple code, the transformer produces an Umple model with additional modeling constructs (abstractions). The input/output relationship for the Transformer component is illustrated in Figure 5.9.

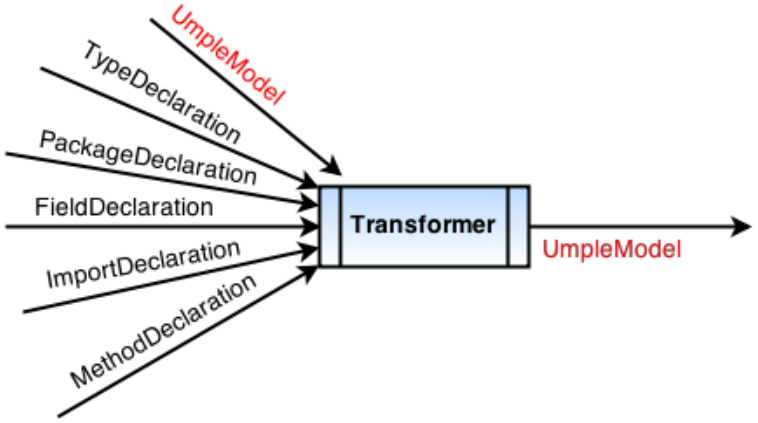

FIGURE 5.9: The Transformer component inputs and outputs

As we have seen in Table 5.1, the Transformer component leverages Drools technologies to implement its rule engine.

5.4.3.1 Drool's Rule Engine

The rule engine interprets and executes the mapping rules on the source model and target model to produce the umplified version of the target model. The Drools engine used by the Umplificator is composed of an inference engine that is able to scale to a large number of rules and facts. The inference component matches facts and data (base language models) against rules to infer conclusions, which result in actions (model

transformations). A rule is a two-part structure (Left-hand-side part and Right-hand-side part) using first order logic for reasoning over knowledge representation.

At a high level structural view, the Rule Engine consists of an: *Inference Engine*, *Agenda*, *Pattern Matcher* and a *Production* and *Working Memory*. The rules are stored in the *Production Memory* and the facts that the Inference Engine matches against are kept in the *Working Memory*. Facts are the data in which the rules act (Model elements in our case). Pattern matching is performed to match facts against rules and is implemented using the Rete algorithm [52]. Facts are evaluated into the Working Memory where they may be modified or retracted. The *Agenda* manages the execution order of the rules. Figure 5.10 shows the difference components of the Rule Engine.

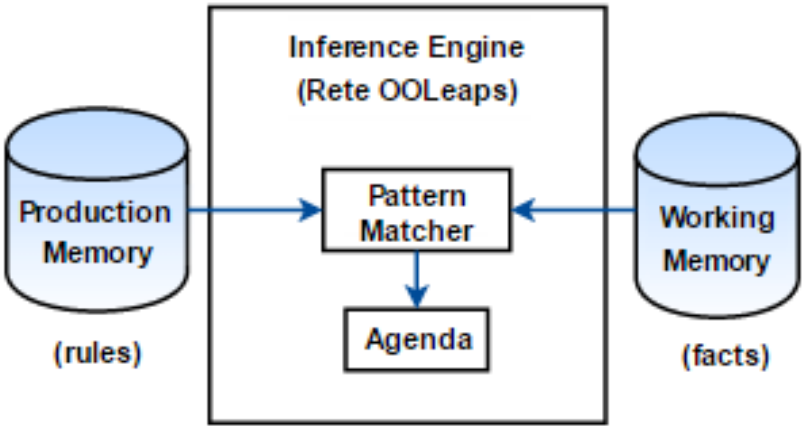

FIGURE 5.10: High Level View of the Drools Rule Engine

Traditionally, rule engines have two methods of execution [53] forward chaining and backward chaining. In forward chaining, the facts are asserted into working memory resulting in one or more rules being concurrently true and scheduled for execution. In backward chaining (goal driven), one starts with a conclusion, which the engine tries to satisfy. Drools is a Hybrid Chaining System because it implements both forward and backward mechanisms. Our Umplificator uses the forward chaining method of operation in which the inference engine starts with facts, propagates through the rules, and produces a conclusion (e.g. a transformation). Figure 5.11 contrasts the two modes of execution. In Forward Chaining, the engine discovers what conclusions can be derived from the data and asserts them (iteratively), whereas in Backward chaining the engine starts with the goals and searches how to satisfy them (as in Prolog).

Consider the scenario of a model transformation in Figure 5.11: if the conditions C1,C2 and C3 apply on a base language element, then we can perform the transformation as dictated bv D1. On the other hand, in backward chaining, we perform the transformation and then attempt to determine if it was correct based on the available information (C1,C2,C3 and input model element).

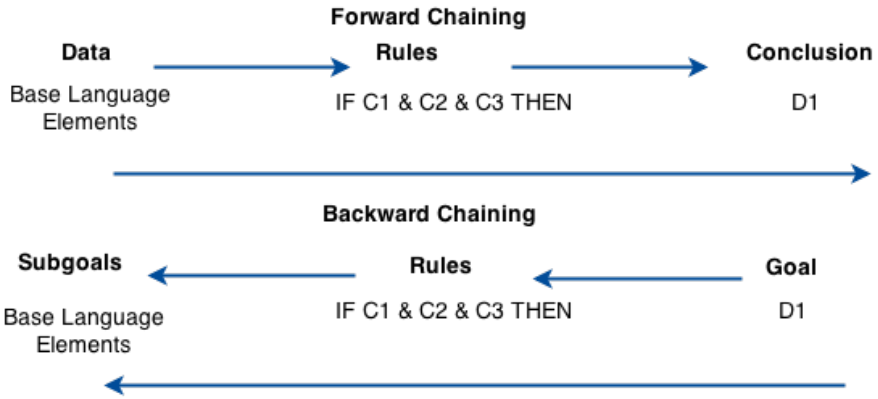

FIGURE 5.11: Forward vs Backward Chaining

5.4.3.2 The Rule Language

The rule engine is initialized with the rules. Drools offers a native rule language, very light in terms of punctuation and supporting Java and domain specific languages.

A rule file in Drools (and in our implementation) is a file with a .drl extension that can have the following elements:

- **Package**: The package name, if declared, must be the first element in the rule file and represents the namespace, which is kept unique for a given grouping of rules.
- **Imports**: These are used to import Java types referenced by the rules.
- **Global Variables:** A Global variable is a variable visible to all the rules inside a rule file. These are not inserted into the Working Memory and are most commonly used to log information on the execution of rules.
- **Functions**: These are used for invoking actions on the consequence (then) part of the rule, especially if that particular action is used over and over again.
- **Queries**: These provide a means to search working memory and store the results under a named value. In the Umplificator, they are used to gather metric

information about the models analyzed. For instance, the query numberOfPublicMethods(..) returns the number of methods having 'public' as modifier. Queries do not have side effects, meaning that their evaluation cannot alter the state of the corresponding executing unit.

The rules as explained in this section are instructions indicating how a piece of the Base language model (Java Model, C++ model, etc.) is mapped to a piece of an Umple model. In the Umplificator, the logic used for model transformations resides in the rules. Moreover, by using rules, we have a single point of truth, a centralized repository of knowledge. Rules can be also read and understood easily, so they can also serve as documentation.

Listing 5.16 shows the basic form of a rule in Drools language, where LHS is the conditional part of the rule and RHS is a block that allows dialect-specific semantic code to be executed. Attributes (Line 2) provides a declarative way to influence the behavior of the rule. We present the rule attributes used in our mapping rules in Table 5.4.

Listing 5.16: Basic rule in Drools

```
1 rule "name"
2   attributes
3   when LHS then RHS
4 end
```

Table 5.4: Rule attributes

Attribute Name	Description
no loop	Avoids infinite loops. When a rule's consequence modifies a fact it may cause the rule to activate again, causing an infinite loop; its default value is false.
lock-on-active	Stronger version of no-loop. If a rule declares this attribute, the rule can be activated once.
Salience	Salience is a form of priority where rules with higher salience values are given higher priority when ordered in the Activation queue; its default value is 0.
agenda-group	Agenda groups allow the user to partition the Agenda providing more execution control. Only rules in the agenda group that has acquired the focus are allowed to fire.

Order of Execution and Grouping

The rules are grouped in files for each of the cases (levels of refactoring) discussed earlier. In other words, there is a rule file containing rules, functions and queries to transform classes, namespace and imports; another file containing those to transform variables into

attributes, another file containing those to transform variables into associations and so on.

To activate the groups on the required order, we used agenda groups. Agenda groups are a way to partition the activation. At any one time, only one group has 'focus' meaning that activation for rules in that group will take effect. In other words, agenda groups provide a way to create a flow between grouped rules. They work as a stack. When we set the focus to a given agenda group, that group is placed on top of the stack. When the engine tries to fire the next activation and there are no more activations in a given group, that group is removed from the top of the stack and the group below receives focus again.

The Umplificator executes the rules to transform classes first, followed by the rules transforming attributes and finally by the rules transforming associations.

We use the attribute agenda-group in the rules to specify the order of the activation. For instance, the rule in Listing 5.17 is a rule belonging to the group that will be executed first. The rule in Listing 5.18 will be executed after any rule belonging to the first level.

Listing 5.17: A rule belonging to Level 1

```
1 rule "transform_Namespace_UInterface"
2         agenda-group "LEVEL1"
3         when
4         // parts omitted
5         then
6         // parts omitted
7 end
```

Listing 5.18: A rule belonging to Level 2

```
1 rule "JavaField_CanBeUmpleAttribute"
2         agenda-group "LEVEL2"
3         when
4         // parts omitted
5         then
6         // parts omitted
7 end
```

Listing 5.19 shows how the rules are inserted into the Working Memory of the Umplificator rule engine. Level 3 will be put on the bottom of the stack, followed by Level 2 rules, and Level 1 rules which will be on the top of the stack. The *KieSession* object represents the working memory of the Rule Engine.

Listing 5.19: Firing the rules in the Umplificator

```
public KieSession fireAllRules()
{
// Agenda works as a stack
kieSession.getAgenda().getAgendaGroup( "LEVEL3" ).setFocus();
kieSession.getAgenda().getAgendaGroup( "LEVEL2" ).setFocus();
kieSession.getAgenda().getAgendaGroup( "LEVEL1" ).setFocus();
kieSession.fireAllRules();

return kieSession;
}
```

More details on the different mapping rules will be presented in Section 5.5.

5.4.4 Generator

The Generator component validates the received UmpleModel and generates Umple code from it. That is, it generates an Umple file for each class or interface in the Umple model.

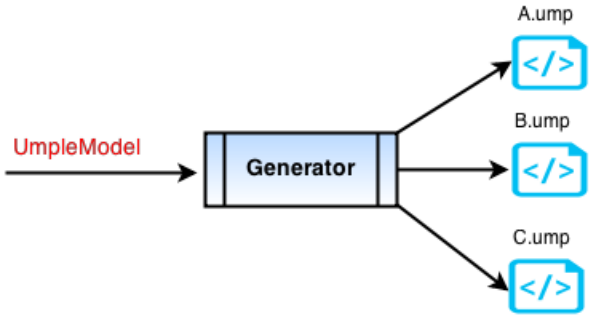

FIGURE 5.12: The Generator component inputs and outputs

The Generate supports different options when it comes to generation of output files. One way to do this is to follow the Java convention of having one .ump file per class. Another common approach is to have one or more files for the model code (just the pure UML elements such as classes with their attributes, associations and state machines) and separate files for the methods; we can in fact have some files for Java methods, and other files for PHP or Ruby methods. The same model can then be used to develop systems that are deployed in multiple base languages. For instance, for the Java input class*A.java*, the two following files would be generated:

1. *A.ump*: containing methods, algorithmic and logic code for class A.

2. *A_model.ump*: Modeling constructs for class A.

The Generator supports also the creation of directories to preserve the namespace structure. For instance, if the namespace of the Umple file is 'cruise.Umple', the Generator will create two directories, 'cruise' and 'Umple' (inside).

5.5 Automated Umplification Example

5.5.1 Initial transformation

As an example of the transformation process using the Umplificator, consider the input Java source code in Listing 5.20. We want to achieve the initial level of refactoring (Level 1).

LISTING 5.20: Input source code

```
1  package university;
2
3  import java.util.Date;
4
5  public class Student {
6
7      private String name;
8      private int studentId;
9
10     public Student (int studentId) {
11      this.studentId = studentId;
12     }
13     public String getName () { return name;}
14
15     public void  setName (String aName) {
16      this.name =  aName;
17     }
18
19     public int getStudentId () { return studentId;}
20
21     public String toString() {
22      return "The student " + name "has id=" + studentId;
23     }
24 }
```

The *Parser* receives the source code above, creates an Abstract Syntax Tree representation of it and transfers it to the Model Extractor. The Model Extractor uses the AST representation to create a Java model which is then traversed and decomposed in pieces by means of a Java class visitor. Table 5.5 presents all the Java Elements collected by the Java visitor.

The Transformer receives the Java model elements in Table above, together with a newly created instance of an UmpleModel and places them into the Working Memory. At

TABLE 5.5: The input Java Model elements

ASTNode Type	Source Fragment
PackageDeclaration	”package university;”
ImportDeclaration	”import java.util.Date;”
TypeDeclaration	”public class Student”
FieldDeclaration	”public String name;”
FieldDeclaration	”public int studentId;”
MethodDeclaration	”public int getStudentId () ...”
MethodDeclaration	”public String getName () ...”
MethodDeclaration	”public void setName(...) ”
MethodDeclaration	”public Student(...)”
MethodDeclaration	”public String toString()...”

this point of time, the *Production Memory* contains all rules but the *Agenda* contains only those belonging to this level of refactoring (those with attribute 'agenda-group LEVEL1').

When the model elements (facts) are inserted into the memory, the pattern matching begins. The rule engine then tries to find objects matching the conditions in the rules. The only rule meeting all the conditions and that can be matched to objects in the Working Memory is the rule named *addClassToUmpleModel*. The rule is presented in Listing 5.21.

LISTING 5.21: Rule 'addClassToUmpleModel'

```
1  rule "addClassToUmpleModel"
2   agenda-group "LEVEL1"
3   when
4    typeDeclaration: TypeDeclaration()
5    umpleModel: UmpleModel()
6   then
7    String typeName = getTypeDeclarationName(typeDeclaration);
8    UmpleClass umpleClass = new UmpleClass(typeName);
9    umpleModel.addUmpleClass(umpleClass);
10   insert(umpleClass);
11 end
```

The rule above simply requires the presence in the Working Memory of an instance of TypeDeclaration (Line 4) and an instance of an UmpleModel (Line 5). As the conditions are satisfied, in the RHS of this rule we create a new instance of UmpleClass, setting its name. To extract the name of the instance *TypeDeclaration* we employ a helper function *getTypeDeclarationName(...)*. After the object is created, we insert it into the session with the '*insert(umpleClass)*' method. This process of matching facts with rules (i.e. inference) is illustrated in Figure 5.13.

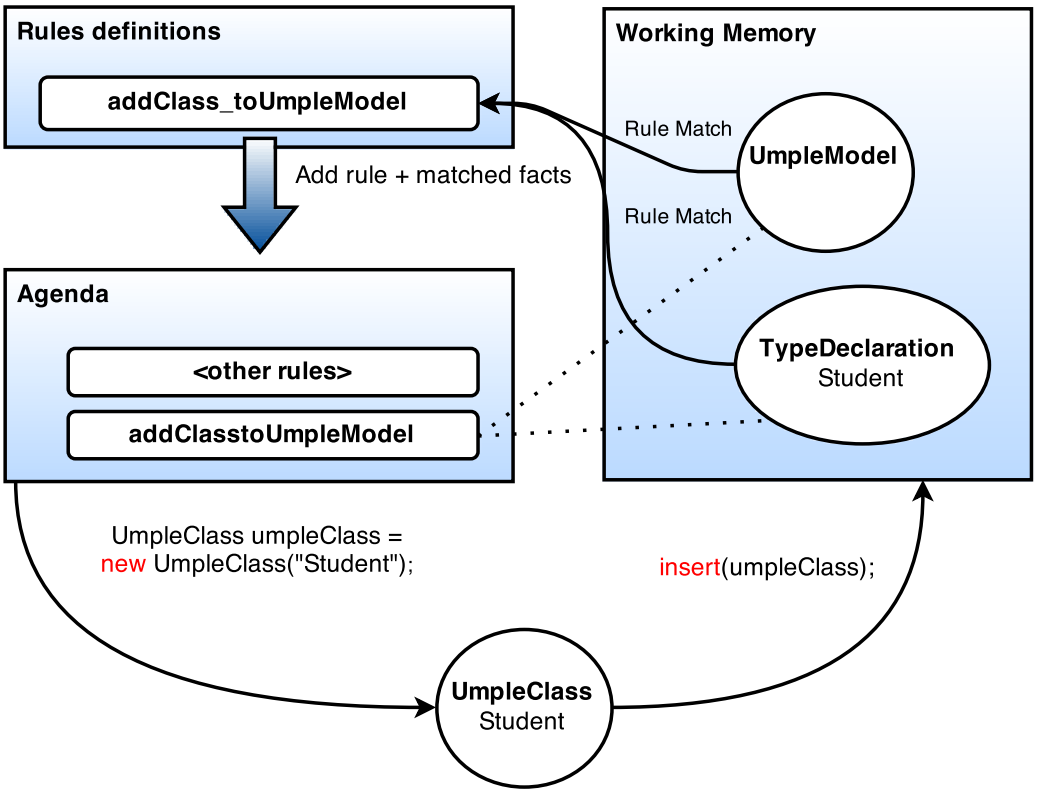

FIGURE 5.13: Pattern Matching and creation of an UmpleClass

After the insertion of the UmpleClass into the working memory, the inserted object can generate more rule matches. The UmpleModel residing in the Working Memory now contains one Umple class. It is automatically updated by the engine.

The rules for the remaining Umple class constructs are then matched. The goal of these rules is to populate the Umple class based on the information obtained from the typeDeclaration.

The rule named *transformImportDeclaration* (Lines 1-11) in Listing 5.22 matches and converts any Import Declaration (Java Language) into an Umple depend construct. The dependency (Line 9) is then added to a matched Umple Class. The Umple Class residing in the Working Memory is then updated at Line 10.

To ensure that the dependency is not added to any umpleClass in the Working Memory but only to the one owning it, we assert that the ImportDeclaration's parent class has the same name as our targeted UmpleClass. The helper function, imported in the first line of the above Listing, is a static function that returns the name of the parent class of the ImportDeclaration. In our case, the name of parent Java class is "Student" which

corresponds to the name of a UmpleClass in memory. The 'eval' clause returns true in this particular case.

Listing 5.22: Rule transformImportDeclaration

```
1  import function cruise.umplificator.rules
2         .TopLevelAnalyzerHelper.getDeclarationContainerName
3
4  rule "transformImportDeclaration"
5   agenda-group "LEVEL1"
6   when
7    importDeclaration: ImportDeclaration()
8    uClass: UmpleClass()
9    eval(uClass.getName()
10        .equals(getDeclarationContainerName(importDeclaration)))
11  then
12   Depend depend = new Depend(getImportName(importDeclaration));
13   uClass.addDepend(depend);
14   update(uClass);
15  end
```

The package declaration is converted then into a namespace with the rule 'transform-Namespace' in Listing 5.23. We again ensure that the package declaration corresponds to the targeted UmpleClass. Note that in this rule we don't need to insert the namespace object into memory since we don't expect any rule to match it.

Listing 5.23: Rule transformNamespace

```
1  import function cruise.umplificator.rules
2         .TopLevelAnalyzerHelper.getDeclarationContainerName
3  rule "transform_Namespace"
4   agenda-group "LEVEL1"
5   when
6    packageDeclaration: PackageDeclaration()
7    uClass: UmpleClass()
8    eval(uClass.getName()
9        .equals(getDeclarationContainerName(packageDeclaration)))
10  then
11   uClass.addNamespace(packageDeclaration.getName()
12                       .getFullyQualifiedName());
13 end
```

As we have assumed an initial level of refactoring. at the beginning of this example, the Transformer will not attempt to transform any variable into an Umple attribute, association end or state machine. However, in the final output code produced for our UmpleClass we require the remaining untreated code to be simply appended. For instance, the rule 'appendFieldDeclaration' in Listing 5.24 extracts information from the field declaration and appends it to the targeted Umple Class. The same behavior is produced from the application of rule 'appendMethodDeclaration' in Listing 5.25.

LISTING 5.24: Rule appendFieldDeclaration

```
1  import function cruise.umplificator.rules
2         .TopLevelAnalyzerHelper.getDeclarationContainerName
3
4  rule "appendFieldDeclaration"
5   agenda-group "LEVEL1"
6   when
7    fieldDeclaration: FieldDeclaration()
8    uClass: UmpleClass()
9    eval(uClass.getName()
10        .equals(getDeclarationContainerName(fieldDeclaration)))
11   eval(!uClass.getExtraCode().contains(fieldDeclaration.toString()))
12  then
13   uClass.appendExtraCode(fieldDeclaration.toString());
14   update(uClass);
15 end
```

The *eval* clauses in the Listing above, ensure that the string representing the field information hasn't been appended before and (as before) that the field string is added to the Umple class owning it.

LISTING 5.25: Rule appendMethodDeclaration

```
1  import function cruise.umplificator.rules
2         .TopLevelAnalyzerHelper.getDeclarationContainerName
3
4  rule "appendMethodDeclaration"
5   agenda-group "LEVEL1"
6   when
7    method: MethodDeclaration()
8    uClass: UmpleClass()
9    eval(uClass.getName().equals(getDeclarationContainerName(method)))
10   eval(!uClass.getExtraCode().contains(method.toString()))
11  then
12   uClass.appendExtraCode(method.toString());
13   update(uClass);
14 end
```

The *eval* clauses in the Listing above, ensure that the string representing the method information hasn't been appended before and (as before) that the method string is added to the Umple class owning it. At the end of this pattern matching process, the UmpleClass with name 'Student' owns a depend, has a namespace and some remaining code that we called extra code. We show the Umple code in Listing 5.26. The extra code in this code excerpt starts from Line 6 to 22.

The code generated by the *Generator*, from the input Umple model, is presented in Listing 5.26.

This concludes the initial transformation step.

LISTING 5.26: Umple code generated – Level 1

```
namespace university.student;

class Student {
 depend java.util.Date;

    public String name;
    public int studentId;

    public Student (int studentId) {
     this.studentId = studentId;
    }
    public String getName () { return name;}

    public void  setName (String aName) {
     this.name =  aName;
    }

    public int getStudentId () { return studentId;}

    public String toString() {
     return "The student " + getName() "has id=" + getStudentId();
    }
}
```

5.5.2 Automated Umplification of Attributes

As an example of the pattern matching for rules in Level 2 (attributes), consider the same code excerpt from Listing 5.20. The process described in the previous example remains identical, except for the transformations phase, which we explain now. Assuming that the rules for the transformation of the package and import declarations have already been performed, we focus exclusively on how the field declarations are transformed into Umple attributes. In this particular example we are expecting the two Java field declarations to be transformed into attributes.

At this point of time, as illustrated in Figure 5.14, the Working Memory contains the Umple model, an Umple Class (Student) and the Java model elements. The Production Memory contains all the 'LEVEL1' and 'LEVEL2' rules and the Agenda only those from 'LEVEL2' since the ones from 'LEVEL1' have already been executed and removed from the stack.

The rule engine then attempts to find objects matching the conditions in the rules. The only set of rules that can be matched to objects in the Working Memory are the rules related to attributes since the only rules in the agenda are those for 'LEVEL2' and the rules for 'LEVEL1' have already been applied. In particular, the unique rule that can

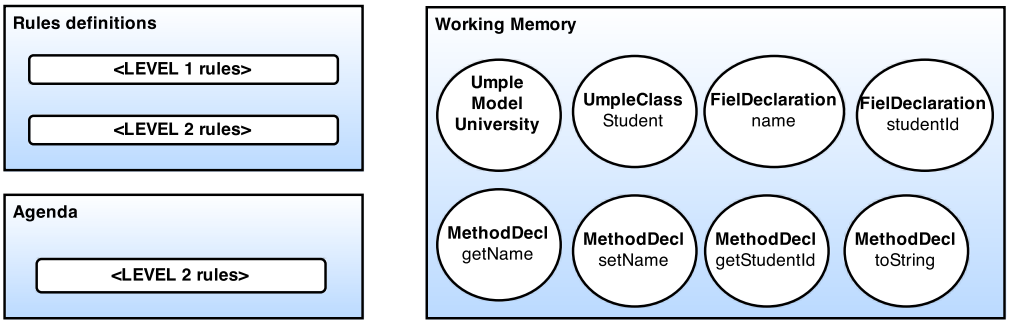

FIGURE 5.14: Rule Engine snapshot after initial transformation

be executed at this moment is the rule named *'Field_CanBeUmpleAttribute'* since all other rules require an instance of an Umple attribute in memory. As we have illustrated before in Figure 5.14, no instances of an Umple attribute exist or have been added so far. Rule *'Field_CanBeUmpleAttribute'* is presented in Listing 5.27.

LISTING 5.27: Rule FieldCanBeUmpleAttribute

```
1  rule "FieldCanBeUmpleAttribute"
2  agenda-group "LEVEL2"
3   when
4          fieldDeclaration: FieldDeclaration()
5          uClass: UmpleClass()
6          method: MethodDeclaration()
7          eval(isPrimitiveOrStringOrTime(fieldDeclaration))
8          eval(uClass.getName()
9               .equals(getFieldCuATTontainerName(fieldDeclaration)))
10         eval(uClass.getName().equals(getMethodContainerName(method)))
11         eval(uClass.getAttribute(getFieldName(fieldDeclaration))== null)
12         eval(hasFieldAGetter(method, fieldDeclaration, uClass.getName()))
13  then
14         String attrName = getFieldName(fieldDeclaration);
15         String attrType  = getAttributeType(fieldDeclaration);
16         Attribute uAttr =
17           new Attribute(attrName, attrType, null, null, false, uClass);
18         uAttr.setModifier("settable");
19         uClass.addAttribute(uAttr);
20         removeClassField(fieldDeclaration, uClass);
21         update(uClass);
22         insert(uAttr);
23 end
```

The rule in Listing 5.27 creates an Umple attribute with information extracted from a field declaration. It does so indeed, only if the following conditions are met:

- In Lines 4,5,6. We require instances of a field, Umple class and method declarations in order to assess whether or not the field can become an Umple attribute. These elements need to be found in the Working Memory.
- In Line 7. The field needs to be of a primitive type.

- In Line 8-9. The field needs to be declared in the Class that derived the current instance of the Umple Class.
- In Line 10. The method declaration needs to be declared in the Class that derived the current instance of the Umple Class.
- In Line 11. The Umple class must not possess an attribute with the name of the current instance of the field declaration. This is to avoid duplicates.
- In Line 12. The field possess a getter in the Class that derived the current instance of the the Umple Class.

If a field (and other model elements) matches the above conditions. A new Umple attribute is created with the information extracted from the instance of the field declaration and associate to the current instance of Umple Class (Line 16). The name (Line 14) and type (Line 15) are initialized as well. By default, our newly created Umple attribute is declared as settable (Line 17). The attribute is then added to a matched Umple Class (Line 18). The Helper function *removeClassField* in Line 18 removes the field declaration from the extra code of the Umple Class since it has been refactored into an Umple attribute. Recall from previous subsection that the field declaration was appended to the extra code of the Umple class. In Line 19, we updated the Umple class residing in the Working Memory. The clause '*update*' is optional but we explicitly invoke it for logging purposes.

Finally, in Line 20 the attribute is put into the working memory so subsequent transformations can be made such as determining if the attribute is lazy or not. In fact, this Umple attribute as we will explain later, meets all conditions to be a 'lazy' attribute. Lazy attributes have been introduced in Chapter 2. This process of matching objects with rules as we have described so far for this transformation step is summarized in Figure 5.15. As can be seen in the Figure, an instance of a the Umple attribute has been added to the Working Memory as a result of the match. Objects in yellow are the ones queried when evaluating the conditions of the rule.

We are not done yet since we need to remove or refactor the getters and/or setters of the field (that became an attribute). For instance, if the rule (omitted) '*hasFieldSimpleGetter*' or '*hasFieldSimpleSetter*' is matched to any field previously transformed into an attribute, the method declaration is removed from the appended code of the Umple

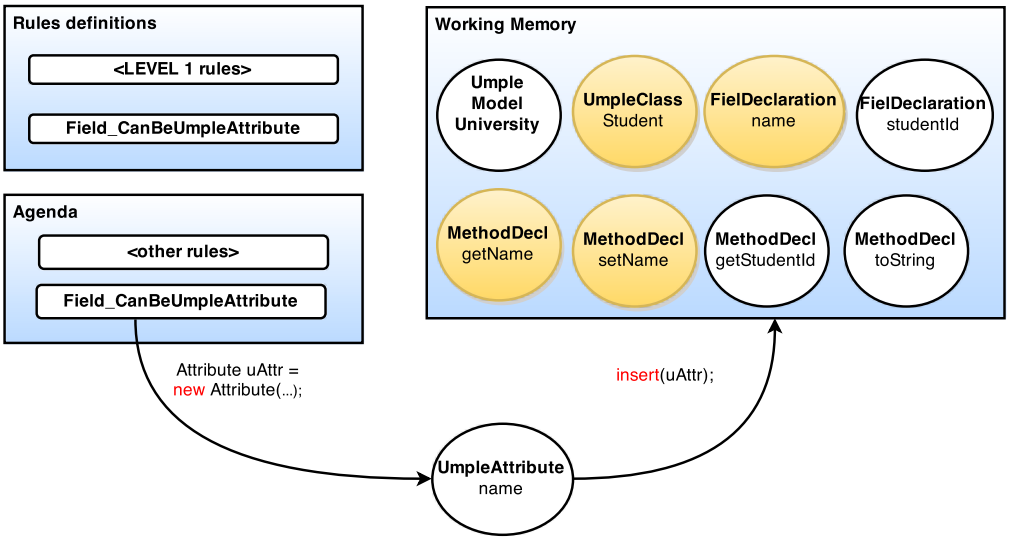

FIGURE 5.15: Pattern Matching and creation of an UmpleClass

class. As is the case in our example, getName(), setName() and getStudentId() are removed from the 'Student' Umple class. When the field has a getter/setter that is not simple, an instance of the class CodeInjection (refer to Umple metamodel) is created to take into account the code differing from the original getter/setter.

Finally, the rule named *isLazyAttribute* in Listing 5.28, rule matches and converts any basic attribute (in memory) that conforms to the required conditions into a lazy attribute (e.g. attribute.setIsLazy(true)). These required conditions are listed next (Lines refer to Listing 5.28):

- In Lines 4,5,6. We require instances of a field, Umple class, method declaration and an Umple attribute in order to assess whether or not the attribute can become a lazy attribute. These elements need to be found in the Working Memory.
- In Line 8. The current instance of the Umple class needs to be the one owning the Umple attribute.
- In Line 9. The field needs to be declared in the Class that derived the current instance of the Umple Class.
- In Line 10. The method declaration needs to be declared in the Class that derived the current instance of the Umple Class.
- In Line 11. A (double) check to ensure that the attribute belongs to the class.

- In Line 12. The current instance of the field declaration is the one used to derived the current instance of the attribute.
- In Line 13: The class in which the field is declared is NOT one of the constructor arguments. As per definition of a lazy attribute.

LISTING 5.28: Rule isLazyAttribute

```
1  rule "isLazyAttribute"
2  agenda-group "LEVEL2"
3   salience 50
4   when
5    fieldDeclaration: FieldDeclaration()
6    method: MethodDeclaration(method.isConstructor())
7    attribute: Attribute(isLazy==false)
8    uClass: UmpleClass(getAttributes().size() > 0 &&
                                          getAttributes().contains(
       attribute))
9    eval(uClass.getName().equals(getFieldContainerName(fieldDeclaration)))
10   eval(uClass.getName().equals(getMethodContainerName(method)))
11   eval( attribute.getUmpleClass() == uClass)
12   eval( attribute.getName().equals(getFieldName(fieldDeclaration)))
13   eval(isFieldInContructor(method, fieldDeclaration, uClass.getName())==
       false);
14   eval(getFieldName(fieldDeclaration).equals(attribute.getName()));
15  then
16   attribute.setIsLazy(true);
17   update(attribute);
18 end
```

Since there are no more rules to execute, the rule engine stops. The update Umple model is passed to the Generator for code generation. Note that the method *toString()* is still part of the extra code of the Student Umple class. The code generated by the *Generator*, from the populated Umple model, is presented in Listing 5.29. This concludes our second transformation step.

LISTING 5.29: Umple code generated – Level 2

```
1  namespace university.student;
2
3  class Student {
4   depend java.util.Date;
5
6   lazy String name;
7   Integer studentId;
8
9   public String toString() {
10   return "The student " + getName() "has id=" + getStudentId();
11  }
12 }
```

In this next section, we provide an overview of the tools currently available to support the umplification reverse-engineering process.

5.6 Umplificator Tooling

The Umplificator is available as an IDE and works within Eclipse; it also operates as a command-line tool to allow rapid bulk umplification and easier automated testing. Both tools are built and deployed using the Ant scripting language; resulting in several executable jars as well as for the Eclipse plugins. Table 5.6 describes the various jars deployed as part of our automated building process. In the table, X corresponds to the version, Y to the revision and Z to the build number. Our current version is '1.21.0.4666'.

TABLE 5.6: Artifacts deployed during the building process of the Umplificator

Name	Description
cruise.umplificaror.eclipse_vX.X.X.jar	Plug-in for the Eclipse IDE
umplificator_X.Y.Z.jar	Command-Line tool for umplification
validator_X.Y.Z.jar	Command-Line tool that checks whether the input Umple code generates compilable base language code.

Umplifying source code by means of the command-line tool can be done using the following command:

```
< java -jar umplificator_1.21.0.4666.jar inputFile -level=1,2,3
      -splitModel -dir -path >
```

where:

- **inputFile** can be an Umple file, base language file (.java, .cpp) or Source directory (containing java/Umple/cpp files).
- **level**: can be 0,1,2 and corresponds to the refactoring that wants to be achieved. 0 for the initial (classes, namespace, imports, etc.); 1 for the refactoring of attributes; 2 for the refactoring of associations. Level 2 includes transformations from level 0 and 1. Level 1 includes (and requires) transformations from level 0.
- **splitModel** (optional): creates two files for each input file; one containing the modeling constructs, one containing the algorithmic and logic code (extra code).
- **dir** (optional): creates directories following the namespace structure.
- **path**: the output directory name where the resulting Umple files will be located.

Additionally, the Umplificator is available as an online tool, called UmplificatorOnline. The tool is under development but will be deployed soon for public access. We have created this project for several purposes. Casual users are able to experiment with the latest version of the Umplificator with no more than a browser and an Internet connection. This allows curious developers to try out the tool. Figure 5.16 presents the initial page of the UmplificatorOnline. An open-source project *downloader* has been implemented as part of this Web tool. Please note that the tool runs the Umplificator main jar (umplificator_1.21.0.4666.jar) for its reverse engineering capabilities.

The preliminary release of the tool allows developers to:

- Select and Open-Source **repository**: User can select projects from GoogleCode, SourceForge or GitHub.
- Select an Open-Source **project** to umplify. The projects listed (in the second combo-box) have been automatically selected based on a number predefined criteria. The criteria for project selection are that project is marked as small or medium-size system and that the project is written in Java or C++.
- Select the **level** of refactoring desired.
- Select one of more **options**. The options have been described during our discussion on the command-line tool.

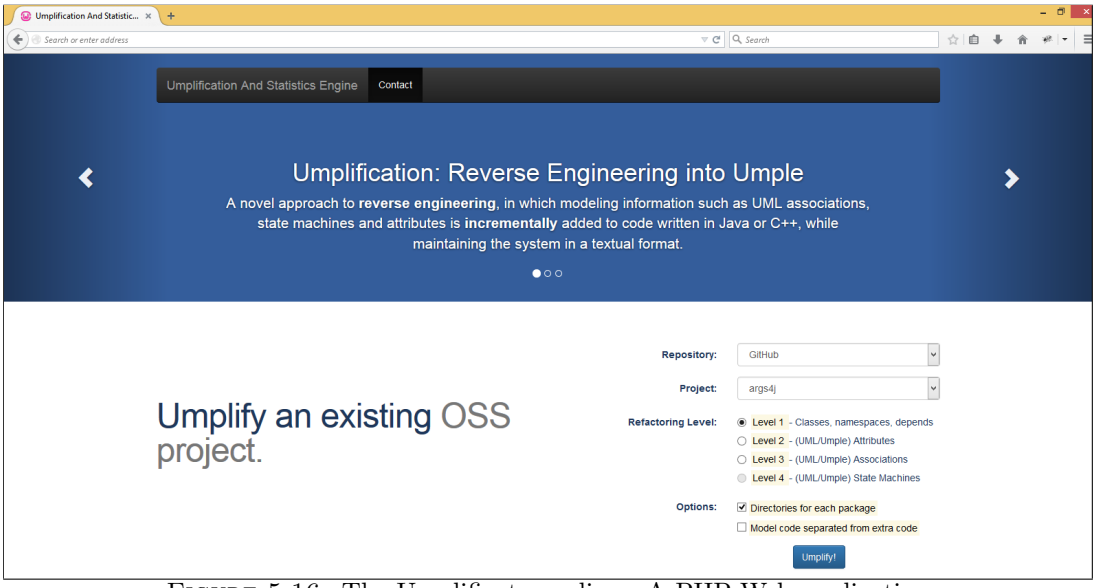

FIGURE 5.16: The Umplificator online - A PHP Web application

Chapter 6

Evaluation

In order to ensure that the results presented in this thesis are of high engineering quality and are as valid as possible from a scientific perspective, several approaches need to be followed. We validated our reverse engineering approach by studying the application of our approach on various software systems. We adopted a **four-phase validation process**with the following steps:

Testing Phase Unit testing is carried out following a Test Driven Development approach (TDD).

Pre-validation Phase Small Java systems written in high quality Java code, with known corresponding models, are employed to validate the accuracy of the transformations performed by the Umplificator.

Initial Phase Medium and large open-source projects are employed to validate the accuracy of the transformations and mapping rules. This set of open source projects will be known as the **'training set'**. The goal of this phase is to ensure the correctness and precision of the transformations on the training set.

Machine Learning-Based Phase In this phase, we umplify a set of randomly selected systems, the**'testing set'** and assess the extent to which our transformations still work. We document the errors encountered during this phase of validation.

In general all four of the above phases are conducted in an iterative manner. In other words, we develop the Umplificator in small chunks that are validated at the same time.

This chapter is organized as follows: in the next sections we present each of the four phases of validation including the results obtained. Finally, we provide extended details on the largest systems that were umplified during the four phases.

6.1 Testing Phase

As illustrated in Chapter 5.4, the Umplificator includes: a **parser**, a **model extractor**, a **transformer** and an Umple code **generator**. Each of the components is independently tested to ensure high quality as illustrated in Figure 6.1. The Umplificator testing process is only capable of testing within the scope of the Umplificator. In other words, we are testing the Umplificator implementation and **not** testing the set of possible umplified systems generated using our tool. In fact, we only test that the outputs (Umple code) are syntactically correct. To achieve the additional level of testing by which you validate the semantics of systems generated by the Umplificator, one must run or build a test suite against those generated systems. At present there are over 135 tests that spans all areas above and are run as part of our automated quality process (continuous integration).

In the subsequent sections we provide an overview of each aspect of the Umplificator's testing approach.

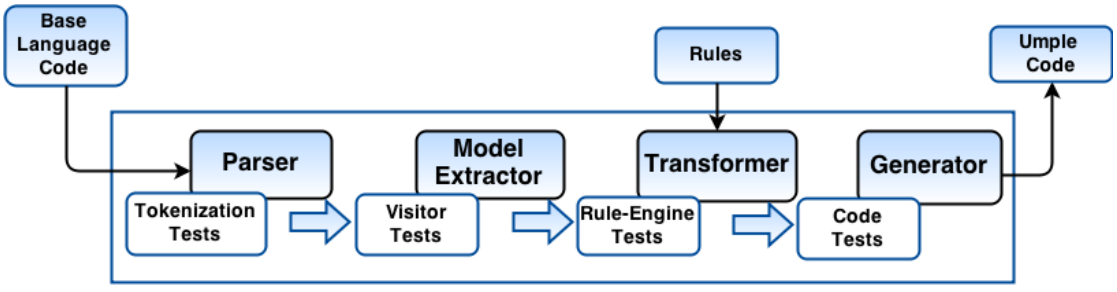

FIGURE 6.1: Umplificator Testing Infrastructure

6.1.1 Testing the Base Language Code Parsers

Testing the Umplificator parser is centered on the creation of the AST DOM from base language code. Our tests ensure that Base Language code is parsed and tokenized as we expect.

A simple parser test is shown below that verifies that the list of detailed problem reports (warnings, or compilation errors) noted by the compiler during the parsing or the type checking of the compilation unit (file) is what we expect. In this particular example, we are expecting two problems (compilation errors) since the input compilation unit contains two errors at two different locations in the code.

```
@Test
public void simpleFileWithTwoErrors()
{
  File testFile = new File(pathToInput+"SimpleFileWithTwoErrors.java");
  String code = SampleFileWriter.readContent(testFile);
  JavaParser javaParser = new JavaParser(); // JDT Parser
  CompilationUnit unit  = javaParser.parseUnit(code);
  Assert.assertEquals(2,unit.getProblems().length());
}
```

The pattern for parser-related test is as follows:

```
@Test
public void parserTestX()
{
  // Step 1: Load external source file (Java or C++ file)
  // Step 2: Parse file (ensure parsing successful)
  // Step 3: Verify tokenization
  // Step 4: Clean up
}
```

6.1.2 Testing the Model Extractor

Testing the model extractor ensures that from the tokens obtained through the parser we obtain a valid base language model representation (e.g. Java model, Umple model, CPP model). In particular, as we have implemented a visitor (software design pattern) to traverse the different elements of the retrieved base language model, our tests ensure that the visitors return the desired number of elements.

For instance, if the test input file contains:

LISTING 6.1: Java input file for test.

```
package cruise.umplificator.visitorTestFiles;

 import java.util.*;
 import java.io.*;

 @SuppressWarnings("unused")
 public class InputForVisitorTest {

  boolean result = true;
  char capitalC = 'C';
```

```
11   byte b = 100;
12   short s = 10000;
13   int i = 100000;
14   double d1 = 123.4;
15   long creditCardNumber = 1234_5678_9012_3456L;
16
17   InputForVisitorTest () { }
18
19   InputForVisitorTest(byte b) {
20    this.b=b;
21   }
22
23   public int getB(){
24    return b;
25   }
26 }
```

in the following unit test we assert that the (Java) visitor returns: 7 field declarations (Lines 17-20), 2 import declarations(Lines 23-27), 3 method declarations (Lines 37-41) and a package name 'cruise.umplificator.visitorTestFiles' (Line 30-34).

```
1  public class JavaVisitorTest {
2
3   String pathToInput;
4   JavaClassVisitor visitor ;
5
6   @Before
7   public void setUp() throws Exception {
8    pathToInput = SampleFileWriter.rationalize("test/cruise/umplificator/
       visitorTestFiles/");
9    File testFile = new File(pathToInput+"InputForVisitorTest.java");
10   String code = SampleFileWriter.readContent(testFile);
11   JavaParser javaParser = new JavaParser();
12   CompilationUnit unit  = javaParser.parseUnit(code);
13   visitor = javaParser.getJavaVisitor();
14  }
15
16  @Test
17  public void field_declarations_returned_in_java_file()
18  {
19   Assert.assertEquals(7, visitor.numberOfFieldDeclarations());
20  }
21
22  @Test
23  public void imports_returned()
24  {
25   int nbImports = visitor.numberOfImportDeclarations();
26   Assert.assertEquals(2, nbImports);
27  }
28
29  @Test
30  public void packages_returned()
31  {
32      String packageName = visitor.getPackageDeclaration().getName().
      getFullyQualifiedName();
33   Assert.assertEquals("cruise.umplificator.visitorTestFiles", packageName
       );
34  }
35
```

```
36 @Test
37 public void methods_returned()
38 {
39  int nbMethods = visitor.numberOfMethodDeclarations();
40  Assert.assertEquals(3, nbMethods);
41 }
```

6.1.3 Testing the Transformer

Testing the **transformer** involves ensuring that our Rule-Engine receives the input, fires the corresponding mapping rules and produces the expected output. For instance, if the input of our tests below is the Java class in Listing 6.1, we expect all our assertions to pass. In particular:

- Line 12-23: In the *setUp()* method of our test, we parse the input file and create an Umple class that is inserted into the working memory of the Rule Engine (Line X). Note that in Line Y the desired **level of refactoring** includes Umple attributes(and excludes Umple associations) since the goal of this test class is to ensure the correct mapping between variables possessing certain characteristics and Umple attributes.
- Line 26-28: The unit test *testNumberOfObjectsInWorkingMemory* ensures that at this point of time, there is only one element in the working memory (the Umple class inserted in Line 22).
- Line 21-63: The unit test *testCorrectMappingBetweenPrimitiveField2UmpleAttribute* validates the mappings between the Java fields (input) and the Umple attributes.
- In Lines 33-35 the fields declarations of the Java class are inserted into the Working Memory.
- Line 37: The DROOLS rules are fired.
- Line 47-62: We assert that the Rule Engine has correctly created the Umple attributes. We ensure that the name and type of field has been correctly assigned to the Umple attribute.
- Line 65-74: The unit test *testCorrectMappingBetweenImport2Depend* also ensures the correct mapping between the input Java import declarations and the Umple depends clause.

- In Line 78 we clean the working memory of the rule engine.

```
1  public class RulesAttributesTypesTest {
2
3   String pathToInput;
4   JavaClassVisitor visitor ;
5   RuleRunner runner  = new RuleRunner();
6   RuleService ruleService= new RuleService(runner);
7   KieSession kieSession;
8   UmpleClass uClass;
9   CompilationUnit compilationUnit;
10
11  @Before
12  public void setUp() throws Exception {
13   pathToInput = SampleFileWriter.rationalize("test/cruise/umplificator/
       visitorTestFiles/InputForVisitorTest.java");
14   File testFile = new File(pathToInput);
15   String code = SampleFileWriter.readContent(testFile);
16      visitor = new JavaClassVisitor();
17   JavaParser javaParser = new JavaParser();
18   javaParser.parseUnit(code);
19   visitor = javaParser.getJavaVisitor();
20   uClass = new UmpleClass("Test");
21      kieSession = ruleService.startRuleEngine(RefactoringLevel.ATTRIBUTES
       );
22   kieSession.insert(uClass);
23  }
24
25  @Test
26  public void testNumberOfObjectsInWorkingMemory() {
27   Assert.assertEquals(1, kieSession.getObjects().size());
28  }
29
30  @Test
31  public void testCorrectMappingBetweenPrimitiveField2UmpleAttribute() {
32   // Insert facts into knowledge base
33   for(FieldDeclaration field: visitor.getFieldDeclarations()){
34    kieSession.insert(field);
35   }
36   // Fire rules
37   kieSession.fireAllRules();
38   // Is not Null
39   Assert.assertNotNull( uClass.getAttribute(0));
40   Assert.assertNotNull( uClass.getAttribute(1));
41   Assert.assertNotNull( uClass.getAttribute(2));
42   Assert.assertNotNull( uClass.getAttribute(3));
43   Assert.assertNotNull( uClass.getAttribute(4));
44   Assert.assertNotNull( uClass.getAttribute(5));
45   Assert.assertNotNull( uClass.getAttribute(6));
46
47   // Type has been set correctly
48   Assert.assertEquals("Boolean", uClass.getAttribute(0).getType());
49   Assert.assertEquals("String", uClass.getAttribute(1).getType());
50   Assert.assertEquals("Integer", uClass.getAttribute(2).getType());
51   Assert.assertEquals("Integer", uClass.getAttribute(3).getType());
52   Assert.assertEquals("Integer", uClass.getAttribute(4).getType());
53   Assert.assertEquals("Double", uClass.getAttribute(5).getType());
54   Assert.assertEquals("Double", uClass.getAttribute(6).getType());
55   // Name has been correctly set
56   Assert.assertEquals("result", uClass.getAttribute(0).getName());
57   Assert.assertEquals("capitalC", uClass.getAttribute(1).getName());
```

```
58   Assert.assertEquals("b", uClass.getAttribute(2).getName());
59   Assert.assertEquals("s", uClass.getAttribute(3).getName());
60   Assert.assertEquals("i", uClass.getAttribute(4).getName());
61   Assert.assertEquals("d1", uClass.getAttribute(5).getName());
62   Assert.assertEquals("creditCardNumber", uClass.getAttribute(6).getName
       ());
63  }
64
65  @Test
66  public void testCorrectMappingBetweenImport2Depend() {
67   for(ImportDeclaration importDecl: visitor.getImportDeclarations()){
68    kieSession.insert(importDecl);
69   }
70   kieSession.fireAllRules();
71   Assert.assertEquals(2, uClass.getDepends().size());
72   Assert.assertEquals("java.util.*", uClass.getDepends().get(0).getName()
       );
73   Assert.assertEquals("java.io.*", uClass.getDepends().get(1).getName());
74  }
75
76  @After
77  public void tearDown() throws Exception {
78   runner.dispose();
79  }
80 }
```

6.1.4 Testing the Umple Code Generator

Testing the code generator involves asserting that from a input file we obtain the expected Umple file. Briefly, we compare the content of an Umple file as generated by the Umplificator and the expected Umple file.

```
1  @Test
2  public void JavaToUmple_VariablesToAttributes_003(){
3   String fileName = "003_JavaToUmple_VariablesToAttributes";
4   File javaFile = new File(pathToRoot+fileName+"_java.java"); //INPUT
5   File umpleFile = new File(pathToRoot+fileName+"_umple.ump"); //OUTPUT
6   // Umplify file. Process must succeed!
7   assertTrue(umplificator.umplifyElement(javaFile));
8   // Get the output content
9   assertOuputAndFile(umpleFile);
10  // Clean files
11  filesToDelete.add(fileName);
12 }
13
14 // Helper Functions
15 public void assertOuputAndFile(File expectedContentFile)
16 {
17  try {
18   String inputFileContent = FileUtils.readFileToString(
     expectedContentFile);
19   String outputModel  = umplificator.getOutputModel().getCode();
20   assertEquals(inputFileContent, outputModel);
21  } catch (IOException e) {
22   fail();
23  }
```

```
24 }
```

The test above, performs the umplification process on the Java input file, and compares the content of the code produced by the Umplificator with the code of the expected Umple file. The comparison is done with the help of method *assertOuputAndFile*.

Testing the different components of our infrastructure allows for better defect management by representing bugs as failing tests, effectively diminishing the time and effort required to perform regression. Furthermore, this multi-level testing helps to make sure that a change or addition of a new feature doesn't break any existing functionality and if there is any bug to quickly locate the defective component. In fact, when a defect is uncovered, it might be one of the following:

1. Defect in the way the base language code is tokenized and converted into an abstract syntax tree.
2. Incorrect population of the base language metamodel instance from the tokenized language.
3. Inappropriate behavior of the rule engine.
4. Syntax errors in the generated Umple code.
5. Semantic errors in the generated base language code (from the umplified model).

6.2 Pre-Validation Phase

As is considered ideal best practice when introducing a new tool in software engineering, we have tested our umplificator using our own repository of examples. This collection of **Umple** examples, currently 42 ranging from Banking systems to Warehouse control systems, is available for review online at [54] and was used to generate the Java 'toy examples'. The process is illustrated in Figure 6.2. The goal of this pre-validation phase is to assert that the **UmpleModel** is semantically identical to the **UmpleModel'** which is the generated output of the Umplificator.

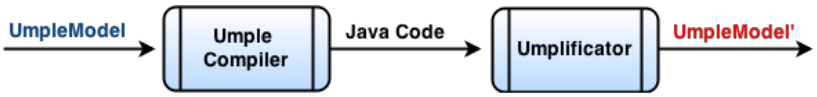

FIGURE 6.2: The Pre-Validation Phase: Comparing UmpleModel and UmpleModel'

Table 6.1 presents the Umple examples used in our first phase of validation as well as some statistics about them (number of lines of code of the Umple model, the number of lines of code of the corresponding Java system and the number of Java classes).

For instance, the *'Access Control Example'*, representing a system for managing access to facilities, is comprised of 6 classes, 8 associations, and 10 attributes. The Umple model is presented in Listing 6.2 together with corresponding visual representation, a UML class diagram generated using our online tool (Figure 6.3). The unit test comparing the input umple model and the umplified model (output) is shown in Listing 6.3.

LISTING 6.2: Umple Model for an Access Control System

```
namespace access_control;

class FacilityType
{
  code;
  description { Menu, Record, Screen }
  key {code}
}

//Functional_Area
class FunctionalArea
{
  String code;
  0..1 parent -- * FunctionalArea child;
  description { Hr, Finance }
  key {code}
}

//Facility_Functional_Area
association
{
  * FunctionalArea -- * Facility;
}

class Facility
{
  Integer id;
  lazy Time t;
  * -> 0..1 FacilityType;
  Integer access_count;
  name;
  description;
  other_details;

  key {id}
}

class Role
```

```
{
  code;
  role_description { Dba, ProjectMgr }

  key {code}
}

class User
{
  Integer id;
  * -> 0..1 Role;
  first_name;
  last_name;
  password;
  other_details;
  key {id}
}

associationClass RoleFacilityAccessRight
{
  * Facility;
  * Role;
  CRUD_Value { R, RW }
}
```

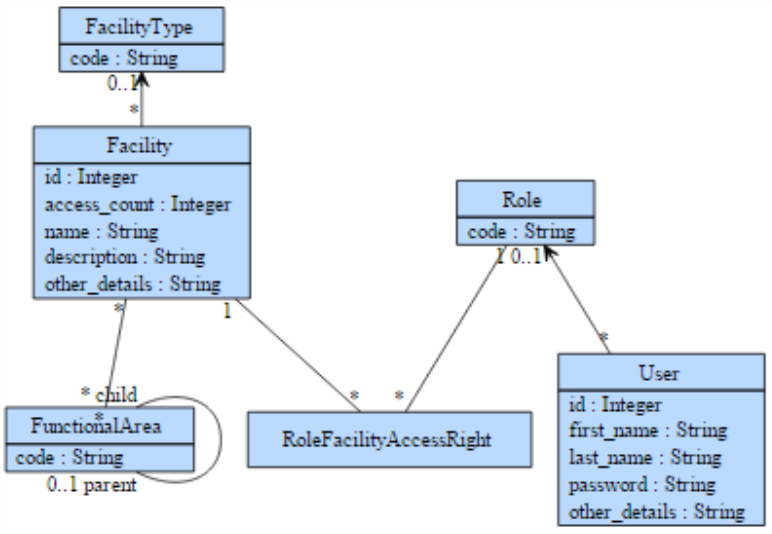

FIGURE 6.3: UML Class diagram of the Access Control system

LISTING 6.3: Unit test to assert the Access Control Example.

```
@Test
public void AccessControlExample(){
 String folderName = "AccessControl";
 File inputFile = new File(pathToRoot+ File.separator +folderName +".ump"
    );
 UmpleFile inputUmpleFile = new UmpleFile(inputFile);
 // Umplify all the files in folder
 List<File> inputFiles = FileHelper.getListOfFilesFromPath(pathToRoot+
    File.separator + folderName, new ArrayList<File>());
 // Umplify files. Process must succeed!
 assertTrue(umplificator.umplify(inputFiles));
 // This is the actual model, the one umplified
 UmpleModel umplifiedModel = umplificator.getOutputModel();
```

```
12  UmpleModel expectedModel = new UmpleModel(inputUmpleFile);
13  expectedModel.run();
14  //1. Class FacilityType
15  UmpleClass facilityTypeA = umplifiedModel.getUmpleClass("FacilityType");
16  UmpleClass facilityTypeE = expectedModel.getUmpleClass("FacilityType");
17  Assert.assertEquals(1, facilityTypeA.numberOfAttributes());
18  Attribute lazyAttributeA = facilityTypeA.getAttribute("code");
19  Attribute lazyAttributeE = facilityTypeE.getAttribute("code");
20  assertEquals(lazyAttributeA.getIsLazy(),lazyAttributeE.getIsLazy());
21  assertEquals(lazyAttributeA.getType(),lazyAttributeE.getType());
22  // 2. Class User
23  UmpleClass userA = umplifiedModel.getUmpleClass("User");
24  UmpleClass userE = expectedModel.getUmpleClass("User");
25  Attribute id = userA.getAttribute("id");
26  Attribute firstname = userA.getAttribute("first_name");
27  Attribute lastname = userA.getAttribute("last_name");
28  Attribute other_details = userA.getAttribute("other_details");
29  Attribute password = userA.getAttribute("password");
30
31  Assert.assertEquals(userA.numberOfAttributes(), userE.numberOfAttributes
      ());
32  Assert.assertEquals("Integer", id.getType());
33  Assert.assertEquals("String", firstname.getType());
34  Assert.assertEquals("String", lastname.getType());
35  Assert.assertEquals("String", other_details.getType());
36  Assert.assertEquals("String", password.getType());
37  // 3. Facility Class
38  UmpleClass facilityA = umplifiedModel.getUmpleClass("Facility");
39  UmpleClass facilityE = expectedModel.getUmpleClass("Facility");
40  Assert.assertEquals(facilityA.numberOfAttributes(), facilityE.
      numberOfAttributes());
41
42  Attribute timeAttr = facilityA.getAttribute("t");
43  Attribute idAttr = facilityA.getAttribute("id");
44  Attribute accessAttr = facilityA.getAttribute("access_count");
45  Attribute nameAttr = facilityA.getAttribute("name");
46  Attribute descAttr = facilityA.getAttribute("description");
47  Attribute otherAttr = facilityA.getAttribute("other_details");
48
49  Assert.assertTrue(timeAttr.isIsLazy());
50  Assert.assertFalse(idAttr.isIsLazy());
51  Assert.assertFalse(accessAttr.isIsLazy());
52  Assert.assertFalse(nameAttr.isIsLazy());
53  Assert.assertFalse(descAttr.isIsLazy());
54  Assert.assertFalse(otherAttr.isIsLazy());
55  // Compare both models, generally
56  assertTrue(areModelsEqual(umplifiedModel,expectedModel));
57 }
```

In the test case above, the level of refactoring includes only attributes (Umple associations have not been extracted) so we are interested in the classes, generalizations and the attributes of each class. We assert that the classes have been correctly detected and that the attributes in each class have been correctly extracted (attribute type, attribute name and additional features). For instance, in Line 69 we assert that the attributes is

lazy, since the variable is not one of the parameters in the constructor of the Java class **Facility** (Java code is not shown here).

More on our approach to validation is presented next.

6.3 Initial Phase of Validation

As part of our second stage of validation, we tested the Umplificator on various open-source systems written in Java. We use freely available systems to ease comparisons and replications of our evaluation. We provide some information on these systems in Table 6.2 including their version, number of lines of code and the number of classes. The last column of the Table indicates whether or not the system has been studied elsewhere.

Note that the data (statistics on the modeling constructs detected) in those external studies is used to compare the result of our automated tool. Furthermore, we perform '*manual*' umplification on the systems that have not been studied before, the results of the manual umplification are then compared to the results of our '*automated*' umplification. In fact, the only project that hasn't been studied (reverse-engineered) before is Weka, a very popular suite of machine learning software written in Java.

More concisely, for each system studied, we have followed these steps:

1. We apply the different transformation steps on the input object-oriented system.
2. We run the test suite available for the system to ensure that code compiles and is semantically identical to the original input source code.
3. We run a custom-made code analyzer on the Umple system generated (umplified) to obtain the statistics of the detected (extracted) Umple constructs (attributes, associations). At this stage, we obtain the number of attributes extracted for each class, their type, as well as the number of all different types of associations.

    - We compare our results with data obtained independently (if any). For instance, JHotDraw has been reverse-engineered and analyzed in other studies [**?**].

- In the case that the system has not yet been studied in other related work, we compare our results with data obtained from manual umplification. That is, we umplify the system without the help of the Umplificator tool. The manual umplification, a very time consuming task, is usually performed by another software engineer (undergraduate students contributing to the project).

4. We compute the **precision** and **recall** of the results previously obtained. Precision assesses the proportion of the constructs (attributes, associations) identified that are valid, while recall assesses the proportion of the independently-identified constructs that are found by our approach (a number near 1 would means very few are missed by our detection algorithms).

5. We refine our mapping rules to improve the precision of our algorithms. This step mainly concerns tuning the Umplificator. In general, tuning the Umplificator to increase the accuracy includes one or more of the following manual steps:

   (a) If there is an Umple construct that was missed from the extraction (false negative), we may add a new mapping rule to cover this case.

   (b) If there is an Umple construct that was incorrectly identified (false positive), we may edit the corresponding mapping rule.

   (c) If one of the methods requires additional transformations (as described in Table 1) was incorrectly refactored.

The following are the definitions we have employed for the precision and recall measures [62]:

$$Precision = \frac{(Documented) \cap (Detected)}{Detected}$$

and

$$Recall = \frac{(Documented) \cap (Detected)}{Documented}$$

In the following sections, we discuss our experiences with **JHotDraw** and **Weka**, the two larger systems we studied.

6.4 Second Phase of Validation

6.5 Results

6.5.1 JHotDraw

JHotDraw7 [55] is an open source graphic editor that supports operations on many graphics file formats. It makes extensive use of software design patterns and has detailed documentation about its design. We selected JHotDraw for umplification to be able to apply our transformations on documented frameworks and to compare results with the documentation of these frameworks and the analyses performed by other tools [19]. For this research we worked with JHotDraw 7.5.1.

Table X shows the results of detection of attributes and associations for the JHotDraw framework. It details the number of classes, the number of attributes and the different types of associations. We also performed a manual analysis to check the accuracy of our algorithms and mapping rules. After improving and refining our rules, we have obtained a precision of 100%. The refinement consisted of adding the Java idioms that our detection algorithms were not able to catch on the first attempt. For instance, not all the setters in the framework always return a void, some of them return a Boolean.

The Umplificator was hence tuned to be able to umplify JHotDraw. With each new system we umplify, we increase the accuracy of the mapping rules as well as the overall effectiveness of the umplificator. In general, tuning the Umplificator to increase the accuracy includes one or more of the following manual steps:

Briefly, the complexity of the tuning depends on the number of false positives and false negatives that the tool generates.

6.5.2 Weka

The next system we focused on was the machine learning tool Weka [63]. As with our the first attempt at umplifying JHotDraw, our first attempt at automatically umplifying Weka resulted in the identificatiojn of the need for improvement to our rules; in

particular some idioms it uses were not yet detected by our tool as it existed. For example, some classes in the classifiers package implement add() and remove() methods with different argument types. Also, the Confusion class declares add(RuleSet) and remove(Antecedent) to add and remove a set of rules from the evaluation algorithm. In addition, we detected, after execution of the test suite, that two classes were not compiling due to an unexpected constructor signature.

Initial Umplification results for Weka nonetheless have a precision of 85% when it comes to attributes and 38% for 1-to-many associations. Table 8 summarizes the results. Note that a precision of 38% doesn't mean that the Umplificator has missed 62% associations of this type. It means that some of them were not correctly transformed into Umple (e.g. incorrect navigability, role names or transformation of accessor/mutator methods). The extensibility and flexibility of our tool allows us to add and refine rules without having to recompile the system.

It is our objective to successively umplify more and more systems, with the hope that eventually our rule base will cover the vast majority of cases needed to successfully umplify new systems the Umplificator is presented with. However, even with a precision in the high 80% range, our tool serves as a useful tool for umplification. Users can leave some variables un-umplified, or can manually umplify the rest.

TABLE 6.1: Small examples used for first phase of validation

Name	**#LOC of Umple Model**	**#LOC of generated Java System**	**# of Java Files**
2DShapes	44	509	9
AccessControl	67	1560	6
Accidents	42	730	4
Accommodations	102	2215	9
AfghanRainDesign	132	2610	13
Airline	51	1800	8
Banking System A	87	2400	13
Banking System B	74	1650	12
Canal System	69	2222	14
Decisions	148	4153	15
Card Games (Oh Hell and Whist)	134	2051	8
Claim (Insurance)	19	408	2
Community Association	68	1591	9
Co-op Education System	69	2420	10
DMM Overview	59	1165	10
DMM Source Object Hierarchy	91	1774	16
DMM Relationship Hierarchy	135	1119	31
DMM CTF	93	932	4
Election System	83	2875	11
Elevator System A	42	1307	4
Elevator System B	56	1971	11
Genealogy A	29	670	2
Genealogy B	32	945	2
Genealogy C	36	1017	3
Geographical Information System	52	1174	11
Hospital	65	1923	9
Hotel	47	1888	10
Insurance	63	1417	10
Inventory Management	44	1753	7
Library	42	1595	10
Mail Order System- Client Order	38	1895	8
Manufacturing Plant Controller	84	3089	11
Pizza System	67	1555	9
Police System	64	3186	10
Political Entities	32	842	5
Real Estate	79	2071	8
Routes and Locations	127	2089	9
School	18	397	9
TelephoneSystem	38	1838	7
University System	32	1206	4
Vending Machine	97	1696	8
WarehouseSystem	83	2831	12

TABLE 6.2: Open-source systems umplified

Name	**Version**	**LOC**	**# of Classes**	**Reference**
JHotDraw [55]	7.5.1	82132	694	Yes
Weka [56]	3.7.13	278642	1370	No
Java Bug Reporting Tool[57]	1.0	2629	36	Yes
JEdit[58]	1.12	59699	234	Yes
FreeMaker[59]	2.3.15	39864	281	Yes
Java Financial Library[60]	1.6.1	1248	27	Yes
args4j[61]	2.0.30	2223	61	No

Chapter 7

Related Work

This chapter surveys previous work in Reverse engineering approaches generating UML. A common theme in much of this work is a choice between two approaches: static and dynamic analysis. These concepts have been presented in Chapter 2. The following section describes the literature review methodology. We then present the results of our findings and a comparison between the different approaches and our own approach.

7.1 Literature Review Methodology

This study has been undertaken as a systematic literature review based on the guidelines proposed by Kitchenham [64]. Key parts of this systematic literature review are presented in this thesis.

7.1.1 Research Questions

The main goal of this systematic review was to identify and classify different techniques for reverse engineering to UML. Specifically, we target the reverse engineering to UML of software systems by means of model transformations. The high-level research question addressed by this study is:

RQ1. What model transformation techniques and/or methodologies for reverse engineering to UML can be identified from the literature?

7.1.2 Search process

To search the databases the, a set of strings was created for each of the research questions based on keywords extracted from the research questions and augmented with synonyms. We designed a two-phase systematic review. In both phases, we first selected the related work using the search engines and cited references in the Table 1. Afterwards, we performed an analysis on the related work. In the second phase, we also conduct-ed a detailed review of a selected subset of initial results. To assure there is not already a literature review answering our research questions, in the first phase, we looked at existing surveys and literature review papers. In the second phase, we focused on studying the existing work on reverse engineering to UML.

The sources for the search were chosen such that they included journals and conferences focusing on software engineering and program comprehension.

The search resulted in an extensive list of potential papers. To ensure that all papers included in the review were related to the research questions, we defined detailed inclusion and exclusion criteria.

7.1.3 First Phase Queries

7.1.4 Second Phase Queries

7.1.5 Inclusion and exclusion criteria

Chapter 8

Conclusions and Contributions

In this thesis we presented our reverse engineering approach called Umplification and the corresponding tool, the Umplificator. Umplification is the process of transforming step-by-step a base language program to an Umple program that merges textual modeling constructs directly into source code.

We presented the evaluation results showing that our approach and its current implementation are effective and efficient enough to be applied in the future to real systems.

Key contributions of this work are expected to be the following:

1. The overall concept of umplification
2. An understanding of how umplification compares with other reverse engineering techniques (incrementality, minimal adjustment of code to prevent disruption)
3. The Umplificator tool itself
4. Case studies of Umplification, demonstrating strengths, weaknesses and opportunities, as well as hopefully demonstrating that the resulting system is easier to understand and has less code.
5. The Transformation that allows developers to easily extend and refine the umplification transformation rules.
6. Another important contribution is the comprehensiveness of our detection of associations and the additional refactoring required to comply with all the different types of associations.

7. Detection of associations (of all different types) and state machines in a body of code. There is little successful work in this area in the literature.

Major advantages of our work, as compared to other reverse engineering approaches, are the concept of incrementally, the ease of addition of mapping rules, and the preservation of the system in a textual format.

For the future, we plan to apply the approach to other open source systems, gradually increasing the ability of the Umplificator to obtain a higher and higher first-pass precision on new systems it encounters. We also will integrate the mapping rules for state machines and refine some of the existing rules to make them more maintainable.

Appendix A

Appendix

A.1 Source Code of the ATM software system

Listing A.1: class ATM

Bibliography

[1] CRuiSE. Umple API summary, . URL http://api.umple.org.

[2] Elliot J Chikofsky and James H Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Softw.*, 7(1):13–17, January 1990. URL http://dx.doi.org/10.1109/52.43044.

[3] Gerardo CanforaHarman and Massimiliano Di Penta. New Frontiers of Reverse Engineering. In *Future of Software Engineering (FOSE '07)*, pages 326–341. IEEE, May 2007. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.15. URL http://dl.acm.org/citation.cfm?id=1253532.1254728.

[4] A Forward, T C Lethbridge, and D Brestovansky. Improving program comprehension by enhancing program constructs: An analysis of the Umple language. *2009 IEEE 17th International Conference on Program Comprehension*, pages 311–312, 2009. ISSN 10636897.

[5] T.C. Lethbridge, A. Forward, and O. Badreddin. Umplification: Refactoring to Incrementally Add Abstraction to a Program. *Reverse Engineering (WCRE), 2010 17th Working Conference on*, 2010. ISSN 1095-1350.

[6] Timothy C Lethbridge, Gunter Mussbacher, Andrew Forward, and Omar Badreddin. Teaching UML using umple: Applying model-oriented programming in the classroom. *2011 24th IEEECS Conference on Software Engineering Education and Training CSEET*, pages 421–428, 2011. ISSN 10930175.

[7] O Badreddin, A Forward, and T.C. Lethbridge. Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity. In *SERA 2013*, pages 129–149. Springer, 2013. doi: 10.1007/

978-3-319-00948-3_9. URL http://dx.doi.org/10.1007/978-3-319-00948-3_9http://link.springer.com/chapter/10.1007/978-3-319-00948-3_9.

[8] Omar Badreddin, Andrew Forward, and TimothyC. Lethbridge. Exploring a Model-Oriented and Executable Syntax for UML Attributes. *Software Engineering Research, Management and Applications SE - 3*, 496:33–53, 2014.

[9] O Badreddin. A Manifestation of Model-Code Duality: Facilitating the Representation of State Machines in the Umple Model-Oriented Programming Language. 2012. URL http://www.citeulike.org/group/18117/article/12461309.

[10] Hamoud Aljamaan, Timothy C. Lethbridge, Omar Badreddin, Geoffrey Guest, and Andrew Forward. Specifying Trace Directives for UML Attributes and State Machines. In *International Conference on Model-Driven Engineering and Software Development*, pages 79–86, January 2014. ISBN 978-989-758-007-9. URL http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0004711500790086.

[11] CRuiSE. Umple online, . URL http://try.umple.org.

[12] CRuiSE. Umple Source Code Repository, . URL https://github.com/umple/Umple/.

[13] CRuiSE. Umple metamodel, . URL http://metamodel.umple.org.

[14] Omar Badreddin, Andrew Forward, and Timothy C. Lethbridge. A test-driven approach for developing software languages. In *Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on*, pages 225–234, Jan 2014.

[15] Matthias Biehl. Literature study on model transformations. *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*, 2010.

[16] Anneke Kleppe. Mcc: A model transformation environment. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 173–187. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-35909-8. doi: 10.1007/11787044_14. URL http://dx.doi.org/10.1007/11787044_14.

[17] Anneke G Kleppe. A language description is more than a metamodel. 2007.

[18] Marcus Alanen, Ivan Porres, Turku Centre, and Computer Science. A relation between context-free grammars and meta object facility metamodels. Technical report, 2003.

[19] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, July 2006. ISSN 0018-8670. doi: 10.1147/sj.453.0621. URL http://dx.doi.org/10.1147/sj.453.0621.

[20] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831 – 873, 2005. ISSN 0747-7171. doi: http://dx.doi.org/10.1016/j.jsc.2004.12.011. URL http://www.sciencedirect.com/science/article/pii/S0747717105000349. Reduction Strategies in Rewriting and Programming special issue.

[21] Jean Bzivin, Frdric Jouault, and Patrick Valduriez. On the need for megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004. URL http://www.sciences.univ-nantes.fr/lina/atl/www/papers/OOPSLA04/bezivin-megamodel.pdf.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994. ISBN 9780321700698. URL http://books.google.ca/books?id=6oHuKQe3TjQC.

[23] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. Modisco: A generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0116-9. doi: 10.1145/1858996.1859032. URL http://doi.acm.org/10.1145/1858996.1859032.

[24] Kun Ma, Bo Yang, and Haiyang Wang. A formalizing hybrid model transformation approach for collaborative system. In *Computer Supported Cooperative Work in Design (CSCWD), 2010 14th International Conference on*, pages 71–76, April 2010. doi: 10.1109/CSCWD.2010.5472000.

[25] Tom Mens, Pieter Van Gorp, Dniel Varr, and Gabor Karsai. Applying a model transformation taxonomy to graph transformation technology. *Electronic Notes*

in Theoretical Computer Science, 152(0):143 – 159, 2006. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2005.10.022. URL http://www.sciencedirect.com/science/article/pii/S1571066106001447. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005) Graph and Model Transformation 2005.

[26] Mikaël Peltier, Jean Bézivin, and Gabriel Guillaume. Mtrans: A general framework, based on xslt, for model transformations. Citeseer.

[27] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Sci.Comput.Program.*, 72(1-2):31–39, June 2008. URL http://dx.doi.org/10.1016/j.scico.2007.08.002.

[28] Dan Li, Xiaoshan Li, and Volker Stolz. Model querying with graphical notation of qvt relations. *SIGSOFT Softw. Eng. Notes*, 37(4):1–8, July 2012. ISSN 0163-5948. doi: 10.1145/2237796.2237808. URL http://doi.acm.org/10.1145/2237796.2237808.

[29] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model driven language engineering with kermeta. In *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, GTTSE'09, pages 201–221, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 3-642-18022-1, 978-3-642-18022-4. URL http://dl.acm.org/citation.cfm?id=1949925.1949931.

[30] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. Polack. The epsilon transformation language. In *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, ICMT '08, pages 46–60, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-69926-2. doi: 10.1007/978-3-540-69927-9_4. URL http://dx.doi.org/10.1007/978-3-540-69927-9_4.

[31] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006. ISSN 01676423. URL http://dl.acm.org/citation.cfm?id=1149670.1149672.

[32] . *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[33] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001. ISBN 0072496681.

[34] H Osman and M R V Chaudron. An Assessment of Reverse Engineering Capabilities of UML CASE Tools. In *2nd Annual International Conference Proceedings on Software Engineering Application*, pages 7–12, 2011.

[35] Andrew Forward and Timothy C. Lethbridge. Problems and opportunities for model-centric versus code-centric software development. In *Proceedings of the 2008 international workshop on Models in software engineering - MiSE '08*, page 27, New York, New York, USA, May 2008. ACM Press. ISBN 9781605580258. doi: 10.1145/1370731.1370738. URL http://dl.acm.org/citation.cfm?id=1370731.1370738.

[36] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akit and Satoshi Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-63089-0. doi: 10.1007/BFb0053381. URL http://dx.doi.org/10.1007/BFb0053381.

[37] Jitender Kumar Chhabra and Varun Gupta. Evaluation of object-oriented spatial complexity measures. *SIGSOFT Softw. Eng. Notes*, 34(3):1–5, May 2009. ISSN 0163-5948. doi: 10.1145/1527202.1527208. URL http://doi.acm.org/10.1145/1527202.1527208.

[38] Russell Bjork. Java ATM System, . URL http://www.cs.gordon.edu/courses/cs211/ATMExample/index.html.

[39] Russell Bjork. Java ATM System, . URL http://www.site.uottawa.ca/~mgarz042/thesis/ATM/src.

[40] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003. ISBN 0321179366.

[41] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Muller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 4(4):501–520, 1994.

[42] TXL Resources. Java Grammar in TXL. URL http://www.txl.ca/nresources.html.

[43] Frdric Jouault, Freddy Allilaire, Jean Bzivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(12):31 – 39, 2008. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2007.08.002. URL http://www.sciencedirect.com/science/article/pii/S0167642308000439. Special Issue on Second issue of experimental software and toolkits (EST).

[44] Object Management Group. XML Metadata Interchange (XMI). URL http://www.omg.org/spec/XMI/.

[45] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321331885.

[46] Matthew Stephan and Andrew Stevenson. A comparative look at model transformation languages. *Software Technology Laboratory at Queens University*, 2009.

[47] Eclipse Marketplace. Java Metamodel. URL http://goo.gl/YV3p83.

[48] Paul Browne. *JBoss Drools Business Rules*. Packt Publishing, April 2009. ISBN 1847196063, 9781847196064. URL http://dl.acm.org/citation.cfm?id=1611309.

[49] Raymond P.L. Buse and Westley R. Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis - ISSTA '08*, page 121, New York, New York, USA, July 2008. ACM Press. ISBN 9781605580500. doi: 10.1145/1390630.1390647. URL http://dl.acm.org/citation.cfm?id=1390630.1390647.

[50] Eclipse. Eclipse Java development tools (JDT). URL http://projects.eclipse.org/projects/eclipse.jdt.

[51] Eclipse Documentation. ASTNode API. URL http://goo.gl/yLt2CF.

[52] Ding Xiao and Xiaoan Zhong. Improving Rete algorithm to enhance performance of rule engine systems. In *2010 International Conference On Computer Design and*

Applications, volume 3, pages V3–572–V3–575. IEEE, June 2010. ISBN 978-1-4244-7164-5. doi: 10.1109/ICCDA.2010.5541368. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5541368.

[53] R.J.K. Jacob and J.N. Froscher. A software engineering methodology for rule-based systems. *Knowledge and Data Engineering, IEEE Transactions on*, 2(2):173–189, Jun 1990. ISSN 1041-4347. doi: 10.1109/69.54718.

[54] CRuiSE. Umple Examples, . URL https://code.google.com/p/umple/wiki/Examples.

[55] Erich Gamma and Thomas Eggenschwiler. JHotDraw. URL http://www.jhotdraw.org/.

[56] The University of Waikato. Weka Repository. URL https://svn.cms.waikato.ac.nz/svn/weka/.

[57] cipov. P. Java Bug Reporting Tool. URL https://code.google.com/p/jbrt/.

[58] SourceForge. jEdit, . URL http://sourceforge.net/projects/jedit/.

[59] SourceForge. FreeMaker, . URL http://freemarker.org/.

[60] Freshmeat. Java Financial Library. URL http://freecode.com/projects/jfl/.

[61] Kawaguchi. K. args4j. URL https://github.com/kohsuke/args4j.

[62] William B. Frakes and Ricardo Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN 0-13-463837-9.

[63] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software. *ACM SIGKDD Explorations Newsletter*, 11(1):10, November 2009. ISSN 19310145. URL http://dl.acm.org/citation.cfm?id=1656274.1656278.

[64] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.