Umple C++ Code Generator

By:

Sultan Eid A. Almaghthawi

MSc Thesis

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

Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Ontario, K1N 6N5
Canada

Abstract

We discuss the design and analysis of a code generator for C++, implemented in the Umple model-oriented programming technology. Umple adds UML constructs and patterns to various base programming languages such as Java and PhP. Umple code generators create code for those constructs, which can include UML associations and state machines, as well as patterns such as immutable and singleton. Base language methods are passed through unchanged along with the generated code. Creating a C++ code generator for Umple posed many challenges, all of which are discussed in this thesis: We had to focus on the appropriate C++ idioms and stylistic conventions to follow. We followed a test-driven development process to ensure that the resulting code was correct. To evaluate the work, we compared our C++ generator with those in other tools such as ArgoUML and IBM Rational Software Architect. We conclude that our C++ generator is superior in many ways to these widely used tools because it is more complete and generates better quality code.

Acknowledgements

I would like to express my gratitude to my supervisor Timothy Lethbridge for the useful comments, remarks and engagement through the learning process of this Master's thesis; he has greatly helped me form my understanding of software engineering.

A very special and well-deserved thanks to CRUISE (Complexity Reduction in Software Engineering) group members for their great collaboration and support, in particular, Andrew Forward, Omar Badreddin, Hamoud Aljamaan and Miguel Garzon.

I would also like to sincerely thank my beloved family and especially my mother, who has supported me through the entire process and for helping me putting pieces together. Sincere thanks to my lovely wife for always being there.

Last but not least, thanks to my sponsor Taibah University for its fund and support through out the academic program.

Table of Contents

Figures

Tables

Chapter 1 Introduction

This thesis discusses the implementation of a C++ code generator in Umple [1, 2] to allow UML-to-C++ code generation.

Umple is a technology for model-oriented programming for UML [2]. It adds modeling abstractions from UML, such as associations and state machines, to programming languages. Code that developers write in the base programming language is compiled unchanged, but Umple generates base language code for the modeling abstractions. Umple allows developers to write software using what we call the *model-oriented programming* approach; by this, we mean that developers write code like other programmers do, but at the same time their programs are structured around modeling abstractions. It supports various languages for code generation such as Java, Php, Ruby and C++. Umple is explained in detail in Section 2.1.

One of the core purposes of Umple is to facilitate generation of better quality code from UML, since existing open source UML tools such as ArgoUML [3] tend to have weak code generation. This research is motivated by the lack of a C++ code generator in Umple when this work started, coupled with the importance of C++. Umple has several features to facilitate the development of code generators in an agile manner, allowing the research to focus on generating quality code.

Implementing the code generator described in this thesis should allow C++ developers to write their systems using Umple in a model-driven manner where they can inject abstract UML elements into C++ code.

This work is part of the research of the CRuiSE group at the University of Ottawa, and builds on previous work by students such as Andrew Forward [4] and Omar Badreddin [5] who have built the Umple parser, its metamodel and code generators for Java, PHP and Ruby.

In this thesis, we address the challenges in producing a C++ code generator in terms of implementation inside Umple and also regarding C++ as a language. We also discuss the challenges from moving from Java to C++. In addition, we discuss the code generator's implementation and compare our approach with other code generators.

A key element in this work is to compare our work with other related tools in the market and see where Umple advances in the state of the art. We have looked into several tools and chose the most widely accepted for the comparison. These tools are IBM RSA [6], ArgoUML[3] and Papyrus[7].

We have also set up a basic LTTng (Linux Tracing Toolkit next generation) tracer for C++ that allows developers to trace their C++ application at an abstract level. LTTng is a tracing tool for C++. It can be used to instrument applications to collect information for various reasons. The challenge here is mainly to create an LTTng generator that generates LTTng code to trace the corresponding model entities such as associations and state machines.This is discussed deeper in section 3.7.1.

In summary, the goal of this thesis is to develop a C++ code generator for Umple that should allow code generation from UML to C++ using Umple with respect to associations and state machines and allow of LTTng tracing for C++. [8-14]

1.1 *Research Questions*

In this section we discuss the research questions we are investigating.

1.1.1 What are the challenges in generating C++ code in the Umple model-oriented technology?

Throughout the development of the C++ code generator, we are interested in pinning down all the difficulties specific to this task. We will be looking to answer this question from several perspectives, including:

1.1.1.1 Changes needed to Umple

Does the Umple architecture and development environment meet all the requirements to implement a C++ code generator effectively? Umple has several generators already implemented. We are interested to see how a C++ code generator would need to be different from other generators in Umple in terms of implementation. In particular, does Umple require any refactoring in order to facilitate the implementation of a C++ code generator? If there are any requirements missing to implement the C++ generator, how much refactoring is required?

1.1.1.2 Quality of generated code

We want to understand the C++ conventions we should use in the generated code. We are interested in several aspects:

Coding standards: Coding standards help manage consistency among software projects, enhance the quality of the code and reduce the probability of generating bugs. They also help the developer understand code written by others, and in Umple they would allow teaching about the generated code. There is a coding standard for almost every aspect of C++. In Umple we are investigating this by looking into several points that include: File names, file format, header files, file headers, naming style, class naming and layout, etc. At a more detailed level we need to consider conventions for such things as use camelCase, vs. underscore separators etc. Umple has a coding standard in all generated code that is being imposed on all generated code for programming languages such as Java, PHP, etc. We want to investigate whether implementing C++ would affect the Umple style of coding.

Readability: One of the philosophies of Umple is that generated code is not to be edited; any extra code (algorithmic methods etc.) is supposed to be injected directly in the Umple source. Readability of generated code does, however, become an issue in two circumstances: The first is when there is a need to audit or inspect the code to validate its safety or security. The second is for educational purposes; to teach students how UML constructs ought to be implemented. There are several factors, which we will discuss in detail later; those have major impact

on the readability. Things like comments above classes, generating API documentation, injecting warnings when applicable at the generated code and the style of coding.

Other aspects: Ultimately from the Umple point of view, the philosophy is that the generated code should be of as good quality as if it was written by hand. We first write systems in C++ by hand, as they ought to be generated by Umple, and discover the issues of interest. To make C++ generation as good as possible we are looking deeper into those aspects that affect the efficiency of the code.

1.1.2 How does our generated C++ code compare to generated C++ code from other tools?

There are several other tools that generate C++ out of UML. Some tools like IBM RSA have a long history of model-driven development and C++ code generation. However, this doesn't necessarily mean that they offer a comprehensive code generation mechanism. Part of the motivation for our work is that these tools could not fully handle associations and state machines. Hence, we are interested in investigating what ways Umple allows us to explore new concepts that are not found in other open source C++ code generation tools. We will answer this by comparing Umple with some well-respected existing tools against a list of criteria that should show the areas where Umple represents an advance on the state of the art, and those where Umple is not there yet in comparison with other tools.

1.2 *Thesis Outline*

The remainder of this thesis is organized as follows:

Chapter 2: Background and Related Work: In this chapter, we discuss Umple and C++, and then investigate other related tools, such as IBM Rational Software Architect, ArgoUML and Papyrus. We also show the metrics collected from an airline system modeled in each tool.

Chapter 3: C++ Code Generator for Umple: Our C++ code generator for Umple has been developed in a model-driven manner. In this chapter, we demonstrate the agile test-driven approach we followed to develop the tool. Also we cover several important aspect of the Umple C++ code generator; we discuss the style of the generated code, the generated API as well as the completeness of UML in terms of

syntax and semantics. We also discuss the development of the LTTng tracer and what does the work we did bring new to tracing using Umple. Note that often people conflate the terms tracing and traceability, which are completely different concepts. Traceability refers to tracing requirements to code while tracing is a process similar to logging; tracing is discussed in more detail at the end of this thesis.

Chapter 4: Comparison with Other Related Tools: In this chapter, compare our approch to C++ code generation to that of related tools. First, we present our criteria of comparison then assess the tools according to these criteria.

Chapter 5: Conclusion and Summary: Finally, we summarize the contribution that has been done to Umple in this context and we try to see if the research questions have been answered.

Chapter 2 Background and Related Work

In this chapter we discuss Umple, C++ model-oriented software development in general and various tools to which we will compare Umple.

2.1 Umple

Umple is a modeling language and tool that is fully developed in a model-driven manner. It adds key features of UML such as associations and state machines in a textual form directly into different object-oriented programming languages such as Java, PHP, Ruby and now C++. Umple tools can also import and/or export other representations of UML, like TextUML, Papyrus XMI and various diagrams.

In this thesis we generate code for essentially all Umple features.

Umple is among many tools that support the model-driven development approach, where developers try to work at a level where very complex systems are represented and maintained through models (whether graphical or textual). This approach generally uses code generation and always uses abstraction of details. Umple adds abstraction on top of programming languages and provides a demonstrably more usable [15] and less complicated modeling language than other similar tools.

Abstraction in software does not stand for vagueness; it stands for reduction of information to the essence by reducing the amount of detail the developer needs to describe or understand. Umple, in this sense, plays a role in the evolution of the development of software and abstraction of programming languages. It adds a new layer above high-level programming languages to ease the development process. If we look at Umple from an abstraction point of view, see Figure 1, we can see that it has several layers of metamodeling. These are:

Umple Metamodel: The core metamodel of the Umple language that describes the construction of Umple models. The Umple metamodel itself was defined in Umple.

Umple Model: An instance of the metamodel, describing Umple elements, attributes, classes, association etc., which are part of an Umple program.

Instance of Umple elements: These are variables, objects, links, states, etc. in a running program.

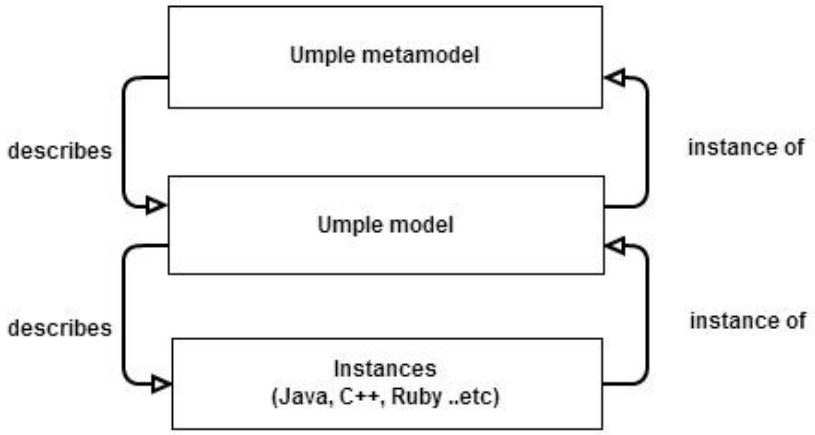


Figure 1: Umple meta-modeling architecture

Working with Umple can improve development in several ways, such as:

- It reduces the number of line of codes a user has to write. Instead of writing boilerplate code, the user can write an Umple model and generate much of the required code instead
- Dealing with fewer lines of code will enhance readability of the system and allow the developer to focus on the logical issues rather than tackling low-level technical problems.
- Writing less code will eventually contribute to avoiding introducing bugs in the system.
- Umple is easy to use. The tool is intuitive in terms of usability of the language and the tool. Also, developers can easily adapt the tool since it relies on intensive testing. This allows them to extend the tool in an agile way.
- Converting you're an existing system to Umple (a process the Umple team calls umplification) can be performed. Tools for this are in development.

The philosophy is that Umple becomes the core element in the system, blending code and UML models. For instance, consider the following three situations that Umple allows you to write: you may write a model-only file, a target language

code only file (say Java, Php, C++, etc) or you can mix a model with target language code in one file. Although Umple allows systems to be converted to Umple (or Umplified), it deprecates round-tripping transformation. As we can see in Figure 2, model-to-code transformation is allowed with no round tripping. Also, code-to-model transformation is under development; this is called 'Umplification'.

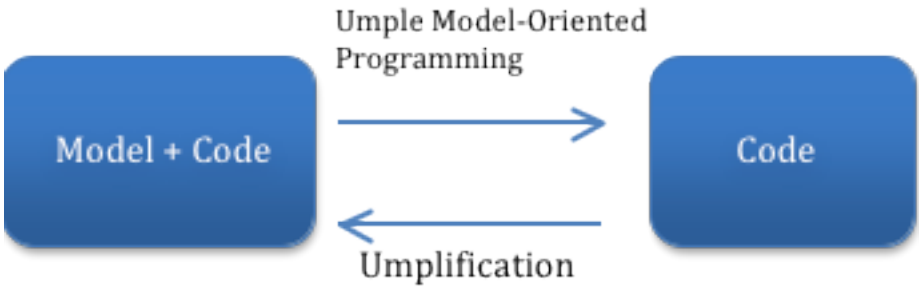


Figure 2: Umple model-oriented programming

2.1.1 Umple Architecture

When the C++ generator was about to be designed, Umple already had several generators implemented. The first generator was the Java code generator and it is considered as the template architectural example for any generator to be added. This contributes in the future evolution of the architecture. It becomes far easier for other developer to understand the code when all generators follow the same style (Umple style) in the architecture. This is reflected in naming of files and naming of methods within the compiler. Also, it can be seen in how the packages communicate among each other. The way Umple works is no different than how most compilers work, it relies on several components in order to parse the model, populate the metamodel and finally generate code for targeted platform.

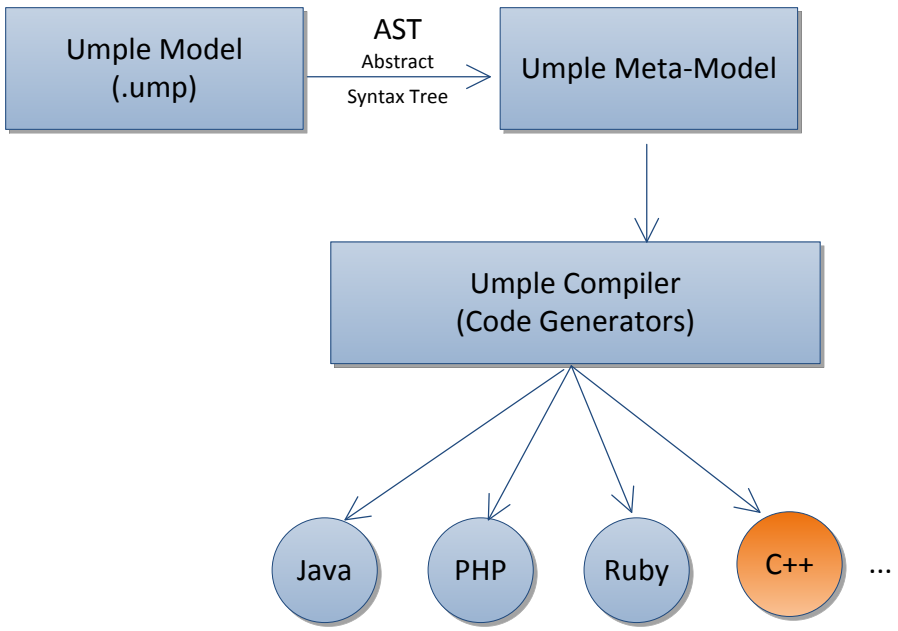


Figure 3: Umple Architecture

2.1.1.1 Umple Grammar changes in in order to allow for C++ generation:

Any language requires a grammar in order to define its syntax. Only one small change was required to the Umple grammar to accommodate C++. When one is writing an Umple model, the desired programming language generators must be specified, otherwise the default generator would be Java. For instance, consider the following example:

```
generate Cpp;
class student
{
  name;
  id;
}
```

This will generate C++ code. However, if the first line were omitted, Java code would be generated by default. In order to add C++ to the set of generators, "Cpp" had to be added to the generate arguments to allow Umple to consider C++ as an

optional generator. This addition, however, does not prevent Umple from generating code for other languages from the same file even though it has the 'generate Cpp' statement. When compiling the model using the command line, one can indicate the desired generator that will be used as an argument. For example, you can generate Php from the previous file using:

```
java –jar umple.jar –g  Php umpleModel.ump
```

The line in the Umple grammar for the generate statement is:

```
generate- : generate
[=generate:Java|Php|Ruby|Cpp|Json|Yuml|Violet|Umlet|Simulate|TextU
ml|Papyrus|Ecore|Xmi|Sql] ;
```

2.1.2 Umple Features:

In addition to the abstraction of UML elements such as classes, associations, attributes and state machines, Umple supports additional features for greater flexibility. These include support for declaring certain design patterns, constraints, and aspect-oriented code injection. These give Umple more flexibility. Some of these features are handled differently for each targeted language for code generation. For instance, some of the design patterns like singleton are implemented differently in C++ than other languages. However, the way Umple maintains aspect-orientation is in a phase prior to code generation.

2.1.2.1 Aspect Orientation

Umple uses aspect-oriented programming in two techniques:

- Umple allows injection of code wrapped in curly brackets before/after a certain pattern is matched. This can be applied to operations done on attributes, associations, methods and state machines.

- Code injection based on pattern matching: Using the before/after statements, Umple allows injection of code in certain places of the code wherever a pattern is matched.

Example:

Assume you want to log the time an attribute was modified. This can be done in Umple using aspect-orientation as follows:

```
generate Cpp;
class X
{
 a;
 whenWasASet;

 after setA {
   setWhenWasASet(getA() + getTimeStamp());
 }
```

The above example will execute whatever between these brackets whenever the value of A was set (modified):

```
after setA {
    …..
 }
```

In the above example, we simply call upon the method 'SetWhenWasSet' and pass some parameters to it. This method will be called whenever A is modified; technically, a method call of setWhenWasSet() is injected as is in the generate code, specifically inside the setA method. Umple also tells you the line number in the original model where the AOP code was written. We could also manipulate aspect-orientation to perform different tasks; logging is an example.

2.1.2.2 Design Patterns:

Umple supports a number of design patterns that can be applied on the model's elements to give them special features. We can use these design patterns to achieve more control over the code. Umple supports the following design patterns:

- **Singleton pattern**: This will restrict a particular class to be instantiated only once at run time.
- **Immutable Pattern**: This will not allow any further modification of the object after it had been constructed. When you declare a class to be immutable in Umple, all attributes of that class will not be modifiable after the construction of object. However, Umple also allows this pattern to be applied on certain elements of a class; for instance, we can have a regular class declared with some immutable attributes.
- **Delegation pattern**: This is accomplished by the use of derived attributes.
- Umple also has support for keys for equality and hashing.

For more details and examples on design patterns for Umple C++ code generator, refer to Section 3.1

2.1.2.3 Tracing

Umple has an internal DSL (domain-specific language) that is part of the Umple syntax and aims to specify tracing at the modeling level; this language is called MOTL (Model-Oriented Tracing Language) [16]. There have been many different techniques to trace code either dynamically or statically and there have been several tools developed for this. However, MOTL can work with different tracers. Based on the way the data is being collected, Umple provides support for tracers ranging from simple primitive tracers that output to a file to more advanced tracing tools. For C++, Umple mainly targets LTTng, which is an advanced tracing tool for kernel tracing and user space tracing for C++ on Linux platforms. The support for these advanced tools is still under development. Umple allows the user to declare the type of tracer to be used at the beginning of the Umple model, currently if one want to change the tracer then the source file should be modified indicating the type of tracer. In tracing, one often wants to collect data about a certain object of the model in which this data may be manipulated for different purposes. Umple allows tracing of the following components:

- Attributes: attributes can be traced through different scenarios, for instance:
  - Whenever a setter/getter of that particular attribute is called.
  - Based on a conditional evaluation a tracer will be triggered.
  - An attribute can be traced for a number of occurrences performed on the attribute (for instance, after an attribute was set 5 times).
  - Tracers can be triggered after/until an attribute value is changed to a particular value.
- State machines: Tracing entry, exit or both of a particular state, as well as invocation of particular events, or occurrence of particular transitions.

The general tracing capability has been developed by other members of the Umple team. However, in this thesis, the contribution to tracing was done by allowing C++ code generation of tracepoints for LTTng for attributes and state machines. This will be discussed in depth at later chapter in this thesis.

2.1.2.4 Example

Umple has a library of examples that can be found at the UmpleOnline website [1]. The following is an example implemented in Umple online as of Jan, 2013; it shows that a school can have several persons that are students.

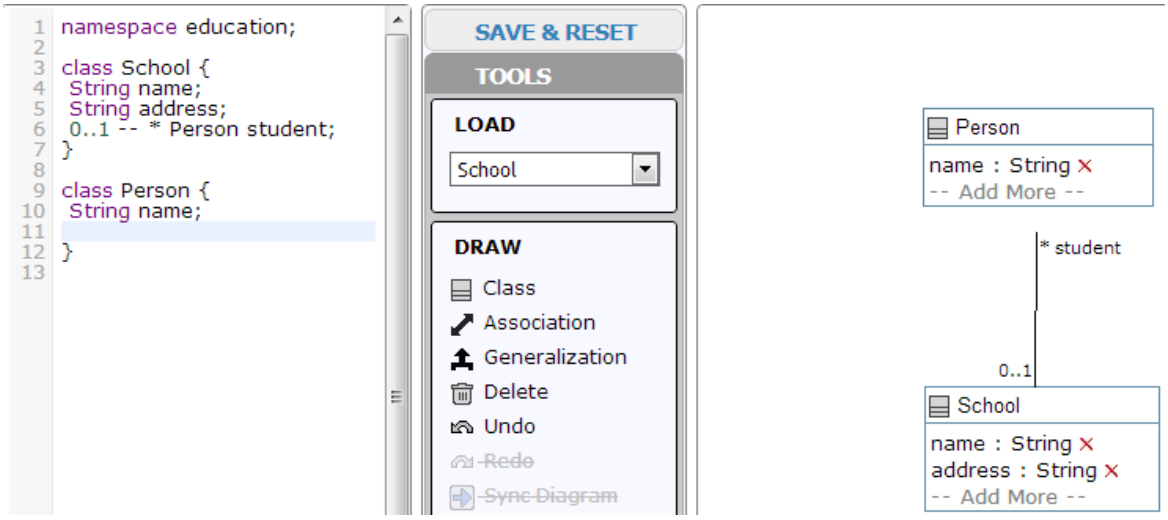


Figure 4: Example of Umple

2.1.3 Umple Tools

Umple as a development tool is available in different forms:

2.1.3.1 UmpleOnline

UmpleOnline, shown in the last section, is a web-based version of the Umple system mostly used for demonstration, teaching, and simple testing, see Figure 4. It has a bookmarking feature, which allows the users to save their models on the server and reloads those using bookmarks.

Moreover, if the user installs Umple locally, the user can manipulate Umple files one one's computer through a web browser. It can hence become a heavy-weight model development tool. UmpleOnline makes a good tool for educational purposes due to the fact that it doesn't require any installation, it has enhanced usability compared to other platforms and a list of various examples of complicated models for different systems makes UmpleOnline ideal for teaching and quick modeling. It can be also effective when used to initialize small projects by generating the code from a domain UML model to the targeted programming language. See Figure 4.

2.1.3.2 Umple Command-Line Compiler

The Umple command-line compiler will compile Umple files and generate the code. It only requires Java 7 to be installed. The tool is available as a JAR file. For instance assume you have an Umple model called exampleModel.ump, you can compile this file using the following command:

```
Java –jar umple.jar exampleModel.ump
```

This will compile the file and return a notification message, if successful:

```
.\exampleModel.ump
Success! Processed exampleModel.ump.
```

If the compiler fails to compile the model and error message is produced. For example, assuming we are missing one curly bracket at the end of a class declaration the result would be:

```
.\ exampleModel.ump
Error on line 25 of file " exampleModel.ump ":
Parsing error: Structure of 'class' invalid
```

Processed .\ *exampleModel.ump*.

2.1.3.3 Umple as an Eclipse-Plugin

An Eclipse plugin is available that allows syntax coloring and compiling of Umple files in order to generate the required code.

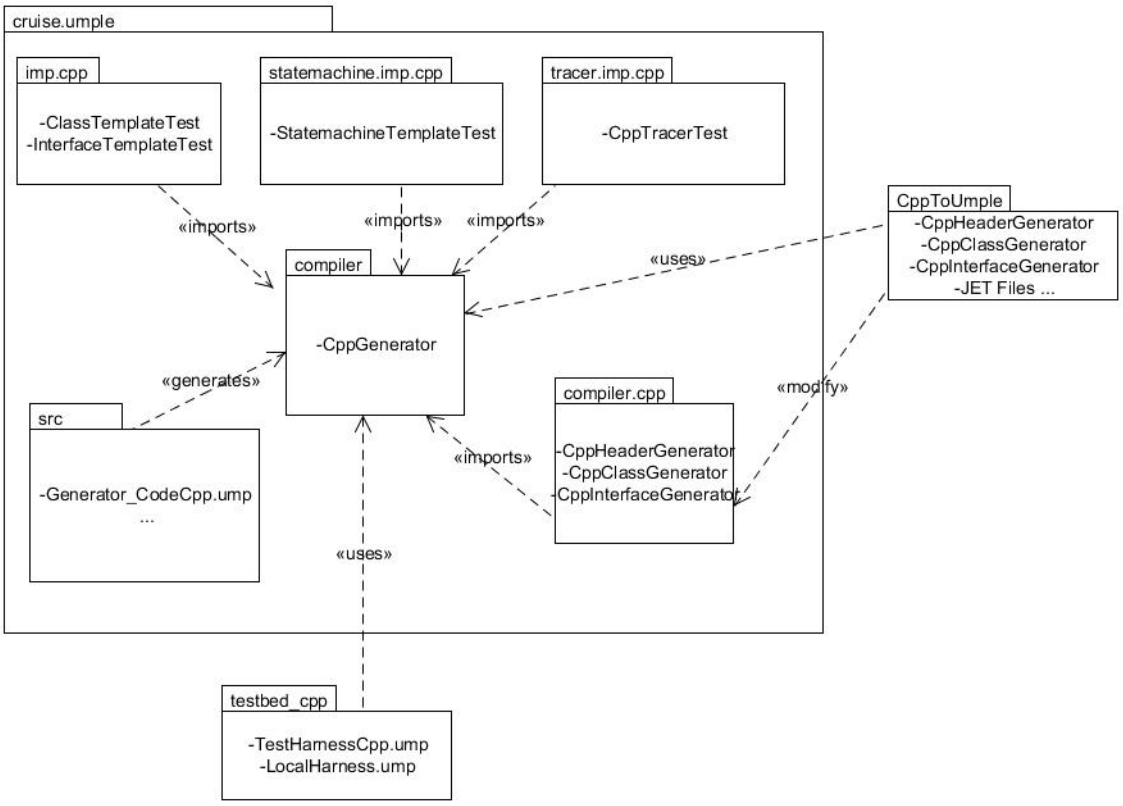


Figure 5: Umple Package view for C++

2.1.4 Generic Files compared to non-generic files within Umple:

We have discussed before the fact that the Umple compiler itself is developed in a model-driven manner. This means that the compiler is actually written in Umple files. For example the file "CppGenerator.java" is actually generated from an Umple model "Generator_CodeCpp.ump". Every component of the compiler is written in Umple, mixing between Umple elements and Java for methods bodies. Hence, any changes to the C++ generator are actually written in the corresponding Umple files. Note: it is considered a bad practice to make changes to Umple-

generated code since they will be overwritten when the code is re-generated the next time Umple is built. Therefore, all changes should be made directly to the .ump files.

The JET (Java Emitter Template) [17] framework is used to implement most of the code generators in Umple. It has syntax similar to JSP and is a Generic template engine that can be used to generate any textual presentation (Java, TextUML, JSP, XML, etc.). JET often generates an implementation class that can be called to translate the model based on the argument passed to the implementation class. The code generator for of Umple C++ as described in this thesis is implemented using JET.

To avoid misunderstanding, the build process for the Umple compiler includes the string ‘gen’ in all the folders that have generated code; this means that such folders will be overwritten.

To keep consistency between the templates of different language-generator projects, Umple uses an 'UmpleToTemplate' project to enforce the template structures; this applies to projects such as 'UmpleToCpp' , 'UmpleToPhp', etc. This means that some JET files are being generated as well. So for instance, some JET files such as ‘Attribute_SetAll.jet’ are generated from UmpleToTemplate, which enforce all Umple JET projects to follow a specific structure. It is very important to understand the generic part of any project in order to differentiate between the generic files and the generated ones in order to know where to make the right changes. The following table lists the generic files of C++ within Umple and its generated elements:

Generic File	Generated File
CodeGenerator_Cpp.ump	CppGenerator.java
UmpleToTemplate: Attribute_SetAll.jet	UmpleToCpp:Attribute_SetAll.jet
UmpleToTemplate: Attribute_GetAll.jet	UmpleToTemplate:Attribute_GetAll.jet
CppClassGenerator.jumpjet + class JET files	CppClassGenerator.java
CppHeaderGenerator.jumpjet + header JET files	CppHeaderGenerator.java
CppInterfaceGenerator.jumpjet + interface JET files	CppInterfaceGenerator.java

Table 1: Cpp related generic files in Umple

2.1.5 The Umple Testing Framework

Umple is developed in an agile manner, applying several agile methods within its development process. The focus is on model-driven development, since the earliest versions of the Umple compiler were written in Java and then Umple was written using Umple itself in a model-driven manner.

In addition, the Umple development process relies on intensive testing and any features added to the tool are driven by test cases; even the User Manual generation process is tested. This allows developers to contribute to Umple in a test-driven manner, which brings a lot of benefits in general and specifically to Umple developers since it's an open-source tool with many developers contributing and making changes to the tool. Therefore, using the test-driven approach allows Umple developers to adapt the existing architectural design and approach development by writing small test cases to specify new changes to be added to Umple.

To clarify this more, we will take a deeper look into Umple testing framework and show how we approached the implementation of C++ within Umple and will discuss the required refactoring. Testing in Umple is done at several levels starting from parser testing (ensuring the correct abstract syntax tree – AST –is built), metamodel testing (verifying metamodel construction from the AST), template testing (checking generated code matches what is expected), language-oriented semantics testing (testing that generated code behaves correctly) and some other tool-oriented testing.

We are mainly interested in levels of testing that directly correspond to C++ artifacts. We say 'artifacts' because the implementation of C++ is not done in one independent package; there is in fact a tailored generator within the compiler as well as other packages in the architecture relating to C++ code.

Figure 6 shows the order of testing in Umple. First is parsing testing, which ensures that the Umple file (umplefile.ump) is correctly parsed according to the grammar. Second, a set of tests ensure that the instance of the metamodel (the model in a test application) is populated correctly.

Third is template testing or code generation testing. In this phase of testing, the generated code is being tested syntactically according to the expected language syntax. This means that for each language, like Ruby or PHP for example, there is a specific testing suite to verify the correctness of the language syntax.

Lastly there is testing the semantics of the generated code. In this phase of testing, Umple makes sure that the targeted language is semantically working and returns the expected values. For a language to successfully pass the syntactic test phase this doesn't mean it is functioning properly; logical errors are only detected with semantics testing. Therefore, for each targeted programming language, an independent testing project is created, usually referred to in Umple as a testbed. For instance in the case of testing C++ we would create a separate project for this purpose called "testbed_cpp". We will be discussing C++ testing in detail in a later chapter.

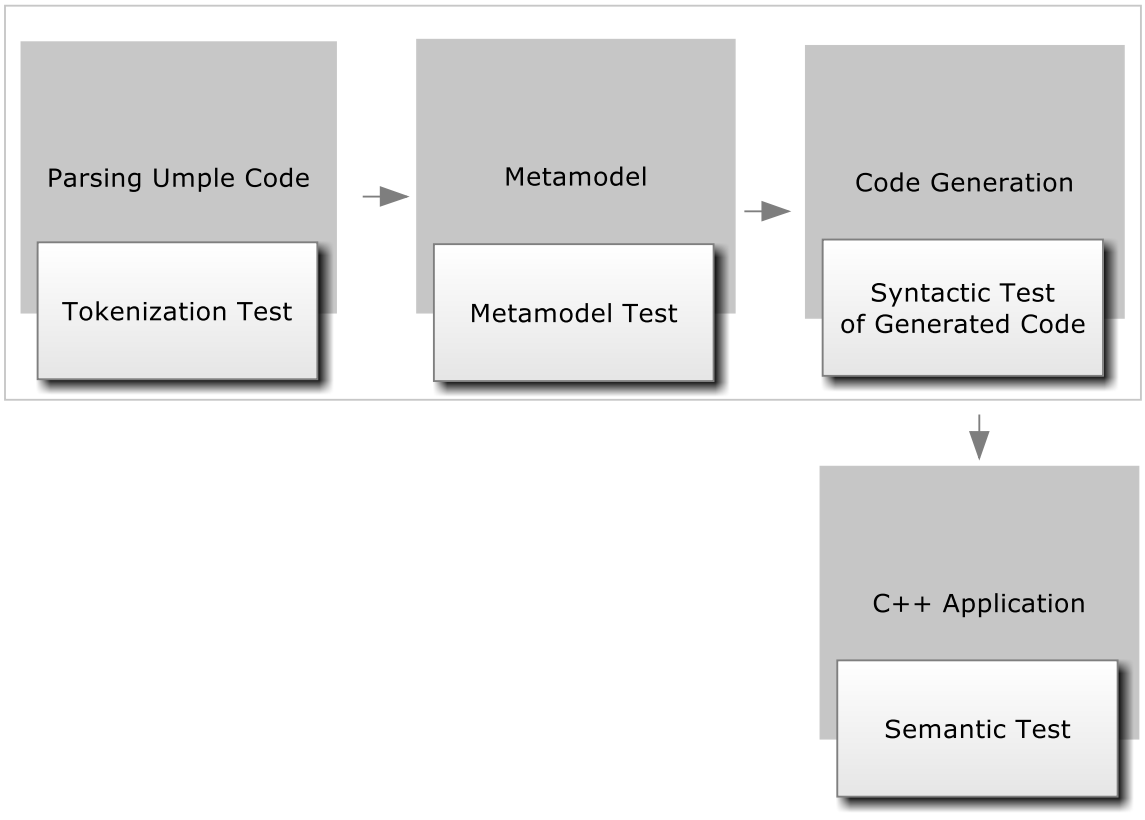


Figure 6: Umple testing framework

Importantly, when it comes to C++, we are interested in these two testing types, since the other testing types are independent of the generated languate:

- Template testing (code generation testing)
- Semantic testing (testbed):

2.2 C++

Umple is not the first tool to target C++ for code generation from UML. There have been several tools with C++ code generators for UML; however, there is no openly available tool that has fully functioning and sufficient solutions for associations and state machines and which generates C++ code. To reach the above conclusion, we looked into closed source tools such as IBM's Rational Software Architect (IBM RSA) [18] as well as open source tools such as ArgoUML[19], and Papyrus [7]. More details on these tools will follow.

Umple targets C++ for several different reasons: First, C++ is a very common language and widely used by many developers. Although many might argue that C++ is becoming less relevant with the move toward more evolved languages, there are still many developers in industry who prefer using C++ for its high performance. According to the open source directory Ohloh.net, C++ is the second-highest language after C in terms of the number of commits, the third top language when it comes to lines of code and the tenth top language in terms of number of projects. Note that these statistics cover only the projects listed in Ohloh.net, but most open source projects are listed there. These statistics (see Figure 7) show that C++ is still considered a very active language and in a good condition to be used for projects that would like to consider C++ as a main development language. Umple, therefore, is targeting C++ to offer a model-driven approach to generate C++ code, and to offer generated code that is of as good quality as that written by hand.

Also, considering the fact that C++ has a higher level of complexity than other languages, such as Java, delivering a model-driven approach with Umple can help the C++ developer avoid many technical issues and focus instead on the high level logic of the system which should eventually provide a good development environment for C++ as a development language.

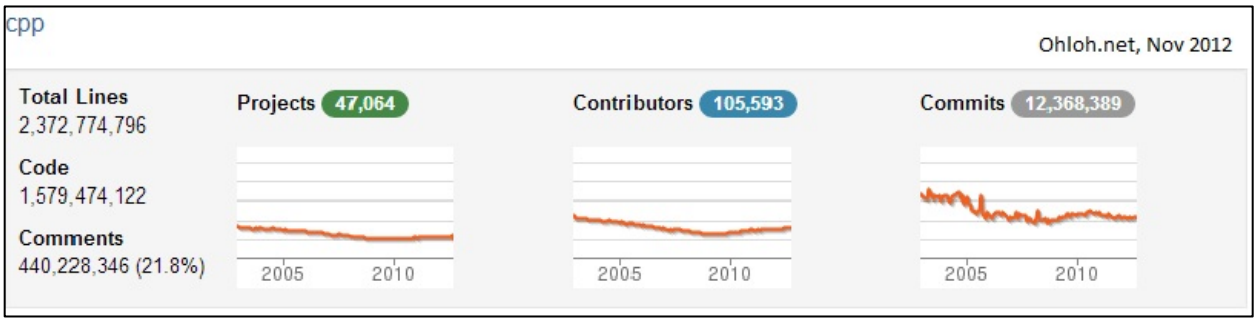


Figure 7: Status of C++ according to Ohloh.net

In addition, TIOBE.com [20] gives C++ an 'A' status according to popularity among other programming languages. We can see from Figure 8 that although the language has decreased from the 3rd position as of Dec 2011 to the 4th as of Dec 2012, it is still active and one of the top languages. According to TIOBE, the language has 9.2% of job advertisements as of Dec 2012 while the first place goes to C at 18.7%.

Another reason for targeting C++ is that the Umple team, consisting of all the researchers working for Dr. Lethbridge at the University of Ottawa, has been a part of a project conducting research on tracing of multi-core systems in which Umple was used as a tool to specify tracing of systems at the modelling level. Since the targeted tracing tool, called LTTng [21], works primarily with C++, this has also contributed to the motivation behind the development of the C++ code generator in order to allow tracing of C++ systems with LTTng in a model-driven manner. More details on this will follow in later chapters.

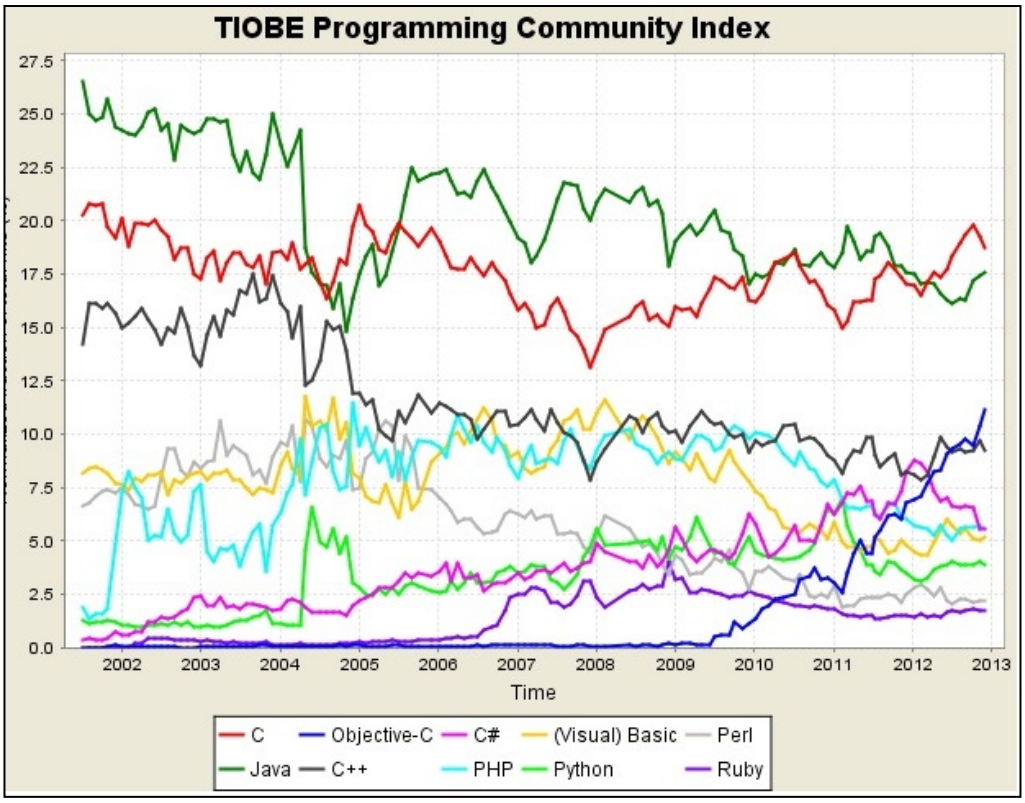


Figure 8: Long term view of C++ popularity according to TIOBE.com

C++ is different from Java in many cases. For a programmer coming from the Java world, there are several issues one has to pay attention to. The following is a list of some of the main differences between C++ and Java in the context of object-oriented programming:

- First, the language requires two files to represent one class. For instance for a class A, we need to generate A.cpp which contains the implementation (body) of methods and A.h which contains the declarations. In Java on the other hand, you only need to generate one file A.java that contains declaration and body.
- Because we write definitions separately in C++, we must manage header file inclusion into the implementation file. In Java we don't need to do that. This in fact introduces a big issue in C++ which is known as "recursive inclusion" which occurs when you try to implement bidirectional associations and erroneously run into an infinitive inclusion. This issue is solved in two different way, either one can use preprocessor guards in

header files or we can use forward declaration where we declare the included class name in the other header file before using it. A Java user won't run in such issues.

- In C++, objects have to be passed either by reference or by pointer; passing by value is not normally an option since itresults in multiple copies of the objects, which would be independently modified and get out of synch. There is no definitive argument as to whether pointers or references are better. References have a simpler syntax, but pointers allow the use of the null pointer, which can simplify many algorithms.
- There is no scope resolution in Java yet in C++ we must use the scope resolution '::' to indicate to which class a particular method belongs to.
- We need to provide a copy constructor when we copy objects of the same class in C++.
- Copying objects in C++ requires deep copying where we need to overload the equals operator. In Java, however, we cannot overload operators like we do in C++, and use equality and hashing methods instead.
- Each instance of a class in Java is an Object, since everything is derived from the root hierarchy 'java.lang.Object'. This concept differs from C++.
- Java uses automatic garbage collectors to clean up memory that is no longer referenced, while C++ by default requires the use of destructors to destroy object when no longer needed. Therefore, it requires more work to ensure memory is cleaned in C++. This doesn't mean Java doesn't suffer from memory leaks though, since unexpected references can prevent garbage collection. Logically, therefore, memory leaks have to be cleaned manually even in Java.
- Some concepts in C++ are not present in Java For instance, the concept of multiple inheritance is not present in Java.
- Interfaces are implemented differently in C++ (i.e. as an abstract class with only pure virtual ethods) but this is conceptually is very similar to Java's interfaces.

2.3 Model-Driven Software Development

Model-driven software development (MDSD) has a long history. Its central idea is that developers create high-level models in a language like UML or SDL

(Specification and Description Language)[22] and then generate code for much of the system from these. Tool vendors have created a variety of modeling tools and it has been increasingly adopted the last few years in domains such as aerospace, telecommunications and automotive software. It is still not used for the majority of software, however.

Model driven development involves creating abstract models of particular domains and software for those domains in order to exploit the abstraction of details and concentrate on the high-level issues of the problem rather than struggling with the details and logic at the low level part of the system. MDD, as a methodology, continues to provide solutions to develop software faster and produce far-more-maintainable products [23].

Many tools have been developed over the last few years based on a pure model-driven development manner, where models become the main focus and representational side of the system. Some existing tools focus on visual representations. IBM has been a pioneer in the development of model-oriented software. Back in the 1990's until early 2000 IBM Rational Rose was one of the first tools that aimed to focus on the visual modeling and visual development using UML. The software developed rapidly and the company eventually released alternatives such as Rational Software Modeler, which are based on the Eclipse IDE. Today the company has some of the most popular software in this field that offer solution for developers targeting UML as their tool for MDD. In addition to Rational Software Modeler it acquired the Rhapsody and Tau tools.

As IBM continued to provide solutions for MDD it has now one of the most powerful tools in the field, IBM Rational Software Architect, also based on Eclipse and providing a model-driven development approach based on UML models with a good support for different architectural domains such as service-oriented architecture and others. The tool has a good mechanism for code generation for different languages, such as: Java, C++, WSDL, etc. More people became interested in the MDD as IBM kept developing its products. However, IBM doesn't offer its products for free; in fact, the products are very expensive which makes it harder for small companies to develop their software using IBM products. This was a great motivation for many software developers to create other open-

source projects that also aims to model software based on UML, other tools such as ArgoUML, Papyrus, Umple, and Umbrello.

MDD increases the maintainability and quality of software systems as it creates a productive environment for software. This methodology, however, is firmly linked most of the time to these main ingredients in any domain-specific recipe [24]:

- **Compilers**: This plays big role in the process of transformation between models and other components in the system. In the case of the Umple C++ code generator, Umple is the compiler we are considering to handle the transformation between UML/Umple model and the final system.
- **Generators**: These are usually part of the compiler and they are responsible for targeting different execution platforms. Most of the systems generate different code based on the selected language and the targeted domain. For instance, Umple generates Java, PHP and other formats. Therefore, for each language to be generated, a customized generator must be tailored for that language within the compiler.
- **DSLs (Domain-Specific Languages)**: Such languages represent the abstraction of the a domain model and can be used to generate specialized code.
- **Transformation languages/Model-to-Text (M2T)**: Such languages handle the code generation from the abstract model to the targeted code. Most MSDS systems use template languages that are tailored to describe the transformation between model and code.

The following sections discuss other tools that have the capability of generating C++ code from UML models and will later on be compared to Umple. For each tool, we will give a brief introduction about the tool, the workflow of the tool and an example of the generated code. In later chapters we will investigate how each tool treats the main components we are interested in of any particular UML model. We are interested to investigate the following:

- Attributes:
  - How does the tool's language declare attributes of different types?
  - Does the tool generate what is expected for attributes? Most developers expect to see good encapsulation of attributes in classes; this can be done by

providing a functional interface to private members of the class. It is unlikely to define public attribute at the model level, however, we are also interested to see whether such functionality is supported by the tool. Also, we want to see if the tool supports design patterns for attributes in UML and to investigate more in the implementation of each design pattern and how this can be declared.

- Associations:
  - What type of associations does the tool support?
  - How does the tool represent associations of different types?
  - Does the tool provide additional features to support associations such as managing referential integrity (i.e. in two-way associations if one object points to another, the other will point back to the first) and multiplicity constraints?
- State machines:
  - Does the tool support state machines? Does it generate good code for state machines?
  - What type of state machines does the tool support?

2.4 Test-Driven Development (TDD)

Test-driven development is an agile method to develop software that focuses on intensive testing and was introduced around 2003 [25]. The idea is that testing components are not only used for testing the functionality of the system but to contribute to the design of the system and its specification as well as its validation. TDD has been widely adopted recently. Conceptually, system development is driven through the creation of test cases that define all aspects and details. However, systems that are agilely developed with TDD require a well-rounded testing infrastructure and good separation of concerns in order to efficiently apply the TDD method. Although TDD has some disadvantages, like the fact that it is a new technique that traditional developers may resist, it offers many advantages to the development process, things like:

- It facilitates the development process and reduces development time due to less debugging and regression.
- The developer ends up with a tested system, which increases its quality.

- It allows you to take small steps toward your goals, which is can help with productivity.
- It verifies whether your design is consistent and clean.
- It allows new developers to adapt the style of former developers and keep consistency of code, since new developers will mostly have to write test cases according to existing testing mechanism written by former developers.

With TDD, developers write their test cases before they write their code. This process allows the developer to end up with a fully tested and functional system. Figure 9 shows the workflow of TDD. Basically, test cases are written first with expected values even though the corresponding code has not been written yet. When these test cases are run, they will eventually fail while the developer is already expecting their failure. Then the developer should write the code to return the expected values for these test cases and do any required refactoring until the test cases pass. It is very important that the tests are specifically written to define certain aspect of the system. This process is gradually repeated over the source code until a satisfactory compilation level is reached. See figure 8.

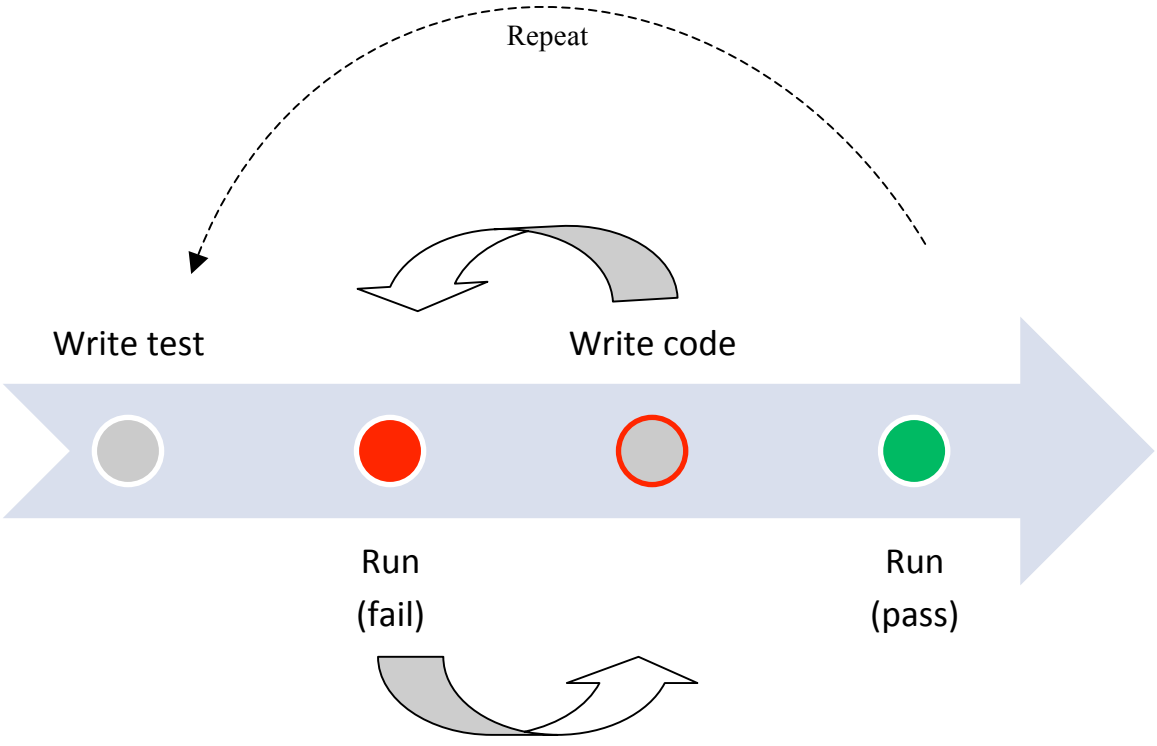


Figure 9: Workflow of TDD

2.5 IBM RSA 8.5 (Rational Software Architect)

IBM RSA is a modeling and development environment that focuses on the development of systems architecture based on UML (Unified Modeling Language). The tool has a long history of support for UML modeling and the development of UML-centric applications. It has been evolving since the 1990s. IBM RSA is considered as a pioneer tool when we are talking about MDD considering its continual contribution to provide solutions for developers using this approach. The tool has the capability to provide many features for developers that use MDSD to write applications and web services. It supports modeling of UML 2.x diagrams. It also supports model-to-code transformation with a list of several OO programming languages such as C++, Java and other formats such as WSDL (Web Service Description Language) [26]. The tool also allows reverse engineering of code, which is known also as code-to-model transformation. However, we are only interested in the model-to-code transformation, specifically, the UML-to-C++ code generator extension.

The tool is built on top of the open-source IDE Eclipse [27] . You have the option to either install IBM RSA separately (as a fresh version of the distribution) or you can extend your existing Eclipse if needed.

Since we are comparing several tools including IBM RSA, this tool is considered as the most targeted tool by related competitors, most of the tools that offer MDD solutions look into IBM RSA and conduct comparative studies since the tool is widely used and well-rounded in terms of stability and integrity.

Let's take a look at the workflow of writing models and code generation of C++ in IBM RSA. In order to generate C++ code, the following steps have to be done:

1. Create a model project that contains all the UML models and diagrams.
2. Create a model, a class diagram of a particular system, say an airline system.
3. Create a container project that will contain all the generated files.
4. Create a transformation file that has all the information about the preferences of the required transformation. For instance, in this file you can map the source model with the targeted container. Also you can set up code-specific

preferences, things like whether you want to generate setter/getter, copy constructor and some other options.

5. Run the transformation file in order to get the generated code.

Figure 10, illustrates the process of generating C++ code using IBM RSA, it basically shows the top view of the projects needed to be created and how the transformation file communicates with the required components in the application.

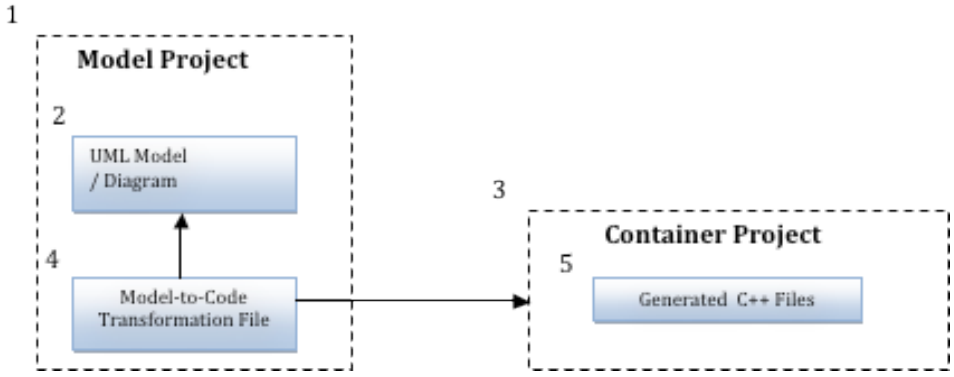


Figure 10: Workflow of IBM RSA 8.5 to generate C++ code

The transformation of UML to C++ in IBM RSA 8.5 will generate two files for each class. One implementation file (class.cpp) and a header file (class.h) that contains all the definitions.

To investigate the transformation further, we will be looking at some examples and examining some generated code. Assume that we have an airline system represented in UML class diagram, see Figure 11, and we want to implement it in IBM RSA 8.5 in order to generate C++ code for that particular model. The model has several classes with association and generalization.

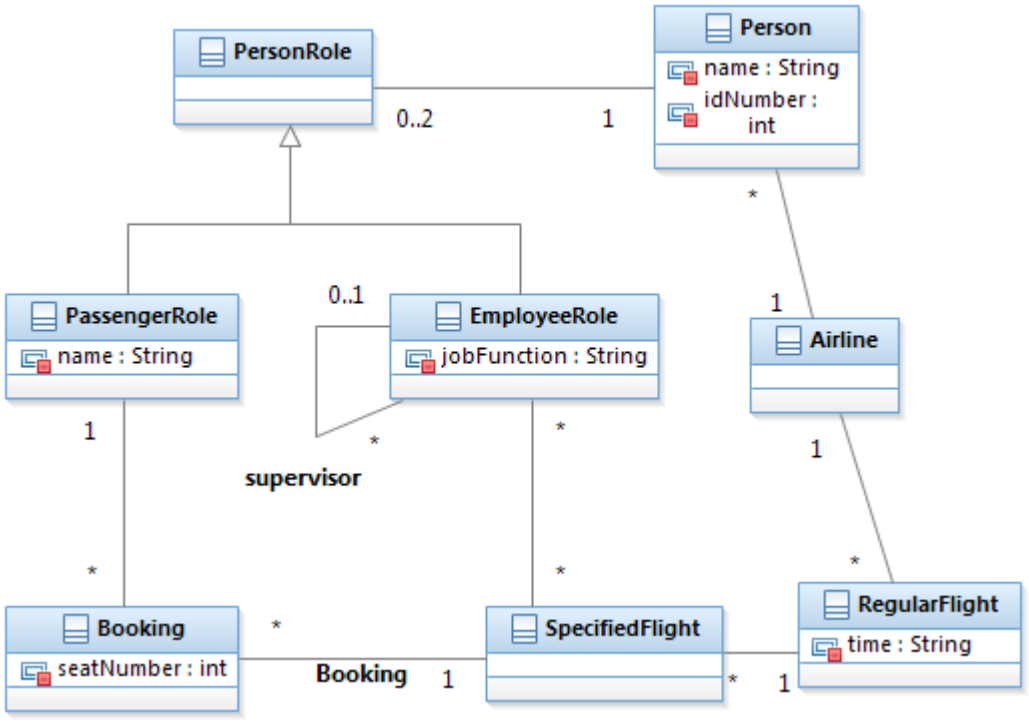


Figure 11: Airline system modelled in IBM RSA 8.5

We will discuss in depth the comparison of this tool against Umple in another chapter. As an example, if we consider the class Person from the model, we get two files generated for that particular class:

- Person.cpp: contains the implementation of the methods
- Person.h: contains the definition and declarations of methods and attributes.

2.6 ArgoUML

ArgoUML is a modeling tool and environment for analysis and design of object oriented software system. It was first released in 1998 (ArgoUML, 2012). It is similar to other UML centric modeling tools. The tool allows modeling of several UML diagrams graphically. It supports class diagrams, use case and others. The tool targets UML as a sufficient OO language to model systems and was implemented fully in Java. There are several points that make the tool comparable to Umple:

- It is free and open-source.
- It supports several open standards: UML XMI, OCL and others.
- It supports associations and state machines (although as we will asee it does not generate proper code for them).
- It is portable across platforms, and is available as Java web start.

Conceptually, the tool is developing a reverse-engineering mechanism between C++ and UML. Also the tool supports round-tripping if a well-grounded mapping between C++ and UML has been defined. This differs from the philosophy of Umple; yes Umple advocates reverse-engineering to Umple, but not round-tripping. The following figure is our classic airline system modeled in ArgoUML.

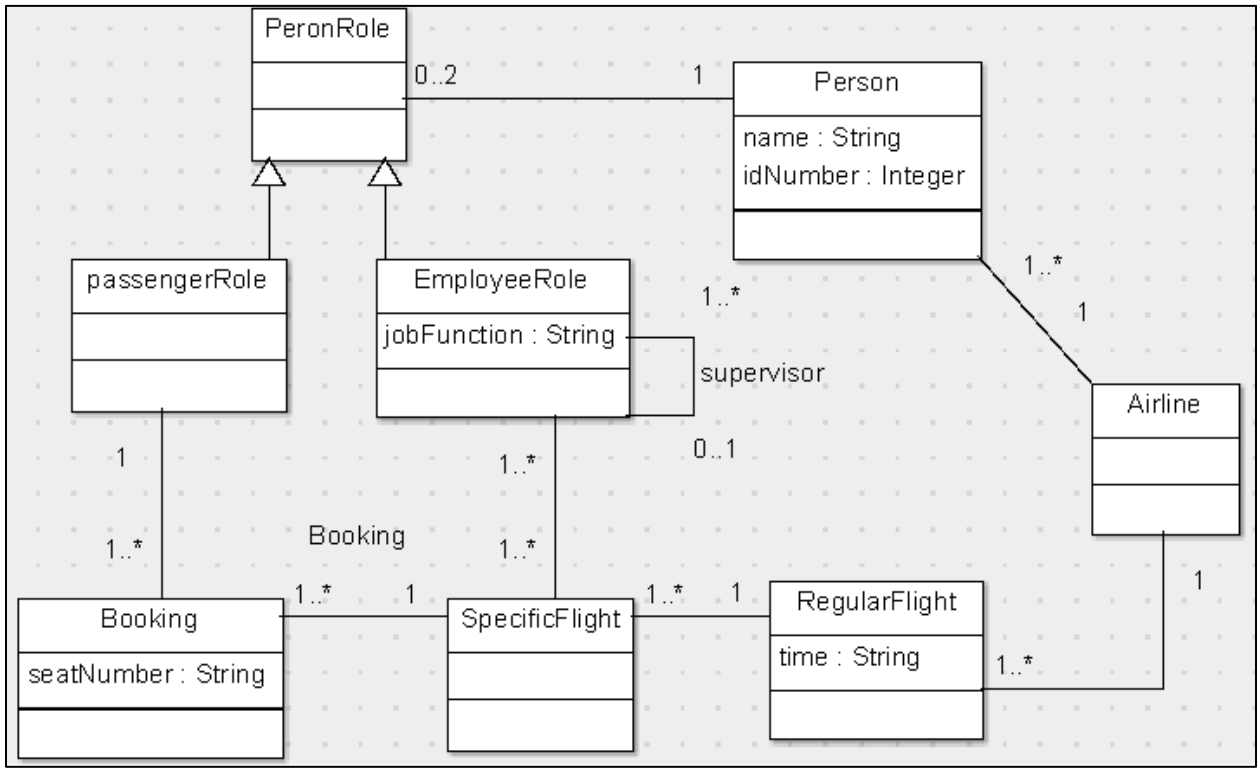


Figure 12: Airline system modelled in ArgoUML

The generated C++ code will be discussed in details in the comparison chapter against our criteria.

2.7 Papyrus

Papyrus is a graphical modeling tool that aims to create an environment to support any kind of EMF (Eclipse modeling framework) tool and specifically UML and target code generation for C++ and other languages. The following diagram is the airline system in Papyrus. Papyrus also supports UML profiles and SysML It uses Acceleo for code generation. It also requires extra add-ons to support code generation for C++. This works by defining a specific runnable configuration within Eclipse. It can be downloaded as an Eclipse plugin or a redistribution of Eclipse that comes shipped with Papyrus.

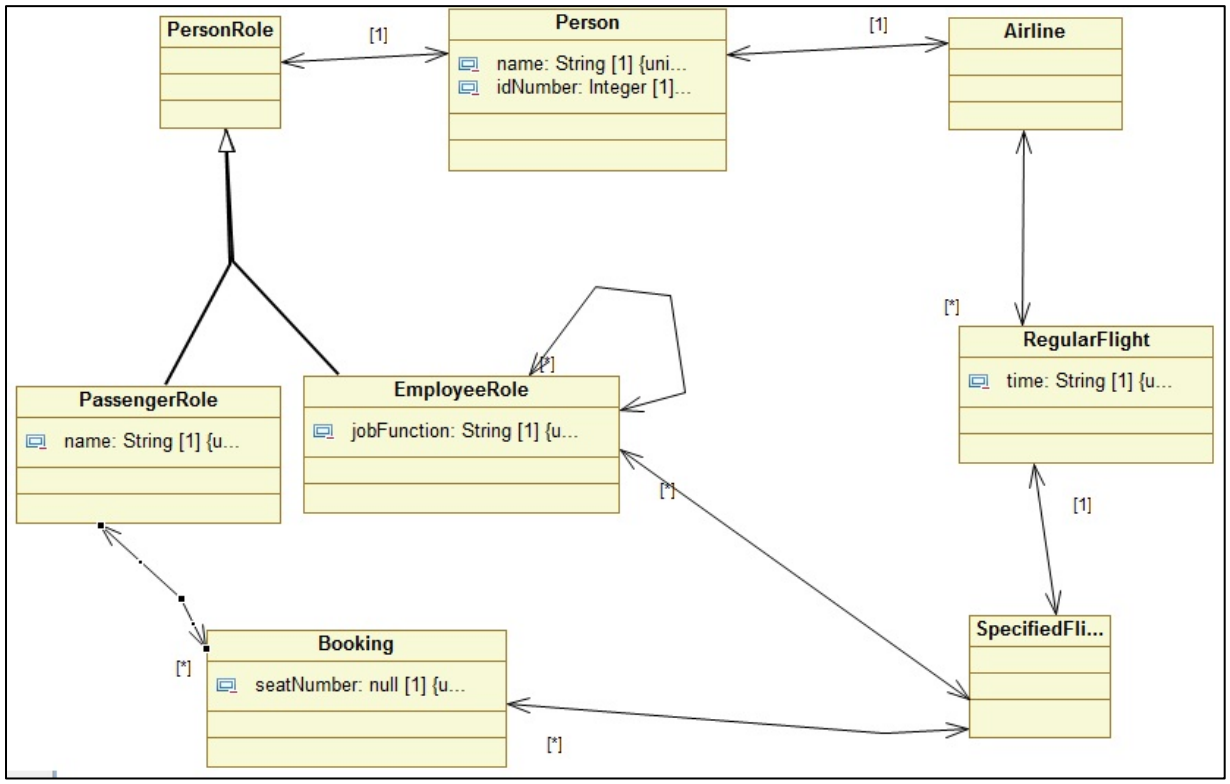


Figure 13: Airline system modelled in Papyrus

Chapter 3 C++ Code generator for Umple

In this chapter we discuss how we have developed the C++ code generator in Umple. We discuss various aspects of Umple, starting with Attributes

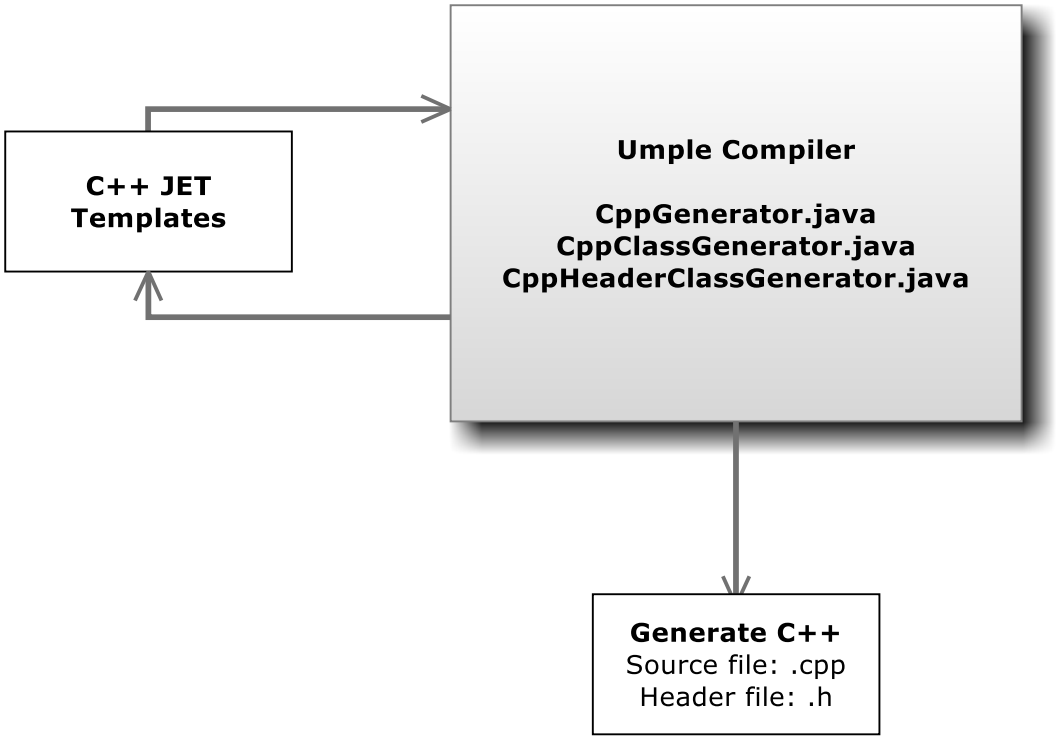


Figure 14: Umple C++ Generator

3.1 Attributes:

In Umple, one can declare attributes by typing the attribute name and type. If no type is specified, Umple will assume the default data is String. For each attribute, Umple provide an encapsulation. This means, all attribute are considered private and a public interface to set and get the attribute is provided. In the Umple C++

code generator, all the declarations of the attributes are private and included in the header file (for example: Person.h). In the implementation file (Person.cpp), the details and implementation for the setter and getter for that particular attribute are generated. Consider the following example where we have a class with two attributes:

```
generate Cpp;

class Person {
 name;
 Integer idNumber;
}
```

This will generate the following declaration in the header file:

```
 //-----------------------
 // Attributes for header file
 //-----------------------
 private:

 //-----------------------
 // MEMBER VARIABLES
 //-----------------------

 //Person Attributes
 string name;
 int idNumber;
```

Note that the attribute name is generated as 'string'. In the Umple compiler, we have a data type map to handle translation of primitive data type from Umple to types in C++'s STL. This is done in *'UmpleToJavaPrimitiveMap'*[1] in *Generator_CodeCpp.ump*. This map includes the following data types:

[1] The method has the word 'Java' in it, because it is modeled after the Java code generator. The other code generators also keep the word 'Java', and we chose to be consistent.

Umple Type	STL type
Integer	int
Boolean	bool
Double	double
Float	float
String	string

Considering the previous example, we should have the following interface generated for these attribute; header file code:

```
//-----------------------
// INTERFACE
//-----------------------

bool setName(const string & aName);
bool setIdNumber(const int & aIdNumber);
string getName() const;
int getIdNumber() const;
```

The details for these methods will be generated in the implementation file. Unless a design pattern was applied on these attributes, the default generated code for any attribute would be as shown above.

3.1.1 Design Patterns

Umple's support for different design patterns at the modeling level gives more control over the system. These design patterns are declared in the Umple model and the desired pattern will be generated in C++ accordingly. For the C++ generator, we have applied the following design pattern based on the java implementation:

3.1.1.1 Singleton Class:

The singleton design pattern allows a class to be only instantiated once or until that instance is destroyed. In Umple, generally, one can declare a class to be singleton by including the following line in the Umple model:

```
class Person
{
   singleton;
}
```

When the Umple compiler asserts that the class is a singleton, the following code should be generated to insure that the rules of the pattern had been applied. In the header file the following declarations are added to the class:

```
//-----------------------
 // STATIC VARIABLES
 //-----------------------

static  A*  theInstance;
A* getInstance();
```

In the implementation file, the details of these methods will be injected as follows:

```
A* A::getInstance()
  {
   if(!theInstance)
   {
     theInstance = new A;
   }
   return theInstance;
  }
```

Note that this implementation that currently being generated in Umple is not a multithread-safe solution; an improved implementation was suggested on an article by Scott Meyers and Andrei Alexandrescu [28] to allow singleton classes to run in a multi-threading environment using a locking mechanism with a double checking technique in C++.

3.1.1.2 Immutable Attributes:

When an attribute is declared as immutable in Umple generally, this means the attribute cannot be modified after construction. This is handled in C++ by limiting the accessibility of the attribute at the generator level. In the case of immutable, the following restrictions are enforced on every immutable attribute:

- The attribute must be private (as is the case for attributes in general)
- The constructor must provide initialization of the attribute
- Only the 'get' method is generated for that particular attribute.

Immutable attributes can be declared as follows:

```
generate Cpp;
class Person {
  immutable String name;
  Integer idNumber;
}
```

This should generate a regular interface for ‘idNumber’ (which is a getter and a setter) yet only a getter method for the attribute ‘name’ ; see the following header file code:

```
//-----------------------
  // INTERFACE
  //-----------------------

  bool setIdNumber(const int & aIdNumber);
  string getName() const;

  int getIdNumber() const;
```

3.1.1.3 Lazy Attribute:

An attribute can be declared as lazy in order to ask Umple to initialize the attribute within the constructor to 'NULL' if no assigned value was provided (or zero if it is a number). The idea is that such an attribute should be populated after construction. One can declare a lazy attribute according to the following:

```
generate Cpp;
class Person {
 lazy String name;
 Integer idNumber;
 }
```

This will basically remove the argument name from the constructor and initialize it to 'NULL' if no value is specified. The lazy pattern in Umple allows you to call the constructor and have the lazy attribute initialized without passing any value. See the following generated constructor for the previous Umple model. Note that the constructor doesn't require you to pass a value for name, it will be initialized to null:

```
 //------------------------
  // CONSTRUCTOR
  //------------------------

Person::Person(const int & aIdNumber)
  {
    name = NULL;
    idNumber = aIdNumber;
  }
```

Umple also allows you to combine multiple design patterns in several cases to have more accurate restriction on the behavior. For instance, you can combine the lazy pattern with immutable pattern. This will result in Umple generating a setter for the attribute yet it can be set only once. This can be done as the following:

```
generate Cpp;
class Person {
 lazy immutable String name;
 Integer idNumber;
}
```

This will generate the following constructor and interface for this attribute, note that the declaration of ‘canSetName’ will be generated in the header file:

```
 //-----------------------
  // CONSTRUCTOR
  //-----------------------

 Person::Person(const int & aIdNumber)
  {
    canSetName = true;
    idNumber = aIdNumber;
  }
```

As we can see, the attribute name is not initialized in the constructor. However, it can be set only if the helper variable ‘canSetName’ is true. As soon as the attribute name is set it won’t be modified. See the following interface generated for this particular case:

```
//-----------------------
 // INTERFACE
 //-----------------------

 bool Person::setName(const string & aName)
 {
  bool wasSet = false;
  if (!canSetName) { return false; }
  canSetName = false;
  name = aName;
  wasSet = true;
  return wasSet;
 }
```

3.2 Methods:

Consider the following class 'CodeTranslator' with the following methods defined in Umple:

```
class CodeTranslator
{
 String translate(String id, Attribute attribute) {
        return "1";
 }
 String translate(String id, AssociationVariable associationVariable) {
        return "1";
 }
}
```

This should generate code in the header file and the implementation file. In this header file we will get the definition of the methods:

```
 string translate(String id, Attribute attribute);
 string translate(String id, AssociationVariable associationVariable);
```

Also, in the implementation file, Umple C++ will generate a method body as provided in the model but in C++. Which means, the data type will be translated into C++ and the method will have a scope resolution of the class it belongs to, see the following code in the implementation file:

```
string CodeTranslator::translate(string id, Attribute attribute){
    return "1";
  }

string CodeTranslator::translate(string id, AssociationVariable associationVariable){
    return "1";
```

3.3 Associations

Proper support for associations is the most sought out feature when we are talking about tools for modeling UML class diagrams. They specify the relationship between classes and other aspects of the model. Associations' complexity in a particular model may range between very simple one-to-one associations between two classes to very complicated figures. However, they play a big factor in improving the quality of the model. A good manipulation of association always reflects good quality of design. Umple supports associations through the following:

Multiplicities:

The type of multiplicities in UML and Umple are:

- 1 : This means one and only one object must be present and linked to the current object.
- * : This means 'many' or unlimited number of objects (including zero) may be linked.
- 0..1 : This is often referred to as 'optional one' and means an object may be linked, but does not have to be.
- 1..* : This means there can be many linked objects, but at least one

In Java, the variables of association ends that have a multiplicity of many (*) are generated using the Interface List<>; when instantiated the class UnmodifiableList

is used. In the Umple C++ generator, they are generated using the STL template vector<>.

Umple has a direct mapping between model and generated code. When we declare an association between two classes, it is going to generate a set of artifacts that are mapped to this particular association. For instance, if we consider the following Umple model:

```
class Person{
  name;
}

class Car {
   color;
   0..1 -- * Person;
}
```

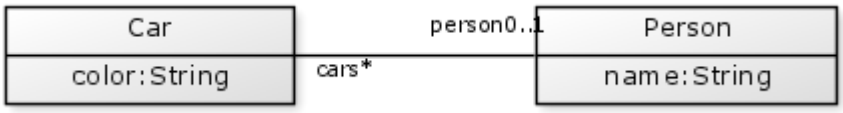


Figure 15: Associations in Umple C++

Associations result in more generated code than attributes. Depending on the type of the multiplicity at both ends of an association, Umple will generate a list of methods (API) to handle associations and has several issues to manage.

In each file of the associated classes, a variable will be defined correspondingly based on the type of the multiplicity. If the multiplicity is 1 or 0..1 then Umple will generate a single object of that correspondent class. If the multiplicity type is of type many, then a vector of that class will be generated. Consider our example above; the following table should demonstrate the implementation of association variables among these classes.

Car	Person
//Car Associations Person* person;	//Person Associations vector<Car*> cars;

Table 2: Association variables implementation

Association variables are implemented using pointers. As seen in the table, in the header file of class 'Person' a vector of type 'Car' is being generated and the association variable is called cars. On the other hand, a single object of type 'Person' called person.

In the class Car, Umple will generate the following API to handle the association variable 'person':

Class: Car	
Person* getPerson();	This will return the object person
bool setPerson(Person* aPerson);	This will also set the value of the object 'person'. Also will add this 'car' to the associated person that had been passed to this method.

Table 3: API for Class Car

On the other hand, the class 'Person' has a vector of cars. This requires more methods to handle the association of type 'many'. The following table demonstrate the API of the class 'Person':

Class: Person	
Car* getCar(int index);	This will return the car based on the index number sent to this method.
vector<Car*> getCars();	This will return the whole vector of cars.
int numberOfCars();	This will return the size of vector.

bool hasCars();	This will check whether this vector is empty. (Whether this person has any cars)
int indexOfCar(Car* aCar);	Query about an index of a specific car in the vector
static int minimumNumberOfCars();	Query regarding the lower bound of the multiplicity (in this example it will return 0 because multiplicity of type '*')
bool addCar(Car* aCar);	This will add a car to the vector
bool removeCar(Car* aCar);	This will remove a car from the vector.

Table 4: API for class Person

Association in constructors/destructors:

If the association is of type 'many', the constructor of the class uses a vector, which is automatically initialized by stl; the constructor code would therefore look like the following.

```
//-----------------------
 // CONSTRUCTOR
 //-----------------------

Person::Person(const string & aName)
 {
  name = aName;

 }
```

When we are deleting a person, we have to make sure that the pointers to each associated car are reset so there are no dangling pointers. Vectors destroy their objects by calling the destructor of that objects implicitly. However, in case the member of the vector is pointer to another object, it has to be deleted using 'delete'

explicitly (manually). Hence, in the destructor of the object, we iterate the vector and assign all its objects to 'NULL'. The following is an example of a destructor for class 'Person' that has a vector 'cars':

```
//-----------------------
 // DESTRUCTOR
 //-----------------------

Person::~Person()
  {
    for(i =0; sizeof(cars); i++)
    {
      cars[i]->setPerson(NULL);
    }
  }
```

Reflexive Associations:

In the Umple C++ code generator, a reflexive association is a class that has an association to itself; this case often happens. Consider the following UML model:

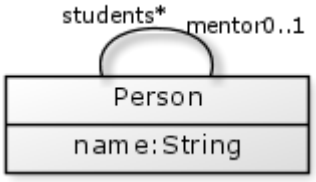


Figure 16: Reflexive Association

The choice of variable names is based on the UML role names. See the following code to understand this more:

```
//------------------------
 // MEMBER VARIABLES
 //------------------------

 //Person Attributes
 string name;

 //Person Associations
 Person* mentor;
  vector<Person*> students;
```

3.4 Generalizations

Generalizations indicate inheritance and specify that the class will inherit all the properties of the ancestor class. It is important to remember that generalization is different from association. Generalization is represented as a filled arrow link without multiplicity while association could be just a link or with an arrow when directional. In Umple, generalization is represented with "isA" annotation within a class.

The following example shows the generalization between classes in Umple:

```
class Person{
   name;
}

class Mentor{
  faculty;
  isA Person;
}

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

This Umple model is a representation of the following UML model, see Figure 17, that demonstrates generalization in Umple C++:

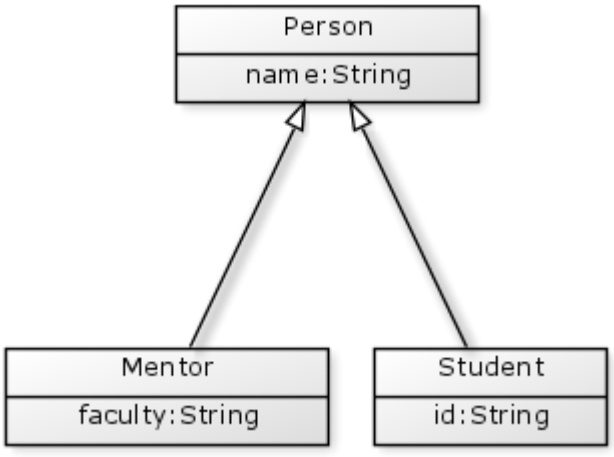


Figure 17: Generalization in Umple C++

A generalization in Umple represents an inheritance relationship. Which is a kind of relationship that states that the class Mentor 'is-a' Person. Technically this should generate a public inheritance relationship between the two classes Mentor and Person as shown below, the same rule applies for Student 'is-a' Person. This will generate the following declaration in the header file of each class:

```
//class Student
class Student: public Person {
.
.
.
// class Mentor
class Mentor: public Person {
```

Now we know that Umple treats inheritance as an 'is-a' relationship, and we know this will generate a public inheritance, what about private inheritance? It is important to note that private inheritance in fact does not represent an 'is-a' relationship. In fact, private inheritance is more of an implementation technique than a design technique and inferior to composition rather than inheritance. Lets look at the behavior of private inheritance to understand this more. There are two

main rules applies for private inheritance, the compiler does not convert the Mentor class into Person when it compiles. Also the member will become private in Mentor even if they were public in Person. This doesn't represent the relationship 'is-a' rather it is an implementation-oriented way of working with classes, often used when a developer only wants to inherit some properties of the base class. Therefor, we are not considering this type of inheritance within Umple context. Some developer ought to use private inheritance to minimize object size when used with libraries; it is not really a big deal within this context. [29]

Interfaces

In Java, the concept of interface was introduced mainly to allow multiple-inheritance. Multiple-inheritance is allowed in the C++ language by its nature. However, in Umple C++ code generator, multiple-inheritance follows the same style as Java, which means it can only be used with interfaces. This could be fixed in future work by extending Umple to have special capabilities if C++ is being generated.

Multiple Inheritance

Consider the same above example except the fact the WingedAnimal and Mammal are both interfaces and the class Bat is inheriting these interfaces ('implement' in Java).

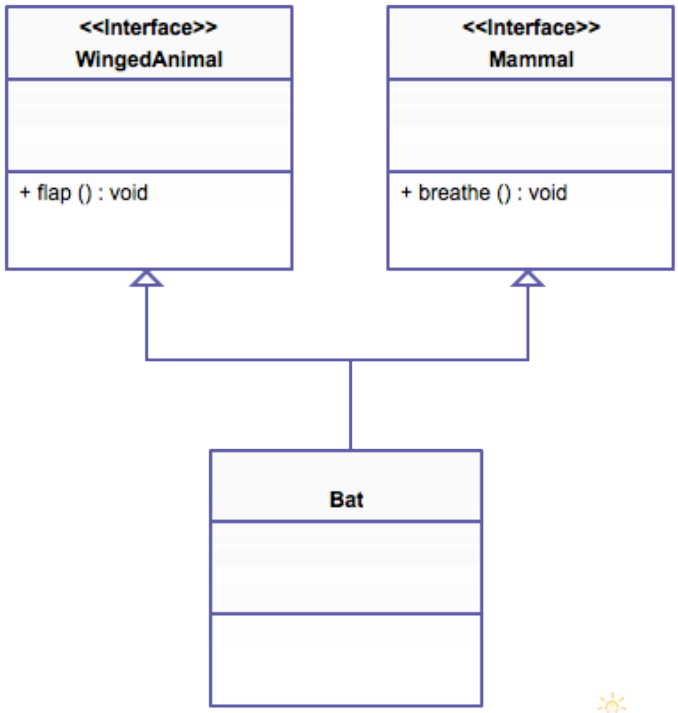


Figure 18: Multiple inheritance in Umple C++

```
generate Cpp;

interface Mammal
{
  void breath();
}


interface WingedAnimal

{
  void flap();
}


class Bat {
  isA Mammal;
  isA WingedAnimal;

}
```

This should generate Mammal and WingedAnimal as interfaces while Bat as a class with multiple inheritance. For any interface, only header files are generated with constant variables and abstract methods. A method is considered virtual when it is declared virtual without a body followed by semicolon and equal zero “; =0 ”. A pure virtual method must be implemented when the interface is overridden. To clarify this more, we can compare this to a virtual method. A virtual function can be overridden yet a pure virtual method has to be overridden.

For the example above, the following code will generated:

Mammal.h

```
/* EXPERIMENTAL CODE - NON COMPILEABLE VERSION OF C++ */
/*PLEASE DO NOT EDIT THIS CODE*/
/*This code was generated using the UMPLE 1.17.0.2937 modeling language!*/

#ifndef MAMMAL_H_
#define MAMMAL_H_

class Mammal
{
  // ABSTRACT METHODS
 public:
 virtual  void breath() = 0;
 virtual ~Mammal(){}
};
#endif
```

WingedAnimal.h

```
/* EXPERIMENTAL CODE - NON COMPILEABLE VERSION OF C++ */
/*PLEASE DO NOT EDIT THIS CODE*/
/*This code was generated using the UMPLE 1.17.0.2937 modeling language!*/

#ifndef WINGEDANIMAL_H_
#define WINGEDANIMAL_H_

class WingedAnimal
{
  // ABSTRACT METHODS
  public:
 virtual  void flap() = 0;
 virtual ~WingedAnimal(){}
};
#endif
```

For the class “Bat” it will generate a regular a class with two files regularly; “Bat.h” and “Bat.cpp”. In this context, Umple will generate implementation to override the abstract methods defined in the two interfaces those were inherited by identical definition in the Student class. Therefore, the code for Bat will include the following methods:

Bat.h

Class declaration would be:

```
class Bat: public Mammal, public WingedAnimal
…
   void breath();

   void flap();
```

3.5 State Machines:

State Machines in Umple, with Java code generation, have been specified by Omar Badreddin in his Phd thesis [4]. State machines in Umple consist of the following:

- State: which is a set of values.
- Transition: an event (method call) and destination that will switch between states. A transition may also have:
  - Action: a block of code to execute when an event is triggered
  - Guard: A condition that has to be evaluated to true in order for triggering to occur.

Umple supports several flavors of state machines :

- Basic state machines
- Nested state machines
- Concurrent state machines
- State machines with doActivity (not currently support in Umple C++)

Here is an example of a basic state machine; we will explain the implementation as we show the generated code. This example is fetched out of the Umple online manual and the implementation of the C++ state machine was driven accordingly. The following state diagram was generated by Umple Online; Consider a state machine for a garage door:

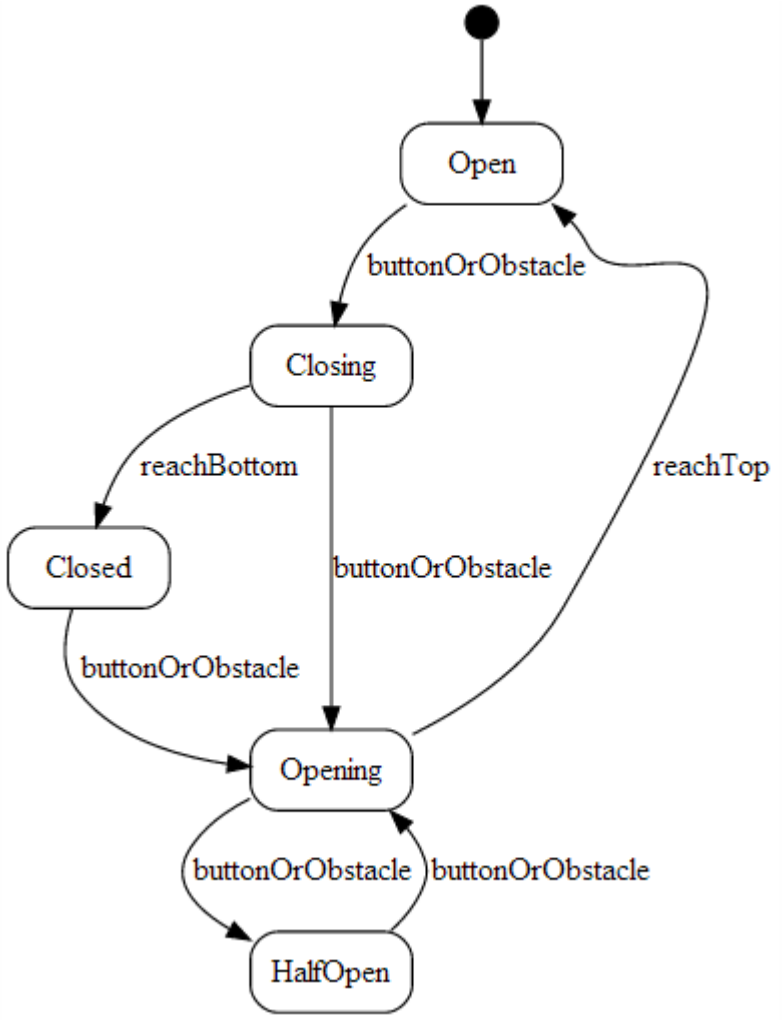


Figure 19: Garage door state diagram generated by Umple Online

As seen in Figure 19, we have a state machine with the following elements:

- States: Open(initial), Closing, Closed, Opening, HalfOpen.
- Events: buttonOrObstacle, reachBottom, reachTop

This state machine can be represented in Umple according to the following syntax:

```
class GarageDoor
{
  status {
    Open { buttonOrObstacle -> Closing;  }
    Closing {
       buttonOrObstacle -> Opening;
       reachBottom -> Closed;
    }
    Closed { buttonOrObstacle -> Opening; }
    Opening {
       buttonOrObstacle -> HalfOpen;
       reachTop -> Open;
    }
    HalfOpen { buttonOrObstacle -> Opening; }
  }
}
```

There are various implementations of state machines in C++. However, one common way to do it is to use an enumeration to represent states. This is how it is being done in Umple Java and other languages generated by Umple. The only issue we had with implementation is the fact that in C++ the enumeration literals for the states actually have numeric values. Which means you cannot retrieve a state name as a string out of an enumeration Processing state machines without having each state represented as string makes the implementation less readable and unsmooth. For instance, consider the above garage door example assuming we are in the initial state, in this case, if we try to run a query regarding the current state the value will be numerical. Performing a comparison between string and a numerical literal does not make sense. Look at the following switch case before we make translate these literals:

```
string GarageDoor::getStatusStringName (Status status)
{
  switch (status)  {
    case “Open” : {….}
    case “Closing” : {…}
        .
        .
        .
    default: {return ""; break;}
  }
}
```

The above code will not work since the value of status (which is an instance of the enumeration) will always hold a numerical value. Hence, to solve this issue we had taken advantage of the fact that we are working in a model driven development approach. Which means the state machine we have is presented in Umple model prior to the code generation phase. This allows us to have more control over the code. We know already that the first enumeration 0 will be ‘Open’ we know that from the Umple model. Hence, we could simply create a switch case that returns the string name of that particular state if the enumeration number matches what we are expecting. We also know the second one will be ‘Closing’. Therefore, we have added extra methods to the state machine API in Umple that should help handling state machines in C++. We will explain all the modification that had been added to the implementation as compared to generated Java code for state machines.

In the header file, an enumeration is being declared and an instance is being created. For the above example that would be:

```
//-----------------------
// MEMBER VARIABLES
//-----------------------

//GarageDoor State Machines
enum Status { Open, Closing, Closed, Opening, HalfOpen };
Status status;
```

We have added the method “getStatusStringName” to get the string name of status. See the implementation of the method:

```
string GarageDoor::getStatusStringName (Status status)
{
  switch (status)  {
    case 0 : {return "Open"; break;}
    case 1 : {return "Closing"; break;}
    case 2 : {return "Closed"; break;}
    case 3 : {return "Opening"; break;}
    case 4 : {return "HalfOpen"; break;}
    default: {return ""; break;}
  }
}
```

We use this method whenever we want to retrieve the literal value of the states. Usually, we need the literal value to compare values within the code. For example, when we ask, if we are in state “Open” then go to “Closing” when “buttonOrObstacle” is triggered. It becomes very difficult if we do it without using the state’s name. For example, if we are in “0” go to “1” if “buttonOrObstacle” is triggered. Hence, the idea behind adding this translation of state names is to support the claim that it enhances the code from usability perspective.

The following table illustrates each method generated for ‘GarageDoor’ state machine:

GarageDoor API	
string getStatusFullName();	This will return a composed name of the states. (in case we have nested states)
Status getStatus();	This will return the current state (number)
string getStatusStringName (Status status);	This will return the state name in string
bool buttonOrObstacle();	An event that is triggered when called
bool reachBottom();	An event that is triggered when called
bool reachTop();	An event that is triggered when called
void setStatus(Status aStatus);	Set the state to the one received as parameter.

Table 5: GarageDoor API (Statemachine API)

For each event created, a method is generated as shown in table 4. Based on the design of the state machine, these methods will direct indicate the change (entry/exit) between states. For instance, consider the event ‘buttonOrObstacle’, the implementation will run a switch case on ‘status’ to know the current state. Accordingly, it will change states. If we look at the code, the event will run an inquiry on ‘status’, in case the current state is ‘Open’ and this event ‘buttonOrObstacle’ was called then it will change state to ‘Closing’. Similarly, if we call the same event again while the current state is ‘Closing’ it will change the state to ‘Opening’, and so on. Ultimately, every time an event method is being call, it will return a Boolean value whether the change was successfully processed. See the code:

```
bool GarageDoor::buttonOrObstacle()
{
  bool wasEventProcessed = false;

  switch (status)
  {
    case Open:
      setStatus(Closing);
      wasEventProcessed = true;
      break;
    case Closing:
      setStatus(Opening);
      wasEventProcessed = true;
      break;
    case Closed:
      setStatus(Opening);
      wasEventProcessed = true;
      break;
    case Opening:
      setStatus(HalfOpen);
      wasEventProcessed = true;
      break;
    case HalfOpen:
      setStatus(Opening);
      wasEventProcessed = true;
      break;
  }

  return wasEventProcessed;
}
```

3.6 Style of Generated C++ Code:

C++ is a rich language and has many features and conventions. This richness, however, may often bring complexity along with it and often makes the code less readable and more error-prone. Style, however, is a way to write code to make it

more readable and understandable by other developers. It also makes the code consistent and easy to debug. The style we are considering for the C++ code generator is based on good object-oriented practices. When generating the code, we are not only considering C++ convention but Umple style as well; this can be seen on several parts of the code, we will go through this in details in this part of the thesis. Umple, despite the targeted platform for code generation, has a style that is being enforced which aims to increase readability and enhance usability at the code level. This can be seen on the following parts of the code:

- Comments: Umple divides the structure of the generated code using comments. Comments that indicates where each group of components is being declared. This structuring highly increases readability. You may refer to the comparison chapter to read more about the analysis of metrics of Umple in terms of comments, chapter 4.

- Naming: Naming style is a very common way to enhance code readability and allow for consistency. Umple uses camelCase style for method although Google standards for C++ suggest PascalCase for naming method [30]. camelCase and PascalCase makes reading and typing methods names easier. However, in some situations they may not serve well; situations like single letter words and special words like ‘iPhone’ in the middle of a method name makes it less readable; ‘getIPhoneNumber’ or ‘getUrl’. Although Umple style is strictly applied on targeted platforms, we may consider small changes if the tradeoff is conventional to the language and worthwhile.

There are several rules we are considering when generating code; those defines how every aspect of the code generation is implemented. Those are:

Files: Generally, for each class in an Umple model we consider generating a header file associated with an implementation file. In some cases when an LTTng tracer was detected more files will be generated. Generated files should be one of the following:

- Header file (Person.h): This file contains definition and declarations of each element in the class. Each header file carries the name of the class. For instance for a class ‘Person’ the header file will be ‘Person.h’. Also, the

'#define' guard name should be identical to the class name, in this case it would:

```
#ifndef PERSON_H_
#define PERSON_H_
…
#endif
```

This is how it is being done in Umple currently, however, this could be improved from readability perspective by adding the project name and directory to the guard name. The '#define' guard helps avoiding unnecessary inclusions.

- Implementation file (Person.cpp): this file contains all the implementation details of functions, constructor and some additional code (such as initialization of singleton-related variables). For a class 'Person' an implementation file named 'Person.cpp' will be generated; this file includes 'Person.h'.

- Tracepoint files (name_tracepoint.tp): Tracepoint files are generated when an LTTng tracer is detected for C++ and a tracing annotation on element was detected. It contains information regarding traced elements. This file is meant to be compiled by LTTng to generate tracepoint files. For each element traced, we generated a tracepoint file. Refer to chapter 3.11 for more details about the content of the file. In state machines, currently one file is being generated for each annotation. However, this could be improved by generating two files; one for entry and one for exit of that state machine if tracing both. This could also be improved by adding the class name in the tracepoint file to avoid conflict with other classes tracepoint files in the same directory. This is still under development and has not been completely polished.

To manage inclusion we always include the header file in the implementation by default. In case an LTTng tracer was detected, we also include generated

tracepoint. For instance, assume a class Person and we are tracing the attribute ‘name’, the inclusion in the implementation file would be:

```
#include "Person.h"
#include "name_tracepoint.h";
```

In case we are tracing more than one attribute, say we are tracing ‘id’ too, the inclusion would be:

```
#include "Person.h"
#include "name_tracepoint.h";
#include "id_tracepoint.h";
```

Note we are not including the tracepoint file ‘name_tracepoint.tp’ but rather the header file ‘name_tracepoint.h’ which will be generated when you compile the tracepoint with LTTng, more details on the tracing chapters. For interfaces, we only generate header file. We will discuss the content of this file later in this chapter.

Declaration Order: In Umple C++, declarations are done within header according to the following order:

- private:
  - Header attributes: Attributes and association variables
- public:
  - constructor
  - operator=
  - interface: setters, getters and helper methods
  - destructor

The implementation file has the same order for methods.

Constructors:

Default constructors are called when a class is being instantiated. A constructor constructs objects and initializes them. In Umple, a constructor with a list of arguments for attribute is generated for each class. The reason why we are defining

constructor with a list of arguments is to ensure the initialization of the attributes when an object is instantiated is done right, it is considered bad practice to have objects created with uninitialized variables. One can use the lazy pattern to ask Umple to assign a value for certain attribute, since lazy attribute are excluded from constructor arguments. Refer to 3.1.1.3 for more information on lazy attribute. Association variables don't need to be initialized in constructor since they are defined as pointer.

The default constructed is the one called by the compiler to initialize attributes and has a default value assigned to it; sometimes has no argument. However, a non-default constructor is that take arguments but the value is passed to the constructor on the time the call has been made. So in a nutshell, consider the following three classes A , B and C:

```
class A {
    A();      // default constructor
};

class B {
    B( int x = 0 , int y = 10);            // default constructor
};

class C {
    C(int x , int y);              // non-default constructor
};
```

We might consider enhancing constructors in Umple C++ by declaring them as 'explicit'. This will allows us to avoid bugs when the compiler performs type conversion on 1-argument constructors. The compiler is allowed to make one type conversion on 1-argument constructor; which could cause the compiler to perform unintended type conversion, this could be simply done by declaring constructor as the following :

```
Class Student {

explicit Student (int x)
{ }

};
```

Type conversion could go wrong when a 1-argument constructor is called by a function like the following example, consider the class student with a constructor that allows implicit conversion:

```
Class Student {
Private:
 int x;

public:
Student (int x) : y (x)
{}

int returnStudent ()
{
     return y;
}
};
```

So when a function calls another method that takes an object of type Student and only passes an ‘int’ like the following:

```
void aFunc (Student student)
{
     ..
}
```

Here is where aFunc is being called by another method called aCaller and aCaller is passing an int instead of an object of type Student:

```
void aCaller ()
{
     aFunc(14);
}
```

The compiler knows there is a constructor in Student that takes one argument of type ‘int’, therefore, it will allow to convert this argument into the expected type; This is called type conversion in C++. Now we know what is type conversion and we already know that it could produce bugs in some cases since the compiler could perform an unintentional type conversion in such situations. That is being said, to

declare a class as explicit will not allow aCaller to pass ‘int’. It will only accept the correct type to be passed to aFunc like this:

```
void aCaller ()
{
    Student student2 = new Student(14);
    aFunc(student2);
}
```

Other than that type of call when the constructor is declared explicit, the compiler won’t allow it. We need to investigate this deeper before making these changes. Therefore, this enhancement at this point is ought to be deferred for future work.

Copy Constructor:

A Copy Constructor is a constructor used to initialize an object with different object of the same class.

The C++ compiler provides a copy constructor by default if no copy constructor was defined. However, the copy constructor provided by the C++ compiler can easily go wrong in several situations. For instance, whenever we try to copy an object using the assignment operator. What possibly can go wrong is that when we copy objects using the assignment operator the compiler actually copies the address that the object is pointing to. In this case when the compiler calls upon the destructor to destroy the first object, it will succeed. However, when it tries to destroy the one that has the same address of the first object, the compiler is destroying an object that has already been destroyed. This issue often happens when using default copy constructor. This is caused because the compiler provides a member-wise (member-by-member) copying while pointer objects require deep copying mechanism. The solution to this problem is to use what is called 'deep copying'. Deep copying creates a new address for the object that is copying and copy the values the pointers is pointing to one-by-one; which solves the problem we mentioned earlier. However, to accomplish deep copying we are also considering defining assignment operator by overloading the operator.

A copy constructor and assignment operator are generated in all cases by default in Umple. We might consider improving the code by generating these only when

required. We only need those two whenever copying pointer objects is needed and this is required for some helper methods when dealing with associations. For instance, whenever we want to add a new member to the vector, we always check if the item is already there. This comparison requires evaluation of two objects of the same class. Therefore, defining an assignment operator becomes futile in this situation. There are several situations where copy constructor is used by the compiler:

- An object is being initialized to an object of the same class.
- Passing or returning objects by value.
- If the compiler needs to generate temporary objects.

  The compiler generates temporary object in several contexts: When a method returns/accepts a value by reference, when the compiler overloads a conversion operator or when the class defines an explicit copy constructor of another class.

Providing a copy constructor would contribute in solving these issues. Therefore, Umple C++ code generator will generate a copy constructor for each class according to the following example:

For a class A with an attribute 'name':

```
//------------------------
// COPY CONSTRUCTOR
//------------------------

A::A(const A & a)
 {
  this->name = a.name;
 }
```

Assignment Operator:

An assignment operator is need when we want to assign an object to another. It is different from copy constructor in a way that it doesn't require construction of new objects. Consider the following three cases:

- Student student; // this will invoke a default constructor
- Student std1(std2); // invoke copy constructor
- std1 = std2 // invoke assignment operator
- Student std1 = std2; // invoke copy constructor and assignment operator

In Umple, we generate an assignment operator as the following:

```
//-----------------------
// Operator =
//-----------------------

A* A::operator=(const A & a)
 {
  this->name = a.name;
  return this;
 }
```

Pointer vs Reference:

Implementing objects using reference or pointers both have their advantages and disadvantages. It all comes down to how to manage the code and what can be more effective for the particular context. In Umple, we implement with pointers. This is because the value Null is a valid input in our context and we need evaluate it. There is no concrete answer to which is better, many, including Google, argue that the difference is more likely syntactic and style.

3.7 Test-Driven Development of Umple C++ Generator

3.7.1 TDD of the Umple C++ Code Generator:

When the Umple project decided to incorporate a C++ code generator as one of its several generators, there were several ways to implement this.

- One possible way was to start the implementation of C++ from scratch, which means we create new JET template files for C++ and write generic

files for Umple starting with an empty template and then write new test cases from scratch for that matter.

- Another possible way would be to use a third party C++ code converter to convert Java, PHP or Ruby code that has been generated from Umple to C++ code. Tools like Tangible [31] and J2C [32] provide Java to C++ converter that could be integrated within Umple architecture. However, this would not be a very wise choice since we won't have control over the generated code. This means, the quality of the code would be in the hand of a third party tool and this might affect the consistency and style of the generated code of Umple. We have tested Tangible by converting an example of Java code generated from Umple to C++ using Tangible version 2.8. We can see several issues with the converted code from Tangible that is not compatible with Umple. For example, the tool generate strings using std::wstring this is not really recommended usage for Linux since it causes issues with byte size; std::wstring is based on wchar_t which supposed to contain a wide character. In windows it holds a 2-byte while in Linux it holds 4-bytes and this could cause problems for developers using the system on Linux or across different platforms and Umple is targeting generating code for different platforms, therefore, issues like this are hard to control since they are managed by third party libraries. However, although converting C++ from a language that is already implemented in Umple is unlikely to be considered, it is still a possible way to implement this C++ code generator.
- Another possible approach, which we have followed in the development of the Umple C++ code generator, was an agile approach that includes a test-driven development technique by duplicating the Java project and gradually converting it to C++. The development mainly included the following major steps:
    - JET Duplication: Duplicate the current Java JET project files (UmpleToJava) and call it UmpleToCpp. This will generate CppClassGenerator.java instead of JavaClassGenerator.java which is the translator that will be used by the compiler later.
    - Preparation of Testing Architecture: Duplicate all the test cases for Java generator and rename them correspondently to C++. For

instance, the file “JavaClassTemplateTest.java” is duplicated into “CppClassTemplateTest.java” and so on.

- Duplicate Code Generator: Duplicate the Generator_CodeJava.ump to Generator_CodeCpp.ump which should generate the CppGenerator.java within the compiler that handles the code generation process in general which contains some language-oriented mappings and code injections.
- Make small translation to the C++ project by changing the expected code from Java to C++. A small change like “Boolean” to “bool”.
- Run test cases knowing they will fail, for instance see Figure 20 :

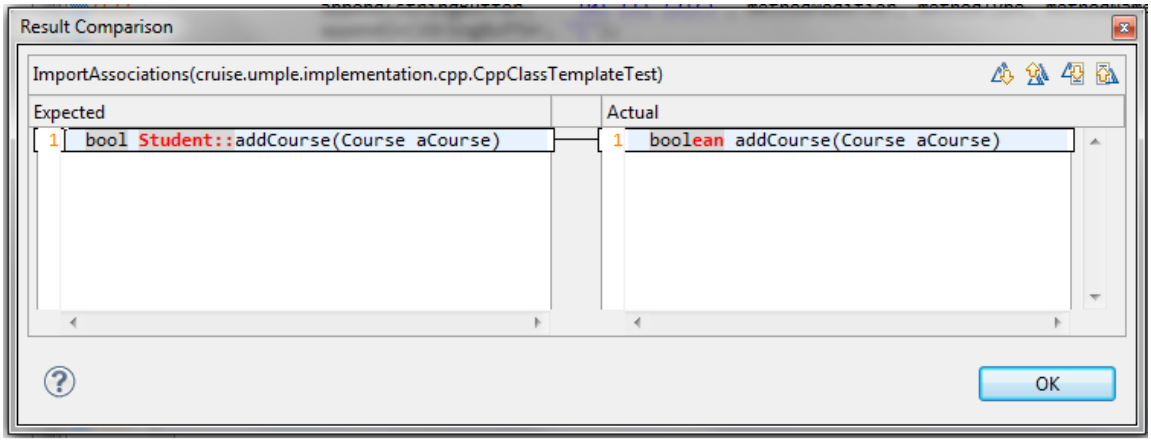


Figure 20: TDD of C++ Code generator

- Do the required refactoring on an iterated pace until all test cases pass.
- Repeat this process of gradual translation until the C++ code reaches the compilation level.
- Do semantic testing: writing C++ test cases based on the style of existing unit testing projects (testbed, testbed_php .. etc) of other languages in Umple ensuring that it compiles and logically behaves as expected in terms of C++ code generation.

The following flow chart, see Figure 21, shows the process of the TDD development of the project as explained earlier.

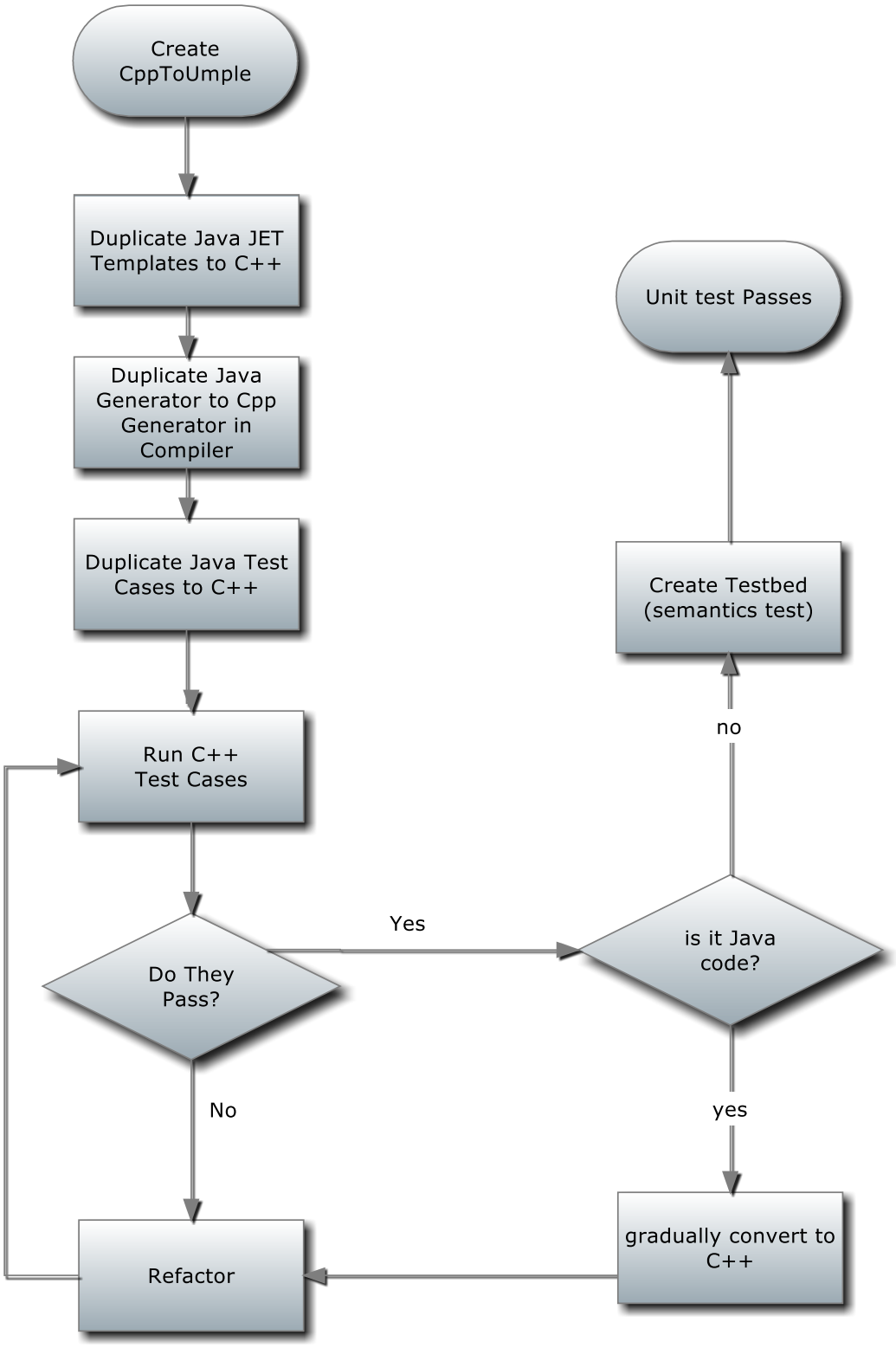


Figure 21: Flowchart of TDD of Umple C++

This process took roughly around 89 iteration to reach the current level of maturity. Processing roughly around ~100 test case. These test cases used for the TDD are syntactic tests that assure our generated code meets our expected code. The semantic test cases are created in a separate projet.

3.8 Tracing

Tracing of the C++ code generator is being implemented on top of MOTL (Model-Oriented Tracing Language), which is an internal DSL language in Umple. It aims to specify tracing at the modeling level, for more details on the specification of MOTL refer to Hamoud Aljamaan's work [16]. We will give a general idea about the concept in this chapter.

Tracing is a technique used to monitor systems to collect more details about the behavior of the system. The rationale behind enabling tracing is that it can allow for debugging problems in the system or maybe detecting suspicious behavior at run time.

Why do we want to trace ? what is it used for ?
It can be used for one of the following reasons:

- Learning about a particular system
- Debugging a system to find errors
- Performance analysis of the code
- Monitoring the system to detect suspicious behaviour

Umple targets the tracing tool LTTng UST [21, 33] by generating tracepoints based on the elements annotated within the Umple model. Allowing tracing at the modeling level may bring a lot of benefit to application-tracing developers, specifically LTTng, in several ways. First, the process of tracing an application can get very technically complicated, therefore, providing a tool like Umple with a tracing capability at the abstract perspective of the system will allow developers to focus on the high level logic of the system and maintain what is to be traced more efficiently. This can be reflected in several ways:

- Umple enhances readability of code. In Umple, developers deal with fewer line of codes, which is easier to maintain and understand; developers don't need to bother interpreting tracepoints or markers.
- MOTL syntax facilitates the process of tracing since developers can simply annotate elements to be traced.

In Umple, different UML elements can be traced: Attributes, associations, state machine etc. In the Umple C++ code generator, currently only attributes and some cases of state machines are being supported. The contribution to Umple in terms of tracing was done in two phases:

1- Porting the tracing work that has been done on Java to C++; this is primarily done by setting up the Umple C++ generator architecture for the tracing.
2- Writing a new a generator for LTTng tracepoint as part of the C++ generator that generates LTTng tracepoints and artifacts.

3.8.1 LTTng:

LTTng is a tool that had been developed to allow highly efficient tracing of applications on Linux. The tool supports two types of tracers:

- Kernel Tracing: To trace the Linux kernel; used to debug systems. We are not interested in this type of tracings at the current time.
- UST (User space tracing): This tracer is used to collect information about a user space activity. This tracer allows developers to inject tracepoint instrumentation within the code to trace specific attributes or methods. This type of tracer is what Umple is targeting for code generation.

Example: The following is a simple tracepoint that shows how the instrumentation is done and how it is compiled with LTTng. We will explain the content of this trace point later in this section with details on each arguments of the tracepoint.

Tracepoint:

```
TRACEPOINT_EVENT(
        sample_tracepoint,   ← The component
        message, // C++ Style comment
        TP_ARGS(char *, text),
        TP_FIELDS(
                ctf_string(message, text) ← tracepoint name and type

                 )
)
/*
 * Longer comments
 */
TRACEPOINT_LOGLEVEL(
        sample_tracepoint,
        message,
        TRACE_WARNING)
```

In the C++ application that we want to trace, a tracepoint call must be written with the correspondent header file according to the following:

```
#include <unistd.h>

#include "sample_tracepoint.h" ← tracepoint header
int main(int argc, char **argv)
{
        int i = 0;

        for (i = 0; i < 100000; i++) {
                tracepoint(sample_tracepoint, message,  "Hello World\n"); ← tracepoint call
                usleep(1);
        }
        return 0;
    }
```

When we run this application it will collect information regarding the component and will print the message ‘Hello world’ as a sample message record.

Umple supports several types of tracers including LTTng, in order to tell Umple what type f tracer to be used one can simply declare the type of tracer to be used within the model. For instance, consider the following example:

```
generate Cpp;
tracer Lttng;

class Person{
  name;
  trace name;
}
```

The second line tells Umple to consider the tracer to be 'LTTng'. Also, by typing 'trace name' this tells Umple to trace that particular attribute. We will discuss the several options to trace elements in Umple later in this chapter.

What is to be generated from Umple to C++ when LTTng tracer is detected?
When we studied LTTng tracepoint instrumentation in C++, we knew that in order to trace an attribute in a C++ application with LTTng there were several changes to be done to the code. These were:

- Creating tracepoints: which is a script file including information about the attribute to be traced and the event created. Tracepoints have extension of (.tp) and they contain information of instrumentation in general. For instance, consider our previous example of tracing the attribute 'name' of class 'Person'; a tracepoint for that particular attribute will be generated 'name_tracepoint.tp' and the content of the file will be as follow:

```
TRACEPOINT_EVENT(
name,
TP_ARGS(char *, text),
message,
TP_FIELDS(
ctf_string(message,name)
  )
)

TRACEPOINT_LOGLEVEL(
message,
TRACE_WARNING)
```

Note this is not C++ code; It is a textual format of a tracepoint; thou it has a structure similar to C++. The first line declares that this is a tracepoint event; various types can be recorded in a trace event. The second line indicates the name of the component to be traced. The tracepoint name is 'message'. 'TP_ARGS' is a macro contains the argument that are passed to the tracepoint, 'char *' is the type and 'text' is the name of the argument. This macro can take several types of argument we will discuss them later in this chapter and describe how they are mapped to Umple types. 'TP_FIELDS' allows you to write fields for the trace event where you can type a certain expression; in Umple this can be treated as trace record. For instance we could type, 'ctf_string (a suspicious name,name)' which will be recorded when LTTng collect this information. 'TRACEPOINT_LOGLEVEL' is an optional addition to the tracepoint to improve the debugging/monitoring process when the log is collected; one can use this to state whether this trace is critical for instance. 'TRACEPOINT_LOGLEVEL' has not been investigated in depth. Since it is optional, we kept it at a very optimal state for future update if needed. The focus was more on tracing UML components in Umple.

Another example of a trace point file, assume we are tracing an id of type integer, we can see how different types are being handled in LTTng tracepoint:

```
TRACEPOINT_EVENT(
id,
TP_ARGS(int, intfield),
intfield,
TP_FIELDS(
ctf_integer(int, intfield,id)
  )
)

TRACEPOINT_LOGLEVEL(
intfield,
TRACE_WARNING)
```

This file should be compiled with the tool 'lttng-gen-tp' [34]. This tool aims to simplify the process of generating the USP tracepoint files. Basically, when the tracepoint file (.tp) is compiled with lttng-gen-tp it will generate the following files: .h , .c , .o; named after the tracepoint file name. The header file can directly included in the C++ file generated from Umple. Umple already includes the expected header file when a component is being traced. For instance, Umple already includes 'name_tracepoint.h' when the attribute name is being traced; although this header file will only be generated after compiling the tracepoint file with lttng-gen-tp. When we had first begun this project, there were two options to handle the tracing process. Either we generate the .tp file and then compile it with lttng-gen-tp to get the tracepoint files or we could have generated these files directly. We have decided to go with the first option since it was easier implement a generator for one file with less line of code. Also, we wanted to avoid conducting frequent changes to align our version of the tracepoint files with LTTng changes. Writing a tracepoint files will allow us to avoid this since we only make changes to one file whenever the tool evolves and LTTng itself will generate the rest.

- Tracepoint header file: this header file imports LTTng header files and contain declarations of tracepoint. This file includes the file 'lttng/tracepoint.h' which has the definition of a tracepoint and 'lttng/tracepoint-event.h'.

- Tracepoint call injection: This is usually a line of code annotating what is to be traced. This call requires inclusion of tracepoint header file in the class implementation file in order to be used. For instance, the call for the attribute name would be injected as follow:

```
tracepoint(name, message, “Hello world”);
```

This line means: trace the attribute name and add the string “Hello world” every time a record is collected.

The injection of this call depends on the trace syntax in the Umple model. Based on the trace annotation, we inject this tracing call according to the following table:

MOTL syntax in Umple	Injection position
trace name; or **trace set name;**	Setter of the attribute
trace get name;	Getter of the attribute
trace set,get name;	Setter/Getter
trace name where name == ”john”;	Setter but activate only when name is set to “john”
trace name for 5;	Setter yet trace deactivate after 5 occurrences
trace name until name == “john”;	Setter but deactivate when name is set to “john”

Table 6: Optional tracing syntax

This is a C++ extension of the work and specification that had been done on Java by Hamoud Aljamaan on MOTL. Currently for C++, there are several issues with the code injection for trace calls, these issues will be fixed in future. The contribution was primarily to write an LTTng generator. However, from a technical point of view, there were several issues to be tackled. For instance, we

had to create a map between LTTng arguments and Umple types in order to match the the data types on the code generation level for LTTng tracepoint.

Types mapping between Umple types and LTTng arguments	
String	TP_ARGS(char *, text)
Integer	TP_ARGS(int, intfield)
Double	TP_ARGS(double, doublefield)

Table 7: Umple types and LTTng arguments map

Chapter 4 Comparison with Other Tools

Previously we introduced several C++ code generators that we are interested in comparing against Umple according to certain criteria. In this chapter, we will define our terms and present the comparison between the different C++ code generators.

First, we need to clarify some definitions to avoid ambiguity or misunderstanding of the meaning.

4.1 What are 'software metrics'?

The collective term 'metrics' is used when we want to describe a variety of concerns regarding measurements in software engineering [35]. However, in this thesis the term is limited to this definition: a software metric measures certain properties of the source code.

Software metrics and benchmarking are often used to measure the quality of the system based on several criteria determined by the evaluator. In this thesis we will be using a list of criteria to measure the quality of the C++ generated code in many terms. We will compare Umple according to these criteria.

4.2 Measurement Scales

In our evaluation of each criterion we are taking into account two types of metrics:

- Ordinal scale: We use this to evaluate subjective aspect of the system; things like consistency cannot be measured using an interval scale. Therefore, we are evaluating any subjective matter according to the following scale: 1 2 3 4 5. These are the following:

  1: Means either the system lacks this quality we are evaluating or it is extremely poorly reflected in the generated code.

  2: The system has the quality being evaluated but it is badly reflected.

  3: The quality being evaluated is not badly reflected in the code and doesn't reflect any notable insufficiency.

4: The quality is quite well implemented in the generated code.

5: The quality being evaluated is excellent, well implemented and bring remarkable efficiency to the code.

- Measurable (interval or ratio) scales: Values on such scales are collected through metrics and the static code analyzer software. The range of such scales varies depending on the criteria being evaluated. We use a tool called Source Monitor [36] to collect metrics for evaluating the such measurable qualities of the system. These include things like the number of line of code (LOC), the number of methods or other similar criteria. Tools tends to be give overly precise values with many significant figures, for instance we get a precise number '456' or a percentage in some cases "35.5%".

If the tool we are using doesn't answer all the criteria we want to run, we augment the results by using other measurement approaches.

The metrics generated from the tool SourceMonitor can be interpreted as the following:

- **Methods/class**: Methods per class, how many methods a class can take, this can be useful to measure the size of the API generated by the tools.
- **Avg statements/method**: This can be useful to measure how big a method is. We could have one method with the size of ten. This can give n idea about how roughly the average size of a method is.
- **Max complexity/Function complexity**: How complicated a method is. This is known also as cyclomatic complexity.

It is very important to keep in mind that we scale the evaluation based on a scale of 5. This allows us to match the result with the evaluation scale used by other tools. For instance, we are considering evaluating our results using GRL [37] (goal-oriented requirements language), converting our collected values to GRL become efficient and more accurate when using a scale of 5 since GRL uses similar scale. We will discuss this in depth in later chapter.

4.3 Metrics Generated from the Airline System

There are many different metrics and criteria one might consider when evaluating a system, and generated source code in particular. When we want to evaluate the system, we make sure that these metrics are related to the C++ source code directly. This means that we are avoiding any other metrics generated by the development environment or where the source code is being hosted. Many IDEs (Integrated Development Environment) may make small changes in different parts of the code or possibly inject certain statements that may affect the result of the metrics. For instance, the Eclipse IDE may inject several lines of comments for the developer that definitely increases the number of lines of code in general. Therefore, in the process of evaluating these systems according to our criteria, we are dealing with the code exactly as being generated from the targeted tool avoiding any additional changes to these files.

Figures 22 and 23 are Kiviat metrics diagram generated by SourceMonitor from the Airline system described earlier in section 2.5, 2.6 and 2.7. A Kiviat metrics diagram is a multi-vector line graph that shows how multiple variables interrelate. A good result in a Kiviat metrics diagrams can be reflected on the line drawn inside the green area, which means that the variables in the green zone are normal.

Figure 22 describes Umple and ArgoUML, while Figure 23 describes code from RSA and Papyrus.

Some things can be noted immediately:

- Umple's metrics are in or near to the 'green zone' considered normal, whereas the other tools generate code that is far from normal since, overall, the metrics shows that they lack sufficiency at some points. For instance, all three other tools provide comments more than the normal average.
- ArgoUML and Papyrus generate virtually no methods.
- We can also note that Umple has higher maximum complexity of methods than other tools where Umple stands at 6 and IBM is standing at 1. Although Umple has a maximum complexity of 6 it still in the green zone that means it is in the normal range while IBM, ArgoUML and Papyrus don't provide enough complexity to reach the normal level.

- When it comes to methods per class, it is useful for following good OO design. We can see from the graph that Umple has an average of 15.88 per class while the normal is between 4-20. This means that Umple is tending toward overloading the class with methods yet still in the normal zone. However, this means that we should be careful about expanding on the API unless it is necessary. Yes the size of API reflects power yet overloading the class with methods that are not used could increase the size for no reason yet currently we only generating what is necessary to run an efficient code for UML elements.

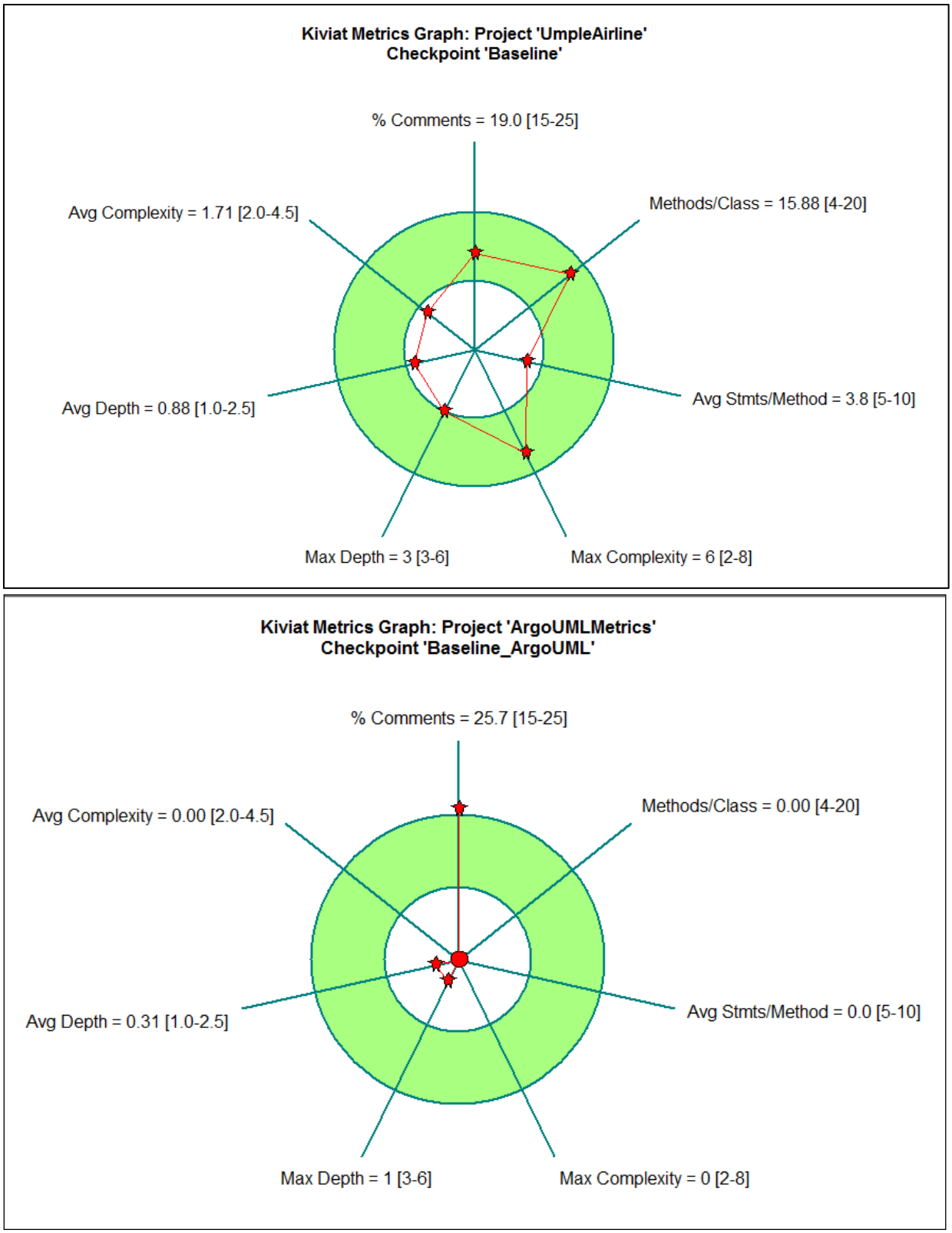


Figure 22: Kiviat Metrics for ArgoUML/Umple Airline System

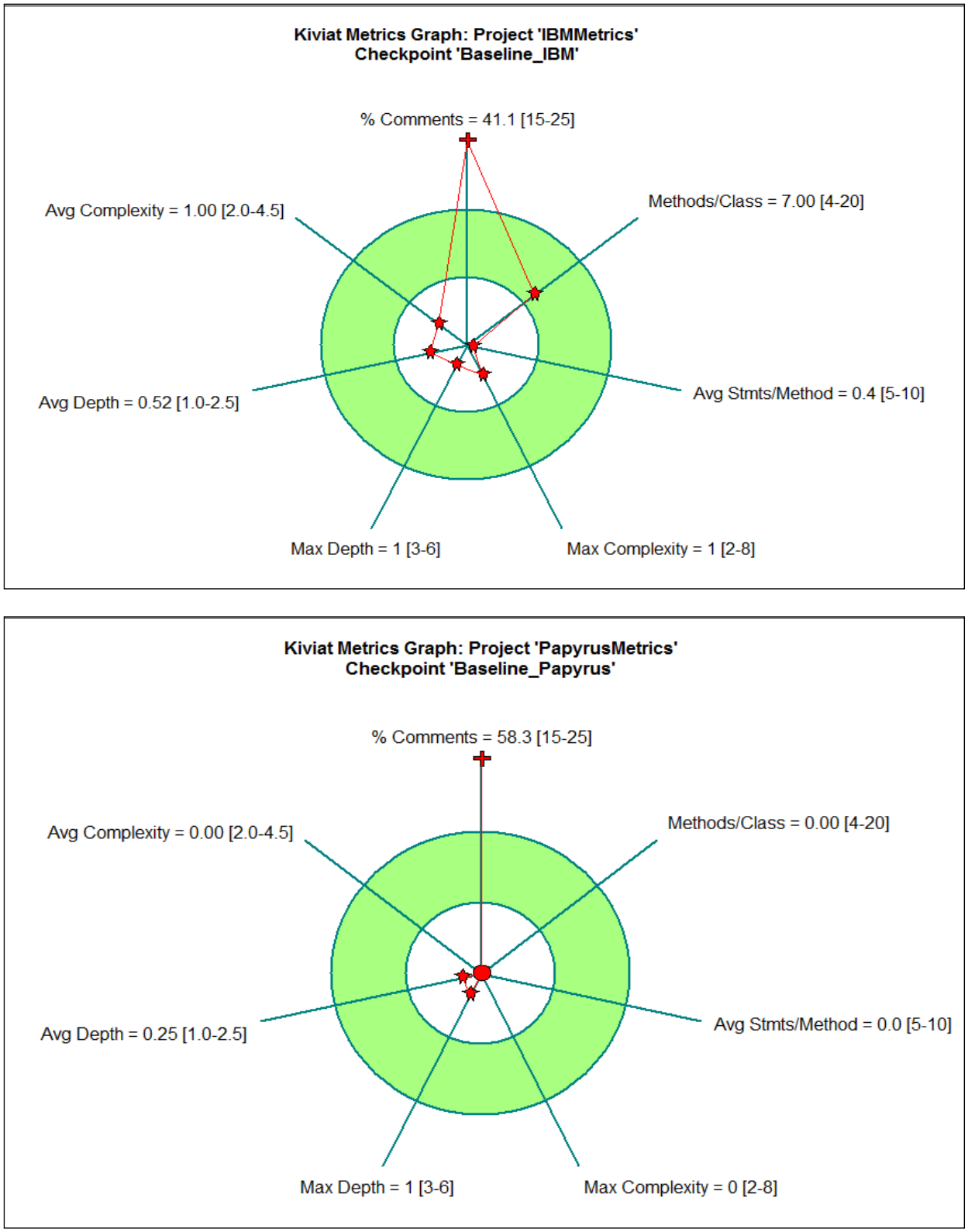


Figure 23:Kiviat Metrics for IBM RSA and Papyrus

4.4 Completeness

We try to measure the completeness of the system in several terms. Completeness of a C++ code generator can be seen in the following:

- UML Syntactic Completeness: How far does the tool support UML? Does it fully support all aspects of UML or some part of it? Any tool that provides UML-to-C++ code generation must have well-rounded support for class diagrams and state machines.
- Semantic Completeness: We can say a feature is semantically complete if it generates all relevant UML semantics. For instance, if the language supports nested state machines but doesn't generate all needed code, then we may say it is semantically incomplete. Another example would be to see whether the tool generates all required API for a certain model construct. So what we need to show here is how far the tool actually support UML aspects with proper implementation.
- Useful general capabilities: We consider completeness to be higher if the tool or language has useful extensions that facilitates its use in the real world. Abilities such as abilities to divide a model into components, to request special cases of code generation (e.g. the singleton pattern) are important here. Generating code that is correct but can't be used because it is not flexible or extensible suggests lack of completeness.

4.4.1 Completeness in Handling Attributes

Attributes are a major feature in UML. Any tool that is targeting modeling with UML must provide support for attribute essentials; things like declaring or adding an attribute to a class and assigning a type to it are features that all tools provide. However, some extended features can be provided based on the perspective and standards the tool is conveying.

We can see in Table 8 that Umple provides several features that are not being implemented in the other tools. For example, as we can see from the table, Umple provides several attribute-oriented design patterns, which gives more power to attributes in some cases and enhance the design of the system. Immutability on

attribute and Lazy patterns are discussed in another chapter, refer to chapter 3 for more details on the implementation of these patterns. We can also see that Umple doesn't allow modification of the access modifier; this means that all attributes in Umple are declared private. This limitation is on the Umple model level which means that it applies to other languages too. For instance, if you want to declare a public attribute, consider the following example in Umple:

```
class A {
  public name;
}
```

This will actually result in generating the following line of code in Java:

public private name;

This is in fact a parsing issue in Umple, check issue number '311' in the Umple bug tracking system [38] to read more about it. However, declaring public attribute still can be done in another way in Umple, one might consider writing the following:

```
class A {
  public static name;
}
```

If you do that in Umple, that line of code will be parsed as extra code, which means that Umple will appear the generated code as is. The tradeoff would be that Umple won't generate any API for that particular attribute since it is parsed as extra code. This can work for Java and maybe other languages too but not for C++, since public attribute has to be included in the public zone of declaration within the class, so in order for this to work in C++ it has to be parsed and managed independently. Umple considers the practice of declaring public attributes as poor OO design, so it is not suggested since the convention should be to hide implementation of the class and encapsulate the properties; therefore, it is not supported currently.

More importantly, we can see clearly that Umple provides more support for attributes than other tools as seen in table 8 below. Design patterns can be very effective in several cases. Also, the functionality of declaring *autounique* attributes in Umple allows the automatic increment of the value of an integer whenever the constructor is being called.

On the other hand, although ArgoUML draws UML attributes and generates declarations for attributes, It doesn't generate interface methods in the source file for declared attributes. In our Airline System example (Figures 10-12), if we look at the class Person we can see that it has two attributes 'idNumber : Integer' and 'name : String'. ArgoUML generates the following declaration for these attributes in the header file:

```
private:
    String name;
    Integer idNumber;
```

This is actually invalid C++ code. The data types have an issue with mapping from the UML model. ArgoUML expects the user to type the language-oriented data type within the model. So, if the user wants to generate C++ code, instead of typing name : String, the user should type name : string or idNumber : int instead of idNumber : Integer. In model-driven development, it is a good practice to avoid including language-oriented information within the model; especially when the tool provides code generation for several languages. Therefore, the tool should have converted these data type to STL types and also included the String.h header file, which is mandatory to use std::string data type.

One may argue that even Umple includes language-oriented information within the model. This is actually false; an Umple model has Umple types which correspond to UML and that are mapped to the targeted-language data type. This is handled within the compiler in the CppGenerator.java file. Umple in fact allows mixing model with code. So a developer can either write a C++ abstract model elements. This means that if you declare an attribute with idNumber : int or idNumber : Integer, these will both generate 'int' as a data type when the code is generated. This is handled well in IBM RSA also but not in Papyrus.

Table 8 indicates the areas where Umple advances in terms of support for UML attributes. We can see that Umple is the only tool that allows Lazy pattern and Immutable pattern. These patterns allow for more flexibility when working with attributes.

	Attribute			
	Umple	RSA	Papyrus	ArgoUML
Declaration	√	√	√	√
Access modifier	**X**	√	√	√
Getter/Setter	√	√	√	**X**
Lazy	√	**X**	**X**	**X**
Immutable	√	**X**	**X**	**X**
Key (unique)	√	**X**	√	**X**
Autonique value	√	**X**	**X**	**X**
Time/Date	√	√	√	√
Constant	√	√	√	**X**
Static	√	√	√	**X**
Overall Evaluation	**4**	**3**	**2**	**1**

Table 8: Support for attributes

Considering the fact the Umple has more support for attributes than other tools, makes Umple ahead in terms of completeness and support for attributes. This evaluation also considers the actual generated code

4.4.2 Completeness in Handling Associations

Table 9 shows the UML features that Umple supports, as compared to the other tools. In general, no other tool properly generates a comprehensive API for adding and removing objects linked by an association. The other tools do not support referential integrity either.

Umple follows the same convention discussed regarding attributes: Methods to access associations are public, and the data structures are private. Other tools offer more flexible visibility.

None of the other tools support UML's notion of association classes. This is fully supported in Umple.

Associations				
	Umple	RSA	Papyrus	ArgoUML
Declaration	√	√	√	√
Visibility	**X**	√	√	√
Role name	√	√	√	√
Directability	√	√	√	√
Referential integrity	√	**X**	**X**	**X**
Multiplicity	√	√	√	√
Navigability	√	√	√	√
Association class	√	**X**	**X**	**X**
Adding/removing objects	√	**X**	**X**	**X**
Overall evaluation 5/5	**4**	**2**	**2**	**1**

Table 9: Comparison of association capabilities

4.4.3 Further analysis of completeness

Lets take a look at the code generated by ArgoUML (refer to the appendix), we can see clearly that the tool has several places where we can detect semantic completeness issues. For instance, the tool draws state machines but doesn't generate any code or implementation for state machines. Also, although the tool generates definitions for classes components (attribute, methods, association. etc.), we can see clearly that the tool doesn't generate the implementation for these definitions. Yes, a header file with definition is important, yet implementation files ought to contain proper code and clarify how everything is being handled.

Now, lets consider IBM RSA, we can also see some semantic completeness issues with some aspects of UML. For example, if we look at the association between Person and PersonRole at the airline example in Figure 11 and Figure 24:

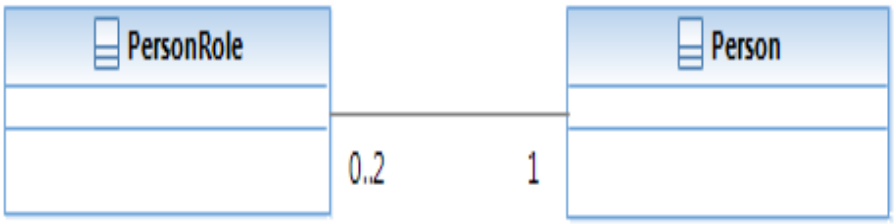


Figure 24: 0..2 to 1 association in IBM RSA

It is obvious that the association between these classes has an upper bound of 2, which is interpreted as: Person can have a maximum of two PersonRole objects. The flaw here is in the generated code; the code doesn't reflect what the association is annotating. Therefore, we can see this as incomplete semantics since you can model this type of multiplicity but it doesn't generate what is needed; if the tool doesn't support this type of multiplicity then the tool should have not allowed applying this feature in the model and it seems that by going to 'properties' you can simply edit the multiplicity to your desired upper bound.

On the other hand, for that particular case, whenever an object is to be added to a container in Umple, it will always compare the number of objects in the container against the multiplicity before adding the new object. For the Above example Umple will generate the following method to add objects when an upper bound on association is defined:

```
PersonRole Person::addPersonRole()
  {
    if (numberOfPersonRoles() >= maximumNumberOfPersonRoles())
    {
      return NULL;
    }
    else
    {
      return new PersonRole(this);
    }
  }
```

Table 10: Comparison of overall completeness

Overall Completeness				
	Umple	RSA	Papyrus	ArgoUML
Classes	√	√	√	√
Interface	√	√	**X**	**X**
Attributes	√	√	√	√
Association	√	√	**X**	**X**
Association class	√	**X**	**X**	**X**
Generalization	√	√	√	√
Multiplicity	√	√	√	√
|Multiplicity bounding	√	**X**	**X**	**X**
Directional Assoc.	√	√	√	√
State Machine	√	√	**X**	**X**
|Nested SM	√	√	**X**	**X**
|Concurrent SM	**X**	**X**	**X**	**X**
|DoActivity	**X**	**X**	**X**	**X**
Design Patterns	√	**X**	**X**	**X**
UML Profile	**X**	√	√	√
Overall	4	**3**	**1**	**1**

When it comes to Papyrus and ArgoUML, both tools have huge issues with completeness; they poorly provide implementation of what is being shown in the model. This can be seen in almost all aspect of the model, we will list a table that illustrates this claim. For instance, lets take a look at the two files generated by ArgoUML for the class 'Airline' from the airline system. We can see that the implementation file is roughly empty.

Ailine.cpp

```
#include "Airline.h"

   /* {src_lang=cpp}*/
```

Airline.h

```
#ifndef Airline_h
#define Airline_h

#include <vector>


class Person;
class RegularFlight;

class Airline {
   /* {src_lang=cpp}*/


 public:

   /**
    * @element-type Person
    */
   std::vector< Person* > myPerson;

   /**
    * @element-type RegularFlight
    */
   std::vector< RegularFlight* > myRegularFlight;
};

#endif // Airline_h
```

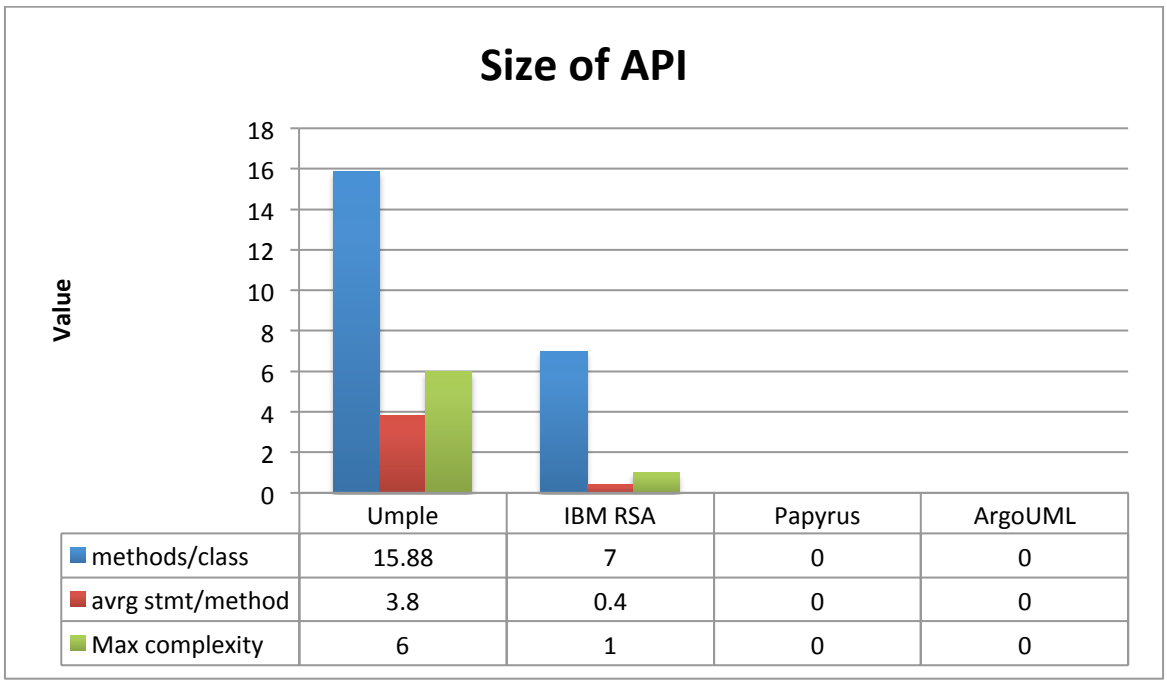
As seen in the two files listed above, we can see that there is an issue with semantic completeness since the tools graphically represents associations yet only generate declaration for these associations which means that if you want to add objects or remove objects you will have to write the code for that. This also mean that you have to edit generated code which is not recommended from an MDD point of view.

Table 10 shows a comparison between Umple and the comparator tools in terms of completeness.

4.4.4 Size of API

Size or richness of the API refers to the overall power of the set of methods that add more support and flexibility to any element of the class. For example, consider setter and getter for private attribute. As long as no redundancy is introduced, the bigger the size of API, the easier it becomes to maintain the property of the class and the more useful it becomes. However, we don't claim that it is always sufficient that the size of API is better in all cases. There are several cases where size doesn't pay off; for instance, when generating code for embedded devices. In this feature, we also try to measure the power of the API by considering its size.

If we look into the Kiviat metric graphs generated from the code analysis for these tools presented earlier, we can see clearly that there is a huge gap between the size of API generated from Umple and the code generated by other tools. Umple has a bigger size of API than these tools in two different ways. The number of methods provided for each class. Secondly, the benefits these methods bring to each class. We know for a fact that a class with too many methods could be overloaded with unnecessary functionalities. However, in Umple case, we have about 15 methods per class for the code generated from the airline example provided in chapter 2. Also, approximately, each method has an average of around 3.4 statement per method, based on the metrics generated from the tool we are using for code analysis. All these methods aim to bring more flexibility and support for handling operations on UML elements such as attribute, associations, etc.



	Umple	IBM RSA	Papyrus	ArgoUML
methods/class	15.88	7	0	0
avrg stmt/method	3.8	0.4	0	0
Max complexity	6	1	0	0

Figure 25: Comparison in terms of the size of API for airline example

We can see from this graph that there is a huge gab between Umple and other tools. ArgoUML and Papyrus barely generate anything. In terms of method implementation they generate nothing. IBM RSA generate some methods but when we want to generate a code that is efficient we should at least generate the implementation for these methods. Umple's philosophy is to generate high quality code; this can be reflected on the size of API.

Currently, Umple doesn't have a plan toward providing mechanism to limit the size of API generated. This feature could benefit those concerned about size.

4.4.5 Other features

Completeness also covers adding additional features of tools. Consider for example, support for a number of design patterns or similar features that add value

to the generated code or the tool itself such as support for aspect-orientation which facilitates the development process.

Umple supports several additional features to support the model that other tools don't. Consider the fact that Umple's support for aspect orientation gives Umple more power to inject code at certain places, which can be very helpful in many situations, such as logging.

4.5 Ease of Use

We will consider five aspects of ease of use: Ease of installation, and flexibility.

4.5.1 Ease of installation

Does the tool require third-party libraries? Does the tool require installation? Or is it possibly available in an instantaneous form such as web-based application. Installation can be facilitated in many different forms. For instance, some tools can be available as a stand-alone application, as a plugin for IDEs such as Eclipse, Web-based applications, Java Web start or any other form. The less the configuration is, the more easy the tools become.

Installation				
	Umple	RSA	Papyrus	ArgoUML
Stand-alone UI	**X**	√	**X**	√
Eclipse Plugin	√	√	√	√
Commandline	√	**X**	**X**	**X**
Web-based	√	**X**	**X**	√

Figure 26: Installation

4.5.2 Flexibility

We consider the availability across different platforms and the pluggability into other tools.

When we look at this from this perspective, a tool that works on different platforms could be more widely accepted. The following tables show the availability of these tools on different platform. All these tools are available on Windows, Mac OS and Linux. However, there is a big gab in the size of installation between these three and IBM RSA. ArgoUML and Umple can be provided as an online version, Umple

Online is available instantly without installation, however ArgoUML requires installation although it is based on Java web-start. Papyrus is only available as an Eclipse plugin.

Installation				
	Umple	RSA	Papyrus	ArgoUML
Stand-alone UI	**X**	√	**X**	√
Eclipse Plugin	√	√	√	√
Commandline	√	**X**	**X**	**X**
Web-based	√	√	**X**	√
Overall	**3**	**3**	**1**	**3**

Figure 27: Comparison of installation options

4.5.3 Readability

In this criteria we try to measure the readability of the generated code and the ability to review the code.

Lets look at the graphs generated from SourceMonitor and see compare the code in terms of Line of Code (LOC) and number of comments. These two things can extremely affect the readability of the code when they are managed well.

	Umple	RSA	Papyrus	ArgoUML
LOC	1766	1088	557	235
Comments	19%	41%	74%	18%
Overall	3	2	3	3

Figure 28: Comparison of LOC and comment in terms of readability

Looking at the metrics generated from airline example, we can make several observations on that in terms of readability. For a system with this size, having

generated 74% of 557 line of code as comment is good if the comments are efficiently injected. Enough comments to describe each block of code in the system can increase the readability of the code. However, generating duplicated comments can be very disturbing in terms of readability of code; IBM RSA has issue with duplicated comments. For instance in the header file, we can see that the tools generate a comment indicating the name of the source transformation file before each declaration. See the code below:

```
//@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
	Booking();

	//@generated "UML to C++
(com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
	~Booking();

	//get seatNumber
	//@generated "UML to C++
(com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
	int & get_seatNumber();

	//set seatNumber
	//@generated "UML to C++
(com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
	void set_seatNumber(int & seatNumber);

	//get specifiedFlight
	//@generated "UML to C++
(com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
	SpecifiedFlight * & get_specifiedFlight();

	//set specifiedFlight
	//@generated "UML to C++
(com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
	void set_specifiedFlight(SpecifiedFlight * & specifiedFlight);
```

The tool could simply group the bulk of code generated from the same source rather than causing this redundancy at

“//@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)" “

Umple is following the same idea of in-code documentation by injecting comments to describe declarations within header class. Umple also allows transformation of comment from model-to-code (since Umple model is textual) which enhance the readability in general. Umple online, compared to other tool, has a very usable interface with color syntax. It makes it ideal for educational purposes is in fact a very readable version in Umple online since it has a dual perspective where user can edit code and model while changes are kept in-sync between both.

Readability can be looked at in two ways, readability of the generated code and readability of model prior to code generation. In Umple's case, readability of the model/code used as input to code generation is beyond the coverage of this thesis since that is up to the individual developer or the designers of Umple syntax; we are only interested here in the readability of the generated code. The following is a snippet of class A with an association to a class B.

```
/* EXPERIMENTAL CODE - NON COMPILEABLE VERSION OF C++ */
/*PLEASE DO NOT EDIT THIS CODE*/
/*This code was generated using the UMPLE 1.17.0.2716 modeling language!*/

#ifndef A_H_
#define A_H_
#include<algorithm>
#include <string>
using namespace std;
class B;

class A
{
  //----------------------
  // Attributes for header file
  //----------------------
  private:

  //----------------------
  // MEMBER VARIABLES
  //----------------------

  //A Attributes
  string name;

  //A Associations
  B* b;

  //----------------------
  // Constructor
  //----------------------
  public:

 A(const string & aName);
 A(const A & a);

  //----------------------
  // Operator =
  //----------------------

 A operator=(const A & a);

  //----------------------
  // INTERFACE
  //----------------------

  bool setName(const string & aName);
  string getName() const;

  B* getB();
  bool setB(B* aB);
  //----------------------
  // Destructor
  //----------------------
```

4.5.4 Embedding: The possibility to merge with additional code.

Is there a need to reverse engineer the code after code generation? And if this feature is supported, how well can it be done?

Umple deliberately doesn't allow for round-tripping. On the other hand, IBM RSA does allow for round-tripping for C++. This can be configured within the transformation file. ArgoUML and Papyrus currently don't support round tripping. Umple, however, has a code-to-model transformation currently being developed called Umplification [39]. This differs from round-tripping in several ways. First, Umplification is applied on particular code once while round-tripping could be applied more. The idea is as soon as we Umplify the system there is no need to go back to the code. We can not judge at the meantime how good the Umplification process is since it is beyond the coverage of this thesis and also it is still under development by other members.

4.5.5 Documentation:

This part is actually divided into two:

Documentation on how to use the tool; which includes the documentation on how to generate C++ code. Umple in this matter comes in a very intuitive and usable version 'Umple Online', which was developed to allow for demonstration and educational purposes. It already comes with lot of examples. Also, Umple allows for warnings at the modeling level, which redirect the user to online documentation for each issue encontered.

Umple has an online user manual and additional documentation on the Umple wiki page [38]; it contains a list of tutorials on how to install the tool and it covers the use of important features. It also includes a developer guide that helps new developers understand the tool and how to get their hands working. Still, IBM RSA has one advantage in terms of tool documentation. Automated-Documentation, which covers the documentation of the API and the possibility to use third-party documenter such as Doxygen [40] or CppDoc [41]. Currently Umple does not

support this for C++ but we are considering the automatic generation of API documentation using of the previously mentioned tools.

4.6 Memory Management

Memory management is a big issue in C++, a tool that generates C++ code must take the following into account:

Although we are clearing the memory allocated for objects, some memory leaks are detected. We have used a tool called Valgrind to find memory leaks on a small set of examples. However, we could come up with a checking algorithm that checks whether a candidate object for deletion is being used by other object or not. This will extremely facilitate the memory leaks issue since it will allow us to deleted unused object after removing them from containers. ArgoUML and Papyrus have no mechanism to handle memory leaks. IBM RSA also doesn't incorporate any mechanism for handling memory leaks.

Chapter 5 Conclusions

The plan of this thesis was to implement a C++ code generator that is very similar to the Java code generator in Umple in terms of semantic completeness and coverage.

In conclusion, the contributions of this thesis can be summarized as the following:

- We developed a C++ code generation capability in Umple. This work allows C++ developers to write C++ within Umple in a model-driven manner. This should allow abstraction of details of UML elements in C++, which brings a lot of benefits to the development environment and process. This generator covers many aspects of Umple. However, not all features of Umple had been implemented in this C++ generator since Umple is in continual development, there were several features those had not been implemented in C++ yet, such as sorting associations, as well as file and console tracers. The contribution is not over however; more features will be added to this work until it completely aligned with other languages in Umple. Also, there are several bugs to be fixed. This code generator supports the following:
    - Associations with respect to the following:
        - Multiplicity bounding
        - Referential integrity
        - API to support associations
    - State Machines that supports:
        - Basic state machines
        - Nested state machines
    - Attributes:
        - Declaring attributes
        - Design patterns: Immutable
    - Generalization:
        - Support for generalization
        - Support for multiple inheritance

- We wrote an LTTng tracer to work with MOTL (model-oriented tracing language) that should allows tracing of C++ application statically. The work involved creating an LTTng tracepoint generator and also handling the trace calls within the application.
- We have also demonstrated our agile approach toward the development of this work.

The responses to our research questions are as follows:

- **Changes needed to Umple**: The main modifications to Umple's general code-generation capabilities are to enable a .cpp and a .h file to be created for each class, rather than a single file for each class. Appropriate mechanisms to ensure including of the correct .h files also had to be generated. We had to also pay special attention to C++ specific issues such as pointers vs. references, use of the standard template library, copy constructors and so on.

  We did find that it was possible to create a C++ code generator for C++ in Umple. No extensions of Umple's syntax or metamodel were needed, other than adding 'Cpp' as a valid generation target, and adding the C++ generation module we developed.

- **Quality of generated code**: A lot of the work in the development of this thesis went into ensuring that the generated code followed good C++ style and was readable.

- **Comparison to other tools**: We can learn from the result of the evaluation of the comparison of the candidate tools against Umple that Umple provides more powerful API and has more power in terms of semantics of UML. Hence, Umple makes a great environment for model-driven development in C++. Although a lot of enhancement is required to improve efficiency and performance of code, Umple still tends to have many advantages in comparison with IBM RSA, Papyrus and ArgoUML.

Umple makes a great environment to implement the C++ generator. Although some refactoring was required to put this into action, the agile development approach in Umple facilitates the development process and allows for TDD. Also, considering the fact that Umple is an open-source tool, implementing the C++ generator within Umple was a good decision.

5.1 Future Work:

Umple has evolved since we began working on the Umple C++ code generator, some features were introduced in Java those were not covered in C++ in addition to other features. The following list of the features discusses the features we weren't able to complete by the time of submitting this thesis:

- Concurrent state machines: We were not able to complete this feature due to compilation failure when defining a Null state in more than one state machine. The C++ compiler fails to compile when the same enumeration entry is defined in multiple enumerations. For instance, if we consider the following concurrent state machines example in Umple:

```
class A {
  sm {
    s0 {
      s1 {
      }
      ||
      s2 {
      }
    }
  }
}
```

This will generate the following enumerations for the nested and concurrent states:

```
//A State Machines
  enum Sm { s0 };
  enum SmS1 { Null, s1 };
  enum SmS2 { Null, s2 };
```

One possible way to solve this duplicate Null issue is to include the name of the state machine before each null. So, for instance, the previous declarations should look like the following:

```
//A State Machines
  enum Sm { s0 };
  enum SmS1 { SmS1Null, s1 };
  enum SmS2 { SmS2Null, s2 };
```

However, this requires us to make modifications to all places where a Null entry is being called or used in the code. This issue is currently being worked on.

- State Machine Actions and Do Activities: These features are currently not supported in the Umple C++ generator created for this thesis. Do activities allow the state to respond to other events while performing a lengthy activity. The implementation requires the code to be multi-threaded. However, multithreading in C++ is currently not supported by the generator. The implementation of multithreading is discussed below. Until we support multithreading, this issue will remain unfixed.
- Multithreading: A mechanism for multithreading in our code generator is to be developed in the future. A 'thread' class was only recently introduced in C++11. Multithreading is required in order to incorporate several features in Umple. There are several possible ways to make the generated code multithread safe. One possible way would be to write the implementation of thread class from scratch and generate its artifacts whenever required. Another way would be to make use of third-party libraries that provides an implementation for managing threads. In this situation one may consider one of the following libraries: the POSIX Threads (pthread) (IEEE Std 1003.1c-1995), which defines an API to create and manage threads in C++. Also Boost provides a set of libraries to handle threads. However, this option requires a third party library and part of Umple's objectives is to avoid the use of third-party libraries. There has not been a decision on how to incorporate this into the Umple C++ code generator. Therefore, it will be fixed in the future.
- We need to write more examples and test cases. One possible way would be to try to manually rewrite existing C++ application using Umple and study the manually umplified version deeper and try to find points of interest and possible areas of improvement.
- In terms of tracing C++ applications with LTTng, there are several potential areas of improvement. For instance, currently tracing state machines has issues.

We need to fix tracing state machines and allow generating for tracepoints for entry and exit of states rather than combining both into one tracepoint. Also we need to write more tracing examples and focus on the use of Umple with the LTTng tracer. This should allow LTTng users to use Umple C++ in real time examples.

- Developing an algorithm to manage the use of objects between each other. We should take advantage of the fact that we are working in a model-driven environment and models, in this context, can tell us important information regarding the system. We can find out hat object is using what. This should allow us to understand the system more and handle memory management better without the use of external libraries.
- Generating API documentation using tools like CppDoc or Doxygen can be very helpful and should enhance the user experience. This is already done in Java, we ought to use a similar mechanism.
- There are many other code generators in Umple, and another group is creating a separate C++ code generator (as yet unpublished). It will be important to analyse the differences between the two generators.

References

[1] Cruise. " Umple Online,", accessed 2013, http://try.umple.org/.

[2] Forward, A. and Lethbridge, T. C. " Umple Language", accessed 2009, http://try.umple.org.

[3] Anonymous " ArgoUML Modeling Tool.", accessed 2009, http://argouml.tigris.org/.

[4] Forward, A. " Computer Science PhD Thesis, Appendices, and Supplementary Material", accessed 2011, http://www.site.uottawa.ca/~tcl/gradtheses/aforwardphd/.

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

[6] IBM. " IBM Rational Software Architect Modeling Tool", accessed 2009, http://www-01.ibm.com/software/awdtools/architect/swarchitect/.

[7] "MDT Papyrus, the Eclipse Project".

[8] Xiangye Ji, Jun Han and Yongwang Zhao. "A Code Generation Toolkit for C++ Web Services Development," in *Intelligent System Design and Engineering Applications (ISDEA), 2013 Third International Conference On,* 2013. pp. 17-21.

[9] de Souza, C. R. B. and Bentolila, D. L. M. "Automatic Evaluation of API Usability using Complexity Metrics and Visualizations," in *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference On,* 2009. pp. 299-302.

[10] Pellegrini, S., Prodan, R. and Fahringer, T. "A Lightweight C++ Interface to MPI," in *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference On,* 2012. pp. 3-10.

[11] Rudahl, K. T. and Goldin, S. E. "Perverse UML for Generating Web Applications using YAMDAT," in *TENCON 2010 - 2010 IEEE Region 10 Conference,* 2010. pp. 1071-1076.

[12] Schade, A. "Automatic Generation of Bridging Code for Accessing C++ from Java," in *Technology of Object-Oriented Languages and Systems, 1997. TOOLS 25, Proceedings,* 1997. pp. 165-180.

[13] Wolf, W. "A Practical Comparison of Two Object-Oriented Languages". 1989. *Software, IEEE,* vol 6, pp. 61-68.

[14] Xinming Tan, Wang, Y. and Ngolah, C. F. "Design and Implementation of an Automatic RTPA Code Generator," in *Electrical and Computer Engineering, 2006. CCECE '06. Canadian Conference On,* 2006. pp. 434-437.

[15] Forward, A., Badreddin, O. and Lethbridge, T. C. "Umple: Towards Combining Model Driven with Prototype Driven System Development," in *IEEE International Symposium on Rapid System Prototyping (RSP),* 2010.

[16] Aljamaan, H. and Lethbridge, T. C. "Towards Tracing at the Model Level," in *19th Working Conference on Reverse Engineering (WCRE), 2012,* 2012. pp. 495-498.

[17] The Eclipse Foundation. " Eclipse Modeling - M2T - Home (Jet Project)", accessed 2009, http://www.eclipse.org/modeling/m2t/?project=jet.

[18] IBM. " Rational Software Architect Design Manager", accessed 2013, http://www-01.ibm.com/software/rational/products/swarchitect/designmanager/.

[19] CollabNet. accessed 2013, http://argouml.tigris.org.

[20] TIOBE Software. accessed 2012, http://www.tiobe.com.

[21] LTTng. " Linux Trace Toolkit - Next Generation", accessed 2013, http://www.lttng.org.

[22] ITU, SDL (Specification and Description Language), accessed 2013, http://www.itu.int/rec/T-REC-Z.100/en.

[23] Sendall, S. and Kozaczynski, W. "Model Transformation: The Heart and Soul of Model-Driven Software Development". 2003. *IEEE Software,* vol 20, pp. 42-45.

[24] Markus Völter (Author), Thomas Stahl (Author), Jorn Bettin (Author), Arno Haase (Author), Simon Helsen (Author), Krzysztof Czarnecki. *Model-Driven Software Development*. July 2006.

[25] Beck, K. *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 2002.

[26] Web Service Definition Language (WSDL). www.w3.org/TR/wsd.

[27] Budinsky, F., Brodsky, S. A. and Merks, E. *Eclipse Modeling Framework*. Pearson Education, 2003.

[28] Scott Meyers, A. A. "
C++ and the Perils of Double-Checked Locking
". 2004.

[29] Scott Meyers, "Chapter 6. inheritance and object-oriented design, item 32 ," in *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Anonymous 2005,

[30] Binkley, D., Davis, M., Lawrie, D. and Morrell, C. "To Camelcase Or Under_score," in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference On,* 2009. pp. 158-167.

[31] Tangible Software Solution. " Java to C++ Converter", accessed 2013, http://www.tangiblesoftwaresolutions.com.

[32] J2C. " Java to C++ Converter", accessed 2013, http://code.google.com/a/eclipselabs.org/p/j2c/.

[33] Desnoyers, M. and Dagenais, M. "LTTng: Tracing Across Execution Layers, from the Hypervisor to User-Space," in *Linux Symposium 2008,* 2008. pp. 101.

[34] LTTng. " LTTng Tracepoint Generator, Lttng-Gen-Tp", accessed 2013, http://lttng.org/files/doc/man-pages/man1/lttng-gen-tp.1.html.

[35] Aggarwal, K., Singh, Y., Kaur, A. and Malhotra, R. "Software Design Metrics for Object-Oriented Software". 2007. *J. Object Technol.,* vol 6, pp. 121-138.

[36] Campwood Software. " SoucrceMonitor Version 3.4", accessed 2013, http://www.campwoodsw.com/sourcemonitor.html.

[37] The Knowledge Management Lab, University of Toronto. " GRL - Goal-Oriented Requirement Language", accessed 2009, http://www.cs.toronto.edu/km/GRL/.

[38] Lethbridge, T. C., Forward, A. and Badreddin, O. "Umple Google Code Project". 2012. Available: code.umple.org

[39] Garzon, M. and Lethbridge, T. C. "Exploring how to Develop Transformations and Tools for Automated Umplification," in *Reverse Engineering (WCRE), 2012 19th Working Conference On,* 2012. pp. 491-494.

[40] Doxygen. accessed 2013, http://www.stack.nl/~dimitri/doxygen/.

[41] Richard Feit. " CppDoc", accessed 2013, http://www.cppdoc.com.

Appendix: Generated Code Examples

The following show examples of code generated from the work in this thesis and by the comparator systems from the airline system (Airline.ump). Only ‘Airline.h’, ‘Airline.cpp’, ‘Booking.h’ and ‘Booking.cpp’ are included. Excessive numbers of blank lines have been supressed

A1: ArgoUML Airline Example

Airline.cpp

```
1   #include "Airline.h"
2
3       /* {src_lang=cpp}*/
4
```

Airline.h

```
1   #ifndef Airline_h
2   #define Airline_h
3
4   #include <vector>
5
6   class Person;
7   class RegularFlight;
8
9   class Airline {
10      /* {src_lang=cpp}*/
11
12
13   public:
14
15      /**
16       * @element-type Person
17       */
18      std::vector< Person* > myPerson;
19
20      /**
21       * @element-type RegularFlight
22       */
23      std::vector< RegularFlight* >
    myRegularFlight;
24  };
25
26  #endif // Airline_h
```

Booking.cpp

```
1   #include "Booking.h"
2
3       /* {src_lang=cpp}*/
```

Booking.h

```
1   #ifndef Booking_h
2   #define Booking_h
3
4   class SpecificFlight;
5   class passengerRole;
6
7   class Booking {
8       /* {src_lang=cpp}*/
9
10
11   public:
12      String seatNumber;
13
14   public:
15
16      SpecificFlight *Booking;
17
18      passengerRole *mypassengerRole;
19  };
20
21  #endif // Booking_h
```

A2: Papyrus Airline System

Airline.h

```
1   /*********************************************************
2    *
3    * Code Generated by Papyrus C++
4    *
5    * CEA LIST
6    *
7    *********************************************************/
8   #ifndef UMPLE_AIRLINESYSTEM_AIRLINE_H
9   #define UMPLE_AIRLINESYSTEM_AIRLINE_H
10
11  /*********************************************************
12              Airline class header
13   *********************************************************/
14
15  /* Owner package header include                          */
16  #include <Umple_AirlineSystem/Pkg_Umple_AirlineSystem.h>
17
18
19  /* Structural includes (inheritance, dependencies...     */
20  #include <Umple_AirlineSystem/Person.h>
21  #include <Umple_AirlineSystem/RegularFlight.h>
22
23
24  /*********************************************************/
25  /**
26   *
27   */
28  class Airline {
29
30
31  /* Public declarations                                   */
32  public:
33
34
35  /* Protected declarations                                */
36  protected:
37
38
39  /* Private declarations                                  */
40  private:
41    /**
42     *
43     */
44    Person* *person;
45    /**
46     *
47     */
48    RegularFlight* *regularFlight;
49
50
51  };
52  /*********************************************************/
```

```
53  /* External declarations (package visibility)             */
54
55
56  /*********************************************************/
57  /* Inline functions                                      */
58
59  /*********************************************************
60              End of Airline class header
61   *********************************************************/
62
63  #endif
```

Airline.cpp

```
1   /*********************************************************
2    *
3    * Code Generated by Papyrus C++
4    *
5    * CEA LIST
6    *
7    *********************************************************/
8   #define UMPLE_AIRLINESYSTEM_AIRLINE_BODY
9
10  /*********************************************************
11              Airline class body
12   *********************************************************/
13
14  /* Header include                                        */
15  #include <Umple_AirlineSystem/Airline.h>
16
17  /* Include from CppInclude declaration                   */
18
19
20  /*********************************************************
21              End of Airline class body
22   *********************************************************/
23  ;
```

Booking.h

```
1   /*********************************************************
2    *
3    * Code Generated by Papyrus C++
4    *
5    * CEA LIST
6    *
7    *********************************************************/
8   #ifndef UMPLE_AIRLINESYSTEM_BOOKING_H
9   #define UMPLE_AIRLINESYSTEM_BOOKING_H
10
11  /*********************************************************
12              Booking class header
13   *********************************************************/
```

```
14
15  /* Owner package header include                          */
16  #include <Umple_AirlineSystem/Pkg_Umple_AirlineSystem.h>
17
18
19  /* Structural includes (inheritance, dependencies...      */
20  #include <Umple_AirlineSystem/SpecifiedFlight.h>
21  #include <Umple_AirlineSystem/PassengerRole.h>
22
23
24  /*************************************************************/
25  /**
26   *
27   */
28  class Booking {
29
30
31  /* Public declarations                                   */
32  public:
33
34
35  /* Protected declarations                                */
36  protected:
37
38
39  /* Private declarations                                  */
40  private:
41    /**
42     *
43     */
44    null seatNumber;
45    /**
46     *
47     */
48    SpecifiedFlight *specifiedFlight;
49    /**
50     *
51     */
52    PassengerRole *passengerRole;
53
54
55  };
56  /*************************************************************/
57  /* External declarations (package visibility)            */
58
59
60  /*************************************************************/
61  /* Inline functions                                      */
62
63  /*************************************************************
64              End of Booking class header
65   *************************************************************/
66
67  #endif
```

Booking.cpp

```
1   /*********************************************************
2    *
3    * Code Generated by Papyrus C++
4    *
5    * CEA LIST
6    *
7    *********************************************************/
8   #define UMPLE_AIRLINESYSTEM_BOOKING_BODY
9
10  /*********************************************************
11              Booking class body
12   *********************************************************/
13
14  /* Header include                                  */
15  #include <Umple_AirlineSystem/Booking.h>
16
17  /* Include from CppInclude declaration             */
18
19
20  /*********************************************************
21              End of Booking class body
22   *********************************************************/
23  ;
```

A3: IBM RSA Airline System

Airline.h

```
1   #ifndef AIRLINE_H
2   #define AIRLINE_H
3   //Begin section for file Airline.h
4   //TODO: Add definitions that you want preserved
5   //End section for file Airline.h
6
7   class Person; //Dependency Generated Source:Airline Target:Person
8
9   class RegularFlight; //Dependency Generated Source:Airline Target:RegularFlight
10
11
12  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
13  class Airline
14  {
15
16     //Begin section for Airline
17     //TODO: Add attributes that you want preserved
18     //End section for Airline
19
20     private:
21
22        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
23        Person * person;
24
25
26        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
27        RegularFlight * regularFlight;
28
29
30     public:
31
32        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
33        Airline();
34
35
36        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
37        ~Airline();
38
39
40        //get person
41        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
42        Person * & get_person();
43
44        //set person
45        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
46        void set_person(Person * & person);
47
48
49        //get regularFlight
50        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
51        RegularFlight * & get_regularFlight();
52
```

```
53
54          //set regularFlight
55          //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
56          void set_regularFlight(RegularFlight * & regularFlight);
57
58
59  };  //end class Airline
60
61
62  #endif
```

Airline.cpp

```
1   #include "Airline.h"
2   //Begin section for file Airline.cpp
3   //TODO: Add definitions that you want preserved
4   //End section for file Airline.cpp
5
6   //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
7   Airline::Airline()
8   {
9      //TODO Auto-generated method stub
10  }
11  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
12  Airline::~Airline()
13  {
14     //TODO Auto-generated method stub
15  }
16  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
17  Person * & Airline::get_person()
18  {
19     //TODO Auto-generated method stub
20     return person;
21  }
22  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
23  void Airline::set_person(Person * & person)
24  {
25     //TODO Auto-generated method stub
26  }
27  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
28  RegularFlight * & Airline::get_regularFlight()
29  {
30     //TODO Auto-generated method stub
31     return regularFlight;
32  }
33  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
34  void Airline::set_regularFlight(RegularFlight * & regularFlight)
35  {
36     //TODO Auto-generated method stub
37  }
```

Booking.h

```
1   #ifndef BOOKING_H
2   #define BOOKING_H
3   //Begin section for file Booking.h
4   //TODO: Add definitions that you want preserved
5   //End section for file Booking.h
6
7   class SpecifiedFlight; //Dependency Generated Source:Booking Target:SpecifiedFlight
8
9   class PassengerRole; //Dependency Generated Source:Booking Target:PassengerRole
10
11
12  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
13  class Booking
14  {
15
16     //Begin section for Booking
17     //TODO: Add attributes that you want preserved
18     //End section for Booking
19
20     private:
21
22        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
23        int seatNumber;
24
25
26        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
27        SpecifiedFlight * specifiedFlight;
28
29
30        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
31        PassengerRole * passengerRole;
32
33
34     public:
35
36        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
37        Booking();
38
39
40        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
41        ~Booking();
42
43
44        //get seatNumber
45        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
46        int & get_seatNumber();
47
48
49        //set seatNumber
50        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
51        void set_seatNumber(int & seatNumber);
52
53
54        //get specifiedFlight
55        //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
56        SpecifiedFlight * & get_specifiedFlight();
```

```
57
58
59          //set specifiedFlight
60          //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
61          void set_specifiedFlight(SpecifiedFlight * & specifiedFlight);
62
63
64          //get passengerRole
65          //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
66          PassengerRole * & get_passengerRole();
67
68
69          //set passengerRole
70          //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
71          void set_passengerRole(PassengerRole * & passengerRole);
72
73
74  };  //end class Booking
75
76
77  #endif
```

Booking.cpp

```
1   #include "Booking.h"
2   //Begin section for file Booking.cpp
3   //TODO: Add definitions that you want preserved
4   //End section for file Booking.cpp
5
6
7   //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
8   Booking::Booking()
9   {
10      //TODO Auto-generated method stub
11  }
12  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
13  Booking::~Booking()
14  {
15      //TODO Auto-generated method stub
16  }
17  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
18  int & Booking::get_seatNumber()
19  {
20      //TODO Auto-generated method stub
21      return seatNumber;
22  }
23  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
24  void Booking::set_seatNumber(int & seatNumber)
25  {
26      //TODO Auto-generated method stub
27      this->seatNumber = seatNumber;
28  }
29  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
30  SpecifiedFlight * & Booking::get_specifiedFlight()
```

```
31  {
32      //TODO Auto-generated method stub
33      return specifiedFlight;
34  }
35  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
36  void Booking::set_specifiedFlight(SpecifiedFlight * & specifiedFlight)
37  {
38      //TODO Auto-generated method stub
39  }
40  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
41  PassengerRole * & Booking::get_passengerRole()
42  {
43      //TODO Auto-generated method stub
44      return passengerRole;
45  }
46  //@generated "UML to C++ (com.ibm.xtools.transform.uml2.cpp.CPPTransformation)"
47  void Booking::set_passengerRole(PassengerRole * & passengerRole)
48  {
49      //TODO Auto-generated method stub
50  }
```

A4: Umple Airline System (code generated as a result of this thesis work)

Airline.h

```
1   /* EXPERIMENTAL CODE - SIMPLE C++ */
2   /*PLEASE DO NOT EDIT THIS CODE*/
3   /*This code was generated using the UMPLE 1.17.0.2938 modeling language!*/
4
5   #ifndef AIRLINE_H_
6   #define AIRLINE_H_
7   #include <vector>
8   #include<algorithm>
9   #include <string>
10  using namespace std;
11  class RegularFlight;
12  class Person;
13
14  class Airline
15  {
16    //------------------------
17    // Attributes for header file
18    //------------------------
19    private:
20
21
22    //------------------------
23    // MEMBER VARIABLES
24    //------------------------
25
26    //Airline Associations
27     vector<RegularFlight*> regularFlights;
28     vector<Person*> persons;
29
30
31
32    public:
33
34
35    //------------------------
36    // Constructor
37    //------------------------
38  Airline();
39
40
41    //------------------------
42    // Copy Constructor
43    //------------------------
44
45   Airline(const Airline & airline);
46
47    //------------------------
48    // Operator =
49    //------------------------
50
51   Airline operator=(const Airline & airline);
52
```

```
53
54    //------------------------
55    // INTERFACE
56    //------------------------
57
58    RegularFlight* getRegularFlight(int index);
59    vector<RegularFlight*> getRegularFlights();
60    int numberOfRegularFlights();
61    bool hasRegularFlights();
62    int indexOfRegularFlight(RegularFlight* aRegularFlight);
63    Person* getPerson(int index);
64    vector<Person*> getPersons();
65    int numberOfPersons();
66    bool hasPersons();
67    int indexOfPerson(Person* aPerson);
68    static int minimumNumberOfRegularFlights();
69
70    RegularFlight addRegularFlight(const Time & aTime);
71    bool addRegularFlight(RegularFlight* aRegularFlight);
72    bool removeRegularFlight(RegularFlight* aRegularFlight);
73    static int minimumNumberOfPersons();
74
75    Person addPerson(const String & aName, const int & aIdNumber);
76    bool addPerson(Person* aPerson);
77    bool removePerson(Person* aPerson);
78    //------------------------
79    // Destructor
80    //------------------------
81   virtual ~Airline();
82
83   };
84
85   #endif
```

Airline.cpp

```
1    /* EXPERIMENTAL CODE - SIMPLE C++ */
2    /*PLEASE DO NOT EDIT THIS CODE*/
3    /*This code was generated using the UMPLE 1.17.0.2938 modeling language!*/
4
5    #include "Airline.h"
6    #include "RegularFlight.h"
7    #include "Person.h"
8
9
10    //------------------------
11    // CONSTRUCTOR
12    //------------------------
13
14    Airline::Airline()
15    {
16    }
17
18    //------------------------
```

```
19  // COPY CONSTRUCTOR
20  //------------------------
21
22  Airline::Airline(const Airline & airline)
23  { }
24
25  //------------------------
26  // Operator =
27  //------------------------
28
29  Airline Airline::operator=(const Airline & airline)
30  { }
31
32  //------------------------
33  // INTERFACE
34  //------------------------
35
36  RegularFlight* Airline::getRegularFlight(int index)
37  {
38    RegularFlight* aRegularFlight = regularFlights[index];
39    return aRegularFlight;
40  }
41
42  vector<RegularFlight*> Airline::getRegularFlights()
43  {
44    vector<RegularFlight*> newRegularFlights = regularFlights;
45    return newRegularFlights;
46  }
47
48  int Airline::numberOfRegularFlights()
49  {
50    int number = regularFlights.size();
51    return number;
52  }
53
54  bool Airline::hasRegularFlights()
55  {
56    bool has = regularFlights.size() > 0;
57    return has;
58  }
59
60  int Airline::indexOfRegularFlight(RegularFlight* aRegularFlight)
61  {
62    int index = find(regularFlights.begin(), regularFlights.end(), aRegularFlight) - regularFlights.begin();
63    return index;
64  }
65
66  Person* Airline::getPerson(int index)
67  {
68    Person* aPerson = persons[index];
69    return aPerson;
70  }
71
72  vector<Person*> Airline::getPersons()
73  {
74    vector<Person*> newPersons = persons;
```

```
75    return newPersons;
76  }
77
78  int Airline::numberOfPersons()
79  {
80    int number = persons.size();
81    return number;
82  }
83
84  bool Airline::hasPersons()
85  {
86    bool has = persons.size() > 0;
87    return has;
88  }
89
90  int Airline::indexOfPerson(Person* aPerson)
91  {
92    int index = find(persons.begin(), persons.end(), aPerson) - persons.begin();
93    return index;
94  }
95
96  static int minimumNumberOfRegularFlights()
97  {
98    return 0;
99  }
100
101 RegularFlight Airline::addRegularFlight(const Time & aTime)
102 {
103   return new RegularFlight(aTime, this);
104 }
105
106 bool Airline::addRegularFlight(RegularFlight* aRegularFlight)
107 {
108   bool wasAdded = false;
109   if (find(regularFlights.begin(),regularFlights.end(),aRegularFlight) != regularFlights.end()) { return false; }
110   Airline* existingAirline = aRegularFlight->getAirline();
111   bool isNewAirline = (existingAirline != NULL && this!=existingAirline);
112   if (isNewAirline)
113   {
114     aRegularFlight->setAirline(this);
115   }
116   else
117   {
118     regularFlights.push_back(aRegularFlight);
119   }
120   wasAdded = true;
121   return wasAdded;
122 }
123
124 bool Airline::removeRegularFlight(RegularFlight* aRegularFlight)
125 {
126   bool wasRemoved = false;
127     int index = find(regularFlights.begin(), regularFlights.end(), aRegularFlight) - regularFlights.begin();
128   //Unable to remove aRegularFlight, as it must always have a airline
129   if (this!=aRegularFlight->getAirline())
130   {
```

```
131     regularFlights.erase(find(regularFlights.begin(),regularFlights.end(),aRegularFlight));
132       delete regularFlights[index];
133     wasRemoved = true;
134    }
135    return wasRemoved;
136   }
137   static int minimumNumberOfPersons()
138   {
139    return 0;
140   }
141
142   Person Airline::addPerson(const String & aName, const int & aIdNumber)
143   {
144    return new Person(aName, aIdNumber, this);
145   }
146
147   bool Airline::addPerson(Person* aPerson)
148   {
149    bool wasAdded = false;
150    if (find(persons.begin(),persons.end(),aPerson) != persons.end()) { return false; }
151    Airline* existingAirline = aPerson->getAirline();
152    bool isNewAirline = (existingAirline != NULL && this!=existingAirline);
153    if (isNewAirline)
154    {
155     aPerson->setAirline(this);
156    }
157    else
158    {
159     persons.push_back(aPerson);
160    }
161    wasAdded = true;
162    return wasAdded;
163   }
164
165   bool Airline::removePerson(Person* aPerson)
166   {
167    bool wasRemoved = false;
168     int index = find(persons.begin(), persons.end(), aPerson) - persons.begin();
169    //Unable to remove aPerson, as it must always have a airline
170    if (this!=aPerson->getAirline())
171    {
172     persons.erase(find(persons.begin(),persons.end(),aPerson));
173       delete persons[index];
174     wasRemoved = true;
175    }
176    return wasRemoved;
177   }
178
179   //------------------------
180   // DESTRUCTOR
181   //------------------------
182
183  Airline::~Airline()
184   {
185    for(i =0; sizeof(regularFlights); i++)
186    {
```

```
187       delete regularFlights[i];
188     }
189     for(i =0; sizeof(persons); i++)
190     {
191       delete persons[i];
192     }
193   }
```

Booking.h

```
1    /* EXPERIMENTAL CODE - SIMPLE C++ */
2    /*PLEASE DO NOT EDIT THIS CODE*/
3    /*This code was generated using the UMPLE 1.17.0.2938 modeling language!*/
4
5    #ifndef BOOKING_H_
6    #define BOOKING_H_
7    #include<algorithm>
8    #include <string>
9    using namespace std;
10   class SpecificFlight;
11   class PassengerRole;
12
13   class Booking
14   {
15     //------------------------
16     // Attributes for header file
17     //------------------------
18     private:
19
20
21     //------------------------
22     // MEMBER VARIABLES
23     //------------------------
24
25     //Booking Attributes
26     string seatNumber;
27
28     //Booking Associations
29     SpecificFlight* specificFlight;
30     PassengerRole* passengerRole;
31
32
33
34     public:
35
36
37     //------------------------
38     // Constructor
39     //------------------------
40   Booking(const String & aSeatNumber, SpecificFlight aSpecificFlight, PassengerRole aPassengerRole);
41
42
43     //------------------------
44     // Copy Constructor
```

```
45    //------------------------
46
47   Booking(const Booking & booking);
48
49    //------------------------
50    // Operator =
51    //------------------------
52
53   Booking operator=(const Booking & booking);
54
55
56    //------------------------
57    // INTERFACE
58    //------------------------
59
60    bool setSeatNumber(const string & aSeatNumber);
61    string getSeatNumber() const;
62
63    SpecificFlight* getSpecificFlight();
64    PassengerRole* getPassengerRole();
65    bool setSpecificFlight(SpecificFlight* aSpecificFlight);
66    bool setPassengerRole(PassengerRole* aPassengerRole);
67    //------------------------
68    // Destructor
69    //------------------------
70   virtual ~Booking();
71
72   };
73
74   #endif
```

Booking.cpp

```
1    /* EXPERIMENTAL CODE - SIMPLE C++ */
2    /*PLEASE DO NOT EDIT THIS CODE*/
3    /*This code was generated using the UMPLE 1.17.0.2938 modeling language!*/
4
5    #include "Booking.h"
6    #include "SpecificFlight.h"
7    #include "PassengerRole.h"
8
9
10    //------------------------
11    // CONSTRUCTOR
12    //------------------------
13
14    Booking::Booking(const String & aSeatNumber, SpecificFlight aSpecificFlight, PassengerRole
      aPassengerRole)
15     {
16       seatNumber = aSeatNumber;
17       bool didAddSpecificFlight = setSpecificFlight(aSpecificFlight);
18       if (!didAddSpecificFlight)
19       {
20         throw new RuntimeException("Unable to create Booking due to specificFlight");
```

```
21      }
22      bool didAddPassengerRole = setPassengerRole(aPassengerRole);
23      if (!didAddPassengerRole)
24      {
25        throw new RuntimeException("Unable to create booking due to passengerRole");
26      }
27    }
28
29    //------------------------
30    // COPY CONSTRUCTOR
31    //------------------------
32
33    Booking::Booking(const Booking & booking)
34    {
35      this->seatNumber = booking.seatNumber;
36    }
37
38    //------------------------
39    // Operator =
40    //------------------------
41
42    Booking Booking::operator=(const Booking & booking)
43    {
44      this->seatNumber = booking.seatNumber;
45    }
46
47    //------------------------
48    // INTERFACE
49    //------------------------
50
51    bool Booking::setSeatNumber(const string & aSeatNumber)
52    {
53      bool wasSet = false;
54      seatNumber = aSeatNumber;
55      wasSet = true;
56      return wasSet;
57    }
58
59    string Booking::getSeatNumber() const
60    {
61      return seatNumber;
62    }
63
64    SpecificFlight* Booking::getSpecificFlight()
65    {
66      return specificFlight;
67    }
68
69    PassengerRole* Booking::getPassengerRole()
70    {
71      return passengerRole;
72    }
73
74    bool Booking::setSpecificFlight(SpecificFlight* aSpecificFlight)
75    {
76      bool wasSet = false;
```

```
77    if (aSpecificFlight == NULL)
78    {
79      return wasSet;
80    }
81
82    SpecificFlight* existingSpecificFlight = specificFlight;
83    specificFlight = aSpecificFlight;
84    if (existingSpecificFlight != NULL && existingSpecificFlight!=aSpecificFlight)
85    {
86      existingSpecificFlight->removeBooking(this);
87    }
88    specificFlight->addBooking(this);
89    wasSet = true;
90    return wasSet;
91  }
92
93  bool Booking::setPassengerRole(PassengerRole* aPassengerRole)
94  {
95    bool wasSet = false;
96    if (aPassengerRole == NULL)
97    {
98      return wasSet;
99    }
100
101   PassengerRole* existingPassengerRole = passengerRole;
102   passengerRole = aPassengerRole;
103   if (existingPassengerRole != NULL && existingPassengerRole!=aPassengerRole)
104   {
105     existingPassengerRole->removeBooking(this);
106   }
107   passengerRole->addBooking(this);
108   wasSet = true;
109   return wasSet;
110 }
111
112
113 //------------------------
114 // DESTRUCTOR
115 //------------------------
116
117 Booking::~Booking()
118 {
119   SpecificFlight placeholderSpecificFlight = specificFlight;
120   this->specificFlight = NULL;
121   placeholderSpecificFlight->removeBooking(this);
122   PassengerRole placeholderPassengerRole = passengerRole;
123   this->passengerRole = NULL;
124   placeholderPassengerRole->removeBooking(this);
125 }
```