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

*"Live as if you were to die tomorrow. Learn as if you were to live forever."*

Mahatma Gandhi

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. We implemented automated umplification in a tool called the Umplificator. The tool is rule-driven: code, including Umple code, is parsed and processed into an internal representation, which is then operated on by rules; transformed textual code and model, in Umple, is then generated. The rules used to transform code to model have been iteratively refined by using the tool on a variety of open-source software systems.

The thesis consists of three main parts. The first part (Chapters 1 and 2) present the research questions and research methodology, as well as introducing Umple and the background necessary to understand the rest of the thesis. The umplification method is presented at increasing levels of detail through Chapters 3 and 4. Chapters 5 and 6 present the tool and evaluation of our approach, respectively. An analysis of related work, and comparison to our own, appears in Chapter 7.

# *Acknowledgements*

Foremost, I would like to express my sincere gratitude to my dear supervisor, Professor Timothy Lethbridge, for guiding me during my graduate studies. I greatly appreciate his vast knowledge, dedication and assistance in writing proposals, scholarship applications, papers and this thesis. Throughout my studies, Tim contributed to my development as a researcher, professor and as a person.

Besides my supervisor, I would also like to thank the members of my PhD committee for the feedback provided at the different stages or this research.

My sincere thanks to Professors Daniel Amyot and Liam Peyton for the continuous encouragement to my work. More than my instructors, they have been mentors who helped shape my career path.

I thank each of the members of the Complexity Reduction in Software Engineering (CRuiSE) research group. Particular and well deserved thanks to Hamoud Aljamaan for all of his constructive feedback.

I wish to deeply thank the University of Ottawa for being my second home for the past 10 years and for offering me all the support required to carry out my research work.

I would also like to thank my whole family for the support they provided me through my entire life. In particular, I would like to thank my mother Patricia, my brothers Laura and Sebastian for their love and encouragement during this busy period of my life.

Last, but not least, I would like to thank my wife Idalia for her understanding and love during the past few years and for her contribution on the graphics of this thesis.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In this thesis, we present the design and analysis of a reverse-engineering technique called Umplification. Our approach makes it possible to incrementally refactor existing programs so that textual Umple abstractions come to be incorporated into the code. Such programs can be maintained using visual UML(Unified Modeling Language) modeling tools or text editors. Umplification hence eliminates the distinction between modeling and programming since the end-product of umplification is not a separate model, but is an incremental change to the code/model.

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, 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 thesis 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. *Umplification* comes from the play on words with the concept of 'amplification' and also the notion of converting into Umple. Earlier work from our research group [5] found that umplifying code is reasonably straightforward for someone familiar with Umple, and with knowledge of UML modeling pragmatics. Moreover, manual umplification of several systems, including the Umple compiler itself, has been performed by other members of our research group. The present thesis focuses on how the umplification process can be performed automatically using reverse engineering technology. This thesis presents an automated approach for detection and transformation of Umple constructs.

In the next section, we state the research problem addressed by this thesis and list the research questions. Then, we present the methodology that we will follow in order to answer our research questions.

## 1.1 Problem Statement

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 partially 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. We present an approach that automatically extracts the model from an object-oriented system in the form of Umple code.

## 1.2 Research Questions

The guiding research questions that have motivated our research are the following:

**RQ1. To what extent can we achieve automated umplification?** Although, in previous work, other members of our research group have found that the process of extracting an Umple model from source code can be relatively easily performed if done manually; our investigation focuses on performing the process automatically by means of a reverse engineering technology. In this thesis, we address this research question by exploring technology alternatives, developing and refining a tool, and then evaluating and assessing the effectiveness of the products of the tool. The latter is described in Chapter 6.

**RQ2. What transformation technology and transformations will work effectively for umplification?** In the course of investigating RQ1, we explore various transformation technologies with the purpose of umplifying a software system. During the initial stages of this work [6] XML-based solutions were explored to implement the reverse engineering capabilities. Alternative approaches are evaluated in this thesis. In Chapter 5, we evaluate ATL and TXL to see to which extent they could fulfill our needs.

**RQ3. What should be the architecture, design and implementation of an umplification tool?** In Chapter 5, we discuss all the design decisions and propose a set of tools and technologies that our tool uses.

**RQ4. What would be an effective process for improving the accuracy of the umplification tool?** To answer this question we will explore an iterative process that can refine the effectiveness of automated umplification. The results of our evaluation are presented in Chapter 6.

## 1.3 Hypothesized Solutions

Our main hypothesis is stated as follows.

**H1: Automated umplification can be achieved on a wide variety of systems.** Our investigations focus on the automation of the umplification process. As we will see later, our evaluation indicates that the approach can be applied on a wide variety of systems such as small-sized and medium-sized open source projects.

This hypothesis is investigated throughout our research activities.

## 1.4 Research Methodology and Activities

This thesis follows the design science methodology [7], in which research is done through the process of building and evaluating products. During our research we follow the five process steps of the design cycle. The knowledge obtained by constructing and evaluating the artifacts is used as input for a better understanding of the problem.

The major steps in the methodology that were conducted to address our research questions and verify our hypothesis are stated below:

1. *Awareness of the problem*: Manually perform umplification to gain an understanding of what will be needed;

2. *Suggestion*: Propose a technique and a set of heuristics to perform automated umplification of Umple attributes and associations;

3. *Development*: 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;

4. *Evaluation*:

   - 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;

   - Iteratively develop more and more transformations to convert additional Base Language code into Umple. Introduce additional case studies until the Umplificator works well on 5-10 reasonably large open-source systems. Java will be the first targeted language in our exploration;

5. *Conclusion* – Compare the work to alternative approaches and outline the areas for future research.

## 1.5 Thesis Contributions

The major research contributions of this thesis are:

- The overall concept of automated umplification;
- An understanding of how umplification compares with other reverse engineering techniques;
- The Umplificator tool itself;
- Case studies of Umplification, demonstrating strengths, weaknesses and opportunities;
- Mapping rules for Umplification and the language for expressing these. These should be general-purpose and easily modifiable to allow future researchers, and even end users, to add to them;
- Detection of associations in a body of source code written in an object-oriented programming language.

## 1.6 Thesis Outline

The remainder of the thesis is organized as follows:

**Chapter 2** presents background research, a brief introduction to Umple and its modeling constructs. Covered in this chapter are the concepts related to reverse engineering and modeling transformations.

**Chapter 3** presents umplification in detail, the core concept of this thesis.

**Chapter 4** presents the mechanisms allowing us to detect Umple constructs. In particular, we present the mapping rules derived from our analysis.

**Chapter 5** presents two 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 well-known 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** presents a multi-stage approach to validate our implementation. Three major case studies, umplification of JHotDraw, Weka and args4j, are presented in order evaluate the effectiveness and efficiency of our approach.

**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** summarizes our research 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 programs 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 [8]. 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 [9] Umple generates many methods to add, delete and query the links of associations.

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 [10]. Umple generates public getter and setter methods, leaving the underlying variable private.

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 [11]. Umple generates event methods, and other complex code to handle all aspects of state transitions, actions, and activities.

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 [12].

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 in the Umple user manual [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 [13], the command line compiler or an Eclipse plugin. The umplification method discussed in this thesis currently focuses on attributes and associations with some generation of code injections. Umplification of state machines is left for future work. 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 is a 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 do not 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 reference 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 Listing 2.2. The definition of rule *classContent* is shown in Listing 2.3.

Listing 2.1: Grammar for Umple classes

```
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 Listing 2.1 'class' is a terminal that must match exactly, as are the open and close curly brackets. Note that the regular parentheses are metacharacters used for grouping, and the asterisk 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

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

The grammar to parse classes (declarations), attributes, and associations is presented in Listings 2.3-2.5 respectively. The basic properties of attributes and the special kinds of associations are presented in Sections 2.1.2 and 2.1.3.

A class definition starts with a name, followed by a curly bracket and any of the items in Lines 2-6 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]? ;
// gpIdentifier = general parameter identifier
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 [14], 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 partial view of the core classes in the metamodel. 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 [15].

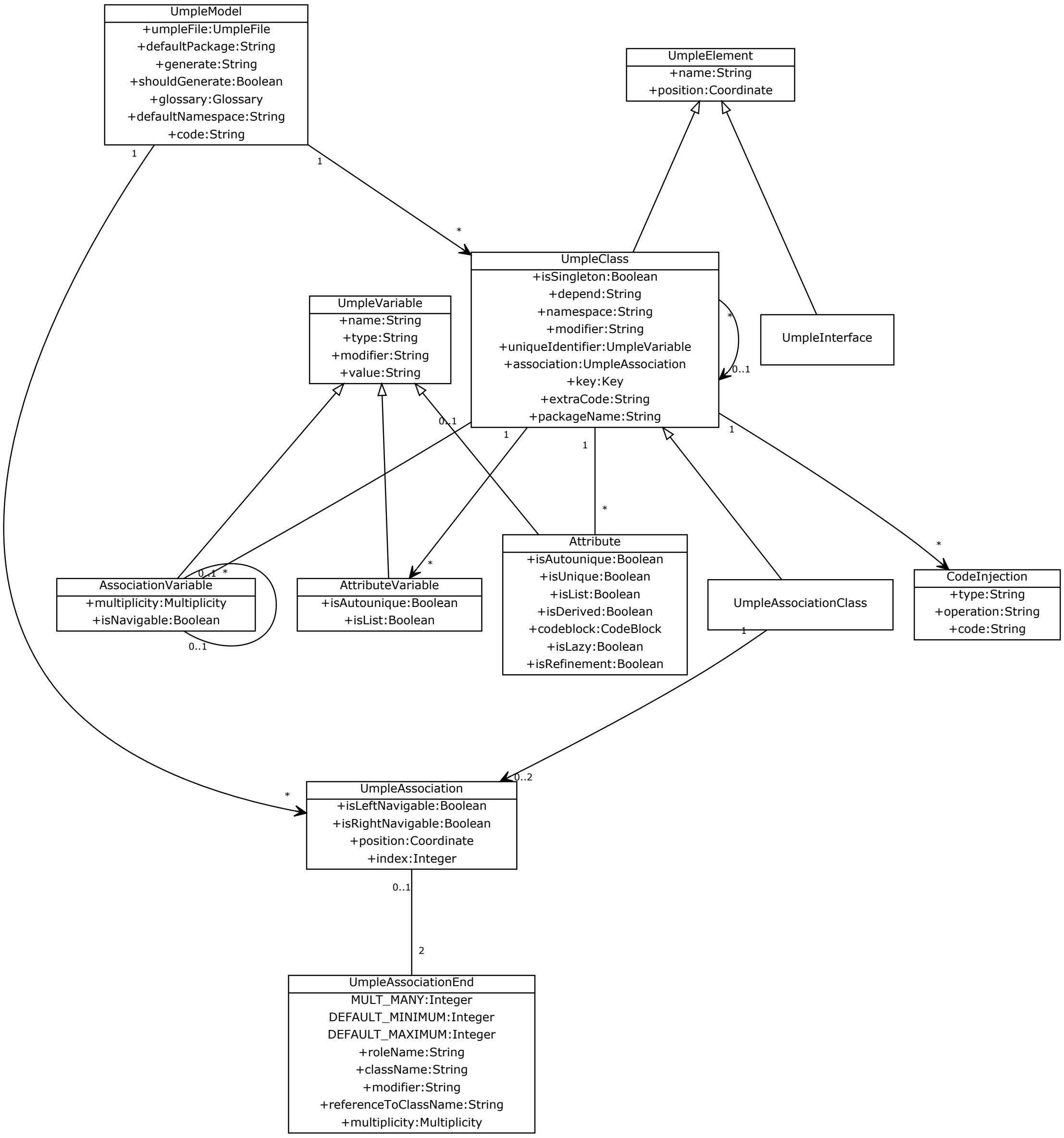


FIGURE 2.1: Umple metamodel (partial)

### 2.1.2 Umple Attributes

An Umple attribute represents some information held by a class. 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 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 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 cannot 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 be 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 declared as a lazy immutable attribute. The setter method of this attribute can be called 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 does not possess any control over the creation of the objects and therefore he cannot 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;<br>if T is boolean | No |
| **Initialized** | Yes | Yes;<br>if T is boolean | No |
| **Lazy** | Yes | Yes;<br>if T is boolean | No |
| **Defaulted** | Yes | Yes;<br>if T is boolean | No |
| **Immutable** | Yes | Yes;<br>if T is boolean | No |
| **Lazy immutable** | Yes | Yes;<br>if T is boolean | No |
| **Autounique** | Yes;<br>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

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;<br>only once. | No |
| **Autounique** | No | No |
| **Constant** | No | No |
| **Internal** | No | No |
| **Key** | Yes | No |

at run time between the different instances of the classes. More specifically, an Umple association is composed of the following information:

- *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 but absent.

The code segment in Listing 2.14 illustrates an association between instances of classes *School* and *Student*. 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;
}
```

Furthermore, Umple supports various kinds of associations as summarized below.

**Reflexive** An association in which both associations ends connect to the same class [16]. In other words, the class is associated to itself. In Line 2 of Listing 2.16, the association is defined between Employee and itself, through the manager/manages role. The relationship means that an Employee in the role of manager can be associated with many managed Employees.

Listing 2.16: A Reflexive association

```
class Employee {
0..1 manager -- * Employee manages ;
}
```

**Compositions** A composition [16] is a sub-type of association where there is mandatory delete on the composition end, regardless of the multiplicity.

The association in Listing 2.17 specifies that a Folder could contain many files, while each File has exactly one Folder parent. The composition constraint states that if a Folder (instance) is deleted, all contained Files (instances) are deleted as well.

Listing 2.17: A Composite association

```
class Folder {
   1 parent <@>-* File files;
}
class File {}
```

**External** To associate a class to another class (external) that is not present or needs to be omitted from the compilation. External classes are useful when writing Umple code that will be connected to code written separately, such as code in a GUI library. In Listing 2.18, class A is associated to class B which is not part of this model. Class B is shown as an external reference.

Listing 2.18: External associations

```
class A {
  1 -- 0..1 B;
}
external B {}
```

**Association Classes** Association classes [16] define the features that do not belong to any of the connected classes but rather to the association itself. That is, a shorthand for two one-to-many associations to a class that contains data that relates the two classes. The two related classes are logically in a many-many relationship. For instance, the association class in Listing 2.19 (Lines 7-11) is used to gather the number of points a player gained on a particular team.

Listing 2.19: An association class

```
class SportsPlayer {
  name;
}
class Team {
  name;
}
associationClass PlayerInTeam {
  Integer points;
  * SportsPlayer player;
  * Team team;
}
```

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 using **after** statements to enforce postconditions or take actions based on the results of other actions. Code injections (after and before statements) can be added into the constructor, generated state machine event methods, 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 2.20: A code injection into the constructor

```
class Operation {
  const Boolean DEBUG=true;
  query;
  before constructor {
    if (aQuery == null)
    {
      throw new RuntimeException("Please provide a valid query");
    }
  }
  after constructor {
    if (DEBUG) { System.out.println("Created " + query); }
  }
}
```

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 2.20 generates the following (Java) constructor:

Listing 2.21: Generated constructor after code injection

```java
public Operation(String aQuery)
{
  if (aQuery == null)
  {
    throw new RuntimeException("Please provide a valid query");
  }
  query = aQuery;
  if (DEBUG) { System.out.println("Created " + query); }
}
```

### 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 include: 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 (with code in other languages mixed in), 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 (instances of the metamodel classes). 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, Ecore, Yuml; 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 [12], generation templates, patterns, aspects and concurrency can also be generated from models.

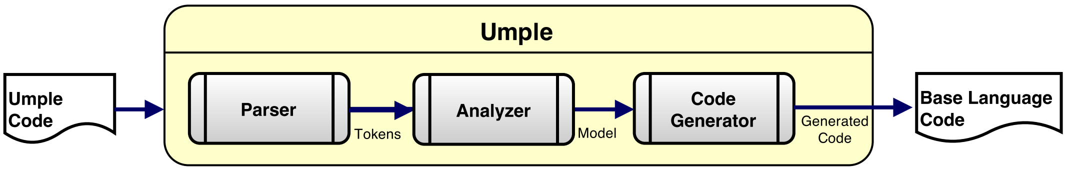
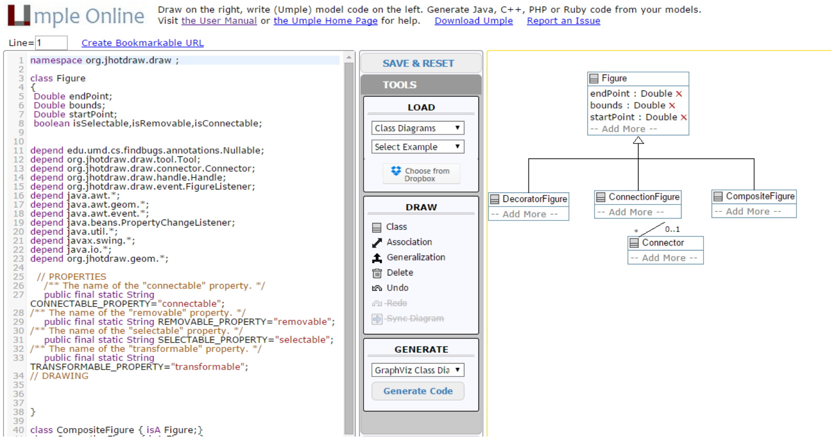


FIGURE 2.2: Umple architecture

Each component is tested independently to ensure that the input is processed correctly and the output produced is valid [17]. 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.

In addition to the Eclipse plugin and command-line based compiler, UmpleOnline [13], a web-based application shown in Figure 2.3, allows to instantly experiment with Umple on the Web.


Draw on the right, write (Umple) model code on the left. Generate Java, C++, PHP or Ruby code from your models.
Visit the User Manual or the Umple Home Page for help. Download Umple Report an Issue

Line=1 Create Bookmarkable URL

```
namespace org.jhotdraw.draw ;

class Figure
{
 Double endPoint;
 Double bounds;
 Double startPoint;
 boolean isSelectable,isRemovable,isConnectable;


depend edu.umd.cs.findbugs.annotations.Nullable;
depend org.jhotdraw.draw.tool.Tool;
depend org.jhotdraw.draw.connector.Connector;
depend org.jhotdraw.draw.handle.Handle;
depend org.jhotdraw.draw.event.FigureListener;
depend java.awt.*;
depend java.awt.geom.*;
depend java.awt.event.*;
depend java.beans.PropertyChangeListener;
depend java.util.*;
depend javax.swing.*;
depend java.io.*;
depend org.jhotdraw.geom.*;

  // PROPERTIES
    /** The name of the "connectable" property. */
    public final static String
CONNECTABLE_PROPERTY="connectable";
/** The name of the "removable" property. */
    public final static String REMOVABLE_PROPERTY="removable";
/** The name of the "selectable" property. */
    public final static String SELECTABLE_PROPERTY="selectable";
/** The name of the "transformable" property. */
    public final static String
TRANSFORMABLE_PROPERTY="transformable";
// DRAWING



}

class CompositeFigure { isA Figure;}
```

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 (even in an agile approach, these would occur, though in tight iterations). 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 2.4 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:

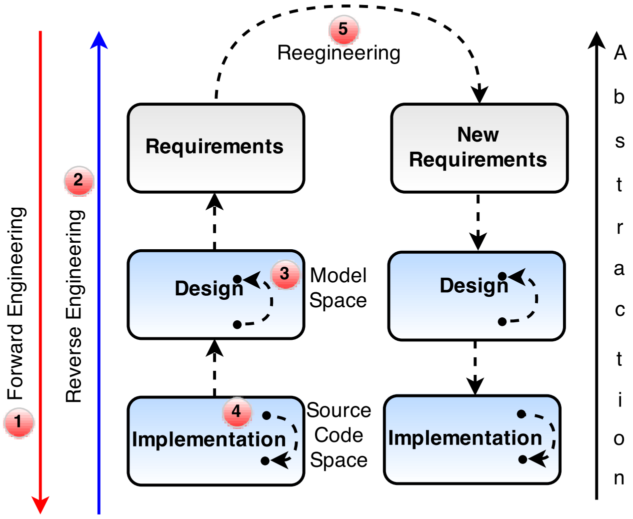


Figure 2.4: Transformations across the different software life-cycle phases

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 has 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. 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 and related code:** 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 variables, collections, and related code:** 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 constraints to Exceptions or other types of code:** Model constraints expressed in the model in *OCL* or in natural language are mapped to code that checks the preconditions, postconditions and invariants in the target programming language.

**Mapping Object models to a persistent storage schema:** Object model is 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:

> "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 is generally used to:

- Cope with complexity: understand large and complex systems.
- Generate alternative views: automatically generate different ways to view systems
- Recover lost information: extract what changes have been made and the reasons for those changes.
- Detect side effects: help understand ramifications of changes.
- Synthesize higher abstractions: create alternative views that transcend to higher abstraction-level software.
- Facilitate reuse: detect candidate reusable artifacts and components.

The umplification process can be used to do all of these, but in particular to synthesize higher abstractions.

Table 2.5 summarizes another categorization of reverse engineering approaches [2]. Umplification includes design recovery, restructuring, refactoring, and a certain amount of data re-engineering.

Table 2.5: Summary of Reverse engineering approaches [2]

| Approach | Description | Related Techniques |
|---|---|---|
| Re-documentation | 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 | The transformation from one 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 | 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 | Small scale restructuring | 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 [18], 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. [19] 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 (entity-relationship) 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 [20], 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 correctness) and are tree based (e.g., parse trees are generated from them). Also, there are a great variety of 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 [20]. Nevertheless, as studied by Alanen et al. [21] 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 [18, 22] on model transformations allows us to conclude that no existing model transformation tool or technique 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 [18]:

- Change of abstraction: Model transformations can change the level of abstraction between the source and the target model. That is, the transformation increases or decreases the level of detail or leaves it unchanged.
- Change of Metamodel:

  - In an *endogenous* [23] transformation, both the source and target metamodels are the same.
  - In an *exogenous* [23] transformation, the source and target metamodels are different.

- Supported technical spaces: Model transformations can operate between the metametamodel, metamodel and model levels [24].
- Supported number of models: A transformation can involve one model (same source model resulting in 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 creates its target as a set of strings while a Model-to-Model transformation creates 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 models have a common property that is not transformed by the transformation [18]. 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 transformation approaches into two major categories [22]: 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 [25].

**Template-Based Approaches:** Template-based approaches are used in the implementation of code generators. The basic idea is to refine and transform 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 metaprograms). This approach can be combined with the visitor-based approach for model traversing.

#### 2.2.3.2 Model-To-Model Approaches

**Direct-Manipulation Approaches:** The approach typically consists of an internal representation of the model (e.g. AST) and some API's to manipulate and query it. The Modisco [26] 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. The *OptimaJ* framework 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 provide 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 transformation 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 do not describe how the transformation 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 [27].

**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 [28]. For instance, the Triple Graph Grammar (TGG) [28] 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 transformed using XSLT or similar technology. However, the use of XMI and XSLT has scalability limitations [29].

#### 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 [30] 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 [31] 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, an XMI-based metamodel and visual syntax for matching element between metamodels. Listing 2.22 shows a partial transformation from a UML class model to an Umple model.

Listing 2.22: A basic QVT transformation

```
transformation uml2Umple(
 in uml : SimpleUML,
 out umple : SimpleUmple
);

main() {
  uml.objectsOfType(Class)->map UMLClassToUmpleClass();
}

mapping Class::classToUmpleClass () : UmpleClass
{
  name := self.name;
  attributes : self.attributes->map attributeToUAttribute();
}
```

```
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 2.23 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 2.23: A basic JET Template

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

To pass arguments to the template method, we use the generate *method* as shown in Listing 2.24. Note that it is possible to pass a reference (a model element) type as argument.

Listing 2.24: Instantiating the BasicTemplate class

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

**Kermeta** Kermeta [32] 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 [33] 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 supports EMF models.

**TXL** TXL [34] 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.

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 |

### 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 [35]. 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 2.25. The resulting source code after the transformations is presented in Listing 2.26. The refactorings are performed one by one, ensuring that the refactoring does not change the intended behavior of the system. For a complete catalog of refactorings refer to [35].

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 2.25: Before refactorings

```java
public class Student {
  private String name;
  public Student(String name) {
   this.name = name;
  }
  public String getName() {
 return name;
  }
}
///----- Class Mentor ------
public class Mentor {
  private String name;
  public Mentor(String name) {
   this.name = name;
  }
  public String getName() {
 return name;
  }
}
```

Listing 2.26: After refactorings

```java
///----- SuperClass ------
public class Person {
  private String name;
  public Person(String name) {
   this.name = name;
  }
  public String getName() {
 return name;
  }
}
///----- Class Student ------
public class Student extends
    Person {
  private String name;
  public Student(String name) {
   super(name);
  }
}
///----- Class Mentor ------
public class Mentor extends
    Person {
  private String name;
  public Mentor(String name) {
   super(name);
  }
}
```

### 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 [36].

**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.

## 2.3 Summary

Umple is a modeling and programming language that incorporates UML concepts. We have focused our discussion about Umple classes, attributes, associations and code injections. This chapter analyzed the syntax of attributes and associations and presented the code generation patterns when these Umple constructs are translated into Java. Furthermore, we provided an overview of the tools currently available to support the creation of Umple systems; as well as the Umple Grammar, metamodel and architecture.

A wide range of reverse engineering and modeling transformation techniques and tools were introduced in this chapter. Both are required to understand 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. The umplification technique will be described in the next chapter.

# 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 technique 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.

The purposes of umplification include a) allowing the user to better understand their system, since the model becomes visible; b) reducing code volume by generating code that would otherwise be repetitive; c) allowing the developer to switch to model-driven development from that point onwards, and to use the modeling abstractions found in Umple more fully.

### 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.

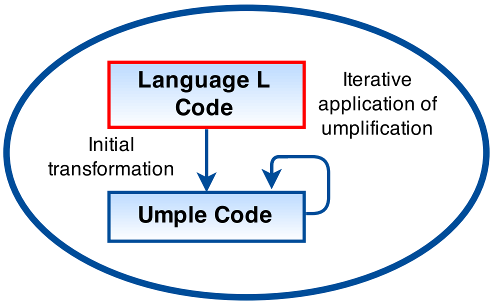


Figure 3.1: The Umplification process generalized

### 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. From a technical perspective, incrementality is reflected in the fact that the tool implementing our approach accepts Umple as input. That is, an Umple model with a higher level of abstraction can be recovered from an Umple model containing fewer umplified constructs.

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 the set of transformation rules it uses 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 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++) 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 code identical 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. We will not cover this aspect of umplification further in this thesis, and will leave the focus on attributes and associations.

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.

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. |

Mutator and accessor methods are detected in source code based on the naming conventions and semantics of the declarations. 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, whereas pure Java examples have solid borders with green shading.

#### 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 relationship 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 1 and 2 (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. In a context in which a field, being part of the association (i.e., one of the associations ends), references a class whose code is not available, the class is declared as *external.* Refer to section 2.1.3 for more details on *external* associations. This is beneficial in a context where the software makes heavy use of a framework for which only binary code is available. 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 Section 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 [37], 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 class diagram. This allows developers to maintain the essential 'familiarity' with their code as they gradually transform it into Umple [38].

### 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 [39] some evidence that reducing the code volume can help to improve program comprehension.

Umplification improves program comprehension by:

- Allowing developers to describe and develop a system at a more abstract level.
- 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 will now illustrate the transformations steps through a small example. This will show how umplification is performed manually. In Section 5.4 we will show how to perform automated umplification. The original Java is presented through Listings 3.1 - 3.3. To help distinguish between Umple and Java code, the Umple examples use boxes with light-blue shading, and pure Java examples use boxes with light-green shading.

Listing 3.1: Person.java

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

Listing 3.2: Student.java

```
package university;

public class Student extends 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 3.3: Mentor.java

```
package university;
import java.util.Set;

public class Mentor extends Person{

 Mentor() {}
 public Set<Student> students;
 public Set<Student> getStudents() {
  return students;
 }
 public void setStudents (Set<Student>students) {
  this.students = students;
 }
 public void addStudent( Student aStudent){
  students.add(aStudent);
 }
 public void removeStudent(Student aStudent) {
  students.remove(aStudent);
 }
 public String toString() {
      return(
          (name==null ? " " : name) + " " +
          students.size()+ " students"
      );
 }
}
```

### 3.3.1 School System: Initial Transformation

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

As discussed earlier, the dependency, package and generalization notations are changed to their respective Umple notations. For instance, the Java code of class Mentor shown in Listing 3.3 would result in Umple implementation shown in Listing 3.4 (in file Mentor.ump). The following gives details of the initial transformation:

- The package declaration in 3.3 becomes an Umple namespace declaration in Listing 3.4.
- The import declaration in 3.3 becomes an Umple depend declaration in Listing 3.4.
- The generalization notation in 3.3 is transformed to the 'isA' notation (Line 5) in Listing 3.4.
- Code in lines 6-26 in Listing 3.3 is not touched by the initial transformation; 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 yet been reverse-engineered.

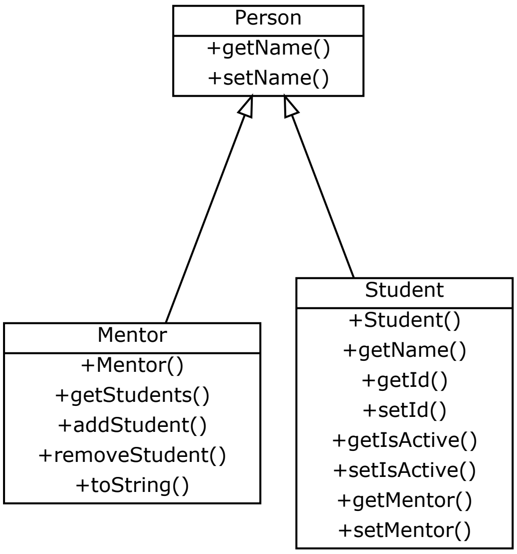


FIGURE 3.2: UML class diagram of the Mentor-Student example - Level 1

### 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   isA Person;
5   Integer id;
6   lazy Boolean isActive;
7   immutable name;
8   const Integer MAX_PER_GROUP = 10;
9   after getName {
10   if (name == null) {
11    throw new RuntimeException("Error");
12   }
13  }
14   public Mentor mentor;
15   public Mentor getMentor() {
16    return mentor;
17   }
18   public void setMentor(Mentor mentor) {
19    this.mentor = mentor;
20   }
21 }
```

The following gives details of Listing 3.7:

- Line 5: Field id becomes an Umple attribute. Getter getId() and setter setId() are removed.

- Line 6: Field isActive becomes an Umple attribute of Boolean type. As the field is not required in the constructor, we mark it as ‘lazy’ so the Umple compiler does not generate a constructor argument for this attribute.

- Line 7: 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 3.2) does not contain code that sets this variable anywhere, therefore it is transformed into an immutable Umple attribute.

- Line 8: 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) or because of the ALL_CAPS convention.

- Line 9-13: As the getter for field name was custom in Listing 3.2, we have adapted it so it conforms to the code that is generated by the Umple compiler. A code injection has been used for this purpose. In this case, the code will inject two lines to be executed before the getName() method returns.

- Code in Lines 51-55 in Listing 3.2 is not touched by this transformation; The same exact code is found in the Umple file in Listing 3.7 (Lines 14-20).

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

```
namespace university;
class Person {
 String name;
}
```

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.

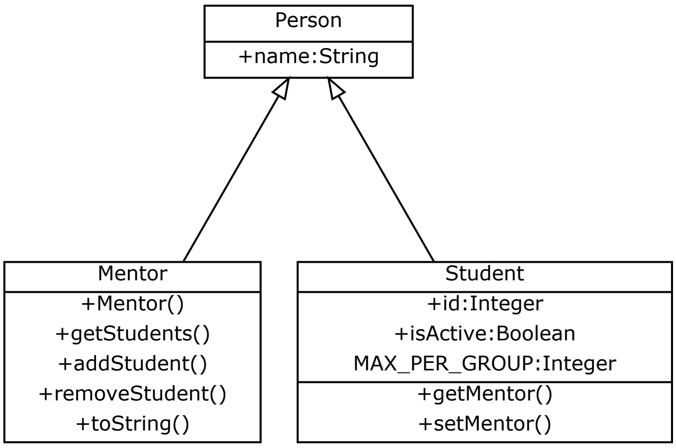


Figure 3.3: UML class diagram of the Mentor-Student example - Level 2

### 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

```
namespace university;
class Student {
 Integer id;
 lazy Boolean isActive;
 immutable name;
 const Integer MAX_PER_GROUP = 10;

 after getName {
  if (name == null) {
   throw new RuntimeException("Error");
  }
 }
}
```

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

```java
public Student(Mentor aMentor){
  mentor = aMentor;
}
```

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.

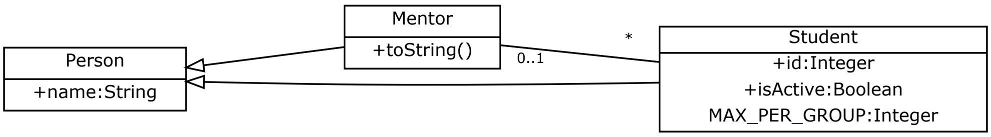


Figure 3.4: UML class diagram of the Mentor-Student example - Level 3

Finally, for this small system, the Umple code extracted (33 LOC) contains substantially fewer lines of code than 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 [40].

## 3.4 ATM System - Manual Umplification Example

In this section, we will manually umplify a second, 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 [41]. 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 [42]), and paste it into UmpleOnline [13] 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 from this thesis 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

```java
package banking;

public class Card
{
 private int number;

 public Card(int number)
 {
  this.number = number;
 }

 public int getNumber()
 {
  return number;
 }
}
```

Listing 3.13: AccountInfo.java

```java
public class AccountInformation
{
 public static final String [] ACCOUNT_NAMES =
    { "Checking", "Savings", "Money Market" };

 public static final String [] ACCOUNT_ABBREVIATIONS =
    { "CHKG", "SVGS", "MMKT" };
}
```

Listing 3.14: Status.java

```java
package banking;

public abstract class Status
{
 public String toString()
 {
  if (isSuccess())
   return "SUCCESS";
  else if (isInvalidPIN())
   return "INVALID PIN";
  else
   return "FAILURE " + getMessage();
 }

 public abstract boolean isSuccess();
 public abstract boolean isInvalidPIN();
 public abstract String getMessage();
}
```

Listing 3.15: Receipt.java

```java
package banking;

import atm.ATM;
import atm.transaction.Transaction;
import java.util.Date;
import java.util.Enumeration;

public abstract class Receipt
{

 private String [] headingPortion;
 protected String [] detailsPortion;
 private String [] balancesPortion;

 protected Receipt(ATM atm, Card card, Transaction transaction, Balances  balances)
 {
  // Code omitted
 }

 public Enumeration getLines()
 {
  // Code omitted
 }
}
```

Listing 3.16: Balances.java

```java
package banking;

public class Balances
{
 private Money total;
 private Money available;

 public Balances(){}

 public void setBalances(Money total, Money available)
 {
   this.total = total;
   this.available = available;
 }

 public Money getTotal()
 {
  return total;
 }

 public Money getAvailable()
 {
  return available;
  }
}
```

Listing 3.17: Money.java

```java
package banking;

public class Money
{
 private long cents;

 public Money(int dollars)
 {
  this(dollars, 0);
 }

 public Money(int dollars, int cents)
 {
  this.cents = 100L * dollars + cents;
 }

 public Money(Money toCopy)
 {
  this.cents = toCopy.cents;
 }

 public String toString()
 {
  return "$" + cents/100 +
  (cents %100 >= 10  ? "." + cents % 100 : ".0" + cents % 100);
 }

 public void add(Money amountToAdd)
 {
  this.cents += amountToAdd.cents;
 }

 public void subtract(Money amountToSubtract)
 {
  this.cents -= amountToSubtract.cents;
 }

 public boolean lessEqual(Money compareTo)
 {
  return this.cents <= compareTo.cents;
 }
}
```

### 3.4.1 ATM System: Initial Transformation

As in the school example described earlier, we start by creating the .ump files, one Umple file per input class. A total of 24 Umple files (7 for the package ‘atm.banking’) are created as result of this initial transformation.

The dependency, package and generalization notations are changed to their respective Umple notations.

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

Listing 3.18: ATMMain.java

```
import java.awt.*;
import java.awt.event.*;
import atm.ATM;
import simulation.Simulation;

// Main program
public class ATMMain
{
   // The rest of the code
}
```

Listing 3.19: ATMMain.ump

```
// Main program
class ATMMain
{
  depend simulation.Simulation;
  depend atm.ATM;
  depend java.awt.event.*;
  depend java.awt.*;
  // The rest of the code
}
```

At the end of this transformation step, the visual representation of the umplified model is shown in Figure 3.5. 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 (umplified).

### 3.4.2 ATM System: Transformations to Create Attributes

Once again, the input for the second transformation step is the set of Umple files produced in the previous step. The next step then recovers the attributes. The visual representation of the umplified model is shown in Figure 3.6.

The following is a summary of the various considerations that were taken into account to perform this transformation step:

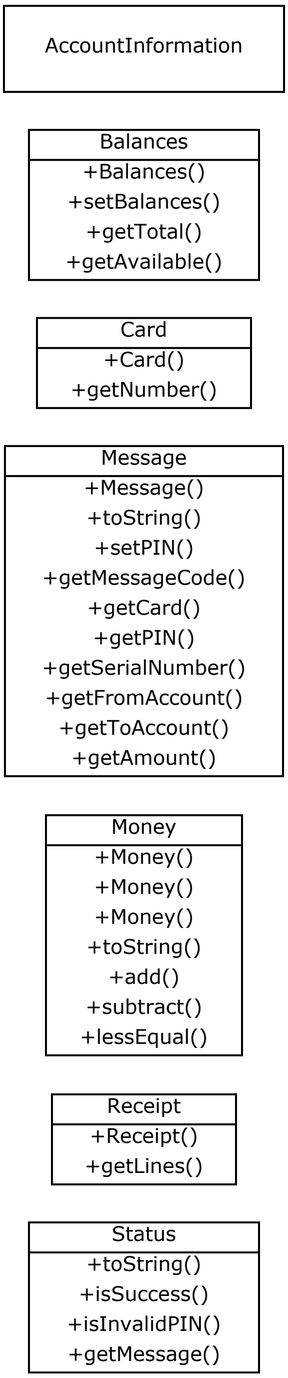


FIGURE 3.5: UML class diagram of the ATM example - Level 1

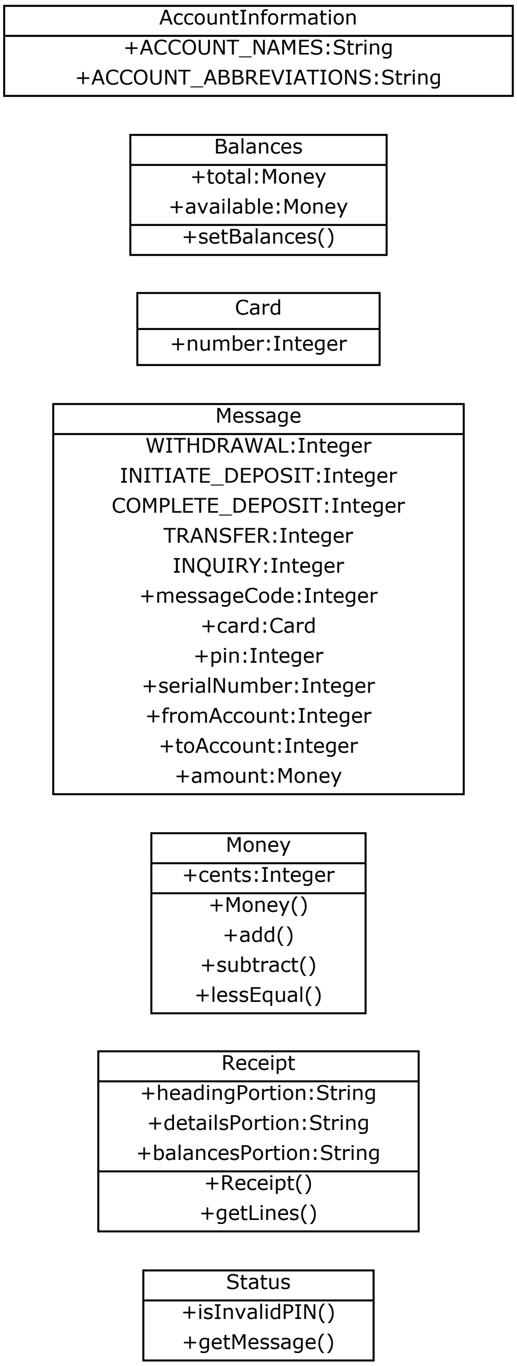


FIGURE 3.6: UML class diagram of the ATM example - Level 2

- Fields (possessing the characteristics described before) were transformed into Umple attributes.

- Constructors were removed from the resulting code. They are not required since Umple generates constructors and in this case they conform to those generated by Umple.

- Mutator and accessor methods were removed from the resulting code. Once again, the Umple compiler will regenerate these.
- *Status* class was transformed into an *abstract* Umple class.
- In class *Balance*, Money is transformed into an attribute since this class contains only fields of primitive data type. Refer to subsection 3.1.3 for more details.

Since there was not a proper set of test cases available, we run a simulation of a series of transactions to verify that the program's semantics were preserved. In fact, the simulation was performed following the 'startup', 'shutdown', 'transaction' and 'session', 'withdrawal', 'deposit' and 'transfer' use cases as described in the documentation of the ATM system [41].

### 3.4.3 ATM System: Transformations to Create Associations

Once the code/model has passed through the two first refactoring steps, we perform the transformation step aiming to recover associations from the input classes.

The result ('atm.banking') produced is shown in Figure 3.7. For conciseness, only classes with associations are shown as part of this class diagram. As mentioned before, the source code of the ATM system can be found at [41].

For instance, the following is a summary of the various considerations that were taken into account to perform this transformation step on classes *ATM* and *Session*:

- The instance variable 'atm' in class *Session* becomes one end of an association. Note that the variable is present in the (unique) constructor of class Session. The multiplicity of this association end is therefore '0..1'.
- We look in class ATM for any instance variable of type Session. Since the instance variable is a collection of Session instances, the variable 'sessions' becomes the other association end. Note that the variable is not present as one of the constructor arguments. Multiplicity of this association end is therefore '*'.
- The rolenames of the associations ends are inferred from the variable names.
- Constructors in both classes are removed.

- Method 'getATM()' is removed from the class *Session*.
- Method 'getCSession(int sessionID)' is removed from the class *ATM* as Umple will generate a method with the following signature 'public Session getSession(int index)'.
- Association in Umple notation becomes: '0..1 ATM – * Session;'.

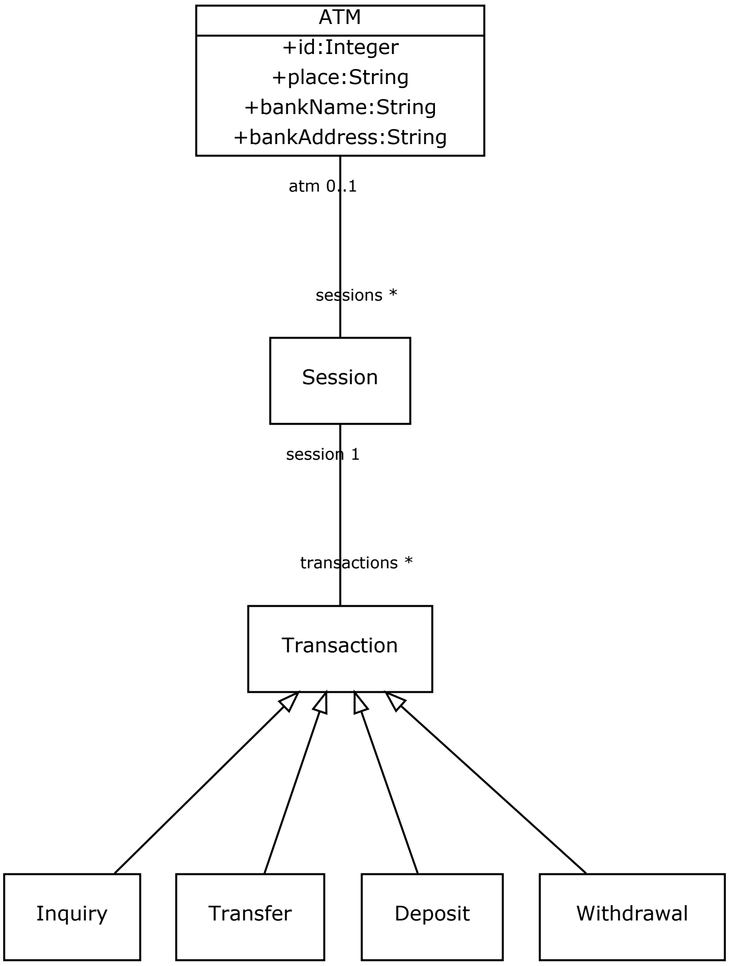


FIGURE 3.7: UML class diagram of the ATM example - Level 3

As in our previous step, the simulation was run to ensure that the program's semantics were still preserved.

## 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 several needs:

The first concerns models that become obsolete very quickly; umplification ensures that from this point on the model can be maintained with the code as it is part of the code. Another need addressed is the desire to simplify existing systems by avoiding the need to write boilerplate code.

In this chapter, we have discussed some transformation steps and demonstrated performing umplification manually. In the next chapter, we will present the mapping rules derived from each of the transformation steps. These transformation rules will be necessary to automate our reverse engineering process. Implementation of the mapping rules is then presented in Section 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 and associations in high-level programming languages mentioned in the literature. In particular, we have considered research that discusses how to generate code from these 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 derived from our analysis, we employ a semi-formal notation. The rules define the transformation from input to output. 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 employ a metamodel-based approach to better express the input/output relationships. The metamodel-based language will be enhanced with OCL [43] expressions.

## 4.1 A Notation for Transformation Rules

Transformation rules presented in this chapter 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

```
Transformation Name (InputModel, OutputModel){
 order n
 params
  ...
 in
   sourceElement: ...
 out
   targetElement: ...
 in-out conditions
  ...
 mappings
     sourceElement.propertyA -> targetElement.propertyA;
     sourceElement.propertyB -> targetElement.propertyB;
}
```

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

- Line 1. Every transformation possesses 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 does not 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.
- Line 9. The conditions that must hold on the source and target model elements are specified following the keywords *in-out conditions*. The mappings are performed only if all conditions specified are true. Conditions are specified using OCL syntax.
- Line 11. 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 (in blue background) 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

```
TypeDeclaration:
                ClassDeclaration
                InterfaceDeclaration
 ClassDeclaration:
      [ Javadoc ] { Modifier } class Name
                        [ extends Type]
                        [ implements Type { , Type } ]
                        { { ClassBodyDeclaration | ; } }
 InterfaceDeclaration:
      [ Javadoc ] { Modifier } interface Identifier
                        [ extends Type { , Type } ]
                        { { InterfaceBodyDeclaration | ; } }
```

Listing 4.3: Grammar for Umple classes

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

Listing 4.4: Rule BaseTypeToUmpleClass

```
1  Transformation BaseTypeToUmpleClass (BaseLanguageMetamodel ,
      UmpleMetamodel){
2   in
3     typeDeclaration : BaseLanguageModel::TypeDeclaration;
4   out
5     UmpleClass: UmpleMetamodel::UmpleClass;
6   in conditions
7     typeDeclaration.oclIsTypeOf(ClassDeclaration);
8   mappings
9     typeDeclaration.name -> UmpleClass.name;
10 }
```

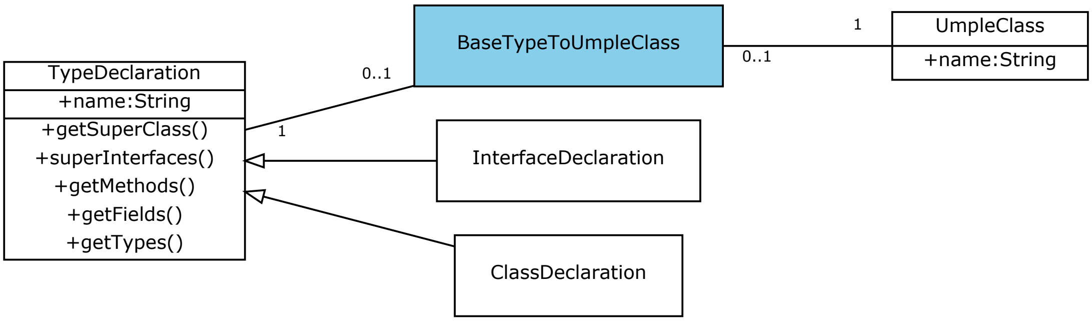


Figure 4.1: Input-output relationships for rule BaseTypeToUmpleClass

Similarly, rule *ImportToUmpleDepend* in Listing 4.5 specifies the transformation of an import declaration into a depend declaration. Import directives in object-oriented languages are used to incorporate information from a type library.

Listing 4.5: Rule JavaImportToUmpleDepend

```
Transformation ImportToUmpleDepend (BaseLanguageMetamodel, UmpleMetamodel
    ){
 in
   importDeclaration : BaseLanguageModel::ImportDeclaration;
 out
   depend: UmpleMetamodel::Depend;
 in-out conditions
 none
 mappings
   importDeclaration.name -> depend.name;
   fieldDeclaration.importDeclaration -> UmpleClass.depend;
}
```

Rule *PackageToUmpleNamespace* in Listing 4.6 specifies the transformation of a package declaration into a namespace declaration in Umple. Package declarations in object-oriented languages are used to organize and group classes.

Listing 4.6: Rule PackageToUmpleNamespace

```
Transformation PackageToUmpleNamespace (BaseLanguageMetamodel,
    UmpleMetamodel){
 in
   packageDeclaration : BaseLanguageModel::PackageDeclaration;
 out
   UmpleClass: UmpleMetamodel::UmpleClass;
 in-out conditions
     none
 mappings
   packageDeclaration.name -> UmpleClass.namespace;
}
```

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

```
Transformation GeneralizationToUmpleIsA (BaseLanguageMetamodel,
    UmpleMetamodel){
 in
   typeDeclaration : BaseLanguageModel::TypeDeclaration;
 out
   UmpleClass: UmpleMetamodel::UmpleClass;
 in conditions
   typeDeclaration.oclIsTypeOf(ClassDeclaration);
 mappings
   typeDeclaration.superInterfaceTypes -> UmpleClass.parentInterfaces;
   typeDeclaration.superClassType -> UmpleClass.extendsClass;
}
```

At this transformation step, we do not attempt to transform any variable into an Umple attribute, association end or state machine. As a result 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

```
Transformation ClassBodyToUmpleClassExtracode (BaseLanguageMetamodel,
    UmpleMetamodel){
 in
   fieldDeclaration : BaseLanguageModel::FieldDeclaration;
   methodDeclaration : BaseLanguageModel::MethodDeclaration;
   typeDeclaration : BaseLanguageModel::TypeDeclaration;
 out
   UmpleClass: UmpleMetamodel::UmpleClass;
 in conditions
   typeDeclaration.oclIsTypeOf(ClassDeclaration);
 mappings
   typeDeclaration->iterate( f: FieldDeclaration |
     fieldDeclaration.toString -> UmpleClass.extraCode);
   methodDeclaration->iterate( f: FieldDeclaration |
     fieldDeclaration.toString -> UmpleClass.extraCode);
}
```

Rules aiming at transforming type declarations into Umple interfaces are very similar to those presented above except that now the 'in-out' condition becomes: '*oclIsTypeOf(InterfaceDeclaration)*'.

Finally, enumerations are umplified into Umple 'enums'. An *enum* in Umple is a state machine that has events.

## 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 or associations variables. Furthermore, we analyze the different patterns supported by existing reverse engineering tools when it comes to the detection of these UML/Umple constructs.

### 4.3.1 Transformations Rules to Create Attributes

The first part of our analysis, as stated at the beginning of this Chapter, consists of analyzing how real projects implement attributes. In previous work done by co-researchers [10] seven projects picked randomly from open source repositories were studied. The seven projects included: from GoogleCode: fizzbuzz, ExcelLibrary, ndependencyinjection and Java Bug Reporting Tool; from SourceForge: jEdit and Freemaker; and from Freecode (formerly Freshmeat): Java Financial Library. Refer to Table 6.2 for details about the systems.

From these projects, we identified 1831 member variables in the 469 classes analyzed. From the member variables found, 1211 were instance variables and 620 were class (static) variables. The results of this study showed that:

- 27% of the instance variables were of a number type. This includes primitive integers, unsigned and signed numbers.
- 14% of the instance variables were of a string type.
- 10% of the instance variables were of Boolean type.
- 2% of the instance variables were of Date and Double type.
- 47% of the instance variables were of a different data type (reference type).

As discussed in Chapter 3 and presented in the study done by co-researchers [10], the type of an (candidate) attribute is either a simple data type such as the types in Table 4.1 or a class that contains itself only instance variables of primitive type. The latter case

Table 4.1: 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. |

is illustrated through Listings 4.9-4.10. In the example, the class *Address* (a reference type) contains only instance variables of primitive type and therefore is more likely an attribute.

Listing 4.9: Person.java

```java
public class Person {
  private Address address;
  // ... Other parts omitted
}
```

Listing 4.10: Address.java

```java
public class Person {
  // ... Other parts omitted
  // Getters, setters and constructors omitted
  private String postalCode;
  private int streetNumber;
  private String streetName;
  private int apptNumber;
}
```

Candidate attributes, those instance variables possessing the characteristics mentioned above, are further analyzed. The instance variables were analyzed for their presence in constructors and in and get/set methods. This is a key fact that helped us determine whether the member variable can indeed become an Umple attribute. Table 4.2 presents the results for this part of the study. Those instance variables with high and medium probability are definitely attributes. The instance variables with a low or very low probability of being attributes are not transformed for now.

Table 4.2: 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 |

Table 4.3: Type of attribute based on their presence in constructor and access method patterns

| Type | Contructor | Setter | Getter |
|---|---|---|---|
| Fully editable | Yes | Yes | Yes |
| Immutable | Yes | No | Yes |
| Lazy | No | Yes | Yes |
| Derived | No | No | Yes |

As presented in Chapter 2, attributes can be categorized into 4 main categories: *lazy*, fully editable, derived or immutable. By analyzing the presence of the variable in the constructor, getter and setter we can determine to which of the categories the extracted attributes belongs to. Table 4.3 presents the different categories of attributes and whether or not they are required as arguments in the constructor, getter and setters. Lazy attributes for instance are those with postponed initialization (third line in Table 4.3). Therefore, when the member variable 1. has a *Get* method, 2. has a *Set* method and 3. is not one of arguments of the constructor, it is matched (converted) into a lazy Umple attribute.

Results of the first part of the analysis on the seven open-source projects can be found at: https://github.com/mgarzon/experiments/mgarzon/thesis/CH4Detection.

The second part of our analysis of attributes consisted of reviewing how other reverse engineering tools recover attributes from source code. Reverse engineering tools in Table 4.4 were employed to recover attributes from (Java) source code. Since all the tools mentioned above generate UML models from source code (and not Umple models) we are not concerned here about the 'lazyness' or 'immutability' characteristics of the attributes. In fact, we are only concerned about the following:

Table 4.4: Reverse-engineering tools evaluated

| Tool | Version | Type |
|---|---|---|
| ArgoUML [44] | 0.32.2 | Open-source |
| Intellij IDEA [45] | 14.1 | Commercial |
| Visual Paradigm [46] | 12.0 | Commercial |
| Rational Rose Modeler [47] | 7.1 | Commercial |
| Fujaba [48] | (Reclipse) 1.0 | Research |
| Ptidej [49] | 5.8 | Research |

1. Is the tool able to correctly recover the type of the attribute?

2. Is the tool able to distinguish cases in which the member variable is of a reference type but that type contains itself only variables of primitive type? It is a

correct recovery if the member variable is extracted as an attribute and not as an association (one-to-one)?

3. Is the tool able to distinguish a 'local variable' from a real attribute by analyzing the presence of an accessor/mutator in the source code? It is a correct recovery if only member variables possessing a getter are recovered as attributes.

4. Is the tool able to deal with all the many different ways of coding an accessor/mutator? 'N/A' in this context means that the tool does not analyze constructors, mutators or accessors when recovering attributes.

5. Is the tool able to distinguish between one (0..1 or 1) and many (*) based on the attribute type? It is a correct recovery if list structures and object types with a plural noun were identified as *many*, all other structures should be identified as *one*. An example of an attribute with *many* multiplicity is 'List<String >ids'.

6. Is the tool able to recover class level attributes (i.e. static)?

Table 4.5 presents the results.

TABLE 4.5: Analysis of reverse-engineering tools for the recovery of attributes

| **Tool** | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| ArgoUML | Yes | No | No | N/A | No | Yes |
| IDEA | Yes | No | No | No | No | Yes |
| Visual Paradigm | Yes | No | Yes | No | No | Yes |
| Rational Rose | Yes | Yes | No | No | Yes | Yes |
| Fujaba | Yes | Yes | No | Yes | Yes | Yes |
| Ptidej | Yes | Yes | No | N/A | Yes | Yes |

By analyzing existing projects and assessing how reverse engineering tools recover attributes, we were able to align our detection mechanisms (mapping rules) with the observed trends, and ensure our tool fills in the gaps that other tools leave.

The mapping rules obtained from our analysis are presented next.

LISTING 4.11: Case 1 - Variable has a getter

```
Transformation FieldToUmpleBasicAttribute (BaseLanguageMetamodel ,
    UmpleMetamodel){
 in
   fieldDecl : BaseLanguageModel::FieldDeclaration;
 out
   attr: UmpleMetamodel::Attribute;
 in-out conditions
  fieldDecl.type.oclIsTypeOf(PrimitiveType)
```

```
     OR fieldDecl.type.oclIsTypeOf(ComplexType) ;
  hasAGetter(fieldDecl);
 mappings
    fieldDecl.name -> attr.name;
    fieldDecl.type -> attr.type;
}
```

Listing 4.12: Is a fully editable attribute

```
Transformation FieldToUmpleFullAttribute (BaseLanguageMetamodel,
    UmpleMetamodel){
 in
   fieldDecl : BaseLanguageModel::FieldDeclaration;
 out
   attr: UmpleMetamodel::Attribute;
 in-out conditions
  fieldDecl.type.oclIsTypeOf(PrimitiveType)
      | fieldDecl.type.oclIsTypeOf(ComplexType);
  hasAGetter(fieldDecl);
  hasASetter(fieldDecl);
  hasAConstructor(fieldDecl);
 mappings
    fieldDecl.name -> attr.name;
    fieldDecl.type -> attr.type;
}
```

Listing 4.13: Is an immutable attribute

```
Transformation FieldToUmpleImmutableAttribute (BaseLanguageMetamodel,
    UmpleMetamodel){
 in
   fieldDecl : BaseLanguageModel::FieldDeclaration;
 out
   attr: UmpleMetamodel::Attribute;
 in-out conditions
  fieldDecl.type.oclIsTypeOf(PrimitiveType)
   | fieldDecl.type.oclIsTypeOf(ComplexType);
  hasAGetter(fieldDecl);
  hasASetter(fieldDecl)==false;
  hasAConstructor(fieldDecl);
 mappings
    fieldDecl.name -> attr.name;
    fieldDecl.type -> attr.type;
}
```

Listing 4.14: Is a lazy attribute

```
Transformation FieldToUmpleLazyAttribute (BaseLanguageMetamodel,
    UmpleMetamodel){
 in
   fieldDecl : BaseLanguageModel::FieldDeclaration;
 out
   attr: UmpleMetamodel::Attribute;
 in-out conditions
  fieldDecl.type.oclIsTypeOf(PrimitiveType)
     | fieldDecl.type.oclIsTypeOf(ComplexType);
  hasAGetter(fieldDecl);
  hasASetter(fieldDecl);
  hasAConstructor(fieldDecl)==false;
 mappings
```

```
    fieldDecl.name -> attr.name;
    fieldDecl.type -> attr.type;
}
```

These high-level rules are implemented in the Drools language and presented in Chapter 5. In the next section, we will consider associations.

### 4.3.2 Refactoring to Create Associations

As we presented in our empirical study that analyzed the generation and extraction of attributes, we discuss in this section associations in practice. We employ the same systems (7) presented in the previous section. From the analysis of finding attributes, we know now which of the instance variables assessed are attributes. The remaining 235 member variables are considered as candidate association ends. In our previous work [9], the distribution of the mutator, accessor and availability in constructors were tracked. The study showed that:

- 29% of the 235 member variables had both a Get and Set method
- 38% of the 235 member variables had at least a Set method (without a Get method)
- 51% of the 235 member variables had at least a Get method (without a Set method)
- 23% of the 235 member variables did not have a Set method
- 17% of the 235 member variables had only a Get method
- 17.9% of the 235 member variables provided Add and Remove methods

In addition, 42 of the 235 member variables were defined using one of the classes that operate on collections such as Map, Set, Hashtable or List classes and hence likely represented associations with an upper bound greater than one. According to previous work done by co-researchers [9], five association patterns categorized by multiplicity were extracted after analyzing 1800 modeled associations in UML diagrams of real systems (MARTE, Flow Composition, ECA, Java, Patterns and rCOS UML profiles) and UML models from our own repository of examples [50]. The five association patterns, ordered according to their frequency of usage, are presented below:

1. one to many (1–*)
2. optional-one to many (0..1–*)
3. many-to-many (*–*)
4. optional-one to one (0..1–1)
5. optional-one to optional-one to optional-one (0..1–0..1)

As it concerns the evaluation of the reverse engineering tools in Table 4.4 when it comes to extract associations from source code, we have used our repository of Umple models [50] as test cases. Each of the examples has been reverse-engineered by the different tools allowing us to determine whether or not the tools support the different multiplicity patterns. Note that Java code has been generated from the various Umple examples. Results are presented in Table 4.6. The outcomes of our evaluation are:

- **Full**: The association multiplicity is correctly extracted from the different Java systems.
- **Partial**: The association multiplicity is correctly extracted in most cases.
- **None**: The association multiplicity is not supported at all.

Table 4.6: Analysis of reverse-engineering tools for the recovery of associations

| **Tool** | **1–*** | **0..1–*** | ***–*** | **0..1–1** | **0..1–0..1** |
|---|---|---|---|---|---|
| ArgoUML | Partial | None | Partial | Partial | None |
| IDEA | Full | Partial | Full | Partial | Partial |
| Visual Paradigm | Full | Full | Full | Partial | Partial |
| Rational Rose | Full | Full | Full | Full | Partial |
| Fujaba | Partial | Partial | Partial | Partial | Partial |
| Ptidej | Full | Partial | Full | Partial | Full |

Based on the findings from our study, we now 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.

As seen in Chapter 3, a variable represents an association if all of the following conditions apply:

Table 4.7: 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 4.8: 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 |

- 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, accesses and/or manipulates instances of the variable type.

In the Umplificator, the tool we will describe in Chapter 5, 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 4.7 and Table 4.8 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. We have considered those collections of elements defined using Map, Set, List and Hashtable classes (from the Java collections framework or the Standard Template Library in C++).

**Inferring role names** Roles names can be easily inferred from the field name. This does not impose a challenge since the field name is always visible during static analysis of source code.

**Inferring directionality** To infer the directionality of the association, we consider both classes linked by the association. Bi-directionality means that each class can access

Table 4.9: Code elements parsed and analyzed for the different patterns

| Pattern | Elements Parsed/Analyzed |
|---|---|
| 1 | Constructor. |
| 0..1 | Set or Get methods. |
| 0..* | Set/Get or Add/Remove methods. |
| 1..* | Constructor. |

the linked object(s) of the other class. In the case one of the classes involved in the association cannot access the linked objects of the other class (member variable is not present) the association is declared as unidirectional.

**Inferring multiplicities** The process of inferring multiplicities from source code starts by analyzing the fields. If the field is an array, or a collection, the multiplicity is many '0..*'. If the field is not an array or collection, the multiplicity is '0..1'. To infer the other multiplicity patterns, analyses of constructors, mutators and accessor methods need to be performed. Recovery of multiplicities in the umplification approach ignores all constraints involving lower-bound (n) and upper-bound (m) since that would require either *dynamically* counting the number of instances created [51] or parsing arbitrary algorithmic code. More specifically, Table 4.9 presents the code elements that need to be inspected in order to infer the different patterns.

Other special cases were taking into account when dealing with associations with an upper-bound greater than one. For instance, in Java code created prior to the introduction generics (i.e. Java 1.4), it was not necessary to typecast objects as much as it was afterwards. For this reason, we need to inspect the add or remove methods to determine the type of object contained (added or removed) by the collection. Most of the other tools we have briefly discussed in this chapter lack mechanisms to infer the type held by a collection. For instance, a model reverse-engineered from Java 1.4 might, in some tools, result in a one-to-one association between *Mentor* and *List* when in reality the association should be a one-to-many association between *Mentor* and *Student*.

The mapping rules obtained from our analysis are summarized in Table 4.10 and implemented in Chapter 5. Helper functions (in light blue) are also presented in the Table.

Table 4.10: Summary of mapping rules for the detection of associations in code, that will be converted into Umple associations

| **Rule Name** | **Description** |
|---|---|
| isAssociation | Base case rule. Returns an association variable with basic information. |
| isOneToUnknownAssociation | Extracts the multiplicity from one of the ends. Updates the association variable. |
| isManyToUnknownAssociation | Extracts the multiplicity from one of the ends. The conditions that must hold for this rule to be applied specify that the upper bound is greater than one. Updates the association variable. |
| isOptionalManyToUnknownAssociation | Extracts the multiplicity from one of the ends. Uses other functions to retrieve the lower bound. |
| isOptionalOneToUnknownAssociation | Extracts the multiplicity from one of the ends. Uses other functions to retrieve the lower bound. |
| MatchOtherAssociationEnd | Attempts to relate associations ends. Returns an association. |
| isNavigable | Updates the navigability of the the association by inspecting both association ends. |
| isCustomMethodDeclaration | Returns a code injection required to adapt the existing method declaration. |
| isCustomConstructor | Returns a code injection required to adapt the existing constructor. |
| isAssociationEndExternal | Returns true if one of the classes involved in the association is not available. See description of external classes in Chapter 2. |
| isVariableInConstructor | Returns true if the member variable is in the class constructor. |
| hasVariableGetter | Returns true if the member variable possesses a getter. |
| hasVariableSetter | Returns true if the member variable possesses a setter. |
| getTypeOfCollection | Returns the type contained in a collection. |
| resetExtraCode | Searches for code that is not required and needs to be removed (fields, methods, constructors) from classes. |
| refactorMethod | Uses code injections gathered from other rules/-functions to refactor methods. |

## 4.4 Summary

This chapter studied how attributes and associations are used in practice; and investigated how these modeling abstractions can be extracted by analyzing source code. We reviewed the capabilities possessed by various reverse-engineering tools for detecting these model elements.

We identified various combinations of multiplicity for association ends and analyzed their impact on code generation and reverse engineering.

Finally, we presented the mapping rules for the detection of constructs in source code, focusing on Java.

In the next chapter, we show more concretely how these rules have been implemented in this research.

# 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 alternative implementations we attempted during the initial stages of our work.

## 5.1 The Umplificator 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 Umple 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 [52]. 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 a manner such that software engineers can better understand the system. In the literature explored in Chapter 7, most 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.

1. [**Support for various input languages**] The tool must allow defining transformation rules from object-oriented source code to Umple. Java, C++ and PHP source code need to be *parsed* since those are the main languages that the Umple compiler can generate. The tool should be able to handle the different idioms and programming conventions of those programming languages.

2. [**Incrementality**] 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.

3. [**Rule execution control**] The tool must offer the ability to specify direct control of the order of rule application. Rule application scoping is needed to restrict the transformation to affect only parts of the model. This is required to support incremental transformations.

4. [**Command-line support**] The tool must offer GUI and/or command line capabilities. The tool should operate as a command-line tool to allow rapid bulk umplification and easier automated testing. Also, command line capabilities are needed for scripting and for back-ends that permit deployment of the tool on the Web.

5. [**Directionality**] The tool must minimally allow interpreting the mapping between the source and target models unidirectionally. Multidirectional languages allow interpretation of the rule in several in both directions, from the source to target

and from the target to source models. As it concerns the umplification approach, multi-directionality is not required.

6. [**Usability of language**] The language created or employed to specify the mapping rules should be as general and extensible as possible. Changing the transformations rules, an activity that we expect to perform very often, should be done without re-compilation the entire system. Moreover, changing the transformation rules should not require changing the language definitions or internal representations.

7. [**Rule organization**] The tool should offer a modularity mechanism to organize the mapping rules, libraries and helper functions.

8. [**Output export**] The tool should be able to export the output in Umple.

**From the developer's perspective:**

9. [**Maintainability**] The tool should be easy to debug. We should be able to quickly identify the location of an error and fix it.

10. [**Extensibility**] The mapping rules should be as general and extensible as possible.

## 5.2 Alternative Approaches Studied

Prior to making our final choice of technology for umplifying software systems, we explored two different and well-known model transformation technologies: TXL [34] and ATL [30]. In the following two sub-sections, we outline the mapping rules, grammar and program directives in these languages for transforming 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" [34].

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 *mapping 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 *LeftHSPattern* 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 a 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* is 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

We describe now 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.

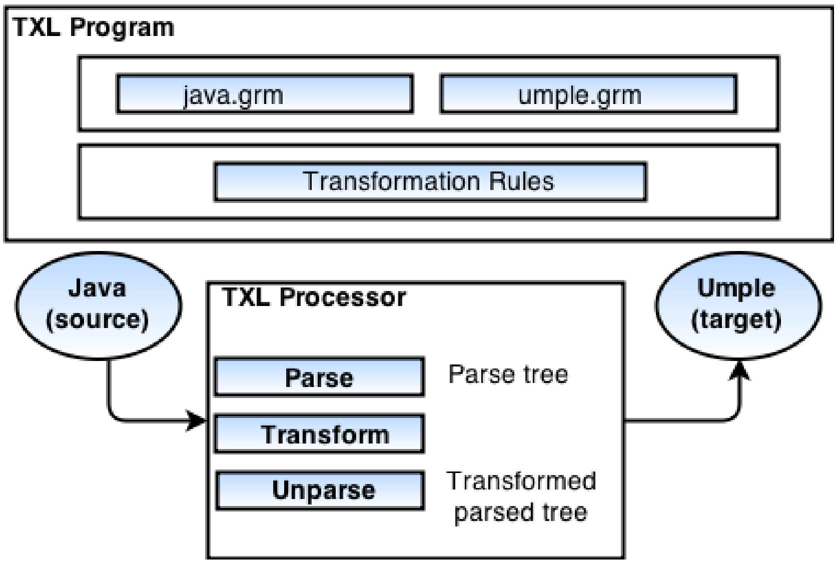


FIGURE 5.1: TXL program for transforming Java to Umple

#### 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 [53]. 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:

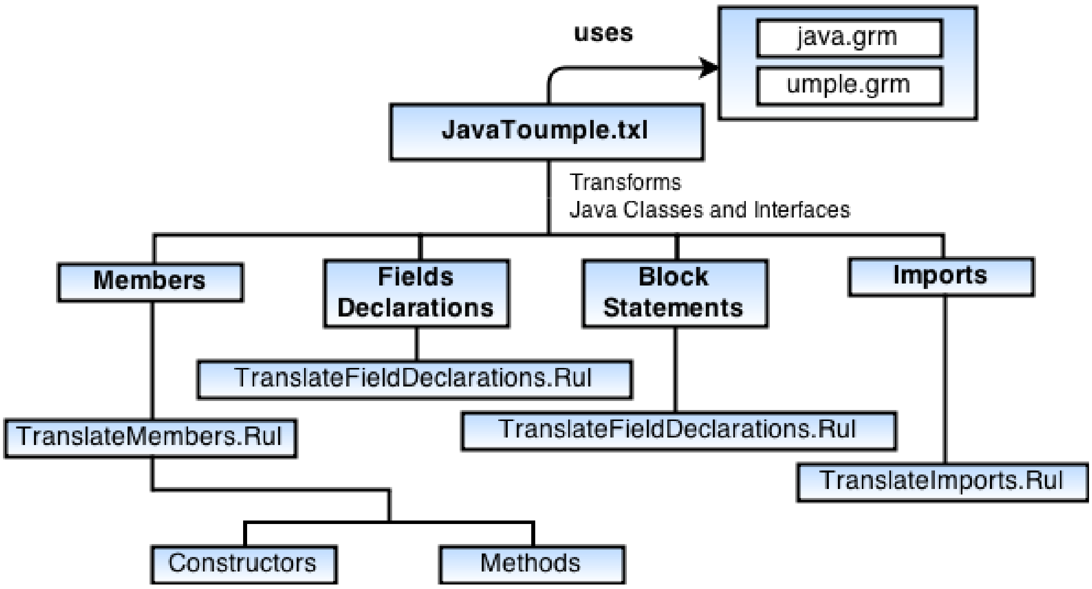


FIGURE 5.2: Structure of the JavaToUmple program

- 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.
- 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 are 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  content
}
// 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 in Listing 5.10, 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 (Lines 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) [30] 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 declarative and imperative ways.

ATL is a set of Eclipse plug-ins created 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 [30]. 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 [54]. The Java model conforms to metamodel *Java.ecore* and the Umple model to metamodel *Umple.ecore*. The ecore [55] 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.

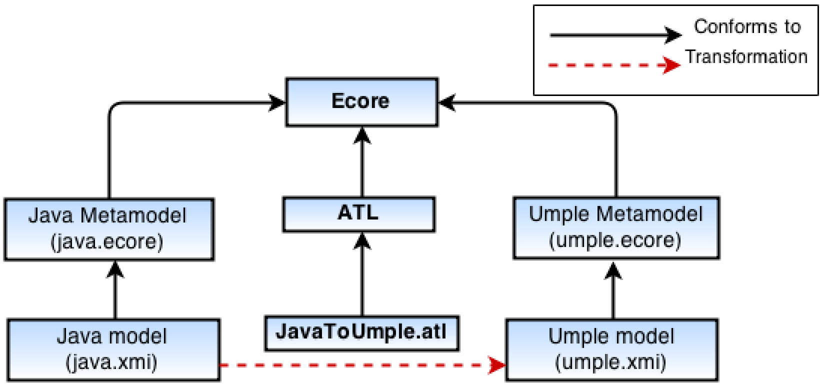


Figure 5.3: The JavaToUmple ATL program

#### 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 cannot 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 [56].

- **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) [30], 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 [57] and has been simplified for the purpose of this transformation example.

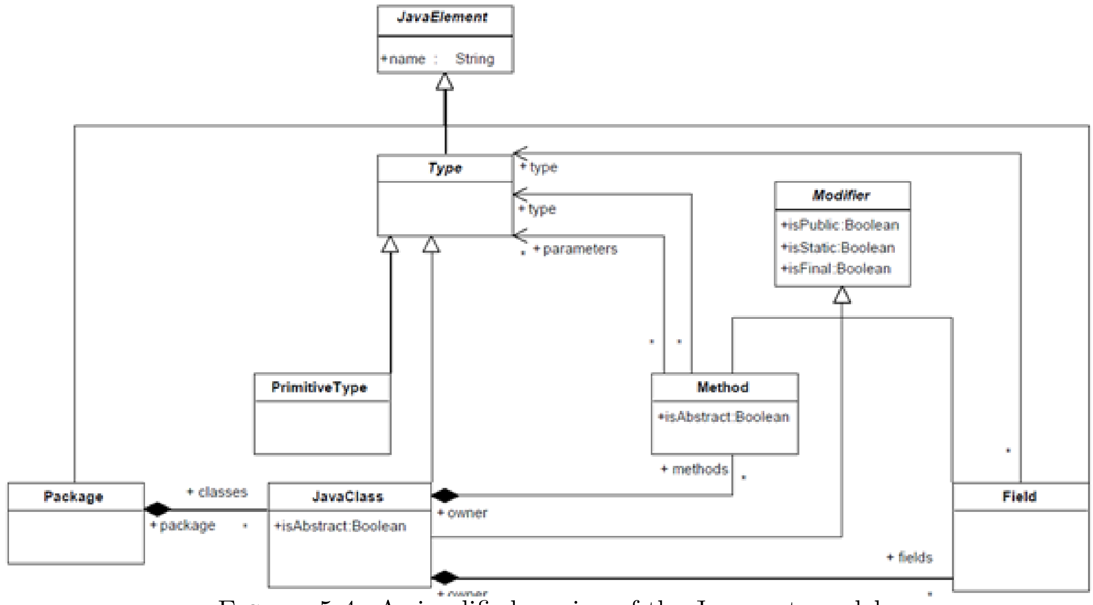


FIGURE 5.4: A simplified version of the Java metamodel

A simplified version of the Umple metamodel (target metamodel) is presented in Figure 5.5. The complete metamodel for Umple can be found at [15]. Both metamodels have been defined in the XMI format, as required by the ATL metamodel loader.

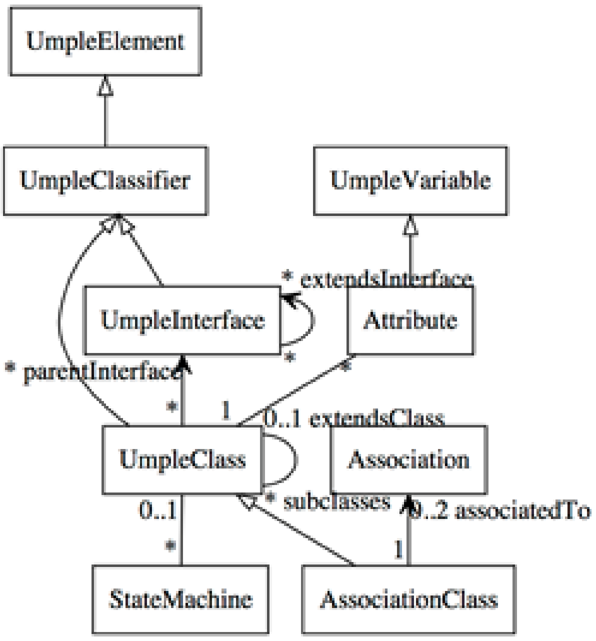


FIGURE 5.5: A simplified version of the Umple metamodel

**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.

The three rules presented below 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).

Listing 5.12: ATL transformation rules

```
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),
 )
}
```

## 5.3 Discussion

In the previous sections, we demonstrated grammar-based (TXL) and metamodel-based (ATL) transformations for the purpose of umplifying a software system. We showed how to represent the input and output language definitions in both ATL and TXL and how to describe the transformations using rule sets. In this section, we evaluate both transformation approaches to determine whether or not they meet the necessary requirements for implementing the umplification approach.

Based on the (not-exhaustive) list of requirements presented at the beginning of this chapter in section 5.1, we now contrast both approaches. The comparison is presented in

Table 5.1. The outcomes of our evaluation can have a positive or a negative effect denoted by symbols '+' and '-' respectively. More symbols indicate higher importance (i.e. '+++' = high importance, '++' = moderate importance, and '+' = low importance). We will next discuss our rationale for assigning the various levels.

Table 5.1: Comparison of ATL and TXL technologies

| **Evaluation Criteria** | **ATL** | **TXL** |
|---|---|---|
| **Support for various input languages** | - | - |
| **Incrementality** | - | - |
| **Rule execution control** | + | + |
| **Command-line support** | - | +++ |
| **Usability of Language** | ++ | + |
| **Rule organization** | +++ | +++ |
| **Output Export** | - | +++ |
| **Maintainability** | + | ++ |
| **Extensibility** | - | - |

**Support for various input languages** In TXL, some of the transformation rules can be reused if our intention is to support a new input source language. As we described before, to define a model transformation from a new input language to Umple, we need to:

1. Construct the working grammar for the input language only since the existing grammar for Umple can be reused.

2. A union grammar has to be written to combine the source and target grammars. For instance, to combine a grammar rule defining a namespace in PHP and the one defining an Umple namespace, we employ a define statement as follows:

```
define namespace
 [PHP_namespace] | [Umple_namespace]
end namespace
```

   The rule above simply means that instances of either PHP namespaces or Umple namespaces can appear during the transformation process.

3. Build the mapping rules. We require to write all the rules to express the relationship between one pattern in the source and its transform in the target language. Some logic from the rules aiming at transform other languages could be reused, but in general they have to be completely rewritten.

In ATL, to support a new input language, we need to:

1. Create a metamodel definition for the new language to be supported.
2. Create the mapping rules. Some of the logic used in helper functions and rules can be reused but as in the case of TXL, the rules must be written from scratch.

TXL offers a (public) collection of grammars including those for PHP, Java and C++ [53], therefore reducing the time taken to develop the transformations.

**Incrementality** As discussed in Chapter 3, incrementality means that we possess the ability to perform the transformation in multiple small steps that produce results quickly. The execution of ATL transformations has always followed a two-step algorithm: 1) matching all rules, 2) applying all matched rules. This algorithm does not support incremental execution. For instance, if a source model is updated, the whole transformation must be executed again to get the updated target model. The only way to implement incrementality in ATL is to regroup the rules belonging to a particular umplification level as lazy rules and then call them from a normal rule. In Listing 5.13, two lazy rules responsible for performing a part of the transformation are called from the rule 'transformLevel1'.

Listing 5.13: Calling lazy rules

```
lazy rule transformAPart {...}
lazy rule transformAnotherPart {...}

rule transformLevel1 {
  from ...
  to umpleClass : umpleMetamodel!UmpleClass { partTransformed <-
    transformAPart(),
                                              otherPartTransformed <-
    transformAnotherPart() }
}
```

Incrementality in TXL can be achieved by regrouping the rules belonging to a refactoring level in functions. For instance, 'function transform LevelX' regroups all rules that need to be matched (applied) in order to transform the input source at a particular level of refactoring.

**Rule execution control** The order of execution of the transformation rules cannot be explicitly controlled in ATL or TXL. Ordering can be partially controlled as explained above by grouping the rules.

**Command-line support** Creating and debugging ATL transformations must be done within the Eclipse IDE. Mapping rules cannot be added on the fly and any rule change requires compiling the entire system. Packaging the transformation files (into a jar) is possible if all Eclipse project dependencies are also packaged. The ATL project offers an Ant task that can be used to chain transformations or to integrate ATL into an existing suite of tools. The major issue when creating a self-contained (and standalone) JAR that can be used in mobile, web applications is that the ATL compiler requires more than 30 dependencies (jars), or bundles as they are called in the Eclipse world. The required bundles are:

- org.eclipse.ui,
- org.eclipse.core.runtime;bundle-version="3.4.0",
- org.eclipse.core.resources;bundle-version="3.4.2",
- org.eclipse.m2m.*;bundle-version="3.2.1";visibility:=reexport,
- And other 22 org.eclipse.m2m.* bundles.

In addition, the 'org.eclipse.core.runtime' requires another 12 dependencies.

On the other hand, TXL can easily be run on the command-line.

**Usability of the language** In TXL, the language used to define grammars can be easily understood if one has prior knowledge of the BNF notation. In our own experience, the challenge concerns the creation of the rules. The language used to define the patterns and replacements is very rich but also very complex. This is an issue since we continuously need to refine and add more rules. Even with the Java-based Graphical User Interface tool that has been recently developed to help debugging TXL programs, the task is (arguably) complex.

On the other hand, the ATL language proposes a mix of Java and OCL-like syntax that can be in our opinion easier to understand, even for end-users.

**Rule organization** In both TXL and ATL, rules can be properly organized in modules. This improves readability and maintainability. Similarly, both approaches allow us to separate concerns in a convenient way; grammars and rules can be independently defined, for example.

**Output export** The TXL compiler can generate any type of text file. For instance, a TXL program can generate Umple files (files with extension .ump).

ATL generates only XMI-based models as shown in Listing 5.14. The generated model needs then to be transformed into an Umple model. Listing 5.14 shows a generated model containing a class named 'Facility' with three attributes (definition omitted).

Listing 5.14: Model generated by ATL

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI
    /2.1" xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" xmlns:uml="
    http://www.eclipse.org/uml2/2.1.0/UML" xmi:id="_model" name="model">

  <packagedElement xmi:type="uml:Class" xmi:id="_Facility" name="Facility
    ">
     <ownedAttribute xmi:id="_Facility-id" name="id" visibility="private">
       <type xmi:type="uml:PrimitiveType" href="pathmap://UML_LIBRARIES/
    UMLPrimitiveTypes.library.uml#Integer"/>
       <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="
    _Facility-id-_upperValue" value="1"/>
       <lowerValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="
    _Facility-id-_lowerValue" value="1"/>
     </ownedAttribute>
     ....
 </uml:Model>
```

Another issue with ATL is that it requires an input model in XMI format. To perform a Java-to-Umple transformation in ATL requires transforming the Java source code into XMI, and afterwards transforming the resulting model from XMI into Umple code. Java and C++ to XMI representations can be performed with external tools such as javaML and srcML respectively, but the fact of adding intermediate transformations can considerably affect the overall performance of the transformation tool. Finally, the generated model needs to be transformed using XSLT or similar approaches to an Umple model.

So although ATL and TXL are suitable for certain kind of transformation tasks, as discussed above, certain concerns cannot be handled naturally when implementing the umplification approach. In fact, the major concerns for any reverse engineering tool used to implement the umplification approach are indeed the extensibility and reusability of the implementation.

To overcome the various issues discussed above, we decided not to use ATL or TXL, and instead use a family of technologies based on ‘xDT’ (where ‘x’ can be J, P, C, standing respectively for Java, PHP and C/C++), and Drools [58]. Each member of the family is intended to deal with a particular kind of transformation task.

The family of technologies proposed in the following section will improve the following aspects, as compared with what we were able to accomplish with ATL and TXL:

- Parsing: we employ existing technologies to parse object-oriented source code. These technologies have been proven to work in different contexts and be very reliable. Java, C++ and PHP source code can be parsed with no effort.
- Visitors: visitors have been developed to traverse the different Abstract Syntax Trees (ASTs) produced by the parsers. This is a required feature to fulfill the requirement for incrementality.
- Matching: we employ a rule-based engine (Drools) that performs the tasks related to the execution of the rules.
- Rule definition: the rules are defined using a Java-like language in an editor with syntax-highlighting and code completion capabilities. Visualization and debugging is performed withing the Eclipse IDE.
- Exporting capabilities: a custom generator has been developed to output Umple code from an Umple model. Code produced is validated.
- Command-line capabilities: a self-contained JAR is produced as part of our automated building process. This executable jar is used in an web application currently being developed and for testing purposes.
- Rule execution: and agenda (a RETE algorithm feature) has been employed to schedule the execution of rules in a deterministic order.

## 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 [58]. Drools is a rule management system with 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 [59].
- Umplificator Workflow: A tool that guides the umplification process within Eclipse.

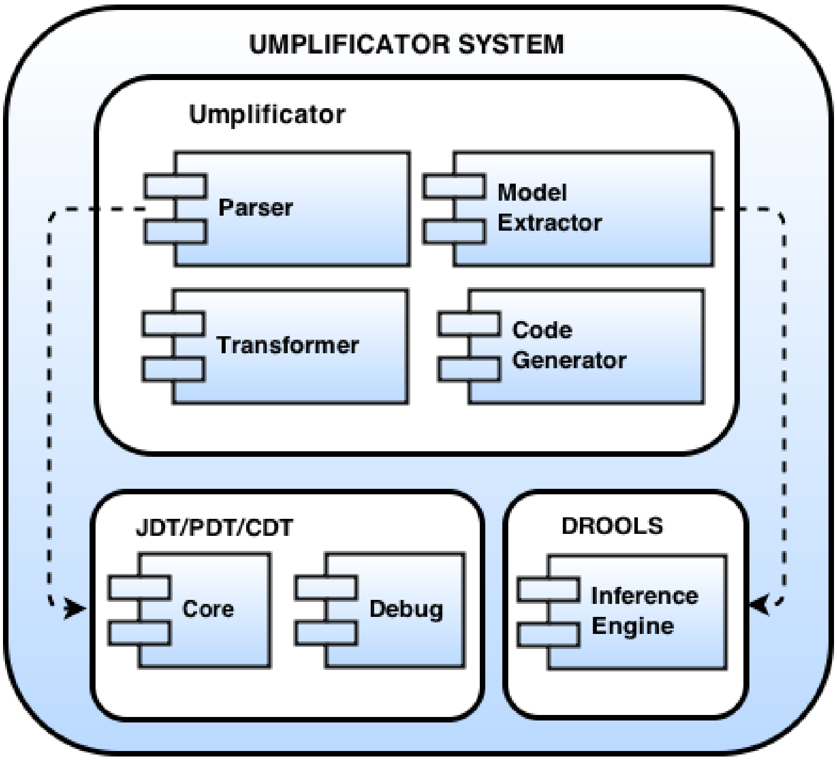


Figure 5.6: The Umplificator components

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 5.6 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 an object-oriented software system in this architecture is described below an illustrated in Figure 5.7.

1. The input is a set of source code files in the base language and/or Umple.
2. (Parser) The source code is parsed.

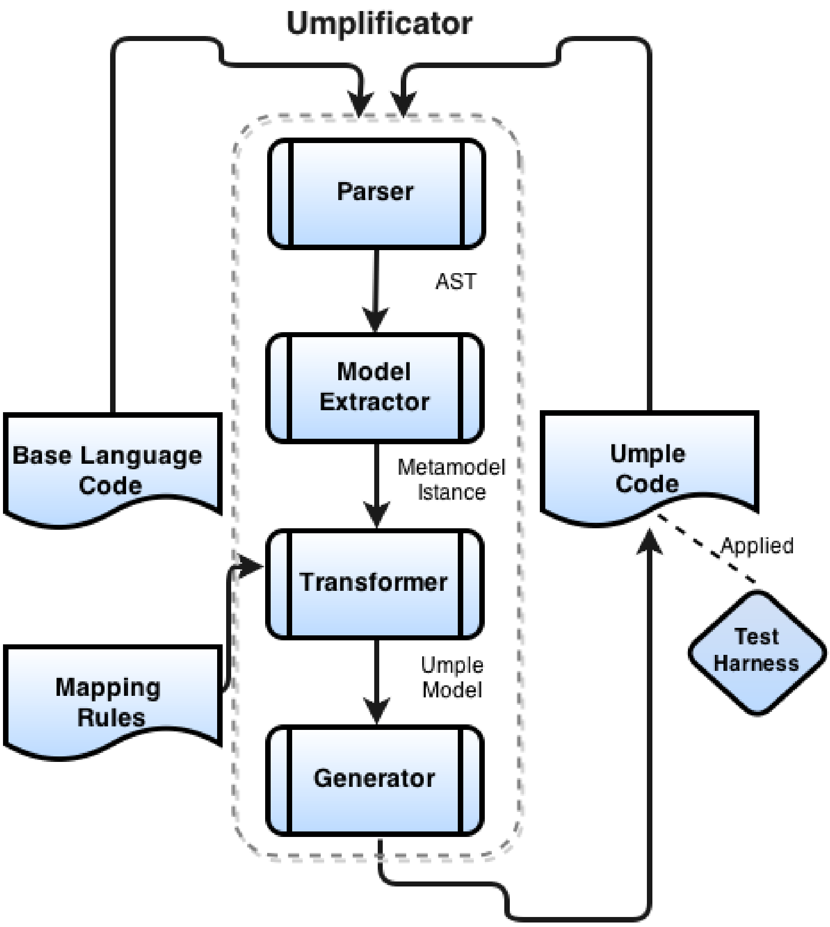


Figure 5.7: The umplification process flow

3. (Model Transformer) The source code is transformed into 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.

The Umplificator employs the libraries and technologies summarized in Table 5.2 to implement its reverse engineering capabilities. The dependencies between the external and internal components of the Umplificator is shown in Figure 5.6, where our *Parser* and *ModelExtractor* components uses the JDT/CDT/PDT projects and the *Transformer* the Drools Rule Engine.

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

Table 5.2: 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 5.3 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 5.3: 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) [60] 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* [61] 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 [25]. 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 5.15. 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 5.15: A visitor for import declarations in Java source code

```

public class SimpleVisitor extends ASTVisitor{

 private List<ImportDeclaration> importDeclarations;

 public boolean visit(ImportDeclaration node) {
     importDeclarations.add(node);
     return false;
 }
}
```

As an example, consider the code of class '*Test*' in Listing 5.16. Once the code is parsed, we used a visitor to collect the desired information. Table 5.4 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 5.16: Test.java

```
package umplificatorTest;

import java.util.Date;

public class Test {
 public int number;

 public int  getNumber() {
  return number;
 }
}
```

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*.

Table 5.4: 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) |
| **ImportDeclaration** | “import java.util.Date” | visit(PackageDeclaration pd) |
| **TypeDeclaration** | “public class Test” | visit(TypeDeclaration td) |
| **FieldDeclaration** | “public String name” | visit(FieldDeclaration fd) |
| PrimitiveType(“int”) | | td.getType() |
| SimpleName(“number”) | | td.getSimpleName() |
| **MethodDeclaration** | “public int getNumber()” | visit(MethodDeclaration md) |
| PrimitiveType(“int”) | | td.getReturnType() |
| SimpleName(“getNumber”) | | td.getName() |
| **Block** | “..” | bl= md.getBody() |
| **ReturnStatement** | “return number;” | stmt = b1.getStatements(0); |

Code comments, which are important from an understanding viewpoint, are preserved in the AST, parsed and transformed into Umple comments. *Comment*, is the abstract base class for all AST nodes that represent comments. There are exactly three kinds of comments: line comments (*LineComment*), block comments (*BlockComment*), and doc comments (*Javadoc*).

The code on the right of Table 5.4 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*.

**C/C++ 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 in C and C++. 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 between 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 program in Listing 5.17 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 5.17: Simple example in C++ - test.c

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

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 traversed using a series of visitors. The input/output relationship of the parser and model extractor components is illustrated in Figure 5.8.

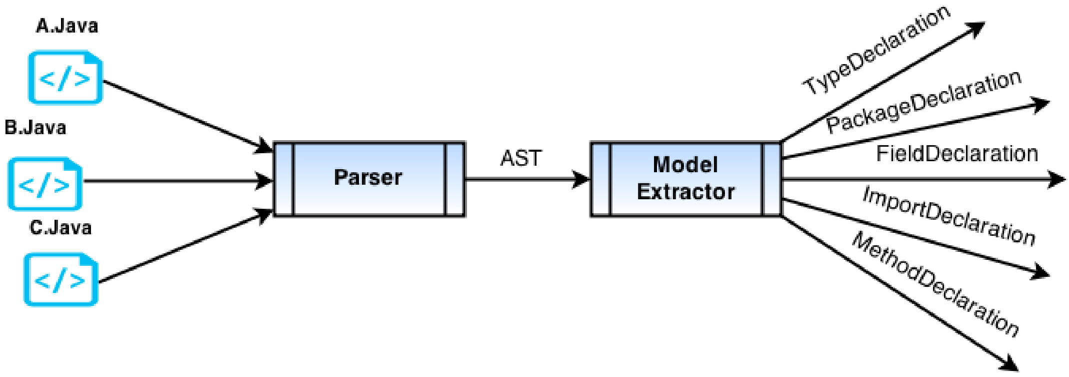


FIGURE 5.8: The Parser and Model extractor components

### 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 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.

As we have seen in Table 5.2, the Transformer component leverages Drools technologies to implement its rule engine.

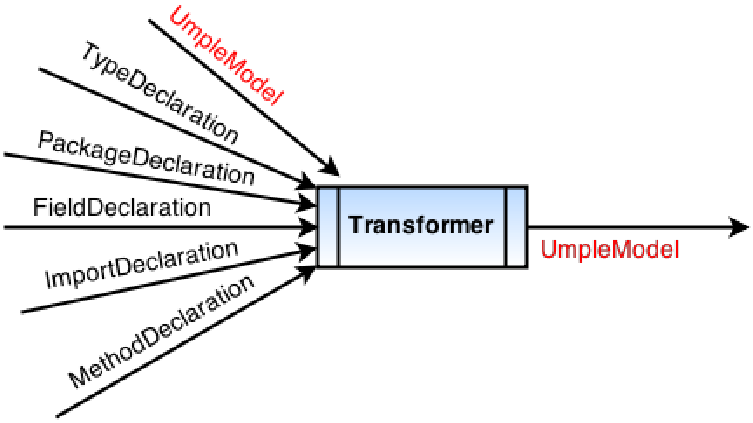


FIGURE 5.9: The Transformer component inputs and outputs

#### 5.4.3.1 Drools 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 a: an *Inference Engine*, an *Agenda*, a *Pattern Matcher*, a *Production Memory* 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 on 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 [62]. 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.

Traditionally, rule engines have two methods of execution [63] 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

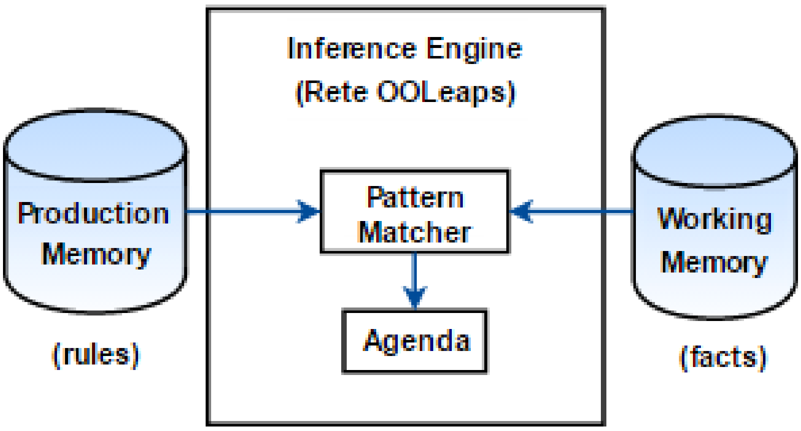


FIGURE 5.10: High level view of the Drools rule engine

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 by 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).

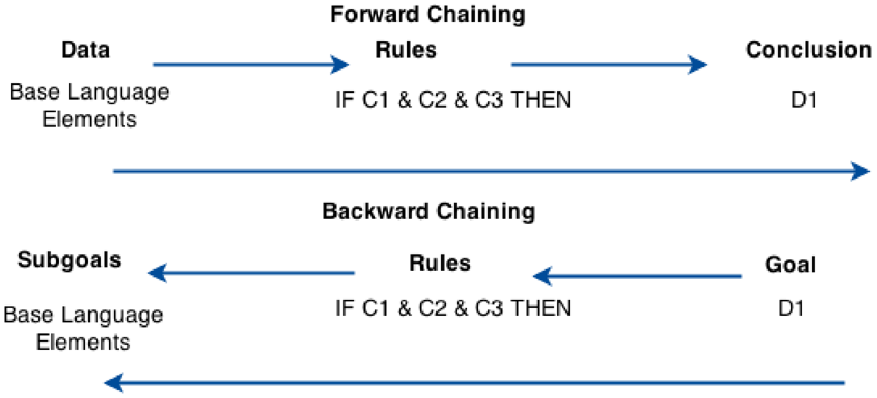


FIGURE 5.11: Forward vs backward chaining

#### 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 metrics 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 5.18 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) provide a declarative way to influence the behavior of the rule. We present the rule attributes used in our mapping rules in Table 5.5.

Listing 5.18: Basic rule in Drools

```
rule "name"
  attributes
  when LHS then RHS
end
```

Table 5.5: 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 have 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 5.19 is a rule belonging to the group that will be executed first. The rule in Listing 5.20 will be executed after any rule belonging to the first level.

Listing 5.19: A rule belonging to Level 1

```
rule "transform_Namespace_UInterface"
        agenda-group "LEVEL1"
        when
        // parts omitted
        then
        // parts omitted
end
```

Listing 5.20: A rule belonging to Level 2

```
rule "JavaField_CanBeUmpleAttribute"
        agenda-group "LEVEL2"
        when
        // parts omitted
        then
        // parts omitted
end
```

Listing 5.21 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 5.21: 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 5.6.

### 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.

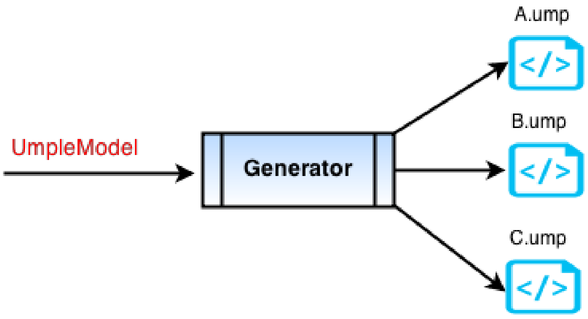


Figure 5.12: The Generator component inputs and outputs

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 C++ 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 Summary of Umplification Technologies

In the previous sections, we have presented the current implementation of the umplification approach. The Umplificator employs different technologies to deal with the various tasks of the umplification process. The advantages of our mixed approach are enumerated below:

**Separation of concerns** The Rule-Engine allows us to break the coupling between the data and logic. The logic is all laid out in rules. With the use of rules, we solve the issues related to hand-coded 'if..then' approaches.

**Speed and scalability** The Rete algorithm provides very efficient ways of matching rule patterns to the internal representations of the input source code. In addition, the Rete algorithm has been proven to be highly scalable. This is important to meet the requirements of industrial projects, where large-scale models must be dealt with.

**Centralization of knowledge** The different rules form a repository of knowledge that serves as a single point of truth. The rules are readable enough to be used as documentation.

**Multi-level testing** Each component in our implementation can be independently tested. This is further discussed in the next chapter of this thesis. Tests can reveal if the addition of rules (or refinements) have invalidated existing transformations.

**Robust parsing** The CDT, JDT and PDT parsers are robust and well-maintained. This is very desired characteristics since languages like Java and C++ are in continuous development. For instance, JDT has been updated to comply with the different Java versions, from 1.4 to 1.8. In addition, the parsers are able to parse specific parts of the code (a method, a constructor, etc). This is a desired feature to comply with the incrementality requirements of the umplification approach.

**Efficiency** The parsing technologies employed are industrial technologies that have been validated with large systems. For instance, Piatov et al. in his study [64], has concluded that the CDT parser is a robust, efficient and actively maintained parsing technology.

**Agile development** The process of building transformations as it stands in our current implementation is compatible with the practices and principles of agile development which are of growing importance in software engineering [65] and systems engineering.

**Extensibility** Rule files (files with extension .drl) can be added on the fly without requiring re-compilation of the entire Umplificator. End-users can then add more rules for their specific needs.

**Reusability** Pattern-matching conditions and actions in (Drools) rules can be reused to accommodate new transformations for different languages.

Furthermore, the estimated cost of our implementation, calculated using the *COCOMO model* [66], with coefficients a=2.4 and b=1.05, is:

- *Codebase Size*: 114725 LOC
- *Duration* = 23.27 months
- *Effort* = 1066.33 person-months

## 5.6 Automated Umplification Example

### 5.6.1 Initial transformation

As an example of the transformation process using the Umplificator, consider the input Java source code in Listing 5.22. We want to achieve the initial level of refactoring (Level 1).

Listing 5.22: Input source code

```java
package university;

import java.util.Date;

public class Student {

    private String name;
    private 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 " + name "has id=" + studentId;
    }
}
```

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 5.6 presents all the Java Elements collected by the Java visitor.

TABLE 5.6: 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()...” |

The Transformer receives the Java model elements in Table 5.6, together with a newly created instance of an UmpleModel and places them into the Working Memory. At 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 5.23.

LISTING 5.23: 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.

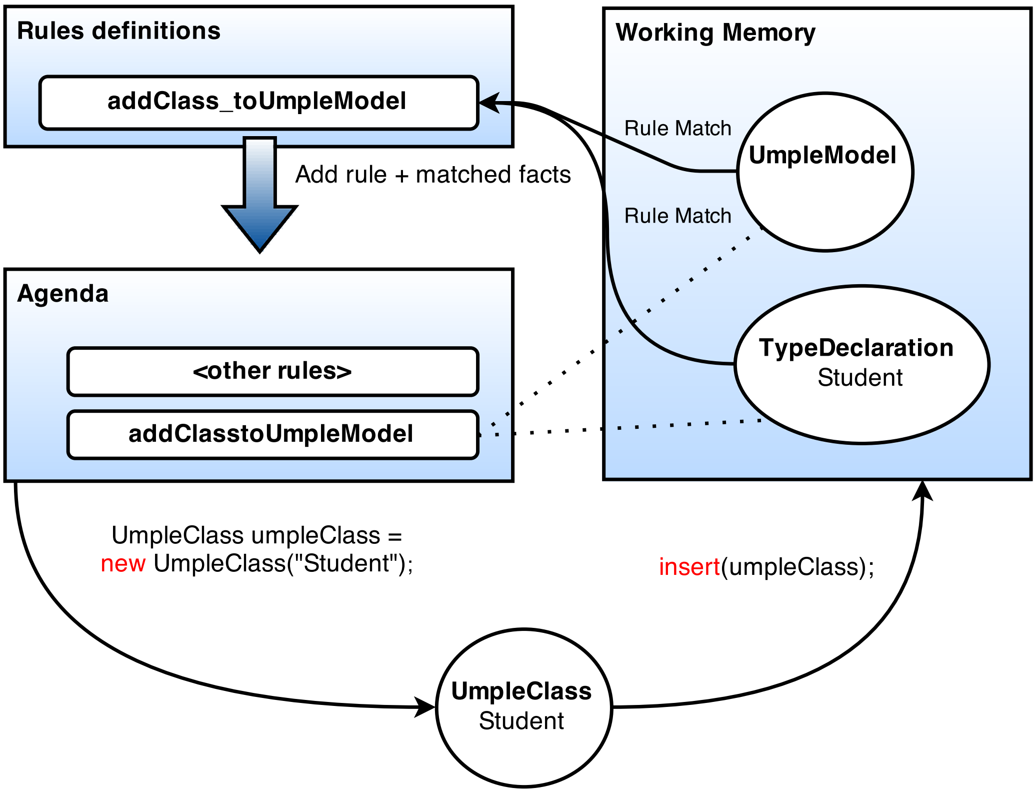


Figure 5.13: Pattern matching and creation of an UmpleClass

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 5.24 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 5.24: Rule transformImportDeclaration

```
import function cruise.umplificator.rules
       .TopLevelAnalyzerHelper.getDeclarationContainerName

rule "transformImportDeclaration"
 agenda-group "LEVEL1"
 when
  importDeclaration: ImportDeclaration()
  uClass: UmpleClass()
  eval(uClass.getName()
       .equals(getDeclarationContainerName(importDeclaration)))
 then
  Depend depend = new Depend(getImportName(importDeclaration));
  uClass.addDepend(depend);
  update(uClass);
 end
```

The package declaration is converted then into a namespace with the rule 'transform-Namespace' in Listing 5.25. We again ensure that the package declaration corresponds to the targeted UmpleClass. Note that in this rule we do not need to insert the namespace object into memory since we do not expect any rule to match it.

Listing 5.25: Rule transformNamespace

```
import function cruise.umplificator.rules
       .TopLevelAnalyzerHelper.getDeclarationContainerName
rule "transform_Namespace"
 agenda-group "LEVEL1"
 when
  packageDeclaration: PackageDeclaration()
  uClass: UmpleClass()
  eval(uClass.getName()
      .equals(getDeclarationContainerName(packageDeclaration)))
 then
  uClass.addNamespace(packageDeclaration.getName()
                      .getFullyQualifiedName());
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, or association end. 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 5.26 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 5.27.

Listing 5.26: Rule appendFieldDeclaration

```
import function cruise.umplificator.rules
       .TopLevelAnalyzerHelper.getDeclarationContainerName

rule "appendFieldDeclaration"
 agenda-group "LEVEL1"
 when
  fieldDeclaration: FieldDeclaration()
  uClass: UmpleClass()
  eval(uClass.getName()
       .equals(getDeclarationContainerName(fieldDeclaration)))
  eval(!uClass.getExtraCode().contains(fieldDeclaration.toString()))
 then
  uClass.appendExtraCode(fieldDeclaration.toString());
  update(uClass);
end
```

The *eval* clauses in Listing 5.26, ensure that the string representing the field information has not been appended before and (as before) that the field string is added to the Umple class owning it.

Listing 5.27: Rule appendMethodDeclaration

```
import function cruise.umplificator.rules
       .TopLevelAnalyzerHelper.getDeclarationContainerName

rule "appendMethodDeclaration"
 agenda-group "LEVEL1"
 when
  method: MethodDeclaration()
  uClass: UmpleClass()
  eval(uClass.getName().equals(getDeclarationContainerName(method)))
  eval(!uClass.getExtraCode().contains(method.toString()))
 then
  uClass.appendExtraCode(method.toString());
  update(uClass);
end
```

The *eval* clauses in the listing above, ensure that the string representing the method information has not 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 5.28. 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 5.28.

This concludes the initial transformation step.

Listing 5.28: 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();
    }
}
```

### 5.6.2 Automated Umplification of Attributes

As an example of the pattern matching for rules in Level 2 (attributes), consider the same code excerpt from Listing 5.22. The process (parsing, extraction, transformation) 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.

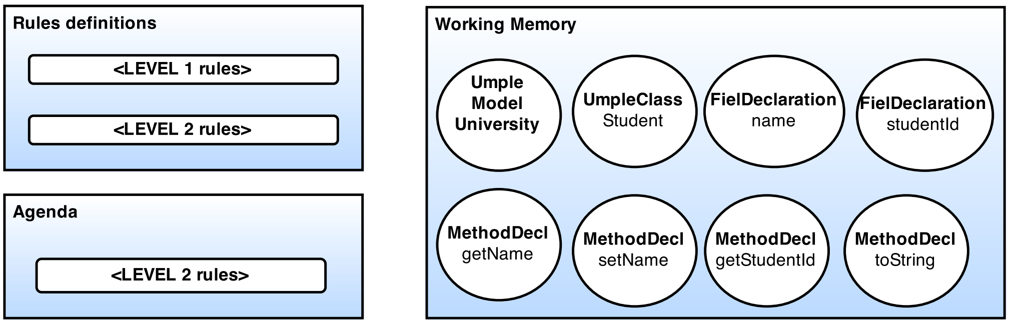


FIGURE 5.14: Rule Engine snapshot after initial transformation

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 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 instance of an Umple attribute exists or has been added so far. Rule *'Field_CanBeUmpleAttribute'* is presented in Listing 5.29.

LISTING 5.29: Rule FieldCanBeUmpleAttribute

```
rule "FieldCanBeUmpleAttribute"
agenda-group "LEVEL2"
 when
        fieldDeclaration: FieldDeclaration()
        uClass: UmpleClass()
        method: MethodDeclaration()
        eval(isPrimitiveOrStringOrTime(fieldDeclaration))
        eval(uClass.getName()
             .equals(getFieldCuATTontainerName(fieldDeclaration)))
        eval(uClass.getName().equals(getMethodContainerName(method)))
        eval(uClass.getAttribute(getFieldName(fieldDeclaration))== null)
        eval(hasFieldAGetter(method, fieldDeclaration, uClass.getName()))
 then
        String attrName = getFieldName(fieldDeclaration);
        String attrType  = getAttributeType(fieldDeclaration);
        Attribute uAttr =
          new Attribute(attrName, attrType, null, null, false, uClass);
        uAttr.setModifier("settable");
        uClass.addAttribute(uAttr);
        removeClassField(fieldDeclaration, uClass);
        update(uClass);
        insert(uAttr);
end
```

The rule in Listing 5.29 creates an Umple attribute with information extracted from a field declaration. It does so if and 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 of that class. 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 used to produce 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 possesses 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 the previous subsections 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 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.

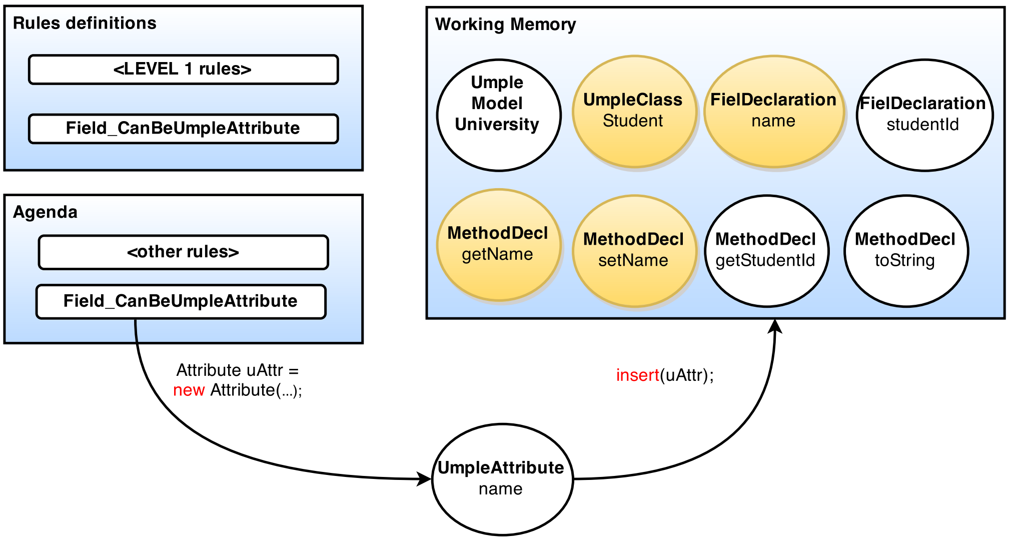


FIGURE 5.15: Pattern matching and creation of an UmpleClass

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 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's metamodel) is created to take into account the code differing from the original getter/setter.

Finally, the rule named *isLazyAttribute* in Listing 5.30, 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 5.30):

- 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 derive 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 5.30: 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 updated 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 5.31. This concludes our second transformation step.

Listing 5.31: Umple code generated – Level 2

```
namespace university.student;

class Student {
 depend java.util.Date;

 lazy String name;
 Integer studentId;

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

### 5.6.3 Automated Umplification of Associations

As an example of the pattern matching for rules in Level 3 (associations), consider the code excerpt from Listings 5.32 - 5.33 . Assuming that the rules for the transformation of the classes, package declarations, import declarations and attributes have already been performed, we focus exclusively now on how the field declarations are transformed into Umple associations. In this particular example, we are expecting two Java field declarations to be transformed into association variables. Note that an Umple association is composed of two association ends. Each of the ends is represented by an association variable. In addition, take special note that the input code in this particular example is Umple code.

Listing 5.32: Student.ump

```
namespace university;

class Student {
 Integer id;
 lazy Boolean isActive;
 immutable name;
 const Integer MAX_PER_GROUP = 10;
 after getName {
  if (name == null) {
   throw new RuntimeException("Error");
  }
 }
  public Mentor mentor;
  public Mentor getMentor() {
   return mentor;
  }
  public void setMentor(Mentor mentor) {
   this.mentor = mentor;
  }
}
```

Listing 5.33: Mentor.ump

```
namespace university;
class Mentor {

 depend java.util.Set;
 isA Person;

 Mentor() {}
 public Set<Student> students;
 public Set<Student> getStudents() {
  return students;
 }
 public void setStudents (Set<Student>students) {
  this.students = students;
 }
 public void addStudent( Student aStudent){
  students.add(aStudent);
 }
 public void removeStudent(Student aStudent) {
  students.remove(aStudent);}
 }
 public String toString() {
      return(
         (name==null ? " " : name) + " " +
         students.size()+ " students"
      );
 }
}
```

The parser receives the source code from the mentioned listings, 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 into pieces by means of a Java class visitor. Table 5.7 presents all the Java Elements collected by the Java visitor. An important fact that we have not revealed so far is that the parser is able to parse a sequence of *independent* Java statements. As is required here, the Java parser will ignore the Umple code from the code excerpts in Listings 5.32-5.33 and will simply take into account the Java elements found. Contrary to what we have presented in the previous subsection (automated umplification of attributes), a TypeDeclaration will not be generated by the parser as shown in Table 5.7.

Table 5.7: The input Java model elements produced

| **ASTNode Type** | **Source Fragment** |
|---|---|
| **FieldDeclaration** | ”public Mentor mentor; ” |
| **MethodDeclaration** | ”public Mentor getMentor() ...” |
| **MethodDeclaration** | ”public void setMentor(Mentor mentor) ...” |
| **FieldDeclaration** | ”public Set <Student>students; ” |
| **MethodDeclaration** | ”public void setStudents (Set <Student >students) ” |
| **MethodDeclaration** | ”public void addStudent(...)” |
| **MethodDeclaration** | ”public void removeStudent(..)” |

The Transformer receives the Java model elements in Table 5.7, together with a newly created instance of an UmpleModel and places them into the Working Memory. At this point of time, as illustrated in Figure 5.16, the Working Memory contains the Umple model, two Umple Classes (Student and Mentor) and the Java model elements in Table

5.7. The Production Memory contains all the 'LEVEL1', 'LEVEL2' and 'LEVEL3' rules and the Agenda only those from 'LEVEL3' since the ones from 'LEVEL1' and 'LEVEL2' have already been executed and removed from the stack. Objects with blue border are those related to class Student and those with red border are related to class Mentor. Depend and namespace declarations have been omitted here for brevity.

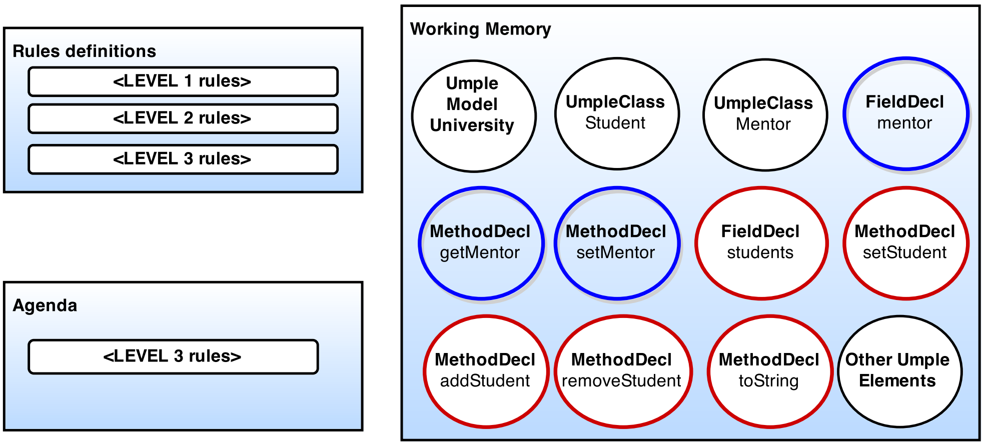


FIGURE 5.16: Rule Engine snapshot before the automated umplification of associations

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 associations ('LEVEL3'). In particular, the unique rule that can be executed at this moment is the rule named *'FieldCanBeUmpleAssociation'* since all other rules require an instance of an Umple association in memory. As we have illustrated before in Figure 5.16, no instances of an Umple association or an association variable exist or have been added so far. Rule *'Field_CanBeUmpleAssociation'* is presented in Listing 5.34.

LISTING 5.34: Rule FieldCanBeUmpleAssociation

```
rule "Field_CanBeUmpleAssociation"
agenda-group "LEVEL3"
no-loop
 when
  fieldDeclaration: FieldDeclaration()
  uClass: UmpleClass()
  method: MethodDeclaration()
  eval(!isPrimitiveOrStringOrTime(fieldDeclaration))
  eval(uClass.getName().equals(getFieldContainerName(fieldDeclaration)))
  eval(uClass.getName().equals(getMethodContainerName(method)))
  eval(uClass.getAssociationVariable(getFieldName(fieldDeclaration))==
    null)
 then
  String fieldType = getFieldType(fieldDeclaration);
  String fieldName = getFieldName(fieldDeclaration);
```

```
15    AssociationVariable assocVar = new AssociationVariable(fieldName,
        fieldType,"", "", null, false);
16    insert(assocVar);
17  end
```

The rule in Listing 5.34 creates an Association Variable with information extracted from a field declaration. It does so if and only if the following conditions are met:

- In Lines 5,6,7. We require instances of a field, Umple class and method declarations in order to assess whether or not the field can become an Umple association variable. These elements need to be found in the Working Memory.
- In Line 8. The field needs to be of a reference type.
- In Line 9. The field needs to be declared in the Class used to produce 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 already possess an association variable with the name of the current instance of the field declaration. This is to avoid duplicates.

If a field (and other model elements) matches the above conditions. A new Umple association variable is created with the information extracted from the instance of the field declaration (Line 14).

The class AssociationVariable has a single constructor which takes the following arguments:

1. String aName: corresponds to the rolename.
2. String aType: corresponds to the type declared.
3. String aModifier: defines the type of association (symmetric, reflexive). This argument is optional.
4. String aValue: correspond to the value provided as initialization (if any).
5. Multiplicity aMultiplicity: upper and lower bounds of the association variable.

6. boolean aIsNavigable: specifies if the association variable is navigable from the other association end.

Also, it has other important fields used to refine certain characteristics of the variable (navigability, composition, etc.).

Finally, in Line 20, the association variable is put into the working memory so subsequent transformations can be made such as determining if the association variable can become part of an association. Given the fact that at this point we are not completely certain that this association variable will become an association end, the association variable is *not* added to the current instance of the Umple class matched (i.e. 'uClass.addAssociationVariable(assocVar)'). In our example, an association variable with name 'mentor' of type 'Mentor' has been created when matching elements of class 'Student' and another one with name 'students' of type 'Student' when matching elements of class 'Mentor'. Refer to rule 'MatchOtherAssociationEnd' which can be found in our main repository [67].

We then attempt to determine whether the association variable is part of an association. Mutator and Accessor methods are analyzed at this point. This has been discussed in Chapter 4. When the conditions are met, the association variable is added to the corresponding Umple class. Subsequent rules determine the multiplicity and navigability of the association ends. For instance, the lower bound of the multiplicity is obtained by analyzing the constructor and the upper bound by analyzing the field declaration itself. In our example, the multiplicity of our association variable 'students' is determined as follows:

- Since the class possesses an empty constructor, so no instances of class Student are created when initializing an instance of class Mentor, the lower bound of multiplicity is '0'.
- Since the field declares a collection of Student objects, the upper bound of multiplicity is '*' (many).

Once the multiplicity has been initialized for both association variables, an instance of an Umple association is created and added to *only one* of the two Umple classes related. By default, if the association is bidirectional, we insert the association in the first Umple class retrieved from the Working Memory. In cases where the association is not navigable from both ends, the association is inserted in the class in which the association is visible.

We are not done yet since we need to remove or refactor the getters and/or setters of the field (that became an association). In fact, the field declarations as well as the method declarations enumerated in Table 5.7 are removed from the 'extra code' of Umple classes 'Mentor' and 'Student', respectively. This is achieved by means of the helper functions 'removeClassField(fieldDeclaration, uClass)' and 'removeMethod(method, uClass)'. The 'removeClassField(fieldDeclaration, uClass)' is presented in Listing 5.35.

Listing 5.35: Function removeClassField

```
public static  void removeClassField(FieldDeclaration field, UmpleClass
   uClass){
 String beforeExtraCode = uClass.getExtraCode();
 String toRemove = field.toString();
 String afterExtraCode = beforeExtraCode.replace(toRemove,"");
 // Clean extracode from class
 uClass.resetExtraCode();
 // Append new extraCode
 uClass.appendExtraCode(afterExtraCode);
}
```

Since there are no more rules to execute, the rule engine stops. The updated 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. This concludes our third transformation step.

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

## 5.7 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 5.8 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 '1.22.0.5146'.

Table 5.8: 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=0,1,2
      -splitModel -dir -path >
```

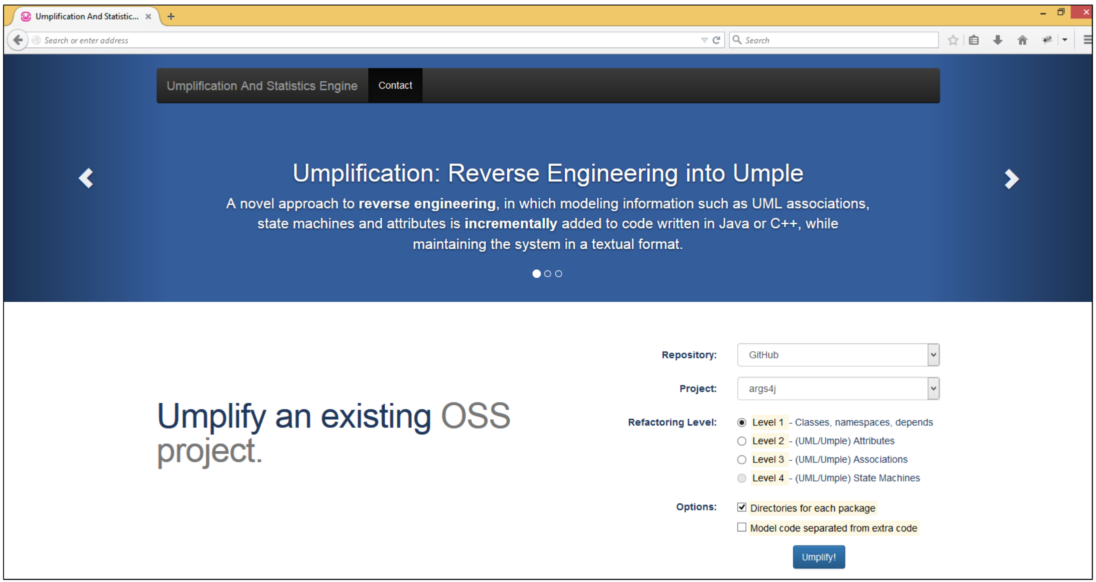
where:

- **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 we want to achieve. 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 5.17 presents the initial page of the UmplificatorOnline. An open-source project *downloader* has been implemented as part of this Web tool. The tool runs the Umplificator main Jar (umplificator_1.21.0.4666.jar) for its reverse-engineering capabilities. An additional service (Scripts) to download projects from Github, SourceForge and GoogleCode to the local system with a specific directory structure has also been developed. This is further discussed in Chapter 6.

The preliminary release of the tool allows developers to:

- Select an Open-Source **repository**: The 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 of predefined criteria. The criteria for project selection are that the project is marked as a small or medium-size system and that it 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 of the command-line tool.


Umplification And Statistic...
Search or enter address
Search
Umplification And Statistics Engine
Contact

Umplification: Reverse Engineering into Umple

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.

Umplify an existing OSS project.

Repository: GitHub

Project: args4j

Refactoring Level:
● Level 1 - Classes, namespaces, depends
○ Level 2 - (UML/Umple) Attributes
○ Level 3 - (UML/Umple) Associations
○ Level 4 - (UML/Umple) State Machines

Options:
☑ Directories for each package
☐ Model code separated from extra code

FIGURE 5.17: 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 Section 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 testing phase explained in this section concerns only the testing process 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.

At present there are over 135 tests that span all components of the tool (parser, extractor, transformer, generator) and are run as part of our automated building process.

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

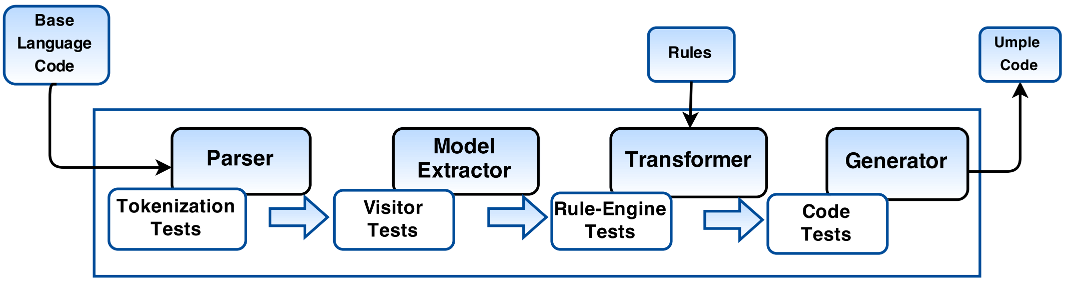


FIGURE 6.1: Umplificator Testing Infrastructure

### 6.1.1 Testing the Base Language Code Parsers

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

A simple parser test 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, is shown in Listing 6.1. In this particular parser test, we are expecting two problems (compilation errors) since the input compilation unit contains two errors at two different locations in the code. The input code has been omitted.

Listing 6.1: A Parser Test

```
@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 presented in Listing 6.2:

Listing 6.2: A pattern for parser tests

```
@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 [25]) to traverse the different elements of the retrieved base language model, our tests certify that the visitor returns the desired number of elements.

For instance, if the test input file contains (Listing 6.3):

Listing 6.3: 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';
  byte b = 100;
  short s = 10000;
  int i = 100000;
  double d1 = 123.4;
  long creditCardNumber = 1234_5678_9012_3456L;

  InputForVisitorTest () { }

  InputForVisitorTest(byte b) {
   this.b=b;
  }

  public int getB(){
   return b;
  }
}
```

in the unit test shown in Listing 6.4 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).

Listing 6.4: A JavaVisitorTest.java

```
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  {
  String packageName = visitor.getPackageDeclaration().getName().
    getFullyQualifiedName();
  Assert.assertEquals("cruise.umplificator.visitorTestFiles", packageName
    );
 }

 @Test
 public void methods_returned()
 {
  int nbMethods = visitor.numberOfMethodDeclarations();
  Assert.assertEquals(3, nbMethods);
 }
```

### 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 presented in Listing 6.3, we expect all the assertions in the test class *RulesAttributesTypesTest* to pass. The JUnit test class, as shown in Listing 6.5, contains a method *setUp()* that specifies the work to be done before running each test such as initializing instance variables.

In Line 18 we parse the input file and create an Umple class that is inserted into the working memory of the Rule Engine. In Line 22, an Umple class is inserted into the Working memory (of Drools Rule Engine). Note that in Line 21 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. Finally, we clean the working memory of the rule engine (Line 28).

Listing 6.5: RulesAttributesTypesTest class

```
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);
```

```
 visitor = new JavaClassVisitor();
 JavaParser javaParser = new JavaParser();
 javaParser.parseUnit(code);
 visitor = javaParser.getJavaVisitor();
 uClass = new UmpleClass("Test");
 kieSession = ruleService.startRuleEngine(RefactoringLevel.ATTRIBUTES);
 kieSession.insert(uClass);
}
 // test cases ...

 @After
 public void tearDown() throws Exception {
  runner.dispose();
 }
}
```

The unit test *testNumberOfObjectsInWorkingMemory* in Listing 6.6 ensures that at this point of time, there is only one element in the working memory (the Umple class inserted before) since the mapping rules have not been fired yet.

Listing 6.6: Test asserting the working memory contents

```
 @Test
 public void testNumberOfObjectsInWorkingMemory() {
   Assert.assertEquals(1, kieSession.getObjects().size());
 }
```

The unit test *testCorrectMappingBetweenPrimitiveField2UmpleAttribute* in Listing 6.7 validates the mappings between the Java fields (input) and the Umple attributes.

Listing 6.7: Test asserting mappings of fields and attributes

```
@Test
public void testCorrectMappingBetweenPrimitiveField2UmpleAttribute() {
 // Insert facts into knowledge base
 for(FieldDeclaration field: visitor.getFieldDeclarations()){
  kieSession.insert(field);
 }
 // Fire rules
 kieSession.fireAllRules();
 // Is not Null
 Assert.assertNotNull( uClass.getAttribute(0));
 Assert.assertNotNull( uClass.getAttribute(1));
 Assert.assertNotNull( uClass.getAttribute(2));
 Assert.assertNotNull( uClass.getAttribute(3));
 Assert.assertNotNull( uClass.getAttribute(4));
 Assert.assertNotNull( uClass.getAttribute(5));
 Assert.assertNotNull( uClass.getAttribute(6));

 // Type has been set correctly
 Assert.assertEquals("Boolean", uClass.getAttribute(0).getType());
 Assert.assertEquals("String", uClass.getAttribute(1).getType());
 Assert.assertEquals("Integer", uClass.getAttribute(2).getType());
 Assert.assertEquals("Integer", uClass.getAttribute(3).getType());
 Assert.assertEquals("Integer", uClass.getAttribute(4).getType());
 Assert.assertEquals("Double", uClass.getAttribute(5).getType());
25  Assert.assertEquals("Double", uClass.getAttribute(6).getType());
26  // Name has been correctly set
27  Assert.assertEquals("result", uClass.getAttribute(0).getName());
28  Assert.assertEquals("capitalC", uClass.getAttribute(1).getName());
29  Assert.assertEquals("b", uClass.getAttribute(2).getName());
30  Assert.assertEquals("s", uClass.getAttribute(3).getName());
31  Assert.assertEquals("i", uClass.getAttribute(4).getName());
32  Assert.assertEquals("d1", uClass.getAttribute(5).getName());
33  Assert.assertEquals("creditCardNumber", uClass.getAttribute(6).getName()
      );
34 }
```

The following gives details of Listing 6.7:

- In Lines 4-6 the fields declarations of the Java class are inserted into the Working Memory.
- The mapping rules are fired in Line 8.
- Line 10-16: We assert that the Rule Engine has correctly mapped the field declarations into Umple attributes. The Umple class must now contain 6 attributes.
- In Lines 19-28 and 27-33, we assert that the name and type of the field declaration has been correctly assigned to the Umple attribute.

The unit test *testCorrectMappingBetweenImport2Depend* in Listing6.8 also ensures the correct mapping between the input Java import declarations and the Umple depends clause.

Listing 6.8: Test asserting mappings of imports and depends

```
@Test
public void testCorrectMappingBetweenImport2Depend() {
for(ImportDeclaration importDecl: visitor.getImportDeclarations()){
 kieSession.insert(importDecl);
}
kieSession.fireAllRules();
 Assert.assertEquals(2, uClass.getDepends().size());
 Assert.assertEquals("java.util.*",uClass.getDepends().get(0).getName());
 Assert.assertEquals("java.io.*",uClass.getDepends().get(1).getName());
}
```

### 6.1.4 Testing the Umple Code Generator

Testing the code generator involves asserting that from an 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.

Listing 6.9: A code generator test

```
@Test
public void JavaToUmple_VariablesToAttributes_003(){
 String fileName = "003_JavaToUmple_VariablesToAttributes";
 File javaFile = new File(pathToRoot+fileName+"_java.java"); //INPUT
 File umpleFile = new File(pathToRoot+fileName+"_umple.ump"); //OUTPUT
 // Umplify file. Process must succeed!
 assertTrue(umplificator.umplifyElement(javaFile));
 // Get the output content
 assertOuputAndFile(umpleFile);
 // Clean files
 filesToDelete.add(fileName);
}
// Helper Functions
public void assertOuputAndFile(File expectedContentFile)
{
try {
 String inputFileContent = FileUtils.readFileToString(expectedContentFile);
 String outputModel  = umplificator.getOutputModel().getCode();
 assertEquals(inputFileContent, outputModel);
 } catch (IOException e) { fail();}
}
```

The test in Listing 6.9, 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 localize any defects that might occur. In other words, this multi-level testing helps to make sure that a change or addition of a new feature does not break any existing functionality and if there is any bug to quickly locate the defective component. 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

We have tested the umplificator using our own repository of 42 small Umple examples. This collection of models covers a wide variety of domains ranging from Banking to Warehouse control, and is available for review online [50].

We generated Java code from these examples using the Umple compiler, and then used the Umplificator to re-generate Umple models for each example. The process is illustrated in Figure 6.2. The goal of this pre-validation phase is to confirm that each original Umple model is regenerated correctly; in other words that it is semantically identical to the *UmpleModel'* generated by the Umplificator.

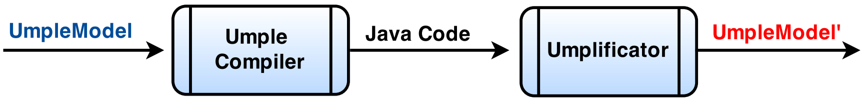


FIGURE 6.2: The pre-Validation phase: Comparing UmpleModel and UmpleModel'

Table 6.1 lists the Umple examples used in this pre-validation phase, 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 6.10 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 is shown in Listing 6.11.

LISTING 6.10: 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}
```

Table 6.1: Small examples used for first phase of validation

| **Name** | **#Umple LOC** | **#Java LOC** | **# 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 |

```
}

//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 }
}
```

Listing 6.11: 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
```

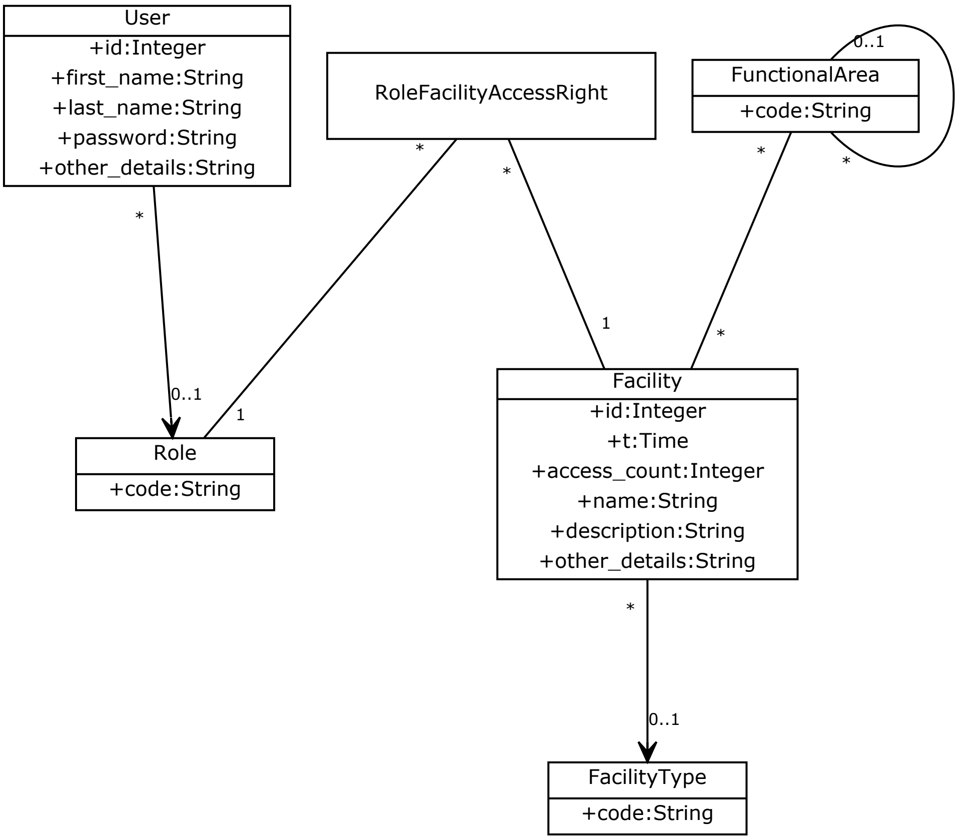


Figure 6.3: UML Class diagram of the Access Control system

```java
UmpleModel umplifiedModel = umplificator.getOutputModel();
UmpleModel expectedModel = new UmpleModel(inputUmpleFile);
expectedModel.run();
//1. Class FacilityType
UmpleClass facilityTypeA = umplifiedModel.getUmpleClass("FacilityType");
UmpleClass facilityTypeE = expectedModel.getUmpleClass("FacilityType");
Assert.assertEquals(1, facilityTypeA.numberOfAttributes());
Attribute lazyAttributeA = facilityTypeA.getAttribute("code");
Attribute lazyAttributeE = facilityTypeE.getAttribute("code");
assertEquals(lazyAttributeA.getIsLazy(),lazyAttributeE.getIsLazy());
assertEquals(lazyAttributeA.getType(),lazyAttributeE.getType());
// 2. Class User
UmpleClass userA = umplifiedModel.getUmpleClass("User");
UmpleClass userE = expectedModel.getUmpleClass("User");
Attribute id = userA.getAttribute("id");
Attribute firstname = userA.getAttribute("first_name");
Attribute lastname = userA.getAttribute("last_name");
Attribute other_details = userA.getAttribute("other_details");
Attribute password = userA.getAttribute("password");

Assert.assertEquals(userA.numberOfAttributes(), userE.numberOfAttributes
  ());
Assert.assertEquals("Integer", id.getType());
Assert.assertEquals("String", firstname.getType());
Assert.assertEquals("String", lastname.getType());
Assert.assertEquals("String", other_details.getType());
Assert.assertEquals("String", password.getType());
// 3. Facility Class
UmpleClass facilityA = umplifiedModel.getUmpleClass("Facility");
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 49 we assert that the attribute 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).

This pre-validation essentially demonstrates with a high level of confidence that the Umplificator works, at a basic level on a significant amount of Java code, but with a known and consistent structure. It was fairly easy to tailor our transformation rules to reverse-engineer code that we had generated ourselves. We know exactly what the structure of this code is; we know that it will be the same throughout each example, and we know that it will not change over time (or if it does change, at least we have control of both forward and reverse engineering). Pre-validation, says nothing about whether the Umplificator can work ‘in the wild’, against unknown code. That is what we have to do in the next stages of validation.

Our pre-validation is similar to the concept of ‘dogfooding’ [68], in that we are testing the Umplificator on our own code. But it is not ‘full’ dogfooding since we have not yet attempted to convert the whole of the Umple compiler back from generated Java to Umple; the Umple compiler is very large and has a number of complexities that will make this a challenge we leave to future work.

Table 6.2: Open-source systems umplified

| Name | Version | LOC | # of Classes |
|---|---|---|---|
| 1. JHotDraw [69] | 7.5.1 | 82132 | 694 |
| 2. Weka [70] | 3.7.13 | 278642 | 1370 |
| 3. Java Bug Reporting Tool[71] | 1.0 | 2629 | 27 |
| 4. JEdit[72] | 1.12 | 59699 | 84 |
| 5. FreeMaker[73] | 2.3.15 | 39864 | 131 |
| 6. Java Financial Library[74] | 1.6.1 | 1248 | 28 |
| 7. Args4j[75] | 2.0.30 | 2223 | 61 |

## 6.3 Initial Phase of Validation

Our next step was to apply the Umplificator to various open-source systems written in Java. We use freely available systems to facilitate 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.

Furthermore, we perform '*manual*' umplification on all open-source systems in Table 6.2. The results of the manual umplification are then compared to the results of our '*automated*' umplification.

We call this phase the 'Initial' phase of validation because this is the first phase where we show that the Umplificator can work on systems that we have not created ourselves.

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.
4. 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 Umple project).

5. We compute the *precision* and *recall* of the results previously obtained. Precision assesses the proportion of the constructs (attributes, associations) identified that are valid – in other words that were also identified by the manual umplification. Recall assesses the proportion of the independently-identified constructs that are found by our approach (a number near 1 would mean very few are missed by our detection algorithms).

6. 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 its 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 requiring additional transformations (as described in Table 3.1) was incorrectly refactored, we may review and correct the refactoring action (a function in the Drools language) that led to the incorrect piece of code.

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

The following are the definitions we have employed for the precision and recall measures [76]:

$$Precision = \frac{|TP|}{|TP \cap FP|}$$

and

$$Recall = \frac{|TP|}{|TP \cap FN|}$$

Where,

1. **TP** is the number of true positives. They correspond to modeling construct that were correctly identified by our algorithms.

2. **FN** is the number of false negatives. In the context of this work, they correspond to constructs that were missed from the automated umplification.

3. **FP** is the number of false positives. They correspond to modeling constructs that were misidentified (false alarms).

The next sub-section presents the results of this validation stage. Specifically, we will present the statistics on the number of Umple constructs recovered from the various open-source projects in 6.2 as well as the *precision* and *recall* measures. It is important to recall at this point that the resulting output artifact from manual umplification and automated umplification is in both cases an Umple model. We run our code analyzer on the resulting umple model to get statistics and the information required for our comparisons. For instance, if we run the code analyzer on two (fictitious) models, one being the result of manual umplification (*ManualModel*) and the other one the result of automated umplification (*AutomatedModel*) of the same input system and we obtain the following results:

For *ManualModel*:

1. ManualModel contains 1 class = {A}

2. Class A contains 3 attributes = {a,b,c}

For *AutomatedModel*:

1. AutomatedModel contains 1 class = {A}

2. Class A contains 3 attributes = {a,b,d}

We can then compute the precision and recall measures. In our fictitious example above, the results are as follows:

1. True positives: 2 relevant attributes were correctly retrieved {a,b}

2. False negatives: 1 attribute was missed {c}

3. False positives: 1 attribute was incorrectly retrieved {d}

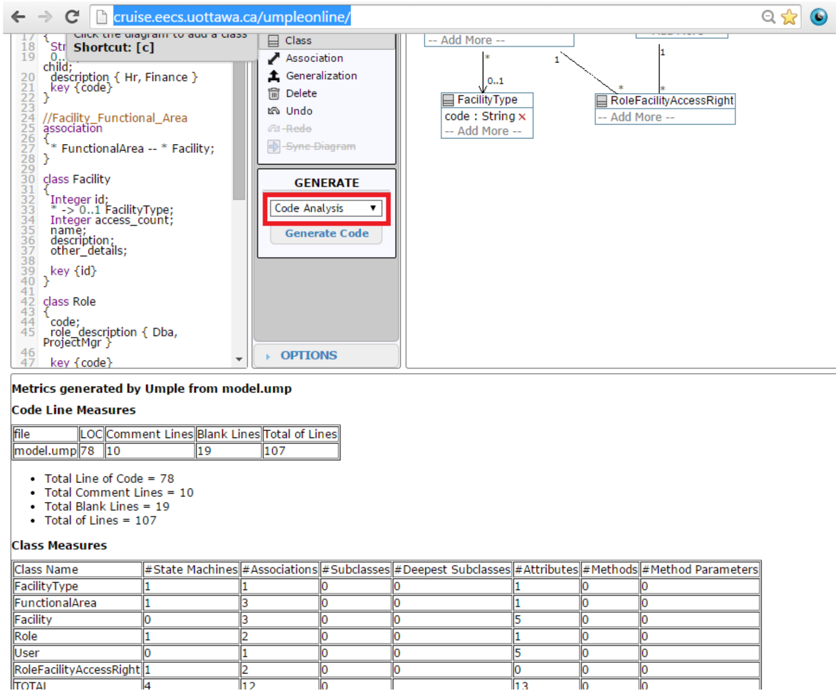
The precision and recall for the recovery of attributes for our fictitious sample system is:

1. **Precision**: TP / (TP + FP) = 2 / (2 + 1) = 66.7%
2. **Recall**: TP / (TP + FN) = 2 / (2 + 1) = 66.7%

Finally, the code analyzer mentioned before is an Umple feature that can be run on the command line:

```
java -jar umple.jar -g CodeAnalysis Test.ump
```

or using our online tool as shown in Figure 6.4.



```
{
  Str
  0..
child;
  description { Hr, Finance }
  key {code}
}

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

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

  key {id}
}

class Role
{
  code;
  role_description { Dba,
ProjectMgr }

  key {code}
```

**Metrics generated by Umple from model.ump**

**Code Line Measures**

| file | LOC | Comment Lines | Blank Lines | Total of Lines |
|---|---|---|---|---|
| model.ump | 78 | 10 | 19 | 107 |

- Total Line of Code = 78
- Total Comment Lines = 10
- Total Blank Lines = 19
- Total of Lines = 107

**Class Measures**

| Class Name | #State Machines | #Associations | #Subclasses | #Deepest Subclasses | #Attributes | #Methods | #Method Parameters |
|---|---|---|---|---|---|---|---|
| FacilityType | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| FunctionalArea | 1 | 3 | 0 | 0 | 1 | 0 | 0 |
| Facility | 0 | 3 | 0 | 0 | 5 | 0 | 0 |
| Role | 1 | 2 | 0 | 0 | 1 | 0 | 0 |
| User | 0 | 1 | 0 | 0 | 5 | 0 | 0 |
| RoleFacilityAccessRight | 1 | 2 | 0 | 0 | 0 | 0 | 0 |
| TOTAL | 4 | 12 | 0 |  | 13 | 0 | 0 |

Figure 6.4: Umple online - Code Analysis generation

## 6.3.1 Results

In the following tables, we will present the number of initially recovered attributes, and associations (of each type) for each of the systems studied. Nine different types of associations were identified in previous work [9]; we present results for three of them, corresponding to the top multiplicity patterns in industry: optional-one-to-many, optional-one-to-one and many-to-many associations.

The results below are a snapshot following the first attempt at automated umplification on these systems. That is, no tuning had been performed on the Umplificator to improve the results. Since then, after tuning the Umplificator, the precision and recall measures for all open-source systems studied in Table 6.2 has been increased to 100%.

Tables 6.3 - 6.9 detail the number of generalizations, the number of attributes and the different types of associations for the systems studied.

The second column of Tables 6.3 - 6.9 presents the number of Umple constructs that were correctly recovered (True Positives). The third column presents the number of Umple constructs that were incorrectly recovered (False Positives). The fourth column presents the number of expected Umple constructs (True Positives + False Negatives, resulting from manual umplification). The last two columns present the precision and recall computations. Note that the number of missed Umple constructs (False Negatives) can be computed by subtracting the number of correctly retrieved Umple constructs from the number of expected Umple constructs.

Let us look specifically at one of the tables, Table 6.3; this shows the results of detection of Umple constructs for the JHotDraw framework. It shows that for JHotdraw, 383 of the attributes were recovered; however only 95% were actual attributes. It turns out that, 20 1-to-1 associations were mistakenly extracted as attributes. More details concerning the Umplification of JHotdraw and Weka will be given in Section 6.5.

Table 6.3: Results of umplification for JHotDraw

| | **TP** | **FP** | **Expected** | **Precision** | **Recall** |
|---|---|---|---|---|---|
| Attributes | 383 | 20 | 363 | 95% | 100% |
| Associations optional-one-to-many | 22 | 0 | 49 | 100% | 45% |
| Associations optional-one-to-one | 115 | 0 | 185 | 100% | 62.2% |
| Associations many-to-many | 30 | 0 | 32 | 100% | 93.8% |

By studying the tables, we can see that the recall rate of our detection algorithms (before tuning) was at its absolute minimum around 38% but neared 100% in many cases. Precision ranged between 95% and 100%.

An additional step we took was to instrument our forward and reverse engineering framework components with timers to measure the time taken to process an input Java

Table 6.4: Results of umplification for Weka

| | TP | FP | Expected | Precision | Recall |
|---|---|---|---|---|---|
| Attributes | 1281 | 0 | 1510 | 100% | 84.9% |
| Associations optional-one-to-many | 168 | 0 | 442 | 100% | 38% |
| Associations optional-one-to-one | 224 | 12 | 285 | 94.92% | 86.8% |
| Associations many-to-many | 89 | 0 | 115 | 100% | 77.4% |

Table 6.5: Results of umplification for the Java bug reporting tool

| | TP | FP | Expected | Precision | Recall |
|---|---|---|---|---|---|
| Attributes | 31 | 0 | 31 | 100% | 100% |
| Associations optional-one-to-many | 9 | 0 | 12 | 100% | 75% |
| Associations optional-one-to-one | 4 | 0 | 4 | 100% | 100% |
| Associations many-to-many | 7 | 0 | 7 | 100% | 100% |

Table 6.6: Results of umplification for JEdit

| | TP | FP | Expected | Precision | Recall |
|---|---|---|---|---|---|
| Attributes | 30 | 1 | 30 | 96.8% | 100% |
| Associations optional-one-to-many | 15 | 0 | 15 | 100% | 100% |
| Associations optional-one-to-one | 22 | 0 | 22 | 100% | 100% |
| Associations many-to-many | 40 | 0 | 48 | 100% | 83.3% |

Table 6.7: Results of umplification for FreeMaker

| | TP | FP | Expected | Precision | Recall |
|---|---|---|---|---|---|
| Attributes | 140 | 0 | 140 | 100% | 100% |
| Associations optional-one-to-many | 37 | 0 | 38 | 100% | 97.4% |
| Associations optional-one-to-one | 29 | 0 | 29 | 100% | 100% |
| Associations many-to-many | 50 | 0 | 52 | 100% | 96.2% |

file and produce the target source code. More specifically, the timer measures the time taken to 1. parse an input file, 2. build (extract) an instance of the base language model, 3. transform the input model based on a predefined set of mapping rules into an Umple model, and 4. generate code (creating files with the .ump extension).

Table 6.10 summarizes the execution times in milliseconds taken to reverse engineer two

Table 6.8: Results of umplification for the Java Financial Library

| | **TP** | **FP** | **Expected** | **Precision** | **Recall** |
|---|---|---|---|---|---|
| Attributes | 15 | 0 | 15 | 100% | 100% |
| Associations optional-one-to-many | 8 | 0 | 8 | 100% | 100% |
| Associations optional-one-to-one | 30 | 0 | 30 | 100% | 100% |
| Associations many-to-many | 12 | 0 | 12 | 100% | 100% |

Table 6.9: Results of umplification for Args4j

| | **TP** | **FP** | **Expected** | **Precision** | **Recall** |
|---|---|---|---|---|---|
| Attributes | 186 | 5 | 186 | 97.4% | 100% |
| Associations optional-one-to-many | 12 | 0 | 12 | 100% | 100% |
| Associations optional-one-to-one | 4 | 0 | 5 | 100% | 100% |
| Associations many-to-many | 28 | 0 | 28 | 100% | 100% |

of the open-source systems in Table 6.2. Note that the reverse engineering component uses Perf4J [77] for calculating (and displaying) performance statistics. The values presented in Table 6.10 correspond to the average of multiple execution times as gathered by this utility. Computation times do not include the time taken to compute metrics on the software systems.

The tests were executed on a machine exhibiting the following characteristics:

- Intel Core i7-4500 CPU @ 3.10GHz
- RAM: 8.00 GB
- Windows 8 - 64 bits

Table 6.10: Reverse engineering execution times

| | ***Execution Time (in ms)*** | |
|---|---|---|
| **Component** | **JHotDraw** | **Args4J** |
| **Parsing** | 50899 | 12500 |
| **Extractor** | 21025 | 3204 |
| **Transformer** | 339327 | 920 |
| **Generating Umple Code** | 1700 | 450 |
| **Total Time:** | 412951 | 14074 |

It can be seen from the above that the transformer is the process that dominates the time, and performs substantially worse than linearly; the other processes perform better than linearly. The execution times for the Transformer process depend on the number of rules applied (based on the level of refactoring achieved) and on the size of input system (138 classes and 22 interfaces as for JHotDraw, for instance) and this explains the variations in the execution times from one system to another. Furthermore, for args4j where the code contains only a few instance variables that can become Umple constructs, the transformation stage has been performed very fast compared to the corresponding time for JHotdraw, a very rich system in terms of modeling abstractions. Note that JHotdraw possesses 81632 lines of code compared to args4j which is made of 2233 lines of code. The net result in the cases shown here is approximate linear scaling; however, given that the transformer takes so much time, it appears that for extremely large systems the time required may become extremely lengthy. On the other hand, umplification is expected to be a task that a project would only perform once per body of code, so extended transformation time might not actually pose a practical problem.

## 6.4 Second Phase of Validation

This phase of our validation approach follows the general approach used in machine learning. We take one set of systems as the ‘training set’ and then take another set as the ‘testing set’ to see how well the Umplificator performs on unknown systems. Our initial training set is the set of systems we worked with in the previous phase.

We adapt this process to be iterative; in other words, we tune the system based on problems found when working with the testing set. One type of problem is failure of the umplification to complete, at its various levels. Another type of problem is automatic umplification that does not match manual umplification of one of the test systems.

The testing set hence becomes a new ‘training set’. The idea is to pick yet another fresh testing set, and repeat. Over time the umplificator should get better and better.

To enable this form of validation, we developed a tool that randomly downloads open-source projects from repositories such as Google Code, GitHub and SourceForge.

The following are the first three steps of our process:

1. *Download* 100 projects (randomly) from the mentioned repositories based on the following criteria:
   (a) The system must be written in Java (we will extend this for C++ as part of future work)
   (b) The system must be of medium-size or small-size. We determine this using the labels provided by the repository hosting service.
2. *Umplify* the projects that have been downloaded into the file system. Attempt to umplify the project fully, and in case of any error, leave behind the result up to the last successful umplification level. For instance, if the Umplification failed at level 2 for one project, then stop the umplification for that project at level 1.
3. *Report* the results following a similar directory structure as the downloaded project. The examples presented next will clarify this idea.

Assuming that we download the projects in a directory called *downloaded_projects*. We then call the script with two positional arguments, the first is the path to the built umplificator, and the second is the path the to projects directory. That is:

```
umplifying_script.sh ~/umplificator.jar ~/downloaded_projects
```

The script, partially shown in Listing 6.12, calls the umplificator.jar to umplify each of the projects at level 0 (Line 3), if it succeeds we continue to the next level and attempt level 1 (Line 12) and so on.

Listing 6.12: Part of the script responsible to umplify and report results

```
1  # Umplify all projects
2  for dir in "${to_umplify[@]}"; do
3   java -jar $umplificator -level=0 -dir -path=$dir/umple_output $dir
4   if [ $? -eq 0 ]; then
5   # if success: output 0-file and continue to next level
6    touch $dir/0.umplify.score
7   else
8   # if fail: output f-file and move to next project
9    touch $dir/F.umplify.score
10   continue
11  fi
12  java -jar $umplificator -level=1 -dir -path=$dir/umple_output $dir
13  if [ $? -eq 0 ]; then
14  # if success: output 0-file and continue to next level
15   touch $dir/1.umplify.score
16  else
17   # if fail: move to next project
```

```
    continue
fi
    # ... code omitted
done
```

The file system is employed to report the Umplification scores; we place a file containing logging information following the same directory structure as the input system being umplified. In other words, we generate a file with a name that follows this structure '*umplify_dir/project/version/myfile*':

Where *myFile* will can be one the following:

- (Doesn't Exist) – If the Umplificator has not even been run on that version yet
- F.Umplify.Score.log – If the Umplificator failed to even umplify the version to level 0
- {012}.Umplify.Score.log – If the Umplificator succeeds on a specific level, a file starting with that level number (a project can potentially have all 3 levels and therefore can have 3 of these files).

For instance, if we are running the script on JHotdraw 7.5.1, and we have successfully umplified it at level 2, then we will be able to find a file named *2.Umplify.Score* as follows:

```
    umplify_dir/org/jhotdraw/draw/7.5.1/2.Umplify.Score.log
```

These files will be used to log the time taken to umplify, the mapping rules activated for the umplification of that unit and/or any errors. Listing 6.13 shows an example of the logging information that could be gathered in one of these files.

Listing 6.13: Logging Information sample gathered by the Umplificator

```
12:30:37 DEBUG cruise.umplificator.UmplificatorMain - Configure Logger
    from log4j.properties
12:30:37 DEBUG cruise.umplificator.UmplificatorMain - Umplification
    process started!
12:30:37 DEBUG cruise.umplificator.core.Umplificator - start time :
    1427733037483
12:30:37 INFO  cruise.umplificator.FileHelper - Files to be umplified:
    Student.java
12:30:37 DEBUG cruise.umplificator.rules.RuleRunner - Instantiate
    RuleRunner-
```

```
12:30:37 DEBUG cruise.umplificator.rules.RuleService - RuleService.
    startRuleEngine
12:30:37 DEBUG cruise.umplificator.rules.RuleRunner - RuleRunner.
    addRuleFile - Add Rules to WM
12:30:39 DEBUG cruise.umplificator.rules.RuleRunner - RuleRunner.
    insertUmpleModel
12:30:39 DEBUG cruise.umplificator.core.Umplificator - Input language is:
     .java
12:30:39 DEBUG cruise.umplificator.core.Umplificator - ---Input---
12:30:39 DEBUG cruise.umplificator.parser.JavaParser - Parsing
    Compilation Unit
12:30:40 DEBUG cruise.umplificator.parser.JavaParser - Initializing Java
    Visitor
12:30:40 cruise.umplificator.visitor.JavaClassVisitor - Visiting
    TypeDeclarations-Student
12:30:40 DEBUG cruise.umplificator.visitor.JavaClassVisitor - Visiting
    FieldDeclarations-private int id;
12:30:40 DEBUG cruise.umplificator.visitor.JavaClassVisitor - Visiting
    MethodDeclarations-setId
12:30:40.DEBUG cruise.umplificator.rules.RuleRunner - RuleRunner.
    insertUmpleClass - Insert uClass into working memory
12:30:40 DEBUG cruise.umplificator.rules.RuleRunner - RuleRunner.
    fireAllRules -     Fire rules LEVEL2
12:30:40 INFO  cruise.umplificator.core.Umplificator - Umplification
    process completed!
12:30:40 DEBUG cruise.umplificator.FileHelper - --UmpleClass has been
    created for output\Student.ump--
12:30:40 DEBUG cruise.umplificator.core.Umplificator - Elapsed Time :
    2643
12:30:40 DEBUG cruise.umplificator.core.Umplificator - Stop Time : start
    [1427733037483]
```

We then perform the following steps to complete this validation phase:

4. *Fix* issues at level 0. We check the results for each of the umplified projects. We inspect the log file for each of the failed projects (F.umplify.score files) to determine what is the root cause of the problem. We fix any issues and proceed with next step.

5. *Fix* issues at levels 1 and 2. We check this time the projects that were not umplified at level 2. We fix any issues and proceed with the next step.

6. *Collect* the statistics on the number of umplified constructs for all projects umplified.

7. *Tune further* by manually umplifying a sample. We select one of the projects for manual umplification. The results of manual umplification are compared with those of the automated umplification using the *CodeAnalyzer*. At this point, we tune the Umplificator (refer to beginning of section 6.3) until there are no discrepancies between the two umplified versions.

8. *Re-umplify.* We attempt the umplification of the initial 100 projects again, and compare the statistical results with those of Step 3. Ideally, tuning the Umplificator to fix issues in steps 4, 5 and 7 has improved the effectiveness for other projects in the set.

9. *Repeat.* We collect 100 more projects and apply steps 1-9.

For the 100 first projects used in this part of our study and after completion of steps 1 to 3, we obtained the following results:

- 100 Projects were successfully umplified at level 0. '*Successfully*' means that the resulting umple code can generate valid Java code.
- 68 Projects were successfully umplified at level 1.
- 55 Projects were successfully umplified at level 2.On-going work is being done to fix the issues (Step 5 above).

We also note the results below, after completion of steps 4 to 8:

- From the 32 umplified projects that couldn't be umplified at level 1, 18 of these umplified projects when compiled generated uncompilable source code. The main problem resided in the usage of generic types. For instance, class *BuiltInBinding* of open-source project *Dagger* in Listing 6.14 declares a generic type and is umplified as shown in Listing 6.15. In fact, this Umple class will generate a Java class that will break all code related to instantiating class *BuiltInBinding* (i.e., return new BuiltInBinding<Object>(...). Since Umple does not yet support template parameters, we have employed Umple's 'top' construct as temporary workaround (in Listing 6.16). Code inside the 'top' construct is generated into a file corresponding to the name of the 'top' construct. Generic classes in Umple will be considered future work.
- After fixing issues related to Generic types. The 18 projects were correctly umplified at level 1.
- The 14 remaining projects were successfully umplified after fixing issues related to misplaced code injections.

- Rules for the detection of one-to-many associations were tuned resulting in an improvement of 4% in the number of detected associations of this type (on average for the 100 projects).

Listing 6.14: BuiltInBinding.java

```java
public class BuiltInBinding<T> {
  private String delegateKey;
  private final ClassLoader classLoader;
  private Binding<?> delegate;
  // The rest of the code is omitted
}
```

Listing 6.15: BuiltInBinding.ump (Invalid)

```
class BuiltInBinding {
  String delegateKey;
  private final ClassLoader classLoader;
  private Binding<?> delegate;
  // The rest of the code is omitted
}
```

Listing 6.16: BuiltInBinding.ump using the Top construct

```
top BuiltInBinding {
 public class BuiltInBinding<T> {
   private String delegateKey;
   private final ClassLoader classLoader;
   private Binding<?> delegate;
   // The rest of the code is omitted
 }
}
```

The code for the script is published online and can be found in our code repository [14] in the following directory:

`cruise.umplificator/scripts/umplificator_all_projects.sh`.

The project surrounding the umplificator that automatically downloads, tracks and reports on various projects can be found at:

`https://github.com/mgarzon/dlproj`.

The umplified systems, statistical results and scoring reports are available online in our code repository [14] in the following directory:

`experiments/mgarzon/thesis/CH6Evaluation`.

## 6.5 Case Studies

We now discuss three case studies in more detail. We start by presenting our experiences with **JHotDraw** and **Weka**, the two larger systems we studied; then we present a third case study regarding **Args4j** in which we perform additional re-engineering tasks.

### 6.5.1 JHotDraw

JHotDraw7 [69] 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 [69]. For this research, we worked with JHotDraw 7.5.1.

Table 6.3 showed the results of detection of Umple constructs for the JHotDraw framework during our first attempt. After improving and refining our rules, we have obtained a precision and recall of 100%. Table 6.11 summarizes some of problems encountered during the incremental transformation of JHotDraw. The last column indicates the type of change required to fix the problem and the component concerned by the change: {Umple.Compiler, Umplificator.Parser, Umplificator.Transformer, Umplificator.Generator and/or Umple.Extractor}. The refinements consisted of adding the Java idioms that our detection algorithms were not able to catch on the first attempt and tuning the Umple compiler to generate compilable code from some of the umplified classes.

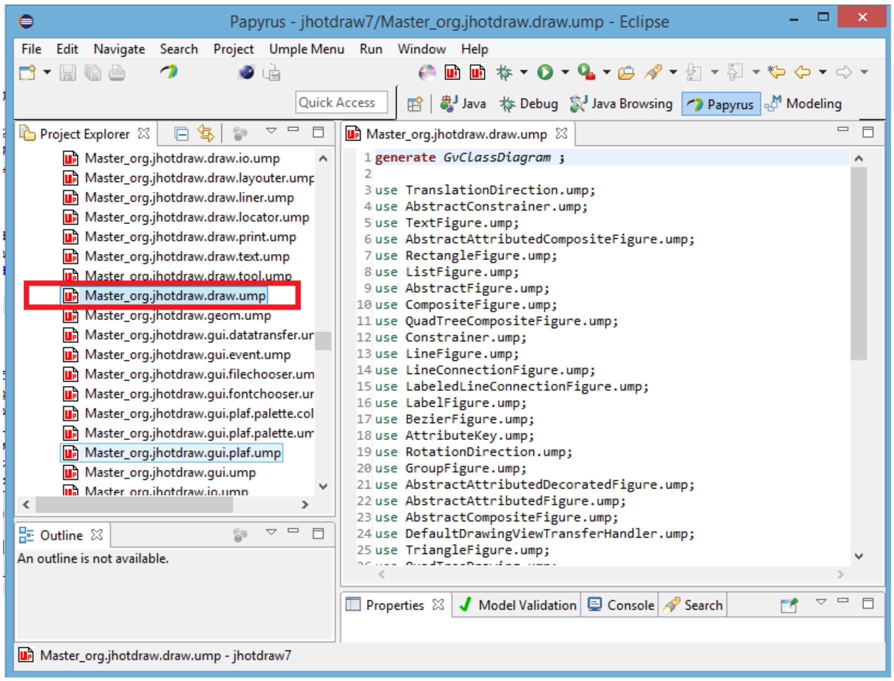
The Umplificator (Generator component) can generate master files to regroup classes per package. The Master file includes other Umple files by means of the 'use' statement. This is shown in Figure 6.5. The file named 'Master_org.jhotdraw.draw.ump' regroups all classes in package '*org.jhotdraw.draw*' and is used to generate the UML class diagram presented in Figure 6.6.

Table 6.11: Problems encountered while umplifying JHotDraw

| Problem Description | Fix |
|---|---|
| **1.** The Umplificator correctly detected Abstract Classes but since Umple did not support abstract classes at the time. Java code generated from the umplified JHotDraw was having compilation issues. | Umple.Compiler: We added the ability to declare classes as abstract classes. Refer to issue 76 in our defect tracking list [78]. |
| **2.** Annotations preceding variable declarations were correctly output by the Umplificator but not handled by the Umple compiler. That is, the Umple compiler was generating invalid code from classes having annotations preceding field declarations such as @NotNullable. | Umple.Compiler: Annotations and comments are now handled and appear at the right place in the generated code. Refer to issue 427 in our defect tracking list [78] |
| **3.** Attributes were not correctly recovered. 23 Attributes were declared as immutable when they were not. By analyzing the cases, we discovered that some fields had setters that were not following the standard conventions (i.e., setX()). For instance, in class *ArrowTip*, the setter for field angle was named 'updateAngle' rather than setAngle(...). | Umplificator.Transformer: The helper, in charge of analyzing method declarations to determine whether or not the method sets a given field, was updated to take into account this naming convention. |
| **4.** Independent enumerations (Enum X in file X.java) were correctly identified by the Umplificator; however the Umple compiler did not handle them since it could at the time only generate classes and interfaces. | Umple.compiler: We introduced a new construct named 'top' to put extra code at the top level. Refer to issue 614 in our defect tracking list [78] |
| **5.** *package-info.java* files that are used to provide a home for package level documentation and package level annotations were not handled correctly. | Umple.compiler: We employed the 'top' construct previously described. |

### 6.5.2 Weka

The next system we focused on was the machine learning tool Weka [79]. As with our the first attempt at umplifying JHotDraw, our first attempt at automatically umplifying Weka resulted in the identification 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.


Papyrus - jhotdraw7/Master_org.jhotdraw.draw.ump - Eclipse
File
Edit
Navigate
Search
Project
Umple Menu
Run
Window
Help
Quick Access
Java
Debug
Java Browsing
Papyrus
Modeling
Project Explorer
Master_org.jhotdraw.draw.io.ump
Master_org.jhotdraw.draw.layouter.ump
Master_org.jhotdraw.draw.liner.ump
Master_org.jhotdraw.draw.locator.ump
Master_org.jhotdraw.draw.print.ump
Master_org.jhotdraw.draw.text.ump
Master_org.jhotdraw.draw.tool.ump
Master_org.jhotdraw.draw.ump
Master_org.jhotdraw.geom.ump
Master_org.jhotdraw.gui.datatransfer.ur
Master_org.jhotdraw.gui.event.ump
Master_org.jhotdraw.gui.filechooser.um
Master_org.jhotdraw.gui.fontchooser.ur
Master_org.jhotdraw.gui.plaf.palette.col
Master_org.jhotdraw.gui.plaf.palette.ur
Master_org.jhotdraw.gui.plaf.ump
Master_org.jhotdraw.gui.ump
Master_org.jhotdraw.io.ump
Outline
Master_org.jhotdraw.draw.ump
Properties
Model Validation
Console
Search
Master_org.jhotdraw.draw.ump - jhotdraw7

An outline is not available.

```
generate GvClassDiagram ;

use TranslationDirection.ump;
use AbstractConstrainer.ump;
use TextFigure.ump;
use AbstractAttributedCompositeFigure.ump;
use RectangleFigure.ump;
use ListFigure.ump;
use AbstractFigure.ump;
use CompositeFigure.ump;
use QuadTreeCompositeFigure.ump;
use Constrainer.ump;
use LineFigure.ump;
use LineConnectionFigure.ump;
use LabeledLineConnectionFigure.ump;
use LabelFigure.ump;
use BezierFigure.ump;
use AttributeKey.ump;
use RotationDirection.ump;
use GroupFigure.ump;
use AbstractAttributedDecoratedFigure.ump;
use AbstractAttributedFigure.ump;
use AbstractCompositeFigure.ump;
use DefaultDrawingViewTransferHandler.ump;
use TriangleFigure.ump;
```

FIGURE 6.5: Master files used to compile JHotdraw

In addition, we detected, after execution of the test suite, that two classes were not compiling due to an unexpected constructor signature.

Another problem encountered while umplifying the Weka classes was related to inner classes. A Class such as *weka.attributeSelection.BestFirst* uses two inner classes to manage the nodes and linked lists in a best-first search. The Umplificator was attempting to match types inside other data types, resulting in an infinite loop. In addition, since the Umple compiler does not currently support inner classes, the solution chosen to remedy this situation was to ignore inner classes and process them as Umple's 'extra code'.

Initial Umplification results for Weka nonetheless had a precision of 85% when it comes to attributes and 38% for 1-to-many associations. Table 6.3 summarizes the results. Note that a precision of 38% does not 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. That is, rules in a external rule file (extension .drl) can be added to the working memory when running the Umplificator on the command line.

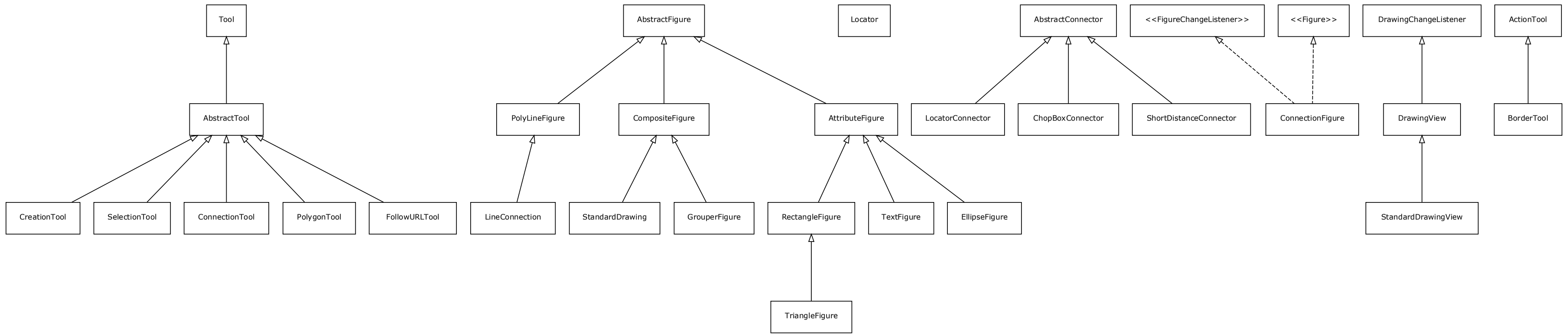


Figure 6.6: UML class diagram of package 'org.jhotdraw.draw' - JHotdraw

### 6.5.3 Args4j - Modernization of the Code Base

We reverse engineered Args4j [80], a small library that enhances the parsing of command line options and arguments in any Java application.

For every class in Args4j, our reverse engineering tool has produced two different files. The model design is expressed as an Umple model (i.e. Model.ump) and any algorithmic code is separated from the model (i.e. Model_code.ump). Developers can now use the code generation technology to generate their system in any chosen programming language from the list of available languages in Umple.

If the modeler chooses to reproduce their system in the same language as it was before the reverse engineering process, we anticipate that the generated code will be of higher quality for a couple of reasons: First, we follow a rigorous test driven development approach in all of our framework components to ensure quality. Second, we have a state-of-the art code generator that respects associations multiplicity constraints and referential integrity [9], and supports complex state machine code generation [81]. Further details on the Umple API generated from various Umple constructs can be found at [1].

In another scenario, if the modeler chooses another programming language, as a target language, to generate the system, then they need to manually convert the algorithmic code in (files of type Model_code.ump) to a new programming language such as C++. The distribution in terms of lines of code is as follows:

- Original Args4j Java source code is composed of 61 classes and 2223 lines of code.
- Umplified Args4j source code is composed of 122 (2 per input class) umple files and 1980 lines of code in total.
- Total number of lines of code in files containing modeling constructs (X.ump) is 312 LOC.
- Total number lines of code in files with algorithmic/logic code (X_code.ump) is 1668 LOC. If we exclude the umple class declarations and curly brackets, the number becomes 1424 LOC.

Consequently, to achieve the goal of translating Args4j into C++, a developer must translate 1518 lines of code (rather than 2223 lines of code). The benefits of umplification for the purposes of re-engineering may be more significant when processing larger software systems. This is marked as future work.

## 6.6 Summary

In this chapter, we have presented evaluation results showing that our approach and its current implementation are effective enough to be applied to real systems.

We have presented a multi-stage validation process evaluating the Umplificator from various perspectives and ensuring the quality of the transformations as well as the systems umplified by our tool.

In the testing phase, we have presented an approach that independently tests the components of our tool. This is a generic approach that could be applied for the testing of programming languages, transformations technologies or compilers.

In the pre-validation phase, we have tested the umplification process using our own set of examples while in the initial phase of validation we have used 7 commercial open source systems. The results show that the precision and effectiveness of the tool improves when more and more systems are umplified, and any needed tuning is performed.

Finally, the second validation phase, validates the Umplificator with a set 100 randomly selected projects. At this point of time, seven open-source and commercial projects have been fully reverse-engineered successfully and 100 randomly selected systems were successfully umplified at level 1 (attributes ).

It is our continuous 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 only in the high 80% range (i.e. not yet at 100%), our tool serves as a useful tool for umplification. Users can leave some variables un-umplified, or can manually umplify the rest.

# Chapter 7

# Related Work

This chapter surveys previous work in reverse engineering approaches, and relates them to the work presented in this thesis. We will take a close look at studies that focused on generating UML models from source code written in object-oriented programming languages. 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 literature review based on some of the guidelines proposed by Kitchenham [82]. Key parts of this systematic literature review are presented in this thesis.

**Research Question** 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 transformations. The high-level research question addressed by this study is:

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

**Search process** To search the databases, a set of strings was first created for each of the research questions based on keywords extracted from the research question, augmented with synonyms.

We designed a two-phase systematic review. In both phases, we searched for papers (including cited references) using the search engines in Table 7.1. We then performed analyses of the related work to select a subset of papers that were truly relevant.

In the first phase we paid special attention to existing surveys and literature review papers to ensure we found any literature review answering our research question. In the second phase we focused on technologies and techniques, as opposed to existing reviews.

Table 7.1: Selected sources for the literature search

| **Source** | **Acronym** |
|---|---|
| IEEE Xplore | IEEE |
| ACM Digital Library | ACM |
| Springer Link | SL |
| Elsevier | EV |
| Scopus | SC |
| Google Scholar | GS |
| Science Direct | SD |

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.

**First Phase Queries and Results** In the first phase of the review we used the following queries: Related terms such as ‘design recovery’ or ‘class diagram’ ‘recovery’ or ‘static’ or ‘dynamic’ were employed when some expected (already known/classified) papers did not appear as part of the results.

- ‘Reverse engineering’ **AND** UML **AND** ‘Survey’ 4610 results
- ‘Reverse engineering’ **AND** UML **AND** ‘systematic review’ 210 results
- ‘Reverse engineering’ **AND** UML **AND** ‘literature review’ 669 results
- ‘Reverse engineering’ **AND** UML **AND** ‘taxonomy’ 22 results

By limiting the search to titles of articles, in the manner discussed below, we reduced the search results to a total of 33 papers. After manual inspection, only 5 papers were retained [83],[18], [84], [85] and [2].

The queries used in the second phase are enumerated below:

### Second Phase Queries

- 'Reverse engineering' **AND** UML
- 'Reverse engineering' **AND** UML **AND** 'class diagrams'
- 'Reverse engineering' **AND** UML **AND** 'attributes'
- 'Reverse engineering' **AND** UML **AND** 'associations'

**Inclusion and exclusion criteria** The databases searches resulted in an extensive list of potential papers. To ensure that all papers included in the review were clearly related and relevant to the research questions, detailed inclusion and exclusion criteria were defined. These criteria are summarized in Tables 7.2 - 7.3. The process followed for filtering the search result was:

1. Use the title to eliminate any papers clearly not related to the research
2. Use the abstract and keywords to exclude additional non-related papers
3. Read the remaining papers and eliminate those that do not fulfill the criterion in Table 7.2.

Table 7.2: Inclusion criteria

| **Inclusion criteria** |
|---|
| Papers that focus on software comprehension |
| Papers that focus on software maintenance |
| Papers that focus on restructuring and re-engineering |
| Papers describing model-to-model transformation techniques |
| Papers describing model-to-text transformation techniques |

Table 7.3: Exclusion criteria

| **Exclusion criteria** |
|---|
| Short-papers, tutorials, mini-tracks and workshop papers |
| Studies whose results are not supported by any evidence or validated |

## 7.2 Results

In the next section, we list and explain the criteria employed to assess the papers retrieved after filtering the results from the second-phase queries. Only 28 papers were retained from the originally 78 retrieved. The papers retained can mainly be categorized based on what type of technique they rely on: static or dynamic analysis of source code. Tables 7.4 and 7.5 list the 21 papers. The third column of the tables indicates if the approach uses dynamic analysis, static analysis or an hybrid approach (static + dynamic).

In the next subsection, we present the most relevant (most cited and very related to our work) papers found. Although the results may convey that this area of research is mature enough, we believe there is still much work that can be done. The important fact is that the underlying research in this area makes us confident that our proposed approach contributes positively to the advancement of the reverse-engineering field.

### 7.2.1 Most Relevant Papers

A number of approaches have been presented in the literature for reverse engineering to discover associations and other related information from source code. Unlike our work, none of them is incremental or produces compilable artefacts. The majority of these produce in one single step a UML model derived from the source code [85].

In [51], Gogolla and Kollman present a technique using both static and dynamic analysis to recover UML class models from C++ source code. This approach defines one of the few techniques for finding bidirectional associations.

In [87], Barowski and Cross propose a technique to extract dependency information from Java bytecode. This approach, however, is not able to express correctly the multiplicity of the recovered associations.

Table 7.4: Filtered results from second-phase queries - Static approaches

| Title | Authors - Year - Ref | Category |
|---|---|---|
| 1. Mining design patterns from C++ source code | Balanyi et al. - 2003 - [86] | Static (Graph-Based) |
| 2. Extraction and use of class dependency information for Java | Barowski et al. - 2002 [87] | Static |
| 3. Recovering concepts from source code with automated concept identification | Carey et al. - 2007 - [88] | Static |
| 4. A Patterns based reverse engineering approach for Java source code | Couto et al. - 2012 - [89] | Static |
| 5. Recovering class diagrams from data-intensive legacy systems | Lucca et al. - 2000 - [90] | Static |
| 6. Reverse engineering co-maintenance relationships using conceptual analysis of source code | Grant et al. - 2011 - [91] | Static |
| 7. A systematic study of UML class diagram constituents for their abstract and precise recovery | Gueheneuc - 2004 - [92] | Static |
| 8. On reverse engineering an object-oriented code into UML class diagrams incorporating extensible mechanisms | Vinita et al. - 2008 - [93] | Static |
| 9. Reverse-engineering 1-n associations from Java bytecode using alias analysis | Kang et al. - 2007 - [94] | Static |
| 10. UML-based reverse engineering and model analysis approaches for software architecture maintenance | Riva et al. - 2004 - [95] | Static |
| 11. Recovering UML class models from C++: A detailed explanation | Sutton et al. - 2007 - [96] | Static |
| 12. Towards an AST-based approach to reverse engineering | Wang et al. - 2006 - [97] | Static |

Sutton and Maletic [96] propose a set of mappings that are intended to recover design-level UML class models from source code. They present their prototype tool, Pilfer, and use it to reverse engineer HypoDraw [107], an open source tool.

Gueheneuc [92] presents a technique that infers associations, aggregation and composition relationships from Java programs using a constraint-based approach. The author formalized the class relationships at the implementation level as conjunctions of four different properties: Exclusivity, invocation-site, lifetime and multiplicity properties. However, their approach requires the availability and analysis of both static and dynamic models to build the class diagrams.

Table 7.5: Filtered results from second-phase queries - Dynamic and hybrid approaches

| Title | Authors - Year - Ref | Category |
|---|---|---|
| 1. A simple static model for understanding the dynamic behavior of programs | Kelsen - 2004 - [98] | Dynamic |
| 2. Discovering Program's Behavioral Patterns by Inferring Graph-Grammars from Execution Traces | Zhao et al. - 2008 - [99] | Dynamic (Traces) |
| 3. CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code | Korshunova et al. - 2006 - [100] | Hybrid |
| 4. Reverse engineering of design patterns from Java source code | Shi et al. - 2006 - [101] | Dynamic |
| 5. Understanding applications through dynamic analysis | Antoniol et al. - 2004 - [102] | Dynamic |
| 6. Reverse engineering of object oriented code | Tonella et al. - 2005 - [103] | Hybrid |
| 7. A fully dynamic approach to the reverse engineering of UML sequence diagrams | Ziadi et al. - 2011 - [104] | Dynamic |
| 8. Mining design patterns from existing projects using static and run-time analysis | Dobis et al. - 2011 - [105] | Hybrid |
| 9. Reverse-engineering of UML 2.0 sequence diagrams from execution traces | Delamare et al. - 2006 - [106] | Dynamic |

Commercial tools and widely used open source tools, including IBM Rational Software Architect [47], Visual Paradigm [46] and ArgoUML [44] lack configurability options and incremental reverse engineering capabilities. Moreover, they detect simple attributes, as well as one-way associations between classes but produce incomplete generated code (e.g. lacking methods and/or referential integrity) and fail to preserve semantics when the UML models derived from the tools are input in their own code generators [108]. Some of the most popular reverse-engineering tools have been already discussed in Chapter 4.

Systa in her paper [109] presented a tool named Fujaba [109] that combines static analysis (graph matching) and dynamic analysis to detect pattern implementations in source code. This is so far the most popular and cited research tool relevant to this research found in the literature.

Bruneliere et al.[26] introduced in their paper a model-driven reverse engineering tool, named Modisco, which has been developed as an Eclipse project and provides a set of generic tools to understand and transform complex models created out of existing systems. MoDisco uses JDT [60] to discover Java elements in Java projects and allows the user to gather metrics and visualize results in a tree representation. Transformations can be performed on the discovered elements using another Eclipse technology, the ATL project [60], a model-to-model (M2M) transformation toolkit that we discussed earlier in this thesis.

TXL [34] is a rule-based language designed for a variety of source-to-source transformations tasks. We experimented with TXL as well as with ATL to implement umplification, and, as described earlier, conclude that the two technologies were not ideal for incremental transformation of multi-language input models (Umple, Java, Umple+Java, etc.). A core concept behind umplification is that we want tool support for incrementality. We need to be able to load and transform a target model, already (partially) transformed by our tool so that users can convert Umple to Umple in steps of their choosing, always passing test cases, and maintaining confidence of the results. Existing model transformation tools proved not readily suitable for the multi-language incremental mode of use.

Paige et al. [110] demonstrated the use of TXL in representing and for implementing efficient transformations between languages. A transformation from UML-to-Java were presented as part of their work.

### 7.2.2 Evaluation for Papers on Reverse-Engineering into UML

In this section, we list and define several important characteristics that we believe contribute to the comprehension of software systems. Our criteria emphasize the extraction of modeling constructs from source code.

The evaluation criteria we will use are as follows:

- Scalability: This refers to how well the solution works when the size of the system being studied increases.
- Incrementality: This refers to the ability of the approach to be performed in incremental and small steps.

- Validation: Has the solution been validated by other software practitioners? Fully if so, Partial if it has been validated only by the authors.
- Usability: Is the tool easy to set up or use? N/A means that a tool is not available.
- Target Programming Language: The programming language(s) that can be input into the tool.
- Attributes Recovery: Is the approach/tool able to infer attributes from source code?
- Associations Recovery: Is the approach/tool able to infer all different types of associations from source code?

The outcomes of our evaluation can be:

- *None* indicates that the criterion is not met or there is no evidence about this in the paper (or papers from the same author on the same topic).
- *Partial (P)* indicates that the criterion is partially satisfied.
- *Fully (F)* indicates that the criterion is fully satisfied and there is clear evidence of this.
- *N* Not applicable.

These criteria focus on how well the related studies solved the main topic of this thesis, i.e, reverse-engineering of object-oriented systems into UML. Note that the expected output from the techniques is UML and not Umple, as we are the first approach recovering Umple models from software systems.

Table 7.6 shows the above criteria applied to the key papers.

What mainly differentiates our work from other work is incremental refactoring, the configurability of the mappings rules, the form of the output (model and code as a single entity), and the preservation of semantics when code is generated from the recovered models.

As the last row of the table indicates, our approach performs well for most of the covered criteria. However, one of the most important characteristics of our approach is that it can be easily extended to cover those missed parts.

Table 7.6: Application of the evaluation criteria to the key papers

| **Authors-Year-Ref** | **Scalability** | **Incrementality** | **Validation** | **Usability** | **Target Prog. Lang.** | **Attribute Recovery** | **Associations Recovery** |
|---|---|---|---|---|---|---|---|
| Barowski et al. - 2002 [87] | N | N | P | N | Java | P | P |
| Grant et al. - 2011 - [91] | P | N | P | F | Java | F | P |
| Gueeheneuc - 2004 - [92] | P | P | F | F | Java | F | F |
| Vinita et al. - 2008 - [93] | N | N | N | N | Cpp | P | P |
| Kang et al. - 2007 - [94] | N | N | N | F | Java | F | P |
| Sutton et al. - 2007 - [96] | N | N | P | F | Cpp | P | P |
| Korshunova et al. - 2006 - [100] | P | N | P | F | Cpp | F | P |
| Umplification | **P** | **F** | **P** | **F** | **Java,Cpp** | **F** | **P** |

# Chapter 8

# Conclusions and Contributions

In this thesis we presented a 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.

## 8.1 Summary of Answers to Research Questions

We outline how the umplification approach and its underlying implementation addresses the research questions identified at the beginning of the thesis.

**RQ1. To what extent can we achieve automated umplification?** In Chapter 5 we addressed this research question by exploring technology alternatives and developing a tool to implement the umplification reverse engineering capabilities. The effectiveness of the transformations were then evaluated in Chapter 6. The evaluation results showed that our approach and its current implementation are effective and efficient enough to be applied to real systems. In particular we were able to successfully umplify many real open-source systems at varying degrees of effectiveness, and were able to iteratively improve this effectiveness by adjusting rules.

**RQ2. What transformation technology and transformations will work effectively for umplification?** In Chapter 5 we discussed our experiments with ATL and

TXL approaches. However in the end we selected an approach that uses a) the mature xDT family of parsing libraries (JDT, CDT, etc.) to process both code and textual models, and build in memory models; b) the Drools tool to transform the in-memory models, accomplishing the core umplification task; and c) our own generator technology to produce revised versions of the system. Details of the approach are expanded in the answer to the next research question.

**RQ3. What should be the architecture, design and implementation of an umplification tool?** The implementation of the Umplificator, combining the technologies mentioned above, was discussed in depth in Chapter 5. The advantages of the selected approach enabled the following aspects, each of which we discussed in Chapter 5.

1. Separation of concerns
2. Speed and scalability
3. Centralization of knowledge
4. Multi-level testing
5. Robust parsing
6. Efficiency
7. Agile development
8. Extensibility
9. Reusability

**RQ4. What would be an effective process for improving the accuracy of the umplification tool?** We presented in Chapter 6 an approach adapted from the model often used in machine learning that considerably improved the precision and recall of our transformations. This validation approach takes one set of systems as the 'training set' and then takes another set as the 'testing set' to see how well the Umplificator performs on unknown systems. The process is iterative; in other words, we tune the system based on problems found when working with the testing set. One type of problem is failure of the umplification to complete, at its various levels. Another type of problem is automatic umplification that does not match manual umplification of one of the test systems. To solve these problems, the Umplificator can be readily tuned by adding new mapping rules to cover 'False Negative' cases, by refining existing rules to cover 'False positive' cases or by correcting the refactoring helper functions when code injections are incorrectly inserted.

## 8.2 Contributions

The main contributions of this thesis are as follows:

1. The overall concept of umplification and the defined levels of refactoring;

2. An understanding of how umplification compares with other reverse-engineering techniques (incrementality, minimal adjustment of code to prevent disruption);

3. The implementation and analysis of integrating different transformation technologies; resulting in the Umplificator tool itself;

4. Case studies of Umplification, demonstrating strengths, weaknesses and opportunities. Results presented in this thesis are reproducible and repeatable. Popular and highly cited open-source projects such as JHotdraw and Weka are now fully rewritten in Umple. We also made improvements to Umple compiler to support abstract classes, interfaces and top-level enumerations, as needed to support the case studies.

5. Mapping rules for Umplification and the language for expressing these. We have employed a popular rule management system known as Drools. Drools offers a native rule language with a simple Java-like syntax. Rules can easily be added and refined by means of an editor with syntax highlighting and code completion.

6. Detection of associations in a body of source code written in an object-oriented; programming language. We have investigated how associations are employed in both forward and reverse engineering and proposed a set of heuristics for the detection of these in source code.

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.

## 8.3 Future Work

Although the results presented here have demonstrated the effectiveness of the umplification approach, it could be further developed in a number of ways:

**Umplify more Projects** 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.

**Incorporate other object-oriented programming languages** We plan to add support for C++ and PHP languages. In particular, we need to create new mapping rules for those idioms that were not already considered for Java. For instance, in C++ we need to analyze the content of header files. The work to be done targets exclusively the *Transformer* component of the Umplificator, since parsing and base language model generation is achieved by means of the CDT/PDT frameworks.

**State Machines Support** We also plan integrate the mapping rules for state machines and refine some of the existing rules to make them more maintainable. Our team's decision to consider state machines as the next model construct to analyze is based on the observation that there seems to be interesting research opportunities in that area.

**IDE Integration** It would be beneficial to improve tools to support better IDE integration including a step-by-step guide to indicate the level of umplification achieved of a software project. The Umplificator Eclipse plugin could be improved by adding an Eclipse perspective to better present the available features of the Umplificator to end-users.

**Evaluation with real developers** To date, we have theoretically demonstrated the benefits of the approach, as well as demonstrated its industrial practicality by reverse-engineering open-source systems with the tool. A next logical step, outside the scope of this thesis, is to evaluate its efficiency and precision with a representative set of software practitioners. In particular, it would be beneficial to subject the Umplificator to evaluation with real developers The process might work as follows: 1) Arrange for a group of users to umplify some small systems, some by hand, and others with the Umplificator. 2) Measure the speed and correctness of the result, and also administer a questionnaire to the participants. 3) Video developers as they umplify larger programs, while encouraging them to think aloud. 4) Analyze the videos looking for places where the participants make mistakes, express frustration, spend a lot of time thinking, have to refer to help, etc. 5) Evaluate in a similar manner the language used to express the mapping rules.

**Exploring other types of software** Investigating the integration of algorithms for the umplification of mobile and Web applications which require enhancing the Umple Language itself and our tool support.

**Umplifying code that matches other model elements** Investigating the detection of additional software patterns aligning with those supported by Umple (e.g keys) is also an interesting research opportunity for future researchers.

# Bibliography

[1] CRuiSE research group. Umple API summary. 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, Jan 1990.

[3] G. Canfora and M. Di Penta. New Frontiers of Reverse Engineering. In *Future of Software Engineering (FOSE '07)*, pages 326–341. IEEE, May 2007.

[4] T.C. Lethbridge A. Forward 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.

[5] A. Forward T.C. Lethbridge and O. Badreddin. Umplification: Refactoring to Incrementally Add Abstraction to a Program. *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 220–224, 2010.

[6] M. Garzon and T.C. Lethbridge. Exploring how to develop transformations and tools for automated umplification. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 491–494, Oct 2012.

[7] T. Tuunanen et al. K. Peffers. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, 2007.

[8] T.C. Lethbridge et al. 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.

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

[10] A. Forward O. Badreddin and T.C. Lethbridge. Exploring a Model-Oriented and Executable Syntax for UML Attributes. *Software Engineering Research, Management and Applications SE - 3*, 496:33–53, 2014.

[11] O Badreddin. A Manifestation of Model-Code Duality: Facilitating the Representation of State Machines in the Umple Model-Oriented Programming Language. 2012.

[12] Hamoud et al. Aljamaan. Specifying Trace Directives for UML Attributes and State Machines. In *International Conference on Model-Driven Engineering and Software Development*, pages 79–86, January 2014.

[13] Umple online. http://try.umple.org. Accessed: 2015-01-01.

[14] Umple Source Code Repository. https://github.com/umple/Umple/. Accessed: 2015-01-01.

[15] CRuiSE research group. Umple metamodel. http://metamodel.umple.org, . Accessed: 2015-01-01.

[16] Uml specification, version 2.0. http://www.omg.org/spec/UML/2.0/, key = Unified Modeling Language,, 2006. Version 2.

[17] A. Forward O. Badreddin and T.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.

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

[19] A. 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.

[20] A. Kleppe. A language description is more than a metamodel. 2007.

[21] M. Alanen et al. A relation between context-free grammars and meta object facility metamodels. Technical report, 2003.

[22] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, Jul 2006.

[23] V. Eelco. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831 – 873, 2005. Reduction Strategies in Rewriting and Programming special issue.

[24] J. Bézivin et al. On the need for megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19$^{th}$ Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[25] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Pearson Education, 1994.

[26] H. Bruneliere et al. 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.

[27] B. Yang M. Kun and H. 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.

[28] P. Van Gorp et al. T. Mens. Applying a model transformation taxonomy to graph transformation technology. *Electronic Notes in Theoretical Computer Science*, 152 (0):143 – 159, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005) Graph and Model Transformation 2005.

[29] J. Bézivin M. Peltier and G. Guillaume. Mtrans: A general framework, based on xslt, for model transformations.

[30] F. Jouault and F. Allilaire et al. ATL: A model transformation tool. *Sci.Comput.Program.*, 72(1-2):31–39, Jun 2008.

[31] X. Li D. Li and V. Stolz. Model querying with graphical notation of qvt relations. *SIGSOFT Softw. Eng. Notes*, 37(4):1–8, Jul 2012.

[32] J.M. Jézéquel et al. 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.

[33] R.F. Paige et al. D. Kolovos. 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.

[34] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, Aug 2006.

[35] M. Fowler, editor. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[36] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001.

[37] 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.

[38] T.C. Lethbridge A. Forward. 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.

[39] G. Kiczales et al. Aspect-oriented programming. In *ECOOP97 - Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.

[40] J. Chhabra and V. Gupta. Evaluation of object-oriented spatial complexity measures. *SIGSOFT Softw. Eng. Notes*, 34(3):1–5, May 2009.

[41] Java ATM System. http://www.cs.gordon.edu/courses/cs211/ATMExample/index.html. Accessed: 2015-01-01.

[42] Java ATM System. http://www.site.uottawa.ca/~mgarz042/thesis/ATM/src. Accessed: 2015-01-01.

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

[44] ArgoUML Modeling Tool. http://argouml.tigris.org/. Accessed: 2015-01-01.

[45] IntelliJ IDEA 14.1.1. https://www.jetbrains.com/idea/help/intellij-idea.html. Accessed: 2015-01-01.

[46] Visual Paradigm for UML. http://www.visual-paradigm.com/. Accessed: 2015-01-01.

[47] Rational Software Modeler 7.1. http://www-01.ibm.com/software/awdtools/modeler/swmodeler/. Accessed: 2015-01-01.

[48] Reclipse - Fujaba CASE tool. https://code.google.com/p/reclipse-emf/. Accessed: 2015-01-01.

[49] Ptidej 5.8. http://www.ptidej.net/downloads/tools/ptidejtoolsuite. Accessed: 2015-01-01.

[50] Umple Examples. https://code.google.com/p/umple/wiki/Examples. Accessed: 2015-01-01.

[51] R. Kollmann and M. Gogolla. Application of UML associations and their adornments in design recovery. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 81–90. IEEE Comput. Soc, 2001.

[52] K. Wong et al. S. Tilley. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 4(4):501–520, 1994.

[53] Java Grammar in TXL. http://www.txl.ca/nresources.html. Accessed: 2015-01-01.

[54] XML Metadata Interchange (XMI). http://www.omg.org/spec/XMI/. Accessed: 2015-01-01.

[55] F. Budinsky et al. D. Steinberg. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009.

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

[57] Java Metamodel. http://goo.gl/YV3p83. Accessed: 2015-01-01.

[58] P. Browne. *JBoss Drools Business Rules.* Packt Publishing, Oct 2009.

[59] R.P.L. Buse and W. 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, Jul 2008. ACM Press.

[60] Eclipse Java development tools (JDT). http://projects.eclipse.org/projects/eclipse.jdt. Accessed: 2015-01-01.

[61] ASTNode API. http://goo.gl/yLt2CF. Accessed: 2015-01-01.

[62] D. Xiao and X. 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, Jun 2010.

[63] 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.

[64] A. Sillitti D. Piatov, A. Janes and G. Succi. Using the eclipse c/c++ development tooling as a robust, fully functional, actively maintained, open source c++ parser. In *Open Source Systems: Long-Term Sustainability*, volume 378 of *IFIP Advances in Information and Communication Technology*, pages 399–399. Springer Berlin Heidelberg, 2012.

[65] R. Matinnejad. Agile model driven development: An intelligent compromise. In *Software Engineering Research, Management and Applications (SERA), 2011 9th International Conference on*, Aug 2011.

[66] S. Sunkle and V. Kulkarni. Cost estimation for model-driven engineering. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12, pages 659–675, 2012. ISBN 978-3-642-33665-2.

[67] Mapping Rules in Umple code repository. http://goo.gl/DFGkZB. Accessed: 2015-01-01.

[68] L. Ash. *The Web Testing Companion: The Insider's Guide to Efficient and Effective Tests*. Wiley technology publishing : timely, practical, reliable. Wiley, 2003.

[69] JHotDraw. http://www.jhotdraw.org/. Accessed: 2015-01-01.

[70] Weka Repository. https://svn.cms.waikato.ac.nz/svn/weka/. Accessed: 2015-01-01.

[71] Java Bug Reporting Tool. https://code.google.com/p/jbrt/. Accessed: 2015-01-01.

[72] jEdit. http://sourceforge.net/projects/jedit/. Accessed: 2015-01-01.

[73] FreeMaker. http://freemarker.org/. Accessed: 2015-01-01.

[74] Java Financial Library. http://freecode.com/projects/jfl/. Accessed: 2015-01-01.

[75] args4j. https://github.com/kohsuke/args4j. Accessed: 2015-01-01.

[76] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Inc., 1992.

[77] Perf4j - performance statistics for Java code. http://perf4j.codehaus.org/. Accessed: 2015-01-01.

[78] Umple Defect Tracking List. https://code.google.com/p/umple/issues/list. Accessed: 2015-01-01.

[79] M. Hall et al. The WEKA data mining software. *ACM SIGKDD Explorations Newsletter*, 11(1):10, November 2009.

[80] H. Aljamaan M. A. Garzn and T. C. Lethbridge. Umple: A framework for model driven development of object-oriented systems. In *Proc. SANER*, pages 494–498. IEEE, 2015.

[81] O. Badreddin et al. Enhanced Code Generation from UML Composite State Machines. In *Modelsward 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, pages 235–245, 2014.

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

[83] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, February 2004.

[84] B. Cornelissen et al. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, 35(5):684–702, Sept 2009.

[85] M. Nelson. A survey of reverse engineering and program comprehension. *arXiv preprint cs/0503068*, 1996.

[86] Z. Balanyi and R. Ferenc. Mining design patterns from c++ source code. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 305–314, Sept 2003.

[87] L.A. Barowski and J. H.Cross. Extraction and use of class dependency information for Java. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 309–315, 2002.

[88] M. Carey and G. Gannod. Recovering concepts from source code with automated concept identification. In *Proc. 15th IEEE Intl Conf. Program Comprehension*, 2007.

[89] R. Couto et al. A patterns based reverse engineering approach for java source code. In *Software Engineering Workshop (SEW), 2012 35th Annual IEEE*, pages 140–147, Oct 2012.

[90] A.R. Fasolino G.A. Di Lucca and U. De Carlini. Recovering class diagrams from data-intensive legacy systems. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 52–63, 2000.

[91] S. Grant et al. Reverse-engineering co-maintenance relationships using conceptual analysis of source code. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 87–91. IEEE, 2011.

[92] Y. Gueheneuc. A systematic study of UML class diagram constituents for their abstract and precise recovery. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 265–274, 2004.

[93] J. Vinita and A. Jain et al. On reverse engineering an object-oriented code into uml class diagrams incorporating extensible mechanisms. *SIGSOFT Softw. Eng. Notes*, 33(5):9:1–9:9, August 2008.

[94] C. Park Y. Kang and C. Wu. Reverse-engineering 1-n associations from java bytecode using alias analysis. *Inf. Softw. Technol.*, 49(2):81–98, Feb 2007.

[95] T. Systa C. Riva, P. Selonen and J. Xu. Uml-based reverse engineering and model analysis approaches for software architecture maintenance. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 50–59, Sept 2004.

[96] A. Sutton and J.I. Maletic. Recovering UML class models from C++: A detailed explanation. *Information and Software Technology*, 49(3):212–229, 2007.

[97] X. Wang and Y. Xiaojie. Towards an ast-based approach to reverse engineering. In *Electrical and Computer Engineering, 2006. CCECE '06. Canadian Conference on*, pages 422–425, May 2006.

[98] P. Kelsen. A simple static model for understanding the dynamic behavior of programs. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 46–51, June 2004.

[99] Z. Chunying and K. Ates et al. Discovering program's behavioral patterns by inferring graph-grammars from execution traces. In *Tools with Artificial Intelligence, 2008. ICTAI '08. 20th IEEE International Conference on*, volume 2, pages 395–402, Nov 2008.

[100] M. Petkovic et al. E. Korshunova. Cpp2xmi: Reverse engineering of uml class, sequence, and activity diagrams from c++ source code. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 297–298, Oct 2006.

[101] S. Nija and R.A. Olsson. Reverse engineering of design patterns from java source code. In *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pages 123–134, Sept 2006.

[102] G. Antoniol et al. Understanding web applications through dynamic analysis. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 120–129, June 2004.

[103] P. Tonella. Reverse engineering of object oriented code. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 724–725, May 2005.

[104] M.A.A. Da Silva et al. T. Ziadi. A fully dynamic approach to the reverse engineering of uml sequence diagrams. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, pages 107–116, April 2011.

[105] Michal M. Dobis and L. Majts. Mining design patterns from existing projects using static and run-time analysis. In Zbigniew Huzar, Radek Koci, Bertrand Meyer, Bartosz Walter, and Jaroslav Zendulka, editors, *Software Engineering Techniques*, volume 4980 of *Lecture Notes in Computer Science*, pages 62–75. Springer Berlin Heidelberg, 2011.

[106] R. Delamare and B. Baudry. Reverse-engineering of uml 2.0 sequence diagrams from execution traces. In *in Workshop on Object-Oriented Reengineering at ECOOP06*, 2006.

[107] HypoDraw. http://www.slac.stanford.edu/grp/ek/hippodraw/. Accessed: 2015-01-01.

[108] P. Selonen et al. R. Kollmann. A study on the current state of the art in tool-supported uml-based static reverse engineering. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 22–32, 2002.

[109] T. Systa. Understanding the behavior of java programs. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 214–223, 2000.

[110] R. Paige and A. Radjenovic. Towards model transformation with txl. *Metamodelling for MDA*, page 162.