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

February 2015

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

Edsger W. Dijkstra

UNIVERSITY OF OTTAWA

Abstract

Faculty of Graduate and Postdoctoral Studies
Computer Science

Doctor of Philosophy

by Miguel A. Garzón Torres

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

Acknowledgements

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many software systems experience growth and change for an extended period of time. Maintaining consistency between documentation and the corresponding code becomes challenging. This situation has long been recognized by researchers, and significant effort has been made to tackle it. Reverse engineering is one of the fruits of this effort and has been defined as the process of creating a representation of the system at a higher level of abstraction [1].

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

There is a long and rich literature in reverse engineering [2]. Most existing techniques result in the generation of documentation that can be consulted separately from the code. Other techniques generate models in the form of UML diagrams that are intended to be used for code generation of a new version of the system. The technique discussed in this paper goes one step further: It modifies the source code to add model constructs that are represented textually, but can also be viewed and edited as diagrams. The target language of our reverse engineering process is Umple [**?**], which adds UML and other constructs textually to Java, C++ and PHP.

We call our approach to reverse engineering a software system umplification. This is a play on words with the concept of 'amplification' and also the notion of converting into Umple. In our previous work [3], we have found that umplifying code is reasonably

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

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

1.1 Research Questions

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

Developers currently often work with large volumes of legacy code. Tools exist to allow them to extract models or transform their code in a variety of ways. However doing so tends to result in a system that is quite different in syntax and structure. They are thus inhibited from using reverse engineering tools except to generate documentation. The Umple technology partly solves this problem by allowing incremental addition of modeling constructs into familiar programming language code. This allows developers to maintain the essential 'familiarity' with their code as they gradually transform it. Converting to Umple (Umplification) has been done manually indeed it was applied to the Umple compiler itself [3] but it ought to have tool support so it can be done in a more automatic, systematic and error-free manner on large systems.

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

1.2 Hypothesized Solutions

1.3 Research Activities

The major steps in the methodology are the following:

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

1.4 Thesis Contributions

1.5 Outline

This thesis proposal is organized as follows.

Chapter 2

Chapter 2 presents background research, a brief introduction to Umple and its modeling constructs. Additionally, it reviews related and relevant work; by understanding the existing technologies and research on the topic we should be able to efficiently navigate our research topics. Covered in this chapter are existing technologies in reverse engineering and model-to model transformations.

Chapter 3

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

Chapter 4

Chapter 4 presents three different technologies that were explored as part of our re-search activities. We evaluate ATL, TXL and JDT to see to which extent they could fulfill our needs. ATL and TXL are two famous model-to-model transformation technologies and JDT is a complete Java Framework used as part of the Eclipse IDE. We discuss all the design decisions and propose a set of tools and technologies that our prototype tool will use. Design and implementation decisions are not final and will be enhanced as our research progresses.

Chapter 5

Chapter 5 presents an analysis of our reverse engineering technique, the design approaches and our prototype tool. We also present the mapping rules employed to perform the transformations.

Chapter 6

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

Chapter 7

Chapter 7 concludes this proposal by listing all the research contributions expected out of our research. We also summarize our research activities and give and outline of our direction for the final thesis.

Chapter 2

Background

This chapter presents the required background knowledge for readers to fully understand the following chapters. We introduce the Umple language and we present the most important concepts about model-to-model transformations and some of the most relevant reverse engineering techniques.

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 [5]. 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 [6].

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 [7].

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 possibly parallel substates, and activities that operate in concurrent threads [8].

4. **Traits**: A trait is a partial description of a class that can be reused in several different classes, with optional renaming of elements. They can be used to describe re-usable custom 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 enabling understanding of the behavior of a complex set of state machines operating in multiple threads and class instances [9].

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, C++ as well as export to XMI and other UML formats. The compiler generates various types of methods including mutator, accessor, and event methods from the various Umple features. A mutator (e.g. set(), add()) method is a method used to control changes to a variable and an accessor (e.g. get()) method is the one used to return values of the variable. An event method triggers state change. An extended summary of the API generated by Umple from attributes, associations, state machines and other features can be found at [10]. Umple can also generate diagrams, metrics, and various other self-documentation artifacts. Umple models can be created or edited using the UmpleOnline Web tool [11], the command line compiler or an Eclipse plugin.

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

2.1.1 Umple Attributes

An Umple attribute is a property of an object. For instance, a Person object might have a *name* and an *address*. Depending on the properties that the attribute may possess, an attribute can be:

2.1.1.1 Basic Attribute

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 through Tables **??** and 2.2, the implications on code generation include a parameter in the constructor and a simple set and get methods to manage access to the attribute. The String datatype in umple is the default type, when no type is specified. The example below shows multiple attributes having different (umple) datatypes.

```
class Demo
{
  name; // String type
  Integer i;
  Float flt;
  String str;
  Double dbl;
```

```
  Boolean bln;
  Date dte;
  Time tme;
}
```

2.1.2 Immutable Attribute

An immutable attribute is the one that does not change during the lifetime 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. Briefly, a constructor argument is required so it can be set at construction time but it cannot be changed after since no setter is generated. The syntax for an immutable attribute is shown below. In this example, the *studentId* must be initialized during construction and cannot changed after it.

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

2.1.3 Defaulted

A defaulted attribute is set in the constructor to the 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.

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

2.1.4 Unique

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

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

2.1.5 Autounique

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.

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

2.1.6 Constant

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.

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

2.1.7 Array

Umple supports attributes that might contain multiple values. The square brackets notation '[]' is used as shown below:

```
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 **??** and 2.2 presents the list of accessor and mutator methods generated from Umple attributes. In Tables **??** 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

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

TABLE 2.2: API generated methods (mutator) from Umple attributes

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

2.1.8 Umple Associations

In Umple (UML) an association defines a relationship from a class to another class. Furthermore, it specifies which links such as references or pointers may exist at run time

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

- **Associations 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:** The roles names are used to clarify the relationship and avoid collision if two classes are associated in multiple ways. Role names are optional except in reflexive associations.

The following code segment illustrates an association between instances of classes *School* and *Person*. In this example, an instance of class School can be associated to zero or more instances of class Student. The 'isA' notation is used to denote an inheritance relationship between the classes (Student is a subclass of Person).

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

Alternatively, in addition to showing an association embedded in one of the associated classes, it is also possible to show an association independently.

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

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

Tables **??** and **??** present the list of accessor and mutator methods generated from Umple associations. In Tables **??** and 2.2, 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.

2.1.9 Umple State Machines

2.1.10 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 you to enforce preconditions and **after** statements to enforce postconditions. Code injections (after and before statements) can be added into the constructor 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 2.1: A code injection into the constructor

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

The following gives details of the above:

Line 2. Declares a constant (static final in Java).

Line 3. Declares a simple (String) attribute.

Line 4-9. Declares a code injection to be inserted at the beginning of the constructor.

Line 10-12. Declares a code injection to be inserted at the end of the constructor.

The code in Listing 2.1 generates the following (Java) constructor:

```
1 public Operation(String aQuery)
2 {
3   if (aQuery == null)
4   {
5     throw new RuntimeException("Please provide a valid query");
```

```
6   }
7   query = aQuery;
8   if (DEBUG) { System.out.println("Created " + query); }
9 }
```

2.2 Reverse Engineering

In the following chapter, we will describe the core concept of this thesis, the umplification technique, a model-to-model transformation technique that aims at incrementally transforming base language code into Umple code. Later, we will discuss which of the above languages proved most useful for umplification.

Chapter 3

Reverse Engineering of Object Oriented Systems into Umple

Developers often work with large volumes of legacy code. Reverse engineering tools allow them to extract models in a variety of ways [5], 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 per-form 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 is

that the end-product of umplification is not a separate model, but a single artifact seen as both the model and the code. In the Umple world, modeling is programming and vice versa. More specifically, for a programmer, Umple looks like a programming language and the Umple code can be viewed as a traditional UML diagram. This allows developers to maintain the essential 'familiarity' with their code as they gradually transform it into Umple [6]. 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 [7] as the abstraction level of the program has been 'amplified'.

3.1 Umplification Process

Umplification involves recursively modifying the Umple model/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. 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 with a small amount of additional modeling information, such as the presence of one new type of UML construct. At each step, the system remains compilable. The approach proceeds incrementally performing additional transformations until the desired level of abstraction is achieved. These incremental transformations allow for user interaction to provide needed information that may be missing or hard to automatically obtain because the input (the source

code) does not follow any of the idioms the automatic umplification tool is yet able to recognize. This characteristic of umplification allows developers, if they wish, to repeatedly re-introspect the transformed program and manually validate each change with an understanding of the incremental purpose of the change.

The approach is **transformational** because it modifies the original source rather than generating something completely new. It first translates the original language (Java, C++ etc.) to an initial Umple version that looks very much like the original, and then translates step-by-step as more and more modeling constructs are added, replacing original code.

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

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

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

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

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

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

Transformation 1: Transformation of generalization/specialization, dependency, 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 representing dependency are transformed

into Umple 'depends' clauses, and notations for namespaces or packages are transformed into the Umple 'namespace' directives. At this stage, an Umple program, when compiled should generate essentially identical code to the original program.

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

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

Transformation 2b: Transformation of variables in one or more classes to UML/Umple associations If variable a is declared in Class A and the type of a is a reference type B, then a is trans-formed 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 bidirec-tional. 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; this is explained in Section 3.3.

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 paper, 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. Not covered further in this paper

In the following chapter, we provide a more detailed view of the transformation cases. To help distinguish between Umple and Java code presented in this thesis, the Umple examples appear in dashed borders with grey shading, pure Java examples have solid borders with no shading. Mapping rules (in the Drool language, that we will describe shortly) appear using double line borders with no shading.

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 a object-oriented programming language. The methodology followed to ensure that our approach is able to identify Umple/UML constructs in source code in most of the situations, involves four main steps:

1. Identify in the literature the typical implementations of attributes, associations and state machines in high level programming languages. We have included the CASE tools aiming at generating code from state machine models (Forward engineering).
2. Identify in the literature the techniques aiming at discovering the modeling constructs in object-oriented source code (Reverse Engineering).
3. Inspect various open source system written in object-oriented programming languages and verify that the existing techniques for reverse-engineering of modeling constructs can detect them.

At the end of the chapter, we will present the set of mapping rules derived from our analysis.

4.1 Initial Refactoring

The first step in umplification (Transformation 0) is to rename the Java/C++ files as .ump files. After this, various syntactic changes are made (Transformation 1) to adapt the code to Umple's notations for various features that are expressed differently in Java and C++. Umple maintains its own syntax for these features so as to be language-independent. First the base language notation for inheritance (e.g. 'extends' in Java) or interface implementation (e.g. 'implements') is changed into the Umple notation 'isA'. This Umple keyword is used uniformly to represent the generalization relation-ship for classes, interfaces and traits. The same notation is used for all three for flexibility so that, for example, an interface can be converted to a class with no change to its specializations, or a trait can be generated as a superclass in languages such as C++ where multiple inheritance is allowed. After this, the dependency notation in the native language (e.g. 'import' in Java) is changed to the 'depend' notation in Umple. Finally 'package' declarations are transformed into Umple namespace declarations. Transformations made as part of these first refactoring steps, are one-to-one direct and simple mappings between constructs in the base language and Umple. No methods need changing. The final output after execution of the above transformations, is an Umple model/program that can be compiled in the same manner as the original base language code. At this point, any available test cases may be run to ensure that the pro-gram's semantics are preserved. For instance, the Java code (in file Student.java) shown below:

4.2 Member Variables Analysis

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

4.2.1 Refactoring to Create Attributes

In this sub-section, we present how member variables possessing certain characteristics are transformed into Umple attributes (Transformation 2a). An Umple attribute is a simple property of an object, but following UML semantics, it is more than just a plain private variable: It is designed to be operated on by mutator methods, and accessed by accessor methods. 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 [12]. In Table 4.1, we present the developed (programmable) heuristics used for the partial analysis of member variables. The instance variables with a low or very low probability of being attributes are ignored for now. Those with high and medium probability are further analyzed.

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

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

Furthermore, we check the type of the candidate attributes (those with a High or Medium probability) and draw a conclusion regarding whether or not the member variable corresponds to an Umple Attribute, because some will be left to be later transformed into associations. If the candidate attribute has as its type either: a) a simple data type, as in Table 4.2 or b) a class that only itself contains instance variables meeting conditions in a and b (for attributes with 'many' multiplicity), then the member variable is transformed into an Umple Attribute.

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 not simple, but are custom. Simple getters/setters are those that only return/update the attribute value. Custom getters/setters are those that provide

TABLE 4.2: Umple Primitive Data Types

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

behavior apart from setting the variable such as validating constraints, managing a cache or filtering the input. Let us now illustrate this refactoring through an example. Assume that we have already trans-formed the Java class into an Umple class, so the input at this point is an Umple file containing Java. In this example code we first analyze the member variables to determine the following: Is the field present in the parameters of the constructor?

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

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

4.2.2 Refactoring to Create Associations

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

TABLE 4.3: Accessor Methods parsed and analyzed

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

TABLE 4.4: Mutator methods parsed and analyzed

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

- 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 class in which the variable is declared, stores, access and/or manipulates instances of the variable type.

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

A simple example is presented now to summarize the main idea behind this transformation step. Assume that Umple code shown below has already passed through the two first refactoring steps. As a result, classes, dependencies, and attributes (if any) have been properly extracted.

4.2.3 Refactoring to Create State Machines

Chapter 5

The Umplificator Technologies

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

5.1 The Umplification tool support goals

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

Our objective is to create an accurate tool that can enable developers to efficiently recover the model from existing software systems written in an object-oriented programming language. The Umplificator should provide extensible mechanisms to create and define transformation rules. In fact, the most important goal for a successful reverse engineering environment is that it must provide an extensible toolset [12]. The extensibility should be present in all the different operations of the tool such as parsing the input source code, transforming the source code and presenting the information. The end-user should be able to provide their own tools for these activities or to extend the ones already provided. The high-level **general** and **specific** requirements for the tool are presented below. General requirements are the ones that every reverse engineering

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

General Requirements

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

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

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

From the developer's perspective:

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

5.2 Alternative Approaches Studied

We have explored two different and famous model transformation technologies with the purpose of umplifying a software system: TXL [13] and ATL [14]. In the following two sub-sections we present the mapping rules, grammar and program directives that allowed us to transform a Java Program into Umple.

5.2.1 TXL

TXL [13] is a programming and rule-based language and rapid prototype system designed for implementing source transformation tasks.

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 parsing the new tree to finally produce the output text. In TXL, grammars and transformation rules are specified in the TXL programming language. The TXL processor is responsible for interpreting both the grammar and mapping rules by using an internal tree-structured bytecode. TXL programs depend on no other tools or technologies and can run on any platform directly from the command line. TXL programs are composed of a **base grammar**, which specifies the syntactic forms of the input structure, a set of **grammar overrides**, which extend the grammar to be used and a set of **transformation rules and functions**, that specify how the input structure will be transformed to produce the desire output structure.

The **grammar** in TXL is a description of the structure to be transformed in EBNF in a context-free ambiguous form.

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

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

Where *LeftHSPattern* and *RightHSPattern* are term patterns. The result of a mapping rule is the instantiation of the *RightHSPattern* and is produced when the term matches the *LeftHS_Pattern* and the condition is true. Rules are applied recursively until they

fail. Functions are similar to Rules but they are applied once on the entire function input.

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

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

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

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

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

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

5.2.1.1 Java to Umple Implementation

In this section, we describe the design process. Next, we describe the implementation of the JavaToUmple program that partially converts Java code to Umple. Lastly, we

provide examples of transformations rules in the TXL language. Figure 5.1 presents the components of the TXL **JavaToUmple** program.

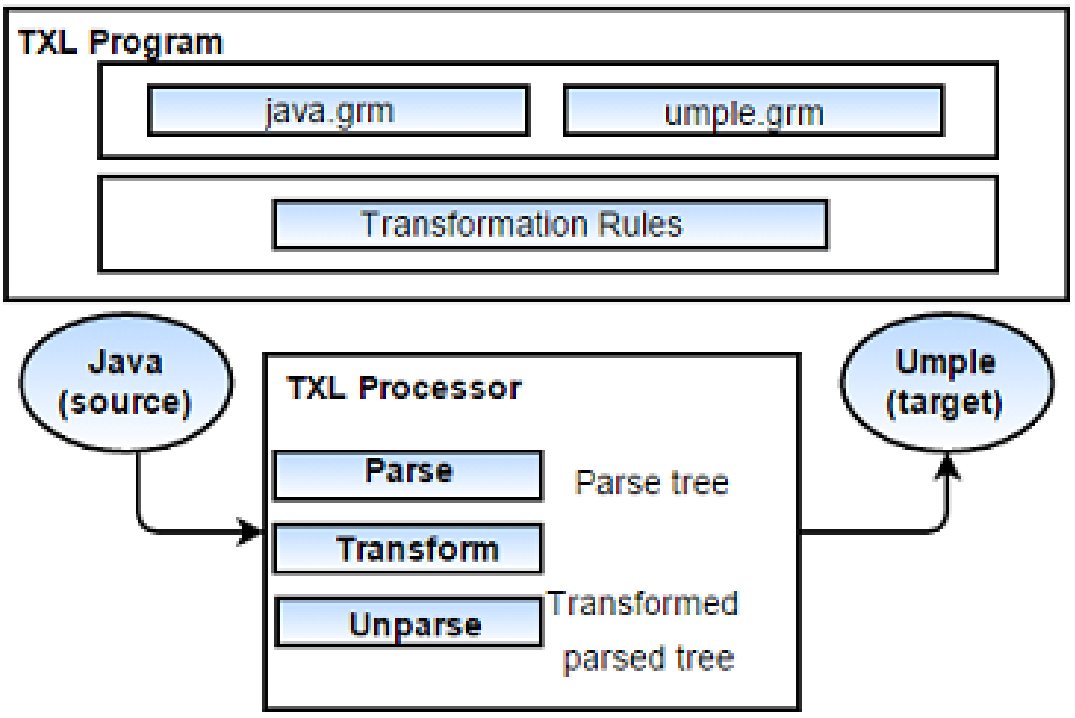

FIGURE 5.1: TXL Program for transforming Java to Umple

5.2.1.2 Design process of the TXL Program

The first step in writing a source transformer is writing working grammars for both the target and the source language and then writing a union grammar that accepts constructs for both languages. A grammar for Java 1.5 is available from the TXL website [REF]. 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 model that starts the program. Below, we briefly describe the different modules:

- JavaToUmple.Txl: This is the main program. It is used by TXL to match an input Java program against the Java Grammar and to call the transformation rules.
- TranslateMembers.Rul: Contains rules and functions to transform nested declarations.

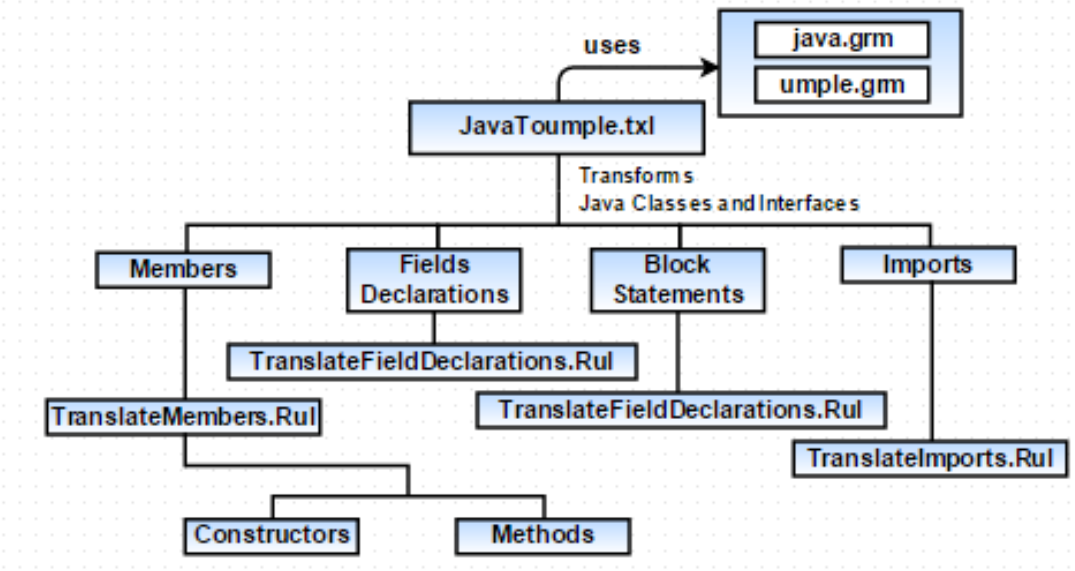

FIGURE 5.2: Structure of the JavaToUmple program

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

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

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

5.2.1.3 JavaToUmple - Transformations Examples

In this 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.

Example 1: Transforming the Class Header

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

LISTING 5.1: "Class definition grammar in BNF form"

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


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

LISTING 5.2: Class definitions in Java and Umple

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

The mapping rule called '*changeClassHeader*' 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
```

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 - 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
```

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**' 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 when the Umple code is generated. The rule below removes all the import declarations. The main program, presented next, illustrates how the program executes the mapping rules in order to produce the output.

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
```

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. 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 it has been shown in Figure 5.2.

5.2.2 ATL

ATL (ATL Transformation Language) [14] is a model transformation language that provides ways to produce a set of target models from a set of source models and allows

users to define model-to-model transformations in both a declarative and imperative way. ATL has been developed in Eclipse as a set of plug-ins by the Institut National de Recherche en Informatique et en Automatique (INRIA) as an answer to the Object Management Group's QVT language request for proposals [REF]. 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 core idea behind an ATL transformation.

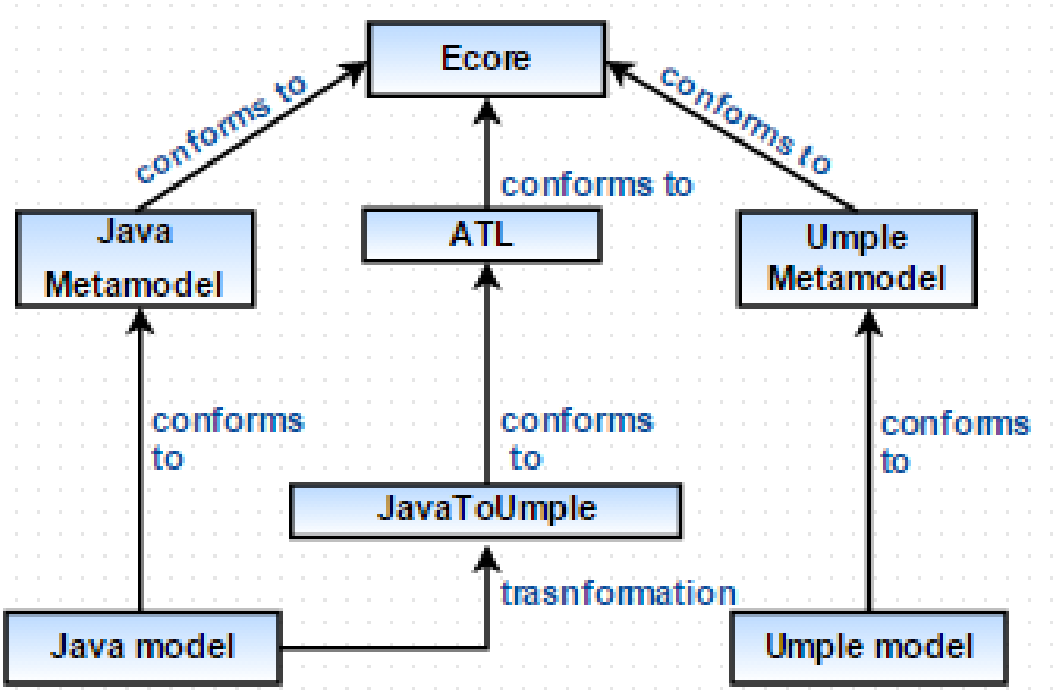

FIGURE 5.3: The JavaToUmple ATL program

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 expressions 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 cant 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 kind of rules in ATL based on the way they are called and how they specify the results: matched rules, lazy rules and called rules.

- **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 imperative code.

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), 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.

5.3 Discussion

5.4 The Umplificator

In this section, we provide an overview of the tool we have developed to support umplification; as well as discuss some of its technical details. Our tool is called the Umplificator.

The 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 [15]. Drools is a rule management system with a forward- and

backward-chaining rules engine. The rule engine is explored in more detail in Section 4.2.

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 [16].
- Umplificator Workflow A tool that guides the umplification process within Eclipse.

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. 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.

5.5 Architecture

The Umplificator has a layered and pipelined 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. The process of umplifying a system in this architecture is described below (Figure 5.5).

1. The input is a set of source code files in the base language and/or Umple.
2. The source code is transformed into base-a model of the base language and Umple constructs.

3. 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 predefined mapping rules.

4. The target Umple model, is then validated.

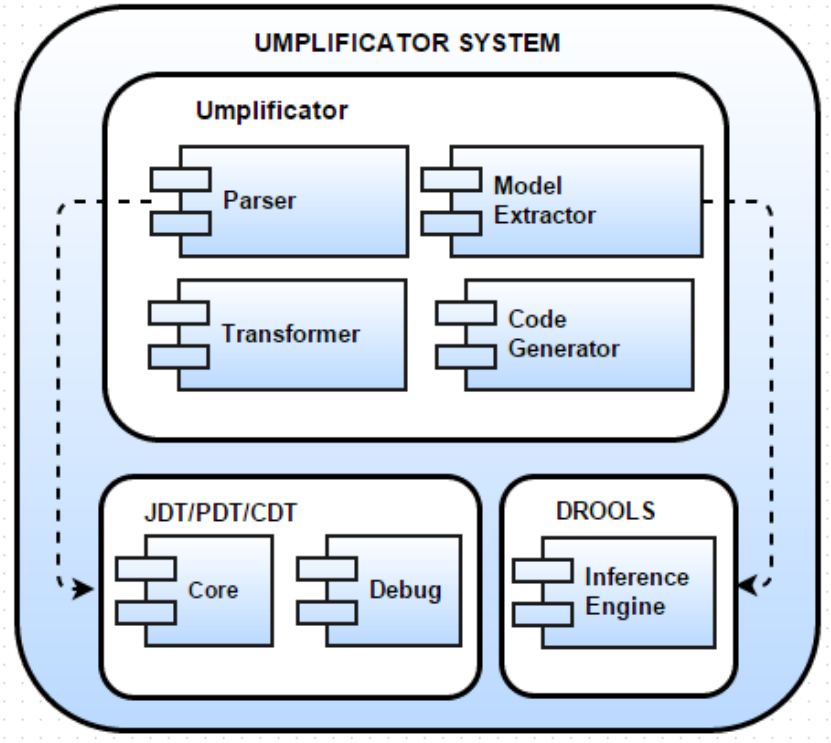

FIGURE 5.4: The Umplificator components

The mapping rules and rule engine are introduced in the following sub-section.

5.5.1 Rule Based Language

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 (LHS and RHS) using first order logic for reasoning over knowledge representation. Pattern matching is performed to match facts against rules and is implemented using the Rete algorithm [15]. The rule engine is initialized with the rules. A Drools rule has the basic form:

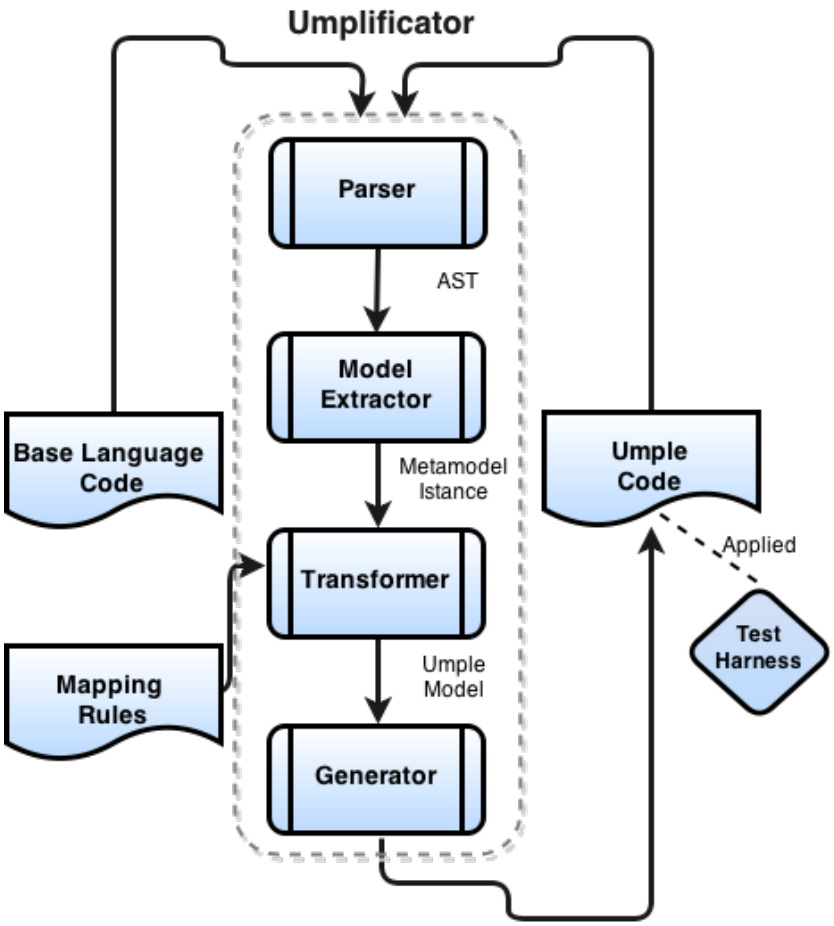

FIGURE 5.5: The umplification process flow

LISTING 5.12: Basic rule in Drools

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

where LHS is the conditional part of the rule and RHS is a block that allows dialect-specific semantic code to be executed. 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 variables into attributes, another file containing those to transform variables into associations and so on. The rules as explained in this paper 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. Additionally, in Drools, one can specify:

- **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. In the

Umplificator, functions are used instead of helper classes so the logic is kept in one place.

- **Queries**: These provide a means to search working memory and store the results under a named value. In the Umplificator, they are used to gather metric information about the models analyzed. For instance, a 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.

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. Traditionally, rule engines have two methods of execution [16]: forward chaining and backward chaining. In forward chaining, the facts are asserted into working memory resulting in one or more rules being concurrently true and scheduled for execution. In backward chaining (goal driven), one starts with a conclusion, which the engine tries to satisfy. Drools is a Hybrid Chaining System because it implements both forward and back-ward mechanism. 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 refactoring).

As an example, consider the rules in Listing . The rule named transformImport(Lines 1-10) matches and converts any Import Declaration (Java Language) into an Umple depend construct. The dependency (Line 8) is then added to a matched Umple Class. The Umple Class is then put into the working memory (Line 9) so subsequent transformations can be made on the object (forward chaining). The rule named JavaFieldIsUmpleAttribute converts Java fields into basic Umple attributes. The attribute is then added to a matched Umple Class (Line 24). The attribute is put into the working memory (Line 25) so subsequent transformations can be made such as determining if the attribute is lazy or not. The rule named isLazyAttribute, not shown here, is used for this purpose. This rule matches and converts any basic attribute (in memory) that conforms to the required conditions into a lazy attribute (e.g. attribute.setIsLazy(true)). The complete set of mapping rules for the umplificator can be found at the umple code repository [17].

LISTING 5.13: Initial Refactoring Mapping Rules

```
1  rule  "transform_Import"
2  when
3   import: ImportDeclaration();
4   uClass: UmpleClass() ;
5  then
6   Depend depend = new Depend(getImportName(import));
7   uClass.addDepend(depend);
8   insert(uClass);
9  end
```

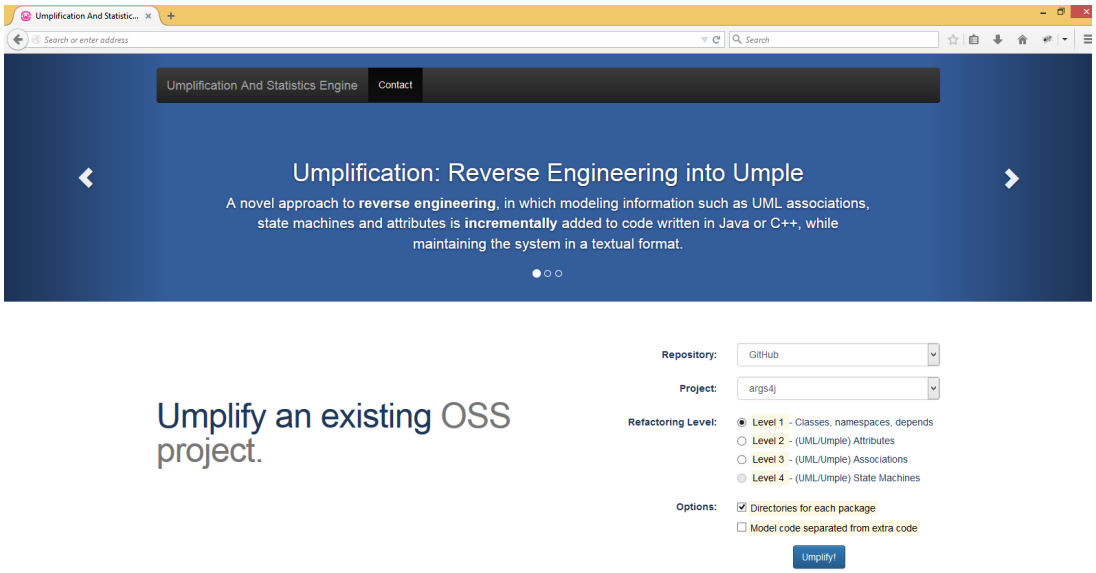

FIGURE 5.6: The Umplificator online - A PHP Web application

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 the transformations steps on various software systems and the results of the detection, adopting a **four-phase validation process**. The four validation phases of our approach are introduced below:

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

Pre-validation Phase In this phase, small Java systems written in high quality Java code are employed to validate the accuracy of the transformations performed by the Umplificator.

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

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

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

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

6.1 Testing Phase

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

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

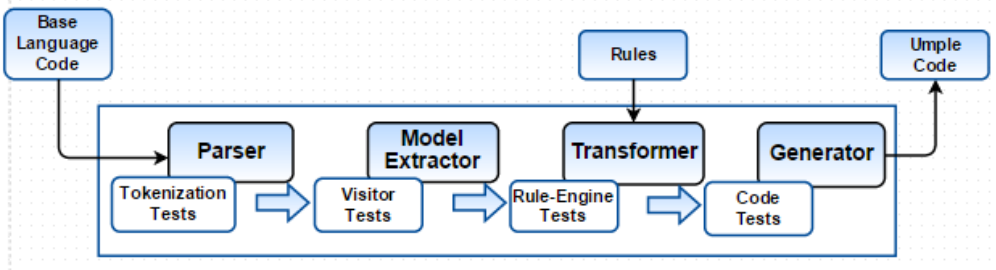

FIGURE 6.1: Umplificator Testing Infrastructure

6.1.1 Testing the Base Language Code Parsers

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

A simple parser test is shown below that verifies that the list of detailed problem reports (warnings, or compilation errors) noted by the compiler during the parsing or the type checking of the compilation unit (file) is what we expect. In this particular example, we are expecting 2 problems (compilation errors) since the input compilation unit contains two errors at two different locations in the code.

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

The pattern for parser-related test is as follows:

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

6.1.2 Testing the Model Extractor

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

For instance, if the test input file contains:

Listing 6.1: Java input file for test.

```
package cruise.umplificator.visitorTestFiles;

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

 @SuppressWarnings("unused")
 public class InputForVisitorTest {

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

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

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

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

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

6.1.3 Testing the Transformer

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

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

- In Line 78 we clean the working memory of the Rule Engine.

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

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

6.1.4 Testing the Umple Code Generator

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

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

```
24 }
```

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

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

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

6.2 Pre-Validation Phase

As is customary when introducing a new tool in software engineering, we have tested our umplificator using our own repository of examples. These collection of **umple** examples, currently 42 ranging from Banking systems to Warehouse control systems is available from review online at [18] and was used to generate the Java 'toy examples'. The process is illustrated in Figure 6.3. The goal of this pre-validation phase is to assert that the **UmpleModel** is semantically identical to the **UmpleModel'** which is the generated output of the Umplificator.

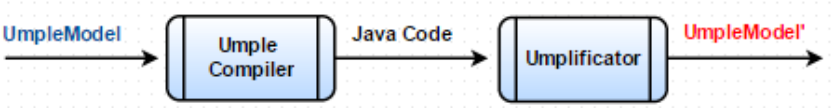

FIGURE 6.2: The Pre-Validation Phase: Comparing UmpleModel and UmpleModel'

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

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

```
namespace access_control;

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

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

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

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

  key {id}
}

class Role
{
  code;
```

```
  role_description { Dba, ProjectMgr }

  key {code}
}

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

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

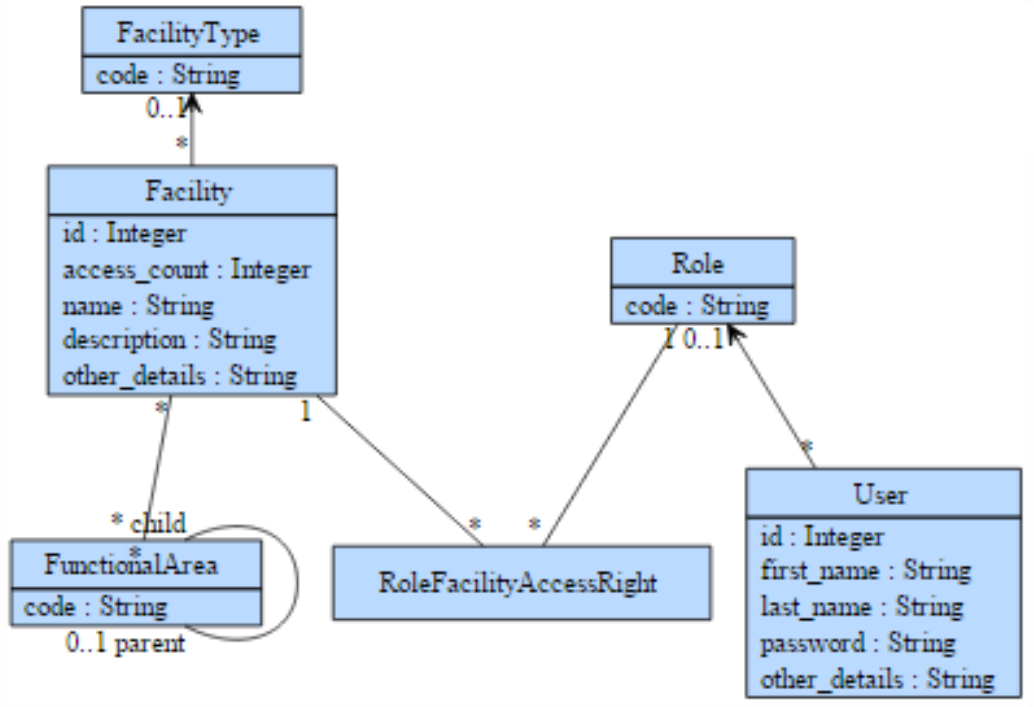

FIGURE 6.3: UML Class diagram of the Access Control system

LISTING 6.2: 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>());
```

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

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

name and additional features). For instance, in Line 69 we assert that the attributes is **lazy**, since the variable is not one of the parameters in the constructor of the Java class **Facility** (Java code is not shown here).

More on our approach to validation is presented next.

6.3 Initial Phase of Validation

Additionally, as part of our second stage of validation, we tested the Umplificator on various open-source systems written in Java. We use freely available systems to ease comparisons and replications of our evaluation. We provide some information on these systems in Table **??** including their version, number of lines of code and the number of classes. The last column of the Table indicates whether or not the system has been studied elsewhere. Note that the data (statistics on the modeling constructs detected) in those external studies is used to compare the result of our automated tool. Furthermore, we perform '**manual**' umplification on the systems *that have not been studied before*, the results of the manual umplification are then compared to the results of our '**automated**' umplification. In fact, the only project that hasn't been studied (reverse-engineered) before is Weka, a very popular suite of machine learning software written in Java.

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

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

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

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

4. We compute the **precision** and **recall** of the results previously obtained. Precision assesses the number of true constructs (attributes, associations) identified, while recall assesses the number of true constructs missed by our detection algorithms.

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

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

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

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

The following are the definitions we have employed for the precision and recall measures [26]:

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

and

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

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

6.4 Second Phase of Validation

6.5 Results

6.5.1 JHotDraw

JHotDraw7 [19] is an open source graphic editor that supports operations on many graphics file formats. It makes extensive use of software design patterns and has detailed documentation about its design. We selected JHotDraw for umplification to be able to apply our transformations on documented frameworks and to compare results with the documentation of these frameworks and the analyses performed by other tools [19]. For this research we worked with JHotDraw 7.5.1. Table X shows the results of detection of at-tributes and associations for the JHotDraw framework. It details the number of classes, the number of attributes and the different types of associations. We also performed a manual analysis to check the accuracy of our algorithms and mapping rules. After improving and refining our rules, we have obtained a precision of 100%. The refinement consisted of adding the Java idioms that our detection algorithms were not able to catch on the first attempt. For instance, not all the setters in the framework return always a void, some of them return a boolean.

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

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

6.5.2 Weka

The next system we focused on was the machine learning tool Weka [27]. As with our the first attempt at umplifying JHotDraw, our first attempt at automatically umplifying Weka result-ed in a precision of less than 100% some idioms it uses were not yet detected by our tool. For example, some classes in the classifiers package implement add() and remove() methods with different argument types. Also, the Confusion class

declares add(RuleSet) and remove(Antecedent) to add and remove a set of rules from the evaluation algorithm. In addition, we detected, after execution of the test suite, that two classes were not compiling due to an unexpected constructor signature. Initial Umplification results for Weka nonetheless have a precision of 85% when it comes to attributes and 38% for 1-to-many associations. Table 8 summarizes the results. Note that a precision of 38% doesn't mean that the Umplificator has missed 62% associations of this type. It means that some of them were not correctly transformed into Umple (e.g. incorrect navigability, role names or transformation of accessor/mutator methods). The extensibility and flexibility of our tool allows us to add and refine rules without having to recompile the system. It is our objective to successively umplify more and more systems, with the hope that eventually our rule base will cover the vast majority of cases needed to successfully umplify new systems the Umplificator is presented with. However, even with a precision in the high 80% range, our tool serves as a useful tool for umplification. Users can leave some variables un-umplified, or can manually umplify the rest.

TABLE 6.1: Toy examples used for first phase of validation

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

TABLE 6.2: Open-source systems umplified

Name	**Version**	**LOC**	**# of Classes**	**Reference**
JHotDraw [19]	7.5.1	82132	694	Yes
Weka [20]	3.7.13	278642	1370	No
Java Bug Reporting Tool[21]	1.0	2629	36	Yes
JEdit[22]	1.12	59699	234	Yes
FreeMaker[23]	2.3.15	39864	281	Yes
Java Financial Library[24]	1.6.1	1248	27	Yes
args4j[25]	2.0.30	2223	61	No

Chapter 7

Related Work

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

7.1 Literature Review Methodology

This study has been undertaken as a systematic literature review based on the guidelines proposed by Kitchenham [28]. Key parts of this systematic literature review are presented in this thesis.

7.1.1 Research Questions

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

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

7.1.2 Search process

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

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

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

7.1.3 First Phase Queries

7.1.4 Second Phase Queries

7.1.5 Inclusion and exclusion criteria

Chapter 8

Conclusions and Contributions

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

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

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

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

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

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

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

Appendix A

Appendix

Bibliography

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

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

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

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

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

[6] O Badreddin, A Forward, and T.C. Lethbridge. Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity. In *SERA 2013*, pages 129–149. Springer, 2013. doi: 10.1007/978-3-319-00948-3_9. URL http://dx.doi.org/10.1007/978-3-319-00948-3_9http://link.springer.com/chapter/10.1007/978-3-319-00948-3_9.

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

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

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

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

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

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

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

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

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

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

[17] CRuiSE. Mapping Rules in Umple code repository, . URL http://goo.gl/DFGkZB.

[18]

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

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

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

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

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

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

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

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

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

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