Exploring Textual Modeling using the Umple Language

By

Dusan Brestovansky

Thesis

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

Ottawa-Carleton Institute for Computer Science

University of Ottawa

Ottawa, Ontario, K1N 6N5

Canada

Acknowledgements

This research is supported by IBM Canada and the Centre of Advanced Studies in Ottawa.

I would like to extend a great thank you to the following:

1. Dr. Timothy C. Lethbridge. As my advisor, supervisor and mentor he has taught me the axioms of proper research, software engineering and pursuit of knowledge. His critique and guidance through this process has been extensive and invaluable.
2. The CRuiSE lab composing of Dr. Lethbridge, Andrew Forward and Omar Badreddin. Our meetings on- and off-line have proven to be most educational in many aspects of the research
3. IBM and associated software developers I was able to interact with and learn from throughout the duration of my Master's term.
4. My family and friends who have supported me in many ways through this difficult journey. Thank you to my father Dusan, mom Tatiana, and sister Natasa.

Abstract

The purpose of our research is to explore the advantages and disadvantages of textual modeling in software engineering, as compared to the traditional graphical modeling languages. A cornerstone of our work has been the development of a text-based modeling language called Umple. Umple has a similar syntax to Java, but it has been enhanced with additional modeling constructs (associations, software patterns, etc.). Umple has the ability to produce working systems by providing a translation into existing object-oriented programming languages (such as Java), and it can also be represented diagrammatically in UML.

Graphical modeling languages have been developed and improved over many years of research. These languages allow software designers and architects to view systems from an abstract and high-level point of view. By using graphical modeling languages we abstract away much of the detail of the implementation of a design, and the process of building large and complex systems is simplified.

Programming languages, on the other hand, are used for fine-grained views of the system, incorporating algorithms, conditions and constraints; in addition to the high-level components (such as classes) which could have been specified using a graphical modeling language as described above.

The separation between the high-level and fine-grained abstractions results in Software Modeling and Software Construction as two separate tasks in the development lifecycle. This research explores ways to reduce complexity and increase the quality of software systems, as well as increase the speed with which systems can be generated, maintained or comprehended.

Table of Contents

List of Tables

List of Figures

Abbreviations

API: Application Programming Interface

OO: Object Oriented

RSM: Rational Software Modeler

ANTLR: ANother Tool for Language Recognition

AST: Abstract Syntax Tree

HUTN: Human-usable Textual Notation

UML: Unified Modeling Language

MOF: MetaObject Facility

MDA: Model Driven Architecture

OMG: Object Management Group

1 Introduction

The purpose of this research is to explore the advantages and disadvantages of textual modeling in software engineering, as compared to the traditional graphical modeling tools and languages. As a major part of the work, we developed a text-based modeling language called Umple. Umple can be directly translated to an object oriented programming language (such as Java), and also to the graphical modeling language UML. This research has been performed in collaboration with the IBM CAS Ottawa, therefore benefiting both the academic and industry communities.

1.1 Motivation and Objectives

Graphical modeling languages have been studied and improved over past decades. They allow designers and architects to view systems from a high-level abstract point of view. This approach simplifies designing large and complex systems.

Programming languages, on the other hand, are used for fine-grained views of the system, incorporating algorithms, conditions and constraints, as well as all of the high-level components, such as classes, that can be specified using a graphical modeling language.

This separation between the two levels of abstraction results in two separate tasks in the development lifecycle: Modeling and implementing. The results of these tasks are the system model and the implementation, respectively. Technically, the implementation can be considered a type of model too, but modeling provides design abstractions and programming provides computer abstraction. However, the separation between system and implementation model leads to inevitable differences between them, which may develop over time or may even be present in the first version of the system.

To elaborate on the points made so far, let us introduce a small example; and will also introduce some of the syntax of the Umple language. Our small RaceCar system has only three classes: Vehicle, Car, and Person. Car is a type of Vehicle, and each Car may (or may not) have a driver (and that driver is a Person). Figure 1 is the class diagram of such a simple system (as generated by UmpleRSM).

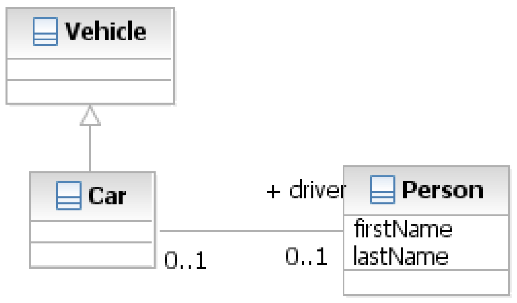


Figure 1: Basic sample system (diagram generated by UmpleRSM).

This very simple system lets us introduce Umple and highlight some of its benefits. To illustrate the power of Umple, Figure 2 shows the code which resulted in this generated UML Class diagram.

```
class Vehicle{
 class Car{
  0..1 -- 0..1 Person driver;
 }
}

class Person{
 firstName;
 lastName;
}
```

Figure 2: Umple code which resulted in the Class diagram in Figure 1.

In the Umple process, one would usually start with the Umple code first and then have the appropriate diagrams generated. However, because we assume the user is familiar with UML, showing the class diagram first makes it easier to draw the links between the two.

Figure 2 shows us the most basic features of the Umple language, but it is enough to illustrate the points made earlier. The Umple code is both concise and expressive. In just nine lines of code we declared classes, inheritance, an association with multiplicities and bi-directionality, class attributes, and optional role name for an association end. In addition to the class diagram, the implementation code is also generated. Class diagrams are the platform independent view of the system, and the implementation code is the platform dependant view, as it contains fine grained details, such as a Façade class and application logic (in this case, over 400 lines of Java code were generated which implement this simple system). Details of all that is generated will be discussed further in the thesis.

Both of these views are kept synchronized as Umple code is updated or modified, not requiring any special attention from the users and letting them focus on the high level modeling using Umple. For example, a change which denotes that a Car has another association to the class Person, which represents the passengers, is only one extra line of code, such as “`1 -- * Person passengers;`”. We will elaborate on each point further in the thesis, but this example illustrates concretely our approach to textual modeling.

Moreover, Umple is an effort to unify programming and modeling, thereby simplifying both. Modeling is simplified through the availability of a simple and easily understandable textual language, and programming is simplified through Umple’s ability to generate system code. This merge of tasks is made possible through adding modeling concepts, such as associations between classes, to a programming language.

We hypothesize that Umple will reduce the time it takes to perform the following software activities:

- Develop – development time should be greatly reduced because the time it takes to write and understand Umple is much less than the time it takes to write and understand the underlying language. As the example shows, it is clear that writing and understanding 9 lines of code takes much less time than to do the same with 400 lines of code. Furthermore, Umple code greatly reduces the number of components one needs to understand to only the high level ones, further facilitating faster development.
- Inspect – because Umple code is both concise and rooted in UML, code inspection should become much easier. This is because there are many fewer lines

to inspect, and the lines represent high-level constructs allowing the user to focus on a few high level components rather than very many low level ones.

- Maintain – Due to the model-code duality of Umple systems, maintenance also should become much easier. In this context, maintenance refers to changing the system and its documentation. This task becomes much faster because Umple regenerates both the system code and the UML diagrams as one task, without requiring the developer to spend time on each.

There are also several disadvantages of graphical representations of design, which have been documented by Dwyer [8]. Of these, the ones which relate to our work are:

- As systems grow in complexity, their diagrammatic representations also grow, sometimes far more than linearly, and generally in two dimensions, making the structure difficult to understand from these diagrams. This problem can be perhaps to some extent addressed using textual representations. The proof of this is that there exist large system implementations that are written entirely in source code without modelling in diagrams, yet are relatively easy to understand and maintain. Understanding such large systems is often made easier in part due to the simplicity of the language used, and through the practice of separation of concerns and other sound software engineering principles.
- As diagrams grow in size, optimal layout becomes increasingly difficult and time consuming to calculate, and may contribute to these graphical representations of a system being difficult to understand or maintain. Layout, however, is not as difficult of an issue when dealing with code, due to the nature of text and textual

editors.

Dwyer has attempted to resolve these with a 3D version of UML. We, however, believe that a textual modeling is better able to solve these points – in fact using three dimensions may in fact be exactly the opposite of what is needed.

Even though testing is not explicitly part of this research, the Future Work section will identify ideas and areas that could take advantage of Umple to simplify testing. In particular, future developments in Umple could greatly simplify testing through automatic generation of test cases from the Umple code.

1.2 Audience

Umple is intended to simplify the development lifecycle of systems. Therefore, the target audience of this research is individuals working with UML or other modeling languages, or those involved in modeling, implementation, or maintenance of software systems.

Furthermore, users of Umple are expected to have some experience with object oriented (OO) programming, and programming languages. This knowledge will make it much easier for users to absorb the Umple approach because Umple follows the OO paradigm, and experience with OO languages and UML will make the Umple language syntax much easier to master.

1.3 Organization

The thesis begins with a review of related background to our research. The background review includes modeling and UML, a brief introduction to object oriented programming related to our research, the third-party tools discussed in this thesis, and two other textual-modeling projects from which we have learned.

Included in the background discussion is a review of modeling practices and methodologies. Model-Driven Architecture (MDA) can be extended to work well with the idea of Umple, and is therefore central to validating Umple as part of development process.

Chapter 3 will then formally introduce the research questions explored within this thesis. The answers to these questions are offered later throughout the thesis.

Next, the thesis discusses Umple as a solution to some of these research questions in Chapter 4. This is a purely theoretical and abstract discussion of the solution. Concrete implementation details are provided in Chapter 5. Chapter 6 then describes the accompanying Umple software. This is a set of tools which were developed as a proof of the Umple concept.

The last chapter of the thesis, Chapter 7, summarizes our efforts and documents some of the future research ideas which could make Umple better able to meet the overall vision. Chapter 7 also lists some of the ongoing efforts by other researchers in the field of

Umple.

2 Background

Much background research has gone into the Umple concept, the language, the process with which Umple would work best, and the supporting software necessary to demonstrate both that the Umple idea is sound, and that it would be functional. This section gives a brief overview of the concepts involved with Umple in order to clarify the need for a tool such as this textual modeling language.

2.1 Overview of Software Modeling

At a high level, software modeling is used to express modeling concepts and ideas within a system at a specific level of abstraction. Modeling has been a part of sound development processes for a long time. In the traditional sense, models would be created after requirements have been gathered. These models along with requirements were the driving force behind development.

The Object Management Group (OMG) proposed Model Driven Architecture (MDA) as a solution to the problem of separation of platform dependant and independent aspects in systems [9, 11]. As Kurtev et. al. state, this approach to software development is still in rapid evolution, with many high-profile institutions racing to provide the complete solution which supports MDA and, more generally, Model Driven Engineering (MDE) [10].

At the earliest times of software modeling, models were created as a part of a design task, usually in the later stages of the process, after requirements elicitation. MDE, however, promotes models as the central artefacts, which drive the development process. As Forward and Lethbridge show in their survey [18], modeling is not yet a universally accepted approach to software development. For some people, on the other hand, models go beyond semi-formal graphical views of the system and include implementation code, XML Document Type Definition (DTD) documents, and a plethora of other artefacts. This is particularly the case when working with the MDE approach. It is easy to see how the Umple language and Umple process fit into MDE. Umple code is another artefact in the system (another view of the system) which is used to derive multiple other artefacts, including the implementation code (in Java). Umple can be used to derive UML diagrams, or vice versa. For further information, France and Rumpe [19] provide an overview of the current state of MDA.

2.2 Overview of UML

Unified Modeling Language (UML) is the most widely used graphical general purpose modeling language. UML is based on the MetaObject Facility (MOF). Both were proposed by OMG, and UML has become the industry standard for modeling.

UML offers a wide range of diagrams which represent different views of a system, and spread across multiple domains. As such, its usability is largely dependant on the tools one uses. As Forward and Lethbridge show in their survey [18], over 50% of developers found modeling tools poor or awful to use for brainstorming or prototyping. Both of these tasks require quick results with many frequent changes, which modeling tools do not appear to support very well.

Even with the use of tools of good quality, large systems are difficult to model and maintain due to the nature of these diagrams and the ability to store a lot of information in relatively small areas. Specific limitations of these diagrams include the fact that constraints and algorithmic code need to be expressed separately and textually. Another limitation is that the links between diagrams are often difficult to trace.

Furthermore, there is no unifying standard which could be used to produce working and complete code directly from UML diagrams. This results in each tool using their own standard which often leads to code that is difficult to read or understand, as well as limited interoperability between code generation tools. As Forward's research shows [18], many tools have the ability to generate code, but developers have been found not to like the code that is generated. Furthermore, the algorithmic code has to be developed separately from the code implementing the UML constructs

2.3 Subset of UML modeled through Umple

This thesis focuses on representing class diagrams using a textual language. This work is primarily done as a proof of concept, and the research is focused on the most common type of diagram in use, which is the class diagram. In the conclusions, we talk about ways to extend Umple into other types of UML diagrams.

UML is very flexible and extendible which might contribute to the complexity of the language. Our intent in developing Umple is to reduce this complexity. The complexity reduction arises from both the textual notation of Umple, as well as limiting the range of possibilities available in class diagrams, while still offering most of the power otherwise available in UML. This concept can then be extended to other parts of UML, beyond Class diagrams.

2.4 Object Oriented programming paradigm

Object Oriented (OO) programming is a paradigm where each type of entity is represented as a class. Generalization relationships among classes are specified, and inheritance of properties occurs. UML extended OO programming to include relationships between *objects* (instances of classes) represented abstractly as associations among classes. UML and OO programming are related in the way they treat objects, however OO is a programming concept, whereas UML is a graphical modelling notation.

2.5 Eclipse IDE

Eclipse is a popular, extendible, open-source integrated development platform (IDE), which can be used to integrate new software in the form of plug-ins. This makes it a good tool for the introduction and prototyping of new ideas.

The Eclipse IDE started as a proprietary project from IBM, and later became part of the open-source community; it is one of the most successful open-source projects in use today. Due to its large user base and easy extensibility, with much online support, Eclipse was chosen as the delivery platform for most of Umple software.

2.6 ANTLR Framework

ANother Tool for Language Recognition (ANTLR) is a parser and lexer generator. This means that given a BNF-like form of rules, it can automatically generate code, which can then parse an input stream based on those rules. ANTLR enjoys a wide audience due partly to its history of success, but mostly due to the fact that it is very easy to use with great tool support (ANTLRWorks is a GUI ANTLR IDE which supports syntax highlighting, a debugging environment, etc). This popularity was heightened by the fact that the author, Terence Parr, has written and published a reference guide for ANTLR as a paperback book [3], which is rarely the case for most open-source pieces of software.

Using ANTLR allows us to quickly and efficiently change the syntax of our language and to add features we discover might be useful or are necessary during the course of research. This tool has proven itself very useful and reliable, which allows for rapid prototype development.

2.7 JET framework

Java Emitter Template (JET) is a component of the Eclipse Modeling Framework (EMF) project. The JET framework takes a set of nested templates and generates an implementation file from it. The templates work using subset of the JSP (Java Server Pages) syntax, where the code within <%<code>%> tags is directly copied over to the resulting file, and code outside of these is translated into StringBuffer.append(…) operations. Within each template one can call other templates to create a template hierarchy.

This is useful to Umple because it simplifies the code-generation process. Instead of thinking about parts of generated code (such as get methods of association variables) in terms of classes, we can now follow a set of templates where we simply fill in bits and pieces which produce the final result. JET is simple and quick to work with, so it was used to generate some of the components of our concrete implementations of Umple. The JET files are compiled, resulting in an intermediate component, which is runnable Java code. This runnable component is the one ultimately responsible for generating system implementations.

2.8 Rational Software Modeller

For Umple to become accepted and used by the software development community, we had to provide at least a subset of standard features offered by other popular tools within the Umple software. On the UML modeling side, there are many tools, proprietary and open-source, which we can interface with to achieve the functionality level we are striving for.

Through our partnership with IBM, we've gained access to their Rational Software family of tools. These tools provide a wide range of capabilities, of which we only needed UML modeling. The decision to use Rational Software Modeler (RSM) over other similar or open-source tools was two-fold:

- RSM has been tested through many applications, resulting in a very mature and relatively bug-free product.
- The second reason is more technical: when modeling in a graphical environment, we are always concerned, and sometimes constrained by the layout [8]. When looking at the possible tools which we could interface with the rest of Umple software, only one we were able to use provides the ability to automatically lay out elements of diagrams. This meant that we do not have to spend extra development time looking at graph theory and the choosing the right algorithm for the task of laying out the diagrams.

These were the main reasons why we chose to integrate Umple software with RSM.

However, extra care was taken to define and document the interfaces between the RSM component and the rest of the software, in order to make it possible to simply replace the RSM modeling package with another which conforms to the same interfaces. This would make it possible, for example, to open Umple to the open-source community. This is currently not possible because RSM is proprietary software.

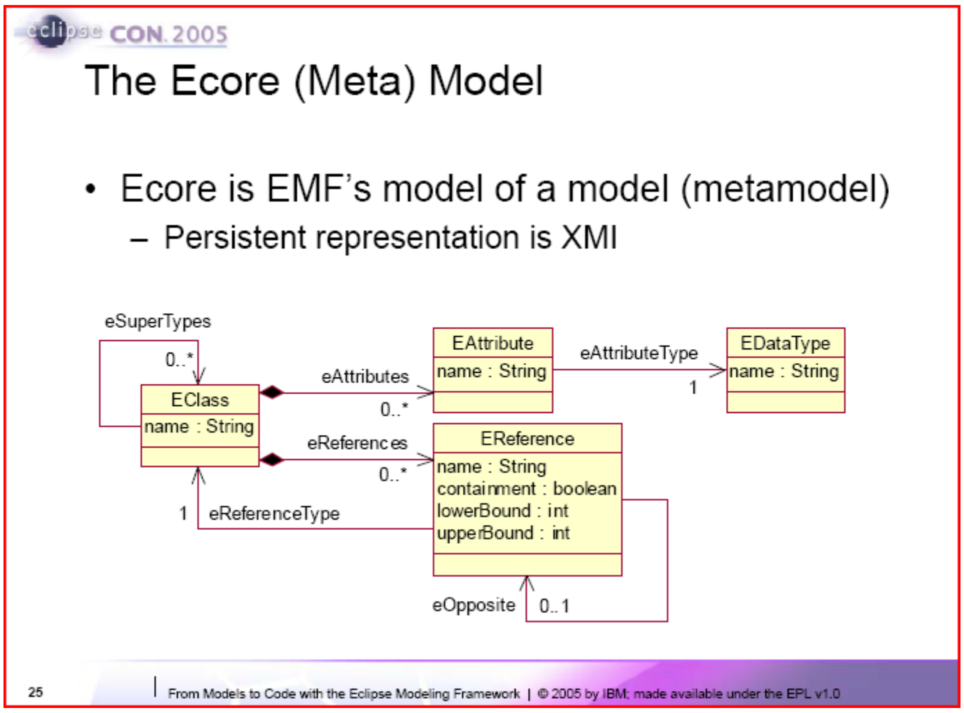
2.9 Other Textual Modeling Languages and Highlights

We are not the first to propose a textual language solution to modeling tasks. This section will discuss some of the well-known previous attempts at textual modeling, and give a brief contrast between these and Umple.

2.9.1 Emfatic and ECore

Emfatic is the language and text editor which can be used to describe an EMF model in a textual modeling language, similar in some ways to Umple [20]. It is using an Eclipse Public Licence (EPL) and is freely available at the Eclipse Modeling Framework Tools website.

EMF provides interoperability, code generation and a persistence mechanism. EMF also claims that the Ecore model is “essentially the class diagram subset of UML”[23], with only a few minor changes. However, these changes are significant enough to make Ecore unfavourable for Umple. The major change of Ecore when compared to the UML class diagram subset is the fact that Ecore does not have association classes.


eclipseCON 2005

The Ecore (Meta) Model

- Ecore is EMF's model of a model (metamodel)
  - Persistent representation is XMI

25 | From Models to Code with the Eclipse Modeling Framework | © 2005 by IBM; made available under the EPL v1.0

Figure 3: ECore metamodel [23].

Emfatic is an Ecore tool, not a UML tool, which is visible in the syntax. For example, in order to create a working model, the user needs to specify nsURI in some of the constructs, such as when declaring a data type, as in:

```
datatype EFeatureMapEntry : org.eclipse.emf.ecore.util.FeatureMap$Entry;
```

Emfatic is able to generate an Ecore model from the code, or code from the Ecore model. This model is then used as an interchange format between other EMF tools. The main disadvantage, besides its usability issues, is that the language is based on the Ecore metamodel, not UML. This metamodel is a very small subset of UML metamodel.

2.9.2 MOF Human-Usable Textual Notation

We have mentioned the MetaObject Facility (MOF) in the UML section. MOF is used as the metamodel for models, and would be considered a meta-metamodel of any system modeled using UML. As with UML, MOF is the work of OMG.

The MOF specification includes the Human-Usable Textual Notation (HUTN) [13]. As the name suggests, this is a textual (code) way to specify instances of a MOF model, which is used to complement the graphical way of doing the same task. During the course of our research, we have reviewed this tool to determine what lessons could be learned and to try to speculate as to how Umple could fix the mistakes made.

The first major difference between the Umple language and the HUTN technology is that HUTN is a way to generically generate language syntax parsers, effectively generating textual modeling languages based on a given model (based on MOF) and a configuration specification. HUTN does not represent a single language, but the set of all languages which could be generated given the above. This approach allows for any model specified based on MOF to be accompanied with a textual notation, which is a very powerful idea. When looking at the actual syntax of these generated languages, however, we can right away see issues which hinder the tool's usability.

As a simple illustration, let us look at an example from the HUTN specification. This example describes a family package metamodel, with classes Family, Person, Dog, Fish, and Car and the relationships we might imagine between these entities. Figure 4 shows the example as taken out of the specification [14].

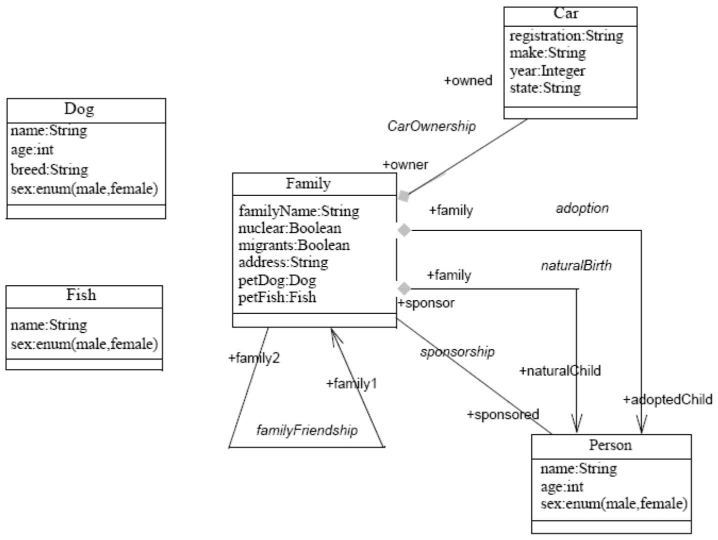
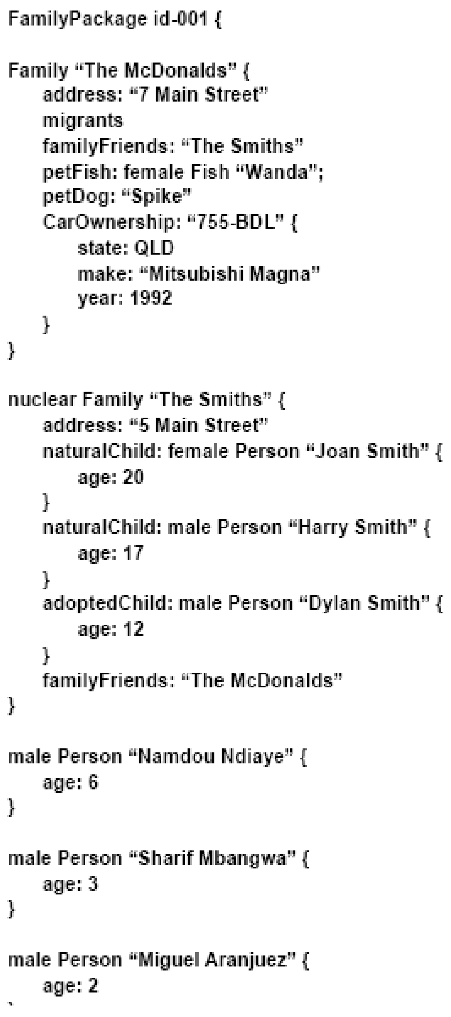


Figure 4: Example of a family package as taken out of HUTN specification.

The example then goes on to show the corresponding XMI specification for this example, which will not be shown here. Lastly, the example gives the code in HUTN-generated language which would use the model described above. Figure 5 shows some of this code.



```
FamilyPackage id-001 {

Family “The McDonalds” {
     address: “7 Main Street”
     migrants
     familyFriends: “The Smiths”
     petFish: female Fish “Wanda”;
     petDog: “Spike”
     CarOwnership: “755-BDL” {
          state: QLD
          make: “Mitsubishi Magna”
          year: 1992
     }
}

nuclear Family “The Smiths” {
     address: “5 Main Street”
     naturalChild: female Person “Joan Smith” {
          age: 20
     }
     naturalChild: male Person “Harry Smith” {
          age: 17
     }
     adoptedChild: male Person “Dylan Smith” {
          age: 12
     }
     familyFriends: “The McDonalds”
}

male Person “Namdou Ndiaye” {
     age: 6
}

male Person “Sharif Mbangwa” {
     age: 3
}

male Person “Miguel Aranjuez” {
     age: 2
```

Figure 5: HUTN code which takes advantage of the family package.

From the example we can quickly identify elements of the language which are difficult to understand or justify, from the point of view of a usable textual modeling language. For example, the need for FamilyPackage id seems unclear. Furthermore, the structure of the code is difficult to follow, and even more difficult to quickly write, which is not the case with Umple. This example helps illustrate the language issues which hinder the tool’s usability.

It is also particularly important to note the different levels of abstraction between the HUTN code shown in Figure 5 and MOF model described in Figure 4. HUTN-generated languages can be used to initialize instances of classes but cannot be used to describe the model itself. This is done using MOF and only after the model is created, using an HUTN tool one generates a language based on that model. This is one of the major differences between Umple, which is a textual language used to create a UML model specifications, and HUTN language which is used to populate a MOF model.

Lastly, HUTN is not used in practice today; this perhaps has more to do with UML than with the tool. The formal specification for HUTN was introduced in August 2004, yet no successful attempts have yet been made to come up with a HUTN specification for UML, which is a child specification of MOF and should be able to be modeled using HUTN. The probable cause of this is that the HUTN specification has to conform to the UML metamodel, which is very extensive and complicated. The HUTN language generated based on UML would then also be very difficult to use. We suspect that even if one tried to only model the class diagrams, as is done in Umple, the XMI-like structure of the resulting language would make it much too difficult to use. One of the goals of using a textual notation over a graphical one is that the textual version should be easier and quicker to use. These goals would simply not be met when using HUTN to model UML, due to UML metamodel size and HUTN's generic nature.

Overall, the HUTN notation provides an improvement over writing XMI for the purpose of describing instance. However, it is not as functional nor as usable as it needs to be, when comparing it to the standards set by other popular non-generic textual languages, such as Java or C++. Added to this is the lack of a UML HUTN specification, forcing the users who do want to take advantage of HUTN to use the MOF, instead of UML. This further hinders HUTN usability and popularity.

2.9.3 UML Action Semantics

As is discussed in detail in Section 5.3.9, every system requires a certain level of application logic. This logic could operate on multiple layers of the system, ranging from GUI logic to controller or even database layer logic. UML was historically very inept at the expression of application logic, constraints, or invariants. The advent of OCL has attempted to change this, but there were still gaps in UML practitioners' ability to express application logic.

The OMG proposed a solution to this problem, named UML Action Semantics [29]. This specification allows UML users to specify a diverse set of actions that can be performed on the elements described in UML diagrams. In theory, this allows users to represent logic within UML. However, this new specification poses some problems which hinder its use and adaptation.

One of these problems is that the specification is for an abstract syntax, relying on concrete implementations. Over time, there have been several attempts at implementing this concrete action language, the most notable of which are Object Action Language

(OAL), Java like Action Language (JAL) and Action Semantics Language (ASL)[30]. These independent implementations introduced several other problems, further inhibiting the use of UML Action Semantics. For example, none of these languages conform completely to the UML standard. As is discussed by Claudius Heitz, et al. [28], “Unfortunately, none of the existing action languages can express all constructs of UML Action Semantics directly.” Each tool chose a subset of UML Action Semantics which resulted in lack of interoperability between tools, a feature this UML standard (or indeed any standard) hoped to provide. Another issue with the languages was that many were proprietary, which limits adaptation by the general professional public. Lastly, all but one of these languages (ASL) are unable to result in 100% generated code. This means that after specifying action semantics and generating a system from the resulting models, one would have to contribute code in the target language. This is a task Umple attempts to omit completely.

As we mentioned, ASL is complete and allows one 100% code generation. However, there are several factors which make it difficult for use in this project. The language itself has been put into public domains but the only parsers and production-quality interpreters of this language are proprietary, and distributed as part of iUML. This makes it inappropriate to be used with core Umple software which is intended to be open-source. Furthermore, the ASL language is not mimetically compatible with Umple or Java, which would decrease Umple’s overall usability. Figure 6 shows us a simple example.

```
# Read a single attribute value (NB "John" must be unique).
johns_age = Person.age where name = "John"

# Read two attributes at a time
[johns_age,johns_height] = Person.[age,Height] where name = "John"

# Example using an instance handle
# Create new object and get handle ...
new_person = create Person with name = new_name

# Later ... Use the instance handle to access an attribute
new_age = new_person.age
```

```
# Read a single attribute value (NB "John" must be unique).
johns_age = Person.age where name = "John"

# Read two attributes at a time
[johns_age,johns_height] = Person.[age,Height] where name = "John"

# Example using an instance handle
# Create new object and get handle ...
new_person = create Person with name = new_name

# Later ... Use the instance handle to access an attribute
new_age = new_person.age
```

Figure 6: ASL Example

As we can see right away, the language uses different style format and syntax, which is in many ways quite dissimilar to Java and Umple. These factors combined led to the decision to not use ASL to specify action semantics in this project.

Another problem with UML Action Semantics is that according to Heitz's research "UML Action Semantics that do not have a direct Java counterpart". Without a direct translation from languages based on UML Action Semantics to an OO language such as Java, the use of this specification is very limited at best. Findings by Heitz were reinforced by Clark, et al., in their review of responses to an OMG Request for Proposal [31]. This review also further questions the suitability of UML Action Semantics. Because of these factors and shortcomings combined, we were forced to not use UML Action Semantics to specify application logic in Umple, and have instead used our own solution, discussed in Section 5.3.9.

3 Research Questions

This chapter is a formal statement of the research questions this thesis explores.

3.1 Abstraction jump in coding

As mentioned in the introduction, there exists a separation of levels of abstraction between the implementation of the system and the models describing the system. The first research question we will investigate is: *RQ1: What are the advantages and disadvantages of adding abstractions found in UML to a Java-like programming language?*

3.2 Bridging gap between models and code

When changing the model of a system, the corresponding implementation has to be separately changed as well, unless a code generator is employed, which is usually not the case. Over time, minor changes are made to the code but are not reflected back as changes in the model, resulting in a gap between the implementation and model.

RQ2: How can we bridge the gap between model and implementation, by making the Umple code represent both? The criteria of success in answering this question would be if the diagrams and models of a system and the implementation of that system are one and the same.

3.3 Advantages of modeling using code

Graphical modeling research was started with the use of flowcharts [1]. Much research has gone into graphical modeling languages since. Abstract ideas and features of software systems have traditionally been shown graphically. The most successful of these graphical techniques is UML, discussed earlier. This thesis will look into another direction of modeling – modeling using text. *RQ3: What are the advantages and disadvantages of text-based modeling?* We intend to incorporate some of the best code language practices into our text-based language.

3.4 Simplify and hasten development

Software development is an imperfect process with many possible areas of improvement. *RQ4: To what extent does the Umple approach simplify development? RQ5: To what extent can the Umple approach speed development?*

3.5 Simplify maintenance

Software systems evolve over time, and over many iterations. Traditionally, a change in system would first have to be modeled and then implemented by one or more developers. This is a two-step approach to the problem of maintenance. Unifying both the implementation and modeling step could greatly simplify maintenance both in terms of time and complexity as, once again, changes in Umple code would result in immediate corresponding changed in the UML models as well as the underlying implementation of the system. *RQ6: To what extent can the Umple approach assist with maintenance?*

3.6 Separate programming language and modeling language

As stated earlier, Umple is a text-based modeling language. This new approach begs the question: Why are we commonly using a programming language separate from the modeling language? *RQ7: To what extent can we create a language that has the full power of both a programming and modelling language, such that it would be sufficient in order to produce a system?*

4 The Umple solution

Umple is an effort to simplify the software development process. Our objective is that Umple would speed up time-to-prototype and the overall time-to-release. This is accomplished through eliminating some of the software development activities, which are often repeated, through automating these tasks. Umple is essentially a modeling language, much like UML, combined with necessary elements of a standard programming language.

UML is a graphical language, offering a set of diagrams and notations to describe a system. Umple is a text based language. One possible process of designing a system using Umple would be to sketch out classes in a textual environment, using very simple and short constructs, and then have the Umple software translate them into their respective UML diagrams. From this point the designer can see both structure and code for the system. The designer can now change the Umple code to refactor the design or change the corresponding UML diagrams. However, the current version of Umple software only supports changing the Umple code. In this way, Umple helps facilitate MDA.

UmpleCore is the translator for the code written in Umple; it translates Umple to an executable platform – currently Java, but possibilities such as bytecode, machine code or other high-level languages are also possible. The syntax of Umple will be discussed in Chapter 5. UmpleCore is made up of many components which will be discussed in detail in Chapter 6.

4.1 Increasing level of abstraction

The conceptual disconnection between modeling languages and programming languages originates from the fact that these work on separate levels of abstraction. When working with UML, one is concerned with the high level interactions within a system, or between separate systems. This is also apparent from the level of formality of syntax of UML. For example, if a parameter is not specified or is omitted, the diagram often still contains enough information to deduce enough about the model to make sense to the software developer reading it. The syntax of UML allows for omission of information in favour of not making the diagrams look overwhelming and cluttering.

Programming languages, on the other hand, do not tend to allow omission of arbitrary details, although there may be defaults for certain elements such as visibility (and some languages lack types). The requirement for all details to be provided makes many programming languages cluttered and verbose. Verbose and complex syntax often gives greater control to the user and allows for more powerful code, but it makes the code more difficult to understand, maintain and debug.

It is then conceivable that a programming language that works also at the level of modeling language abstractions could share benefits of both modeling and textual programming languages, while eliminating some of the drawbacks of each. This would be a textual language which takes advantage of simpler syntax, which assumes defaults where not specified, as in UML. It would also contain constructs found in UML, which would result in the required increase in abstraction.

4.2 Umple language

The Umple language is the text-based modeling language referred to throughout this thesis. This language incorporates ideas and lessons learned from many object-oriented languages used in practice, as well as concepts from UML.

Umple combines both modeling and OO principles. For this reason, Umple code should in theory translate to any OO programming language. However, each target language results in a slightly unique dialect of Umple, since the design of Umple focuses on translating UML features to the target language, but passes through algorithmic code without interpretation. The name Umple refers to the syntax common to all dialects. Derivative such as JUmple (with file extension .jump) represents a specific syntax which generates Java code, and expects algorithmic elements to be written using Java-like syntax and semantics. As part of this thesis, we have defined this dialect of Umple, leaving other potential dialects for future work. Where a concept applies to the whole Umple approach, the term Umple will be used. If, however, a feature or a concept applies only to JUmple, we will use this term.

The first observation we made was that every language has to balance power and simplicity. As stated in the HUNT specification [14], "the syntax of a language can have a strong effect on the speed and efficiency of its use for an expert user, but the syntax features associated with this speed and efficiency often lead to a more difficult learning curve for the novice user. While it is not impossible to deal with both, a certain trade-off between these two features is apparent in many common programming languages.” Because the focus of this research is to provide a simple solution to the complex task of system development, some of the features and powers of the target programming language had to be limited; in other words standard solutions are chosen to certain programming tasks, limiting the power of the programmer somewhat.   Most particularly, Umple takes care of generating essential target-language code needed to implement high-level concepts specified in Umple; this is often referred as “boilerplate” code, and is discussed below. All systems, however, require a certain level of customization. This customization comes mainly from algorithm and arithmetic operations, often referred to as application logic, performed by the system.

Most new languages and advances in the programming languages have simplified the development process, allowing better efficiency, flexibility, or power to the user. Umple is a demonstration of how the next step can be taken. Umple benefits from the observation that much of the work developers of systems do can be simplified and automated.

Boilerplate code is code which is necessary in a program to implement a certain concept and is repeated many times. Classic examples are getter and setter methods in a class. Such code does not hold much value when one is trying to understand the behaviour or architecture of the system. It can, and indeed must, be reused in numerous similar contexts, often without any need for alterations other than simple variable substitutions.

Of particular interest in this thesis is boilerplate code for declaration of variables representing associations and the methods needed to add and delete links of those associations. Sometimes such code can be quite complex, but other than referring to different variables, it is basically the same for all associations that have the same patterns of multiplicity.

Another observation is that users do not take full advantage of modeling tools. In academia and most of the successful software companies, it is agreed that modeling helps to communicate most types of systems and simplifies understanding of a system [24]. However, as discussed in the Section 1.1 there's a conceptual disconnection between programming languages and modeling languages. For example, the idea of associations is one of the major areas of study – what is an association in Java? In UML, the idea of association is very well documented and described, however there is no explicit association construct in Java, C++, C, PHP, etc. Umple aims to bridge the gap between programming languages and modeling languages, to provide a "model-code duality". This of course brings many advantages:

- Use of models in software development
- Systems will be easier to understand because constructs such as associations or design patterns will have a standard implementation.
- There will not be a need to separately update design and code because now the design is the code is Umple, which can be automatically rendered as diagrams

describing the system. This, coupled with javadoc-like documentation practices, could greatly simplify and quicken this process.

Umple is an abstract idea which can be applied to any OO language. This means that components of the supporting software can be interchanged to work with or produce code in Java, C++, C#, etc. For the purposes of this thesis, however, Java will be used as a point of reference. Java is a widely used OO language, which makes it a good reference point.

Models are vital to the Model Driven Development approach. Even though there are many model transformation tools out there (JET, MTF, OptimalJ, etc..), they are often limited to allow the user to specify transformation rules and carry out a transformation operation based on these rules, and some other parameters. Umple assumes the transformation rules, and lets the user focus on specifying the model of the system, instead of how to transform it to executable code. In other words, using Umple we are able to specify what is the system, instead of how is the system constructed. By assuming these transformation rules (Umple-to-UML and Umple-to-Java), Umple standardizes the implementation of a system, which could result in:

- Use of best practices uniformly among all developers. If Umple makes a decision about the best way to implement a pattern, for example the Proxy pattern, then all systems created using Umple will behave the same way, resulting in consistency, and hence greater maintainability. Furthermore if defects are found in how Umple implements a concept, the fix can be applied across all systems when they are recompiled.

- Simplification of system through abstraction. Even modern languages, such as Java, result in the need to write boilerplate code, which has to be repeated many times in various parts of the system. By eliminating some of this boilerplate code, Umple lets the user focus on the more important structure (and in later stages of Umple, also behaviour) of a system. Therefore, Umple raises the abstraction to the level of UML.

4.3 Coding on the level of UML

Code in the current version of Umple is mainly composed of three types of constructs: classes, associations, and association classes. These directly mirror what is immediately visible in a UML Class diagram. In Umple, classes describe the objects within a system, associations describe the links between these objects that can or must exist at runtime, and association classes are a combination of both. As in UML, classes in Umple can have attributes and associations to other classes. These associations define multiplicities and optional role names for each link between classes.

These features are directly extracted from UML. This shows how coding in Umple is like modeling in UML class diagrams. The user is able to think on the abstract level of UML while writing code, which can instantly become the implementation of the system.

4.4 Coding and modeling as one task

Coding in Umple follows the same process as modeling in UML. The key feature of

Umple is that it directly generates both UML diagrams and system implementation. Any change in JUmple code will result in immediate changes to the diagrams and generated code. Umple unifies these tasks as one.

4.5 Modeling concepts within a programming language

Working with UML and Java, it became obvious that not all modeling concepts are incorporated into the OO paradigm. The ones omitted will be discussed in the next section. Even though OO programming languages are powerful enough to simulate the ideas, there is no standardized way to do so.

Through Umple tools, we are able to standardize the way modeling concepts are implemented in a programming language. Some concepts, such as classes, belong to both modeling and OO domain. However, concepts such as state and associations do not. This results in inconsistent, complex and often faulty code written from scratch each time.

4.5.1 Mapping code to modeling

Umple offers a standard way to implement concepts in UML that are not explicitly present in pure OO programming. Each code-generation tool provides its own way of implementing a set of specific concepts, and there is no complete standard or reliable list of sound practices. This results in the code of a system generated by tool A being incompatible with code generated by tool B, and requiring “glue code” to bring these two together. The result is that the same systems generated by multiple tools have different properties in terms of cohesion, coupling, complexity of generated code, lines of code, security, performance, etc.

Umple provides a possible set of standards that, if adopted by other tools, could be used interchangeably with other code generated by tools based on UML. These standards are based on lessons learned from other code generation tools as well as original ideas following the Umple methodology of simplicity and understandability.

Mapping attributes

There is a direct mapping between UML class attributes and Java instance variables. Umple translates an attribute declaration to an instance variable and the standard Java get and set methods, where appropriate.

Umple goes further, however, in generating code which checks certain conditions before interacting with attributes of a class or ensuring an invariant. For example, a case where setters cannot be generated is when we declare an attribute as “immutable”. This means an attribute can only be set once and must stay the same from then on. This setting is done by supplying the value for the attribute in the constructor of the generated class.

An example of condition-checking code generation presents itself when we use the keyword “unique” to describe a class attribute in Umple. This is shown in Umple code below.

```
class Flight{
 //Represents the unique identifier
 unique flightID;
}
```

Figure 7: Umple code snippet using "unique" keyword.

The above snippet of code declares a class named Flight. This class has an attribute flightID which we state is unique. This corresponds roughly to the UML notion of a 'qualifier' and conceptually means that at no time shall any other class associated to Flight have multiple objects of class Flight with the same flightID. The code which ensures this is always true is generated in the setFlightID() method of Flight, which contacts all of Flight's neighbours and makes sure they do not already have a flightID variable set with the given value.

In the simplest of cases, an attribute in an Umple class simply translates to an instance variable in a Java class. Umple, however, offers very simple, quick, and easy-to-understand ways to modify how attributes are handled and effect class interactions with these attributes.

Mapping classes

Mapping between Umple classes and Java classes is handled in accordance with the Umple methodology. Once again, there is a direct mapping between Umple classes and Java classes. Each class construct in Umple code inevitably becomes a class in Java. Umple handles all tasks required to create a complete class, such as constructor generation, import declarations, and so on.

Creating class hierarchies can be accomplished using the keyword “isA”. This syntax was chosen as being synonymous with the “is-a” test, which is often taught as a very simple test to determine if a class is a subclass of another class.

Mapping associations

Most of the power of Umple comes from the way it handles associations between classes. At the very lowest level, a one-way association in Java is simply a variable in a class that has the type of the other class. Umple associations go far beyond this point, to translate UML association concepts to Java. The following are useful examples.

```
class Employee{
 unique ID;
}
association{
 * Employee -- 1 Employee supervisor;
}
```

Figure 8: Reflexive association example.

```
class Fare{}
class City{
 name;
}
association {
 * Fare faresFrom -- 1 City fromCity;
}
association {
 * Fare faresTo -- 1 City toCity;
}
```

Figure 9: Multiple associations between two classes.

In Figure 8, we are given two classes and two associations between these classes. We immediately notice concepts which are not directly present in Java. The first of these concepts is multiplicity, which follows the UML syntax. There are many ways to program the behaviour of 1..* multiplicity in an association, but Java does not offer a standard or an explicit construct to support this concept. Much of the logic generated for Umple associations is related to ensuring multiplicities are respected. In the above example:

- Multiplicity end 1 simply translates as an instance variable in the opposing class (the opposing class is the class where the 1 is written when an association is drawn in UML). Set and get methods are generated to facilitate access to this variable. The object of the class needs to be supplied in the constructor to the opposing class. This is to ensure that each instance of the opposing class at all times has a link to an instance of this class.
- Multiplicity end 0..* translates to a set in the opposing class. Simple get(), set(), add() and delete() methods are generated to manipulate this set.
- Multiplicity end n..m is a combination of the first two types of multiplicities. A list has to be passed to the constructor with at least n elements. Add and delete methods make sure the consequent interactions with that list do not cause it to decrease below the lower bound or increase above upper bound.

Next, we see the role name (toCity, fromCity and supervisor) in Figure 9 and Figure 8 respectively, which is a label put on an association end in class diagrams. Role names often simplify understanding of an association, such as in Figure 8, where we are describing that each Employee has a supervisor. Role names can also distinguish between associations, such as in Figure 9, where a class has a fare to city and from the city. These and similar cases have made it necessary for role names to be part of Umple. The translation using role names simply means that variable name is declared using the role name provided, instead of the class name.

As we've shown, the UML concept of association does not have a trivial translation to Java. However, such a translation is possible and is standardizable. Umple proposes one such translation based on satisfying the functional requirements implied by the semantic meaning of associations, as well as simplicity requirements imposed by the overall goal of Umple.

4.6 Change of paradigm

Umple inevitably presents a change in paradigm. This change is necessary in order to reach our goal of simplifying the development process. Unlike many other revolutionary ideas, which in the past required a radical change in both the way developers think and what they do, the paradigm proposed through Umple is more of a merge between the modeling and programming paradigms.

4.7 Coding becomes modeling

As we suggest throughout this thesis, Umple coding becomes UML modeling. In its current stages, this modeling is limited to class diagrams, but ideas on how to further this research will be given in the conclusions chapter.

By bringing UML concepts and ideas to a programming language, we have created a bridge between creating a model of a system, and implementing that system. Even more so, we have introduced a way to complete each task faster, both in terms of time to create and time to understand and maintain.

This is a potentially key improvement in the paradigm of software development, while not requiring great amount of new knowledge or a new paradigm to be absorbed by the developer. We have reused the semantics and ideas from both modeling and programming paradigms, allowing someone who is technically able to quickly learn to use and work with Umple.

4.8 Model change over time

As modeling and programming become a single task, there is no need for separately updating a class model and the underlying implementation code due to change in the requirements or for any other reason.

System change over time is inevitable. Systems evolve from the first day of development to the last day before decommissioning. Therefore dealing with these changes quickly, efficiently, and keeping the related models in perfect synchronization is a significant advantage of Umple.

4.9 Generated system as black box

In order to preserve the concept of working strictly at a high level of abstraction, we needed to pay close attention to the generated system in terms of simplicity of the code, coding standards and best practices, and the way one is meant to interact with our generated system. As Forward and Lethbridge determine in their survey [18], developers look for code generation in their tools but most find the generated code very cryptic and difficult to work with. This is an area where Umple can make significant improvements.

An Umple system needs to have a complete set of functionality. It needs to provide all the functional and non-functional features required by the customer. This can be accomplished using the syntax provided by JUmple. Umple systems behave as a black box, by providing a set standard API to interact with the system, and a clearly defined point of contact. The developer simply needs to be aware of the JUmple code which describes structure, interactions, and application logic, and the developer can simply plug this Umple generated system into an existing one, or combine it with another Umple generated system.

Umple incorporates a few features to help accomplish this goal. The first of these is the ability to separate code into namespaces. A namespace is a logical grouping of classes which relate to a common domain. Separating your Umple code with namespace declarations translates to each class declared after a namespace declaration being put in the package with the specified name. This form of separation of concerns helps facilitate understanding of both Umple code and the underlying code, where package structure contains information about the organization of code.

Another feature that helps developers treat Umple generated code as a black box is the generation of a Facade class. This is an example of the widely used and very useful Facade design pattern, which states that one should provide a point-of-entry class which handles interacting with a system. This is a standard practice for any system, whether generated or manually created, and so it was a necessary feature of Umple-generated systems.

The Facade mechanics are fairly restrictive. A façade can never return an object whose type is a class specified in the Umple code since external code would not know about the generated classes. The only types of return values one can expect the Facade to return are Strings, JSONObjects (will be discussed in few paragraphs) and primitive data types. Primitive data types include int, double, and Boolean. This was done deliberately to decrease the coupling between an Umple-generated system, and another layer or system that might use it. For example, an outer system might want to call a getRegularFlight() method from our Airline example (see Appendix 2) in class called Airline. To do this, the system has to go through the Facade class, but the resulting return is not the RegularFlight object, but instead a unique identifier of that object, relative to the system. This unique identifier is different then the attribute declared as "unique", which was discussed earlier. A unique identifier is assigned to each object created within our Umple-generated system, and is stored in the *system registry* collection. This identifier is then removed from the registry when an object is deleted.

This registry class is also automatically generated with each system. It is implemented as a simple hash table (using the Java Map interface). Facade and registry work closely together because most calls to a Facade method require a context, which is the unique ID of an object we are accessing.

Another feature of the Facade class implementation is that when asked to return a list, for example a list of all the RegularFlights associated with an Airline, it returns an iterator to the list of the ID's of each object making up the original list. This serves two purposes. The first purpose is to, once again, adhere to the encapsulation of our system. The second is to return as little information as necessary. An Umple-generated system is meant to be usable in any application, including local or remote. If our system is invoked remotely, using for example RPC or as a Web Service, returning a whole list could negatively affect the performance very quickly. This is why a call for a list to the Facade in an Umple-generated system returns an iterator to the list of ID's making up the underlying list.

Lastly, to simplify interacting with the system, the Facade class provides two versions of each method that would return an ID of an object. First, as stated, is an implementation which returns the ID. Second is an implementation which returns the ID as well as the values of all of this object's visible fields in JavaScript Object Notation [2] (JSON). This is done using the string representation of the JSONObject transfer object open-source implementation. Returning a transfer object is done to minimize interaction on a scenario where an outer object, such as a GUI, wants to display all available information, or a subset of that information, about an object. Without our transfer object approach, the GUI layer would need to make numerous calls to the Facade class, which is a performance issue. This way, however, the GUI can simply call a single get-type method if just a single attribute is required, or else a single get-type method that returns the whole attribute content of an object.

Interacting with an Umple generated system is made as simple as possible, while respecting restrictions that result in good software practice and design. The generated system is treated as a black box because even though Umple is created to allow as much power as is available in the underlying language, there are many systems developed on constant basis with these languages. Instead of trying to replace them all, Umple complements them with the interfaces it provides in the generated code, but also offers many advantages to development of new systems within Umple instead of the traditional medium.

5 Umple Language

As mentioned before, the aim of this research is to provide means of simplifying software development. Our solution that allows us to do that is a new language which combines terminology and ideas from both OO programming languages and the graphical modeling language UML. If we hope to achieve the stated goal, we need to present a language that is simple to understand in terms of both syntax and semantics, yet powerful enough to allow for the construction of systems of an arbitrary size and complexity.

When coming up with the language, we used UML as one of our main sources of inspiration. This is because we feel syntax based on a graphical language is more intuitive and quicker to understand than syntax mostly based on programming languages. After all, many would argue that the underlying programming language we use (Java) is already a very clean and simple language to understand.

As is shown in a usability study of UML [4], users saw it as easy-to-use. This is a desirable feature of Umple as well. What was interesting and perhaps counter-intuitive, however, is that users did not see the UML diagrams as easy-to-use. For example, the study states that when rating class diagrams, the mean rating given to these was 4.295 out of possible 7 with a standard deviation as 0.673. This result is strong evidence that developers see the advantages and positives within using UML, but find the particular diagrams difficult to use, which leaves much room for improvement.

Furthermore, the study identified features that developers found easy to use and features that they had problems with. To us, this means that we should follow the way the easy-to-use features are done in class diagrams and perhaps reinvent or simply just improve on the way the problematic features are done. It is also interesting to note that some of the frustrating parts of diagrams will be inherently improved when one uses textual representations, as Umple does, instead of the UML graphical notation. The study does not state how to improve these diagrams or what developers thought might be possible improvements, but we believe we can attempt to fix these issues through careful analysis and iterative comparisons of Umple features.

5.1 Main Concepts

The main idea which makes writing Umple simple is to describe what one could see on the UML diagram counterpart in as few words as possible while taking advantage of default values, or assumed semantics (see Table 1). This is a uniform concept, which can be applied when writing Umple of any diagram, not just class diagrams. This is important, as Umple shows large potential scope and we therefore cannot and should not confine it to simply class diagrams. A section in the conclusions chapter will discuss our thoughts and ideas on furthering Umple into other popular UML diagram types.

Context	**Assumption**
variables	- assumes String type if not specified - providing a default value removes it from constructor signature. This value can be set using its accessor method, but is no longer required to be set by the

	constructor - if not specified, variables will have both a set() and get() method - if the "immutable" keyword is used, variable will be read-only, and will need to be set in constructor unless default value is given - variables are assumed to be settable. Explicitly specifying a variable as "settable" has the same effect as leaving it out. - variables set as "internal" are considered private
role names	- if not given, role name is assumed to be the class name in lower case - if not given, singular will be used for single object, plural for a set
namespace	- if not given, name of file is used as namespace - first namespace specified is the container of registry and façade class - each class belongs to the last namespace specified before the class declaration
association sets	- treated as lists - implemented as array lists - if the class which is to be content of a set representing multiplicity has "unique" identifier, then this set is implemented as hash table with that id as key
inheritance	- multiple inheritance is not supported

	- isA declaration has higher priority than implicit inheritance declaration - last isA declaration determines final inheritance structure. If multiple isA rules are declared within the same class, only the last one will be implemented.

Table 1: Some of the most common Umple defaults and assumptions.

Another important concept not just in Umple but in all OO is separation of concerns [5]. This states that concerns which are conceptually different should be dealt with in different components, with as little functional overlap as possible. Using this approach allows one to focus on one particular aspect of a system at a time, without the need to conceptually understand all details of the rest of the system because, as Dijkstra notes [22], from one aspect's point of view, the others are irrelevant. Umple facilitates separation of concern in many ways, allowing the breakdown of even the most complex of systems, which will be discussed in Section 5.3.5. Following this principle is a standard OO practice, but it needs to be handled with care. Even though conceptually separation of concern allows us to understand or debug a system quicker, the way in which one can do separation of concerns must be simple and consistent. In other words, allowing for too many different ways to specify separation of concerns may cause our code to become confusing because it is then difficult to see the “big picture”, which is the overall system or its higher level structure.

Frederic Richard and Henry F. Ledgard [16] state in their work that “distinct features should have distinct forms”. Furthermore, McIver and Conway [17] refer to multiple forms for the same features as syntactic homonyms. The Umple language minimizes these while offering a few syntactic homonyms along with clear rules for use in order to improve cohesion in the Umple code.

5.2 When to start using Umple

When keeping in mind that Umple can be used as simply a code representation of a class diagram, modeling with Umple becomes quick and easy to learn. We start writing Umple the same way we start drawing a class diagram. One doesn't simply jump in and start drawing boxes and associations between them, unless one first has a conceptual view of the entities and their relationships within this system. This is the case with Umple as well. This is another advantage of Umple – because it lends itself very well to and mimics the process of creating diagrams (even though it is done textually instead of graphically), and it does not call for a drastic change of process employed by a development team. If one wishes to use Umple as part of the software lifecycle, the time to introduce and start writing in Umple is the same time that team would have created UML diagrams. This is almost exclusively at the beginning of the development lifecycle, with changes to these models as the development progresses. This way we can very easily identify the appropriate time to use Umple. The transition from pure UML form of models used by a development team to an Umple form of models (which generates UML models) should be quick and intuitive.

5.3 Features of Umple

After we identify when is the appropriate time to start writing Umple, we can then look at the actual features of this language and how they interact with one another. The features of the language in this context refer to language constructs, their meanings, and the particular concrete syntax that comprises Umple. We find the simplest way to explain Umple is to constantly relate it to UML. To view the whole syntax of Umple, please consult Appendix 1.

5.3.1 Hello World Umple

Classes and associations are the basic building blocks of UML Class diagrams and we will start with simple representation of these in Umple. As in UML, the most basic and minimalistic class is one with just a declared name, therefore Umple as well allows for such basic class with the declaration shown in Figure 10.

```
class Airline{}
```

Figure 10: Hello World Umple example.

This very simple line of code accomplishes exactly what was described above, which is a minimalistic class with a name (see Appendix 2 for the complete version of the Airline example). In this case, the name of the class is Airline. When building class models, we most often think of the entities within a system, then their relationships, and then iteratively fill in the more detailed features such as multiplicities of associations, attributes, or operations. This is also the way one might want to write Umple, where we first brainstorm all the entities and possible object classes of the system, quickly throw them into the model with simple constructs as we saw above, and then iteratively fill in bits and pieces as they are discovered.

The syntax used to describe a class is as follows:

```
classStruct   :       'class' IDENTIFIER '{' (classItem)* '}'

classItem
        :       isARule
        |       varDecRule
        |       singletonRule
        |       implicitAssociationDeclaration
```

5.3.2 Class attributes

Each class in Umple can have its own set of attributes of a certain data type. Umple limits the number of possible data types to Integer, String, Float, Double, Boolean, Date and Time. We believe these are enough to convey the most kind of class attributes one might wish to use.

Date and Time are in fact objects in the underlying implementation but are used by Umple as primitive data types. This is done because of the observation that many real systems take advantage of some time or date functionality, and we felt adding support for these natively simplifies and minimizes some of the work required by the developer.

Data may also be declared as “immutable”. Immutable fields are ones which can only be set once, and never again. If a default value is provided for an immutable variable, then only a getter method is generated for this field. This is done in the following example:

```
class Elevator{
  immutable String prefixID = “QW12”;
}
```

Figure 11: Elevator class with immutable attribute.

This will create a field called prefixID in the Elevator class which has a default value and getter method. Declaring a field immutable without specifying a default value will mean the value has to be provided in the constructor of this object, and as before, can never be changed again after.

Umple tries to simplify and speed up coding in many ways. One of these is that when the type of a field is not given, it is simply assumed this field is of type String. UML supports supplying this form of incomplete information because it often still conveys the information necessary for understanding the intended function of a class, and this has caused it to appear in Umple as well. We could not get rid of types altogether and use type inference to fill in types of fields, because that technique requires observing how variables are used to determine the type of a variable. However, it is Umple that dictates how a variable is used because it is Umple that generates the code that interacts with these data types. This puts an upper bound on how much work we can simplify for the user. Only having to specify the type of a field is a very attractive trade-off for the ability to generate intelligent code that interacts with this field.

Variable scope can be controlled with the “settable” and “internal” keywords. A variable is assumed to be “settable” by default, which means it can be read-write accessed by anyone with access to the containing object. Variables declared as “internal” are treated the same way private variables are in OO, with only the class able to access and change these.

An additional feature available in Umple is the ability to declare fields as “autoUnique”. Auto-unique fields are unique in each instance of the class. The code ensures that only one static master-variable is used and each new instance of the class is assigned the new incremented value of the static variable. This is useful for attributes such as ID, which need to be unique in each instance of a class.

The syntax dealing with variables in Umple is:

```
varDecRule : regVarDec |  autoVarDec

regVarDec :   (uniqueDec)?(attributeModifier)? (attributeType)? IDENTIFIER
(EQUALS value)? ‘;’

uniqueDec :   'unique'

attributeModifier :     'immutable'|'settable'|'internal'

autoVarDec : 'autoUnique' IDENTIFIER ‘;’

attributeType :         'String'|'Time'|'Integer'|'Float'|'Date'|'Double'|'Boolean'
```

5.3.3 Class Relationships - Hierarchies

The UML usability study [4] mentioned previously found that class relationships are both a likable and frustrating part of UML. Umple gives us the opportunity to improve the understanding of these in a model. Umple allows only for single inheritance, as is done in OO languages like Java.

There are two ways to declare hierarchical relationships in Umple. The choice between the two could be based on the separation of concern principle, or simple user/team preference. Assuming we are modeling a hierarchical relationship between parent class Person and its two child classes Student and Teacher. In OO terms, this means that both Student and Teachers inherit some features and relationships of Person.

The first way is to declare this relationship explicitly, using the “isA” construct. This would result in a form of code as follows:

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

Figure 12: Inheritance example using isA keyword.

The choice of using “isA” as a keyword for explicitly declaring inheritance comes from the “isa” test. This is a rule which is used as a heuristic to conceptually check the correctness of inheritance hierarchies. Because the 'isa' rule is a fairly standard teaching tool when explaining inheritance in OO, it seems like good choice for a keyword in Umple.

There is another way to specify inheritance in Umple, which we call implicit declaration. If working with the same example from the explicit declaration section, we would write the Umple code as

```
class Person{
 class Student{}
 class Teacher{}
}
```

Figure 13: Inheritance example using implicit inheritance declaration.

This code behaves the same way the earlier version does. If we wish to declare another level of inheritance, for example, to say that a Student class can also have a HonourRollStudent child, we would simply insert the declaration of that class into Student, and keep going in a recursive fashion. Each version has a set of advantages and disadvantages which determine when one might use implicit or explicit inheritance declarations.

One of the advantages of the explicit form is that because it explicitly states the 'isA' keyword, it might be little more readable and identifiable as a child of class Parent. Another advantage is that if we are adding a class to the system which is a part of some inheritance hierarchy, we can simply declare a class as being a child of another class already declared in the system, instead of altering existing code, which we might not have access to.

On the other hand, an advantage of the implicit declaration is that it is less verbose, more compact representation, and it allows us to very quickly identify all the children of a class, or the parent of class. It also helps to increase the cohesion of our Umple code, because we are keeping classes conceptually related by an inheritance hierarchy together. Cohesion is a combination of separation of concerns and bringing similar things together, and general good organization for understandability.

The advantages of the implicit inheritance declaration become more apparent when we are dealing with larger inheritance hierarchies, and we would like to visually query the system. This is done very easily in UML, where in one diagram one can immediately see the way classes are structured. Explicit declaration would require the user to jump all around possibly multiple Umple files (file extension .ump), or to search through the same file. Implicit declaration mimics the locality of Class diagram inheritance hierarchies which might make large inheritance hierarchies more readable and understandable in Umple.

The syntax dealing with hierarchies in Umple is as follows:

```
classStruct:   'class' IDENTIFIER '{' (classContent)* '}'

// Class declared inside another class – implicit declaration
classContent :          classItem | classStruct | appLogic

// Using the isA rule – explicit declaration
classItem: isARule | varDecRule | singletonRule | implicitAssociationDeclaration;
```

5.3.4 Class Relationships - Associations

Another concept for a construct which describes the relationship between classes in UML is an association. Associations also appear in Umple. Associations declare that instances of one class will have references to instances of another class.

Associations were among the top most frustrating features ranked by users in [4]. This gives us an opportunity to introduce an improvement to Umple over the underlying language, UML. Associations are not an explicit structure of our OO language, Java, which already makes Umple associations an improvement over Java.

An association is more than simply a reference to a class from within another (or possibly the same) class. Associations also offer an accounting mechanism in the form of multiplicities and association constraints to better express the relationships between objects.

Once again, there is an explicit and implicit way to declare associations in Umple. The advantages of each once again dictate the use.

Let us imagine a simple Airline example, where the only entities present are an Airline, Employees and Planes (similar to the full Airline example in Appendix 2). One possible Umple representation of this system could be

```
class Airline{}
class Employee{}
class Plane{}

association {
 1 Airline -- 1..* Employee workers;
}
association {
 1 Airline -- * Plane;
}
```

Figure 14: Explicit association declaration example.

This piece of Umple code declares three classes mentioned earlier, and two associations. The first association is between the Airline and Employee classes. This association states that an airline has at least one employee, and each of these employees only belongs to one airline. The second association works similarly to the first, but the lower bound on number of planes an airline can have is zero. Another feature we can see is the role name declaration “workers”. This has the same semantics as a role name in UML.

The implicit way of declaring associations is somewhat similar to the way implicit declaration was done for inheritance. To express the same relationships using the implicit declaration style, we might write something like the following:

```
class Airline{
 1 – 1..* Employee workers;
 1 -- * Plane;
}
class Employee {}
class Plane {}
```

Figure 15: Implicit association declaration example.

Once again, this piece of code behaves the same way as the explicit version. We can see similarities between the implicit inheritance declaration and implicit association declaration.

As was the case before, each style of declaring associations presents its own set of pros and cons. We have found that most of the time, however, these two styles should be used in conjunction with one another, to help preserve the separation of concerns principle, increase cohesion and decrease coupling. One possible way to organize a system like this is to group all of the operational entities (flights, airline, employees), accounting entities (frequent flier points, fares), and possibly details of actual flights (flight legs, actual instances of regularly schedule flights). If we assume this is how we choose to separate our system, then we can declare associations between classes within each subsystem implicitly, such that each subsystem does not know about any of the other parts, and then declare associations between classes from different subsystems using the explicit declaration style. In this way, we accomplish very clean separation between each of the components, and we can quickly observe the exact points of coupling between each subsystem, because those are the associations we have abstracted and declared them explicitly. Appendix 3 on the accompanying website shows full code for many of Umple test examples.

The way we declare multiplicities in Umple is very similar to the way we declare them in UML. This notation is fairly easy to understand, and ported quickly and easily to Umple. The underlying generated system code ensures boundaries are respected, and that we generate the appropriate interface classes (for example, it would be incorrect to generate an addAirline() method in Plane class, because there can only be one airline associated to each plane).

To declare that an association is one way only, meaning that one class knows of the other but that one does not know of the first, can also be done in Umple. To do this using the explicit declaration, we simply add the “->” construct, as in

```
…
association {
 1 Airline -> * Plane;
}
```

Figure 16: Explicit association directionality example.

This states that an Airline has a one-way association to Plane, and Plane does not have an association back to Airline. The Airline end with multiplicity one is simply conceptual, and is not enforceable by code, because Plane does not have a variable of type Airline in the generated system. This is, however, a feature of UML and so it was also put into Umple.

To accomplish the same behaviour using the implicit declaration style, one might write something like:

```
class Airline{
 1 -> * Plane;
}
class Plane {}
```

Figure 17: Implicit association directionality example.

This, once again, declares that Airline has a one-way association to Plane, with zero as lower bound and no upper bound. The notation of the "->" was used to mimic the way a one-way association looks in a class diagram. We have already seen "--" which is a two way association.

It is also possible to declare a reflexive association. A reflexive association is one of which both ends end in the same class. These association types are very common. If we wish to model in Umple a class Employee, in which each employee has a set of zero or more subordinates, and one supervisor, we might write something as the following:

```
class Employee{
 * -- 0..1 Employee supervisor;
}
```

Figure 18: Reflexive association example.

The only syntactical difference between a reflexive association and a regular one is that a role name must be provided. This can be useful for documentation purposes, but is also syntactically necessary, because without the role names we would not be able to distinguish multiple reflexive associations from each other.

The Umple syntax dealing with associations is as follows:

```
langStruct :   classStruct | associationClass |association
classStruct :  'class' IDENTIFIER '{' (classItem)* '}'

classItem:     isARule
         |     varDecRule
         |     singletonRule
         |     implicitAssociationDeclaration

implicitAssociationDeclaration : multiplicity implicitAssociationDirectionality
        multiplicity IDENTIFIER (IDENTIFIER)? ‘;’

implicitAssociationDirectionality :  ('--'|'->')

association  : 'association' '{' (associationItem)* '}'

associationItem:      associationLine

associationLine : multiplicity IDENTIFIER (IDENTIFIER)?
        implicitAssociationDirectionality multiplicity IDENTIFIER (IDENTIFIER)? ‘;’

multiplicity:    (NUMBER ('..'!( NUMBER | '*'))?) | '*'

roleName:     IDENTIFIER
```

This syntax describes both implicit and explicit associations. At the high level, one can declare a class, an association class or an association. A class can contain an implicit association declaration, the syntax of which is very similar to the higher level counterpart, with the exception of the first identifier, which is omitted in the implicit version. The name of the class in which this inline association is declared is assumed to be the identifier value.

5.3.5 Umple Separation of Concerns

We have already introduced a few ways in which Umple helps to promote separation of concerns: grouping classes working in the same sub-domain together, explicitly or implicitly declaring associations depending on whether the association is intra- or inter-component, or implicit vs. explicit class hierarchies. Another way is to use namespaces.

Namespaces are declared using a line “namespace <namespace name>” in an .ump file. Namespaces separate the code into logical components, and in the case of Java they are separated into packages. Umple automatically takes care of including all necessary packages in the generated code so the user is free to create as many namespaces and divide the system into as many components as is desired with minimal work.

The Facade and registry class are placed in the very first namespace declared. If no namespaces are declared at all, then a default package name is used, which is the name of the first Umple file read. A class declared in Umple code is placed in the namespace defined by the latest preceding namespace declaration. It is sometimes desirable to separate out the facade and registry classes completely from the rest of the generated code, which could be done by using the following code

```
//Facade and Registry will be in PackageA
namespace PackageA
namespace PackageB
class SomeClass{
 //This class would be in package PackageB
 ...
}
```

Figure 19: Namespace example.

This simple trick lets us completely separate out the entry point classes of the system generated by Umple. Without the first namespace declaration, both SomeClass and the facade class would be put in the PackageB.

Another way to use separation of concerns is to separate the different components on the level of Umple files. One can declare a subsystem A in package PackA in a file called SubsystemA.ump, and another self contained subsystem B in package PackB in a file called SubsystemB.ump. The associations between these two systems (essentially what some might refer to as glue code) can be declared separately, or within one of the two, but in a different namespace. The user has a lot of freedom in the way the separate their system.

When we combine all of these mechanisms to clearly separate components, we allow the user to increase cohesion of both of the Umple code and the resulting generated code, as well as decrease coupling by abstracting out the inter-component associations. When these mechanisms are used sensibly and with sound software engineering practices in mind, the systems one creates should be much easier to understand, manage, adapt and alter.

The syntax for namespace declarations is as follows:

```
item:   langStruct | namespaceDecl
namespaceDecl :     'namespace' namespaceExpr
namespaceExpr :     IDENTIFIER('.' IDENTIFIER)*
langStruct :     classStruct | associationClass |association
```

A namespace declaration is at the highest level, as are class and association declarations. A namespace expression can contain dots, as for example “umple.cruise” namespace.

5.3.6 Application Logic

Application logic in this context refers to any custom programming-language code that cannot be generated. Umple allows the user to specify this logic in Java-like syntax. This is only a subset of possible Java, however, since Umple does not allow manipulating of arbitrary objects within the application logic.

This logic can be either inline (specified within the classes in .ump files) or extracted separately in .jump files. Which of these methods is used is determined by how much logic is required. A rule of thumb could be that if any more than a small number (e.g. 15) lines of code are required then use the external method, otherwise use the inline method.

At the present time, application logic is used for such elements as constraints (invariants, precondition checking, etc.) and algorithms of various kinds. As Umple develops to incorporate states the need for application logic should decrease. The exact form of the logic is also expected to evolve, as it is anticipated to make some of it consistent with, and drivable to and from OCL, as is touched on in section 7.2.1.

5.3.7 Design Patterns in Umple

Umple lends itself well to incorporation with design patterns. Design patterns are proven modeling structures which offer a solution to a specific type of problem. Currently, there are two native patterns incorporated into Umple.

The first pattern is the Singleton pattern. This pattern ensures that there is only one instance of a class which is declared to be a singleton. In Umple, this is declared using an optional expression “singleton;” in the declaration of a class. This will ensure the code which ensures this task is generated in the class declared as singleton.

The second important pattern employed by Umple is the Façade pattern. This pattern dictates that there is a class used as an entry point to the rest of the system. This pattern was already discussed in section 4.1.9. Umple generates a Façade class automatically for any system compiled.

Umple provides the opportunity to incorporate many other design patterns, to both make them easier to use in common practice, and establish standard ways these patterns are implemented.

6 Umple Software

In order to test the powers and limits of Umple, we need a tool which is up to par in terms of quality and current standards put forth by other tools which attempt to make software development easier. There were several tools developed for this project, both to facilitate development using Umple and testing the final generated system. The following is a component diagram showing the abstract dependencies of the 4 major components of Umple.

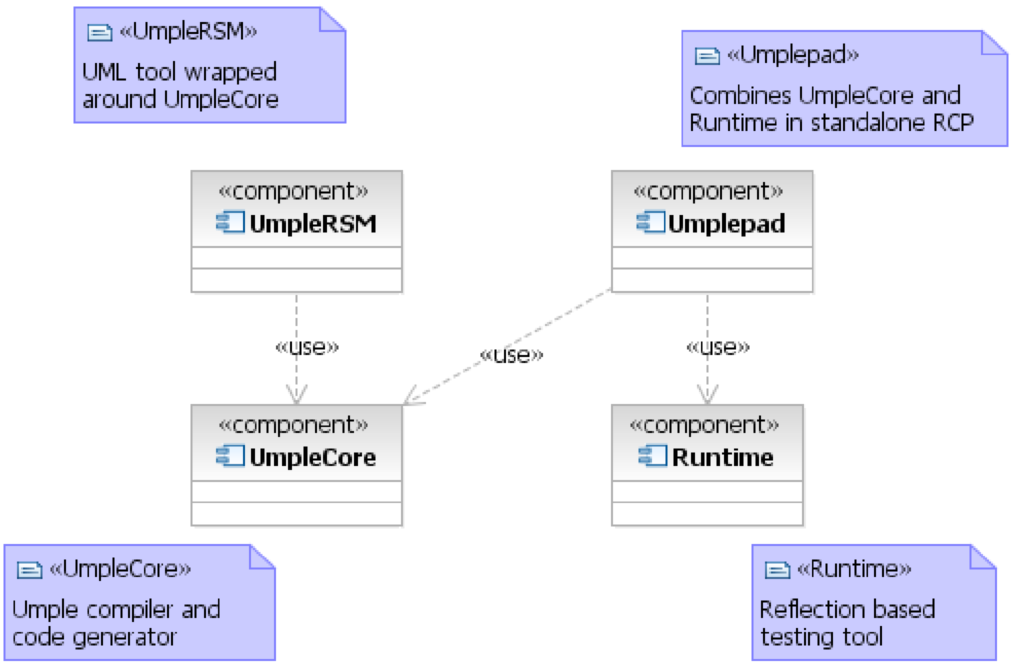

«UmpleRSM»
UML tool wrapped around UmpleCore
«Umplepad»
Combines UmpleCore and Runtime in standalone RCP
«component»
UmpleRSM
«component»
Umplepad
«use»
«use»
«use»
«component»
UmpleCore
«component»
Runtime
«UmpleCore»
Umple compiler and code generator
«Runtime»
Reflection based testing tool

Figure 20: Umple Software components.

Most of Umple software facilitates model transformation. If we define a model to be on a higher level of abstraction above an implementation, then Umple code is a model of the system in Umple which is then transformed using the tool. In this case, the transformation is two fold – includes both model-to-model transformation from Umple to UML and model-to-text transformation from Umple to code. The rules of each transformation stem from the Umple methodology, and the attributes of each transformation are specified in Umple code.

6.1 UmpleCore

This is the major piece of software of the Umple project. All of the other software components of the Umple project tie-in with or use this component in some way. UmpleCore is an Eclipse plug-in. The architecture of this plug-in was originally very monolithic with a hand-written parser and interpreter. This has evolved to a very component-oriented design allowing for quick localized changes. Figure 21 shows a very high-level organization of this plug-in.

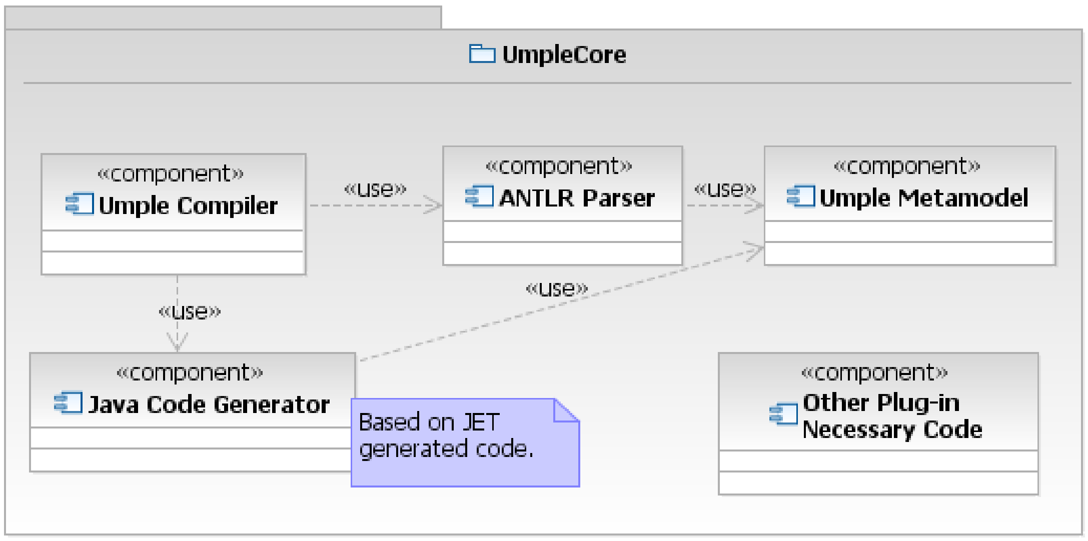


Figure 21: UmpleCore high-level overview.

UmpleCore is responsible for parsing Umple code, creation of a model according to the contents of that code, and generating the system implementation.

6.1.1 Umple Metamodel

It is important to note the Umple metamodel. This metamodel has similarities to the aspects of the UML metamodel that deal with class diagrams. There are minor changes related to the associations between classes in the Umple metamodel; however, all the classes of the Umple metamodel (classes which are created as a result of system specified in .ump code) hail from UML metamodel. This is an important feature of Umple. Umple aims to bring a modeling approach into development, in order to breach the practical and theoretical gaps between modeling and coding. Current languages don’t lend themselves well to this goal, which is why there is a need for Umple. There are many existing metamodels we could reuse in Umple software, but having a separate metamodel related to UML allows us greater control and ability to experiment.

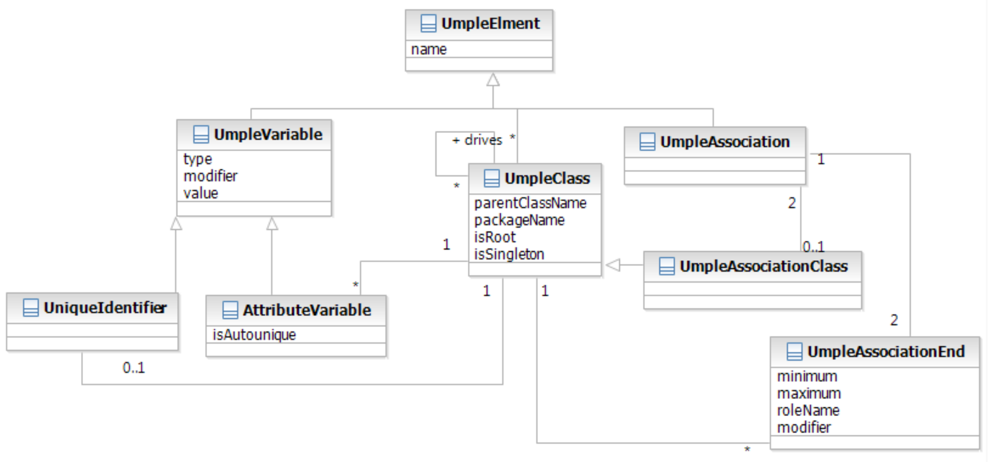


Figure 22: UmpleCore metamodel.

One such metamodel could have been the Ecore metamodel of EMF, which we introduced in section 2.9.1. Section 2.9.1 describes Emfatic in the context of EMF and Ecore. The discussion which follows here is a discussion of Ecore and its accompanying advantages versus Umple metamodels.

As mentioned before, associations and association classes are thought of as first class citizens in Umple. This means we need to have a representation of them in models based on the Umple metamodel. Association classes are crucial, and Ecore model does not explicitly keep track of them, whereas the UML model does. An option we have considered was to add the association class representation into the model. If we add association classes into the model, however, we lose most of the advantages promised by EMF. We can no longer take advantage of interoperability of tools because now Umple will be using a metamodel different from Ecore and the XML based serialization could not be interpreted properly unless the interpreting tool and the creation tool follow the same metamodel.

With interoperability eliminated, we now look at code generation. EMF generates fairly powerful java code, including UI code. Again, the EMF functionality does not quite align with the underlying needs of Umple.

First of all, the UI code generated by EMF is very basic – the GUI is simple a tree-based Eclipse view. If one wants to generate a UI of acceptable quality, one has to turn to the GMF framework. The GUI generated by EMF is nearly useless and so it wouldn’t add any value to Umple if we were able to generate that kind of UI along with the rest of the code UmpleCore would generate.

Secondly, EMF claims that the implementation of the system that is generated is easy to follow. As mentioned in one of the presentations about EMF by IBM, “Generated code is clean, simple, and efficient”[1, slide 39]. This is easily questioned when looking at a very simple example. Assuming an example consists of only one class Book, with attributes “title” and “pages”, then the following code is generated using the EMF generation tool [2]:

```
public class BookImpl extends EObjectImpl implements Book
{
  ...
  protected static final int PAGES_EDEFAULT = 0;
  protected int pages = PAGES_EDEFAULT;
  public int getPages()
  {
    return pages;
  }
  public void setPages(int newPages)
  {
    int oldPages = pages;
    pages = newPages;
    if (eNotificationRequired())
      eNotify(new ENotificationImpl(this, Notification.SET, ...,
oldPages, pages));
  }
  ...
}
```

Figure 23: EMF generated code.

Here we already see a few troubling features. First, is that each generated class extends EObjectImpl class which is part of EMF. This means each system ever created through Umple would have to be accompanied with the EMF framework. This is a constraint, even if EMF is an open source project. That is how the generated code gets much of its power: through dependence on and interaction with EMF. On lines 15 and 16 we see a call to possible observers. This code was neither necessary nor required for the implementation of a simple class Book with pages and a title, which makes the user wonder “why would it be there?” Umple aims to deliver a simple front end to generating code and models, to speed up development and simplify the systems and understanding of these systems. If, however, our implemented systems contain any code that appears to not belong there or is difficult to explain, then this hinders the simplicity of the system, and the power it may add the user did not ask for.

EMF is a Java framework. The code it generates is Java only, and the framework only comes in Java. This would not fit with Umple, which attempts to abstract away from programming languages and be based purely on object orientation and modeling. Therefore if we used Ecore metamodel as a basis for the Umple metamodel, and did use the EMF code generation tool, this code would only be useful in Java. If we then extended Umple to another language (for example Ruby or C++) to prove that it is in fact a viable concept in any language, we would have to reimplement the wheel for this particular language. Furthermore, generated Ruby code would also have to now have all the advantages and strengths that its Java counterpart would offer (due to the use of EMF). If Umple is to be independent of lower-level languages, it cannot bind itself to Java as it is done by EMF and Ecore.

Other features EMF-generated code offers is persistence and serialization, automatic notification of model changes, and a reflection API. These are features which Umple plans to support and for which EMF might in time prove itself to be more useful then it is currently, but for now the Umple metamodel is sufficient, flexible, efficient, and lends itself nicely to the nature of Umple. However, we have learned some lessons from EMF, and even used one of its components. This component is the JET framework which was briefly described in section 2.7.

6.1.2 Umple compiler

This is the heart of the UmpleCore. It is responsible for overlooking the process of turning Umple code into an in-memory model based on the Umple metamodel. This model can then be serialized in the form of generated code or in the form of UML diagrams (using the UmpleRSM plug-in). The driver class of this package is the UmpleSystem. This is where all the actions required to compile and generate code converge. The main data structure of UmpleSystem is the classHash hash table. This is where each UmpleClass is stored, with its name as the key.

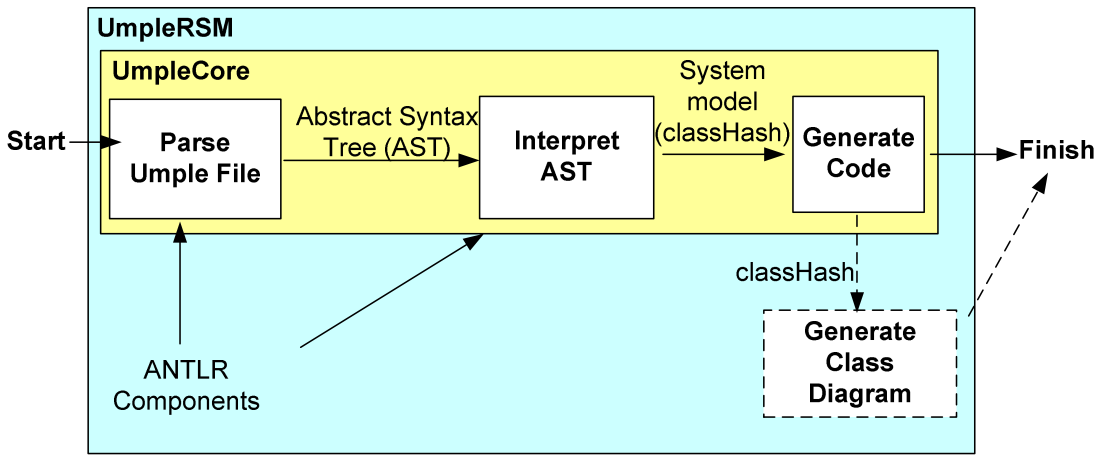


Figure 24: Compilation process.

Only classes are stored, because everything in Umple is related to classes. This stems from the way we observed UML class diagrams to be structured. For example, a UML class diagram with just an association or just a multiplicity, or even a set of multiplicities and associations would not be a valid diagram. The same is true for Umple – Umple code consisting solely of a declaration of an association would fail to compile. This implies that classes are the central building blocks of class diagrams and are therefore also building blocks of Umple code, even though Umple has associations as explicit elements. By storing only the classes with their appropriate associations and attributes in the classHash, we have access to everything that could be declared in an Umple file, while still offering the majority of the diversity and features most commonly used within class diagrams.

When we say "associations as first class citizens", we are referring to the fact that one can explicitly declare an association between two classes, independently of the declaration of the classes. Umple declares semantics and syntax for these, and they can even appear in .ump files separate from the files where the classes are declared.

6.1.3 ANTLR Parser

Another major player in the UmpleCore is the ANTLR-generated parser and evaluator component. A pass through these is the first step in Umple code transformation. Figure 25 shows the parser generation.

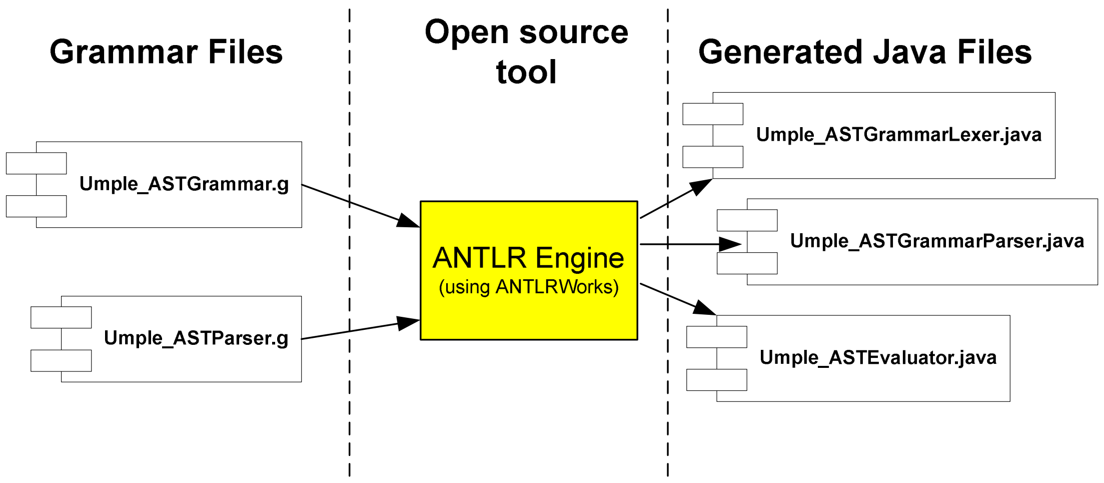


Figure 25: Parser generation using *ANTLR*.

Umple_ASTGrammarLexer.java is the first file used in the compilation process. This turns the stream of characters into tokens. That means the parser which is passed this stream of tokens is now only concerned with seeing EOL (End of Line) following an IDENTIFIER, without having to worry about the values in these tokens. This separation is a standard practice in language parsing.

Most languages are constructed to form a tree. A special type of tree most often used for language parsing is called Abstract Syntax Tree (AST). This tree would, for example, store data such as (CLASS String name John). This form simply states that at the root of the tree is a CLASS token, meaning the tree represents a class construct from Umple. The rest of the arguments are children of the CLASS node, and they declare there is an identifier name, which is a string, and has a default value “John”.

Umple_ASTGrammar.java is the generated parser which turns the stream of tokens into an abstract syntax tree, where the root node is identified as ROOT and all the children sub-trees represent the constructs given in .ump file, and the child sub-trees of these are the constructs given within each class, association or association class. In this way, the structure of the code provided and what it is meant to do is very easily understandable by both humans (during debugging) and computers. Umple_ASTGrammar.java simply turns a flat stream of tokens into an abstract tree, which is to be parsed by an abstract syntax tree evaluator.

Furthermore, every time an association or an association class is encountered, it is put in a queue. This queue is a list of all structures which need to be parsed in the secondary phase of compilation. This is because associations and association classes depend on other classes. Each association is between two classes, which means that these classes need to be present in the system at the time of compilation of the association, otherwise we wouldn't know what class to assign the association to. This is why associations and association classes are stored in a workQueue for later use. However, an important distinction had to be made when dealing with association classes. Because association classes are classes as well as associations, the features inherited from classes are processed during the first stage, along with the rest of the regular classes, and association features are processed at the later stage, along with the rest of the associations. The queue stores the AST's of all associations and association classes.

Another feature of ANTLR is that it easily allows for reuse. We reuse the same structures from classes and associations to compile association classes, with only having to make minor changes in the actions to take once an association class is parsed.

Umple_ASTEvaluator.java is the file which now traverses an AST and takes an action based on what is encountered. Firstly, the AST containing all classes is traversed. This is because in Umple, classes are the only constructs that do not depend on other constructs (at least at the beginning), so they offer a good place to start. Every time a new class is encountered in the tree, a new UmpleClass is created and stored in the classHash, mentioned above, and the key being the name of the class. After the creation of the class, the necessary attributes are created within the class as they are encountered during the traversing of the tree. The result of this step is the set of all simple classes specified in the Umple code stored in a hash table, and each class also has all the attributes specified in the code. As Umple only handles single inheritance, each Umple class stores a name of its parent, if one is specified.

Next, each AST in the queue is passed to the evaluator and parsed, similarly to the way classes were parsed. The required classes are retrieved from the classHash as needed, and the appropriate references are created using the UmpleAssociation and UmpleAssociationClass objects.

At this point we have created an in-memory representation of the model specified in Umple code. Lastly, the native code file has to be parsed, if any were specified. This concludes the parsing portion of the process, the result of which is an in-memory representation of the system specified in .ump or .jump file.

Now that the in-memory model is created, the system is able to act upon it. This process is started by the UmpleSystem.generate() call. The process consists of three steps:

- Run some pre-generation operations on the model. These will be discussed below.
- Iterate through the set of all classes in classHash and pass each class to the generation package.
- Iterate through the set of all classes in classHash and pass each class to the model creator package (if a diagram is being drawn as well).

The pre-generation operations are specific to the implementation language used. In the case of Java, the pre-generation process creates constructor signatures, which are then reused in some methods and method signatures. This has to be done in a certain order. Unlike above in the first phrase, where we simply iterate over the whole set of classes, this cannot be done that way when generating method signatures. This is because class A may depend on class B if B is a child of A – therefore if compiling B before A, then the method signatures will be incomplete. Therefore, the following algorithm is used when to generate the appropriate signatures:

```
For each UmpleClass in classHash {
  If class is root //has no children
    Then: generate signature for class, put class in readyForGenQueue
    Else: put class in waitingForAnotherClassQueue
}

While(waitingForAnotherClassQueue.size > 0){
  Find class in waitingForAnotherClassQueue which depends on a class which has
its signature generated (i.e. is a member of set readyForGenQueue)
  Generate signature for that class, take out of waitingForAnotherClassQueue and
put in readyForGenQueue)
}
```

Figure 26:Pre-generation algorithm.

This algorithm ensures that each class processed is either a root or has its parent class method signatures already available. This is necessary for Java because, for example, when calling the constructor of a class B which is a child of A, we need to include the arguments passed to A in the signature of B. A concrete example would be:

```java
public class Entity {
  public Entity(String ID){
    ...
  }
}
public class Student extends Entity {
  public Student(String ID, String name){
    super(ID);
    ...
  }
}
```

Figure 27: Java signature example.

Here, the Entity class would be processed first, having one of its method signatures generated as “String ID” and another would be “ID”. Because “String ID” is required in the signature of the constructor for Entity’s child, Student, when processing Student, the Entity’s signatures are obtained and added to the signature of the Student. Of course, the “super(ID);” is also generated, but this doesn’t happen until later in the process.

The actual process of generating a signature is delegated to the code-generation package. This is because all OO languages might have different style of method signatures, and although they follow from the same process (stemming from inheritance which is one of the features of Object Oriented languages), the actual syntax will differ.

6.1.4 Java Code Generator

The Java Code Generator component is the one responsible for all code generation. It starts off by calling the RunPregenOpsGen.getCode() method, to perform the initial operations discussed above. This class is generated through the JET framework, which was introduced in Section 2.7. The class creates two different types of signatures possible for java methods and caches them in the storage facility, making them available for other classes and methods to use. This is done so:

- Classes do not have to recalculate the same information multiple times, therefore saving time; and
- By creating the constructors first, we can make sure that all the required information to construct a valid constructor is available (When trying to create constructors on a class-by-class basis, we run into the problem of needing constructor information about classes which have not yet been compiled, as

mentioned above).

The different types of signatures are:

- A formal signature – this signature is used in method headers and contains both variable type and variable name. The name follows a convention where the variable name uses “a” as prefix of the type if no more appropriate name is specified. An example formal signature is: “String name, RegularFlight aRegularFlight”.
- An informal signature – this signature is different from the formal version in that it does not contain types. These are used when calling a method. An example informal signature is: “name, aRegularFlight”.

Creating and caching these is more efficient than creating only the formal and then altering it each time to produce the informal, because the method signatures are used very frequently.

The process of creating the appropriate signature is relatively simple. If the current class is not a root then the constructor signature or the parent is fetched and added to a String variable. Then the list of all attributes within that class is traversed, and each attribute which did not have a default value specified in the Umple code is also given a name and inserted into the signature variable. Similarly, the list of associations the current class has to other classes is parsed. If the association specifies that the current class has to have at least an X of the other class, where X >0, then this means the object we have an association that has to be specified in the constructor. This means, for example, that if a

RegularFlight class has a 1-* association to Airline, then at no given time can there be a RegularFlight without an Airline, and to make sure of this, the Airline is specified in the constructor, via the aforementioned process (see Appendix 2 for the complete Airline example).

There is one thing to note when dealing with 1-* associations. Umple has a special name for each end of the association. The 1 end is called the driver, and the * end is called the subordinate. When looking at the example mentioned earlier, we see that Airline is the driver of RegularFlight, and RegularFlight is a subordinate of Airline. It was necessary to name these two ends of an association because currently there is no other way to refer to these, other than simply “end 1” and “end 2”. The driver/subordinate relationship affects how the generated code behaves. When the code generated by Umple is setting a link of a bi-directional association between two classes, the driver class instance will set its reference to the subordinate instance and then call a specially-designated method in the subordinate instance (with the driver itself as an argument) to make sure the subordinate sets its reference to the driver. Without the distinction between driver and subordinate, both classes would have to set its reference to the other class and have to be called by a third entity (maybe resulting in undesirable associations), or each of the classes would have to set its reference to the other class and call the other class to tell it to set its reference to itself, resulting in infinite recursion. Therefore, if working with our standard airline example, when adding a RegularFlight to the system, the process of creating links between classes involved in associations is initiated (driven) by the driver, which is the Airline class in this example. A call to addRegularFlight() method which is generated part of Airline will create a new RegularFlight using the arguments supplied to the addRegularFlight() method, and in the process it will pass the context Airline object (using this Java keyword) to RegularFlight so that it can set its association end to Airline. From here on, any other associations RegularFlight might need to accommodate can be handled in a similar fashion.

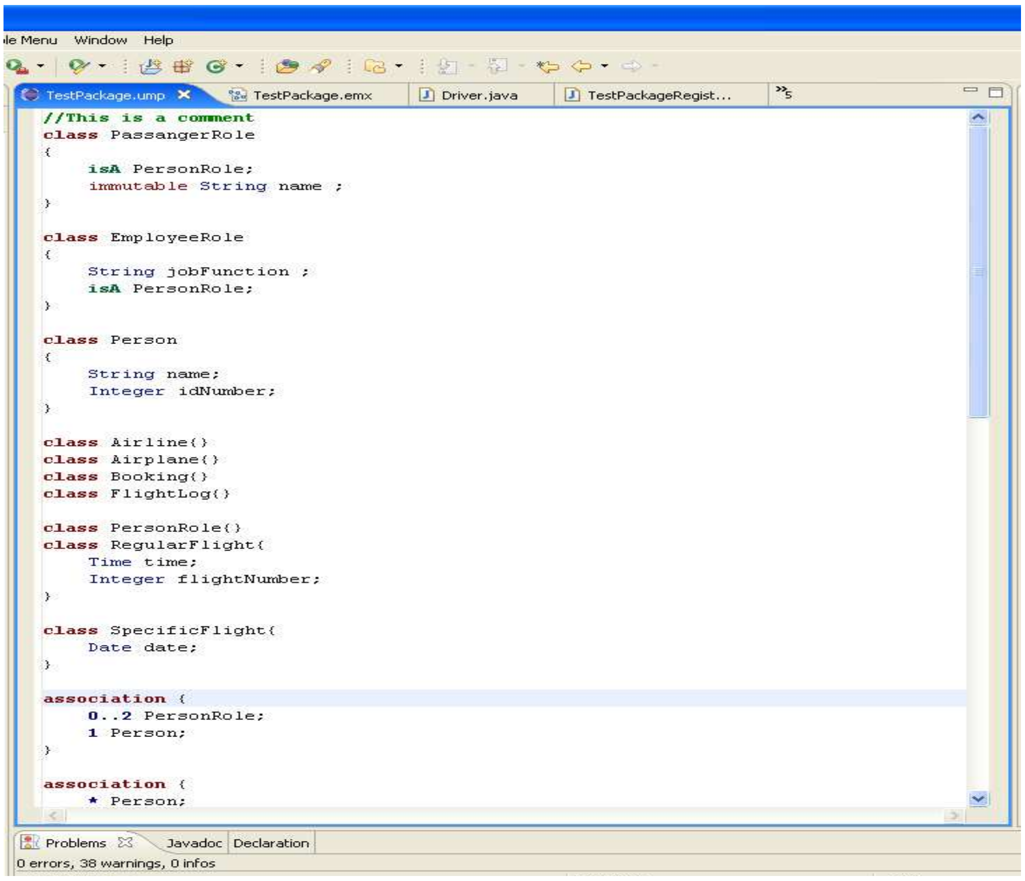
This idea is then extended to all types of associations where the following algorithm is used to determine the driver and the subordinate:

- 1-1 association → the first end specified in the association is the driver.
- 0..1-anything → the 0..1 end is the driver
- Anything else → the first end specified in the association is the driver.

Each generated signature is then slightly altered to contain the three different versions of each class constructor signature. The difference between the signatures is simply in details. Where one method signature would be “String ID, String name”, another signature type would be “ID, name”, etc. The signatures differ depending on the context in which they are used. The first version of the signature would be used as part of the constructor, but the latter would be used when calling this constructor by another class. In Java, only the method signatures are needed prior to constructing code, so that is all that is done.

6.1.5 Other Eclipse plug-in related concepts

The editor for UmpleCore supports basic syntax highlighting to make the language easier to read and scan for important constructs. The colours and styles were chosen to match with those of the standard Eclipse java editor. By using familiar styles, we hope to convey the idea that Umple is in many ways similar to Java. This is to help combat developer pushback, by supplying some of the features many Eclipse users have come to expect.



```
//This is a comment
class PassangerRole
{
     isA PersonRole;
     immutable String name ;
}

class EmployeeRole
{
     String jobFunction ;
     isA PersonRole;
}

class Person
{
     String name;
     Integer idNumber;
}

class Airline{}
class Airplane{}
class Booking{}
class FlightLog{}

class PersonRole{}
class RegularFlight{
     Time time;
     Integer flightNumber;
}

class SpecificFlight{
     Date date;
}

association {
     0..2 PersonRole;
     1 Person;
}

association {
     * Person;
```

```
0 errors, 38 warnings, 0 infos
```

Figure 28: Some of the syntax highlighting used by UmpleCore editor.

Figure 28 shows only some of the highlighting available to the user, but it is enough to show that it is similar to the syntax highlighting of Java, and will hopefully result in impressions of familiarity.

6.2 UML tool – UmpleRSM

The discussion up to this point has been focused on textual representation of an object-oriented system. Umple, however, offers a bridge between text and models, so it is necessary to explain how Umple handles the modeling aspect of the process. UML diagrams are potentially very ambiguous artefacts [15]. The Umple language is used to clear up the ambiguity, while leaving the generated UML diagram somewhat ambiguous, in the interest of only showing the most important structure or features of the system.

UmpleCore is the driving software component in the Umple process. It offers abstract interfaces to allow for interchange of many of the internal and external components. One such component is the code generation package, discussed above. Another interchangeable component is the UmpleRSM.

UmpleRSM is the modeling package of our Umple software. UmpleCore calls this package after it has finished creating the in-memory model from the textual description given in .ump and .jump files, and has finished generating code based on this model.

As the name suggests, this modeling component is based on Rational Software Modeler (RSM), created by IBM® Rational. RSM is a very versatile and capable subset of tools offered by IBM Rational Software Architect. This subset is sufficient for the purposes of Umple, which is why it was chosen as the modeling tool. Some of the other UML modeling tools we looked at were Green UML[6], and UMLet[7] but these tools simply didn't meet our expectations of software quality.

One of the very useful features of RSM is the "Arrange All" function which simplifies the necessary work done by the developer, because we do not need to worry about laying out classes in a class diagrams. The task of efficiently laying out classes in a diagram automatically has been studied and is a difficult problem to solve. So when we were given a chance to work with RSM, the choice was easily justifiable.

UmpleRSM has a very simple structure. Two out of the three classes in the only package in that component are responsible for running the UmpleRSM plug-in. The other class (RsmModelCreator) is the one responsible for the work required to generate a class model.

RsmModelCreator implements the ModelCreator interface. This interface only dictates that whatever package is responsible for modeling an Umple system, it must do so using the update(UmpleClass), update(UmpleAssociationClass), and update(UmpleAssociation) methods. Additional serializeModel() method needs to be provided which would save the in-memory representation of class diagram into a file, so that it could be read by a displaying tool. The interchange ability of components stems from this interface, because UmpleCore is not concerned with how the diagrams are generated or displayed, it simply provides the data used to create these diagrams. UmpleCore requires a modeling component at all times, which is why it contains its own version of a modeling component in UmpleCore.models. The default implementation of the modeling component, however, is empty. This is an example of the Null Object pattern.

Just as the with code generation component, the modeling component cannot exist by itself. The Eclipse system running UmpleRSM needs to have UmpleCore plug-in running. However, the difference between these two components is that the modeling component is distributed separately and builds on top of UmpleCore. When the plug-in is loaded, it simply initiates the UmpleCore component and passes in itself as the model creator, therefore overriding the default one.

RSM models are based on UML2 implementation of the Ecore model. UML2 is an EMF-based implementation of the UML 2.x metamodel. The component hierarchy is illustrated in Figure 29e.

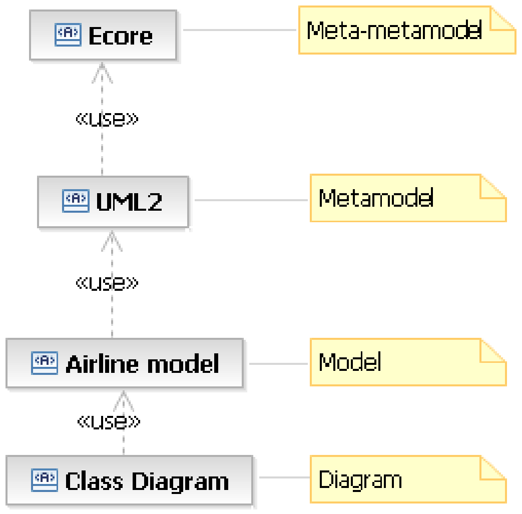


Figure 29: Dependency diagram for an Airline example.

If we consider an Airline system example, the final result after running UmpleRSM would be a class diagram (this class diagram is shown in Appendix 2). Because UML2 is a more functional model then Ecore, it now provides the objects and classes we require, namely the Associations, Association Classes, and Classes. The process of generating diagrams is simple transformation from UmpleClass to UML2.uml.Class, UmpleAssociationClass to UML2.uml.AssociationClass, and so on, which are inserted into a diagram.

The difficult task when creating the modeller component was primarily the exploration of RSM. In particular, exploring how to programmatically create diagrams which have until now been created using a GUI, imported from another tool, such as Rational Rose, or using XML. Adding to this task was the lack of any recent tutorials and documentation. RSM has gone through radical architectural changes, which seem to have been merged with the old architecture, which meant that the old tutorials called for method calls which were still present in the system, but were either deprecated or not behaving as expected. Some of the mandatory operations required were not mentioned in any of the tutorials encountered, adding to the confusion and difficulty.

Generating a class diagram requires a few steps. The first step is to translate an UmpleClass into an Uml2.uml.Class and then into a Node. GMF(Graphical Modeling Framework subcomponent of RSM) uses Nodes to represent any multi-edge object, and uses Edge objects to represent links (such as associations or generalizations). After a blank node is inserted into the diagram object it then had to be populated with fields and, if necessary, information within those fields. However, in our early prototypes these nodes were never showing up properly in the diagram, always resulting in an “error box”. After further investigation, it became apparent that all diagram changes need to be executed within transactional domain as a RecordingCommand. This was not mentioned anywhere in any of the tutorials and walkthroughs and FAQ’s ever encountered. Much of the knowledge about what needs to be included and how data should be structured in an RSM file came from reverse engineering a generated .emx file, which was serialized in XMI.

However, after experimenting with the modeling component and manipulating it enough to give the required results, UmpleRSM is able to generate both Java implementation of the system specified in Umple code, as well as a class diagram of said system. This is done simply by clicking on a “compile” icon in the Eclipse GUI, without the need for any additional setup. These two features combined offer the user the aforementioned model-code duality, where the implementation and the documentation (in form of diagrams) of the system can be updated each time the code of the system is updated.

6.3 Testing

As with any software system, there are many parts that need to be tested in different ways. Umple software is broken up into three separately testable components: UmpleCore, UmpleRSM, and Java Code Generator. Each component is distinct in its nature and requires a different method of testing.

Common to the testing of all three components, however, are sample systems. These are a set of systems defined in Umple that we use to test the different components, and that we use for regression testing once a change is made to any of the subcomponents of Umple software. Umple is designed to be able to handle systems that can be easily modeled using UML class diagrams, so it is obvious that to test the various components of the Umple project, we need a set of systems that contain various use cases we might wish to test.

In addition to the system use cases, we also made use of unit tests to test some of the important low-level functionality. These were only used in testing of UmpleCore. These unit tests were made to test functionality such as adding a class or an association into the Umple metamodel, or actions taken when encountering a special construct in Umple code, such as encountering the “singleton” pattern keyword, or “unique” keyword.

The Umple code component has remained fairly stable, with a few changes being added as we progressed on the project. Most of the changes we have made to the syntax of Umple did not affect backward compatibility, which made our use cases an adequate test of the overall Umple functionality. When additional functionality was added to the language, new use cases were often created which took advantage of this feature, or existing systems were altered to do the same.

However, the generation component changed very often, with new features and improvements being put into place almost weekly. As will be described later on, the code testing framework is set up so that if changes are made to the way code is being generated, the tests will pass as long as the original generated code is still present. If the Java Code Generator component is changed and the generated code still passes all of the tests, then the new code is generated and overwrites the old version. This of course means that the deltas (changes) in new generated code need to be inspected individually by the developer, but this is an unavoidable drawback, which is to be expected due to the difficulty in testing code generation components automatically.

There are 17 sample systems currently in our testing framework. These test cases include simple hierarchies such as the ones in the Airline example, to much more complex hierarchy in the 2D Shapes sample system. Other systems, such as the Political Entities, focus more on complex associations between objects. This provides us with a fairly solid testing base when compared against the intended scope for Umple systems. The sample systems are all available in Appendix 3.

6.3.1 Testing of UmpleCore

Instead of using unit testing to test each of the sub-parts of UmpleCore, which is very labour intensive, we use a much easier approach with very similar results. What is important about UmpleCore is the behaviour – generation of UmpleClasses, UmpleAssociations and UmpleAssociationClasses of a model, and setting of the appropriate attributes of each. The simplest way to test whether these were actually created in accordance to the model specified in Umple language, is to query the system after code generation for the number of each entity created. Each time one of these is added to the model (for example, because a “class” is specified in the Umple code), the appropriate counter is incremented. When we pass one of the aforementioned sample systems through UmpleCore, we know how many of each construct are to be created, which results in a check at the end of expected versus actual number of created constructs.

This approach is much simpler and faster then JUnit testing for each class and method in UmpleCore, but as with any trade off it also offers less power then JUnit testing. In particular, if class attributes weren't properly generated in the model, they wouldn't be caught by the UmpleCore testing. However, they would be caught by Java Code Generator testing later in the lifetime of the test case, so overall we did not lose much of the power offered by rigorous JUnit testing.

6.3.2 Testing generated code

When testing and evaluating the generated code, we are presented with a few difficulties. Code of most programming languages is very flexible. This flexibility comes from many sources. For example, allowing for any number of spaces after an identifier, allowing the open parentheses on the same line of a method declaration or after, etc. can result in an infinite number of possibilities which all exhibit same behaviour.

Another difficulty is performance; to parse generated code and evaluate it on a token-by-token basis is as time consuming as parsing the original Umple code, with the added complexity of having to perform this operation potentially dozens of times because each Umple file could turn into many long Java files. Furthermore, it is not enough to simply check that the generated code follows the right syntax. If this approach is chosen, then the right semantics must be checked as well, which would involve building a tree of paths and checking them all, same way the Java compiler would do.

A complete generated code testing platform would provide all of the following:

i. Syntax correctness – syntax has to be correct from the point of view of the language being tested. However, a failing syntax check does not necessarily mean failing system, as will be discussed later.
ii. Semantics – ensuring the generated code functions as expected.
iii. All code elements present – comments are not considered when parsing code, which means they must be checked externally by another tool.
iv. Code style – The generated code has to be readable and follow certain syntax.
v. Fast performance – fast testing will result in the ability to check more test cases more often.

Some of the points listed are difficult to accomplish. Point ii, for example, would require associating a set of test cases with Umple code and running each generated code through a set of test scenarios to ensure this is met.

Our final solution for testing of the generated code followed a simplistic approach, which was both flexible and efficient, while catching most of the bugs in our generated system. This approach was to check generated code against our pre-checked standard file, line-by-line, while omitting all whitespaces. Whitespace related bugs are often detected during the second stage of our testing. This gives us at least a general idea that if each line which appears in the pre-checked standard also appears in the generated code, we are fairly certain that if there is an error, it is in new code (code which has not yet been put into the pre-checked standard but is generated in the current iteration). If a line is expected in the standard but is not present in the tested code, this test fails and process continues to the next file.

The second portion of the process is to pass the newly generated code through a Java ANTLR parser. If the parser fails somewhere along the way, this test once again fails and process restarts using the next available file.

Standard files are replaced every time we have reached a milestone in the project, at which point the generated files are manually checked and replace the old standards. Even though this way of testing does not come close to being complete, it provides enough feedback to let us quickly highlight bugs in our generated code, while not allowing us to focus our research efforts on the continued development of the Umple software prototype.

6.3.3 Testing of the UML component

The testing of this component was the most simplistic. Rather than programmatically testing the presence of elements within the UML2.0 model we would be working with, the behaviour is simply visually inspected.

Visual inspection is the most appropriate way to check the generated class diagrams because it allows us to view all the visual properties of these diagrams. When dealing with class diagrams, features such as box size and layout are important to help visualize the system as well as convey the most important features of it quickly and efficiently.

6.3.4 UmpleRuntime

UmpleRuntime is a testing tool of the resulting generated code. This testing is performed on the Umple level of abstraction, giving the option to test on the level of Java code. This is made possible through Java reflection. This makes UmpleRuntime the only piece of Umple software which is bound to a particular language, as the rest of the software can be easily incorporated with other OO languages.

UmpleRuntime proves useful during regression testing because it allows for automated execution of commands on a system. The user creates the list of commands according to the structure of system being tested. If the code structure doesn't correspond to the desired UML structure, the test would fail and user is given feedback about the results. In this way, we can create, save, and load execution scripts for each system.

As mentioned in the previous section, our testing approach does not catch all possible defects in the code. Even if some code compiles and conforms to the expected code, there could still be semantic errors which were not caught in previous testing stage. UmpleRuntime is useful for highlighting these types of errors, in that it actually executes the code and observes the behaviour. This is done through insertion, deletion and manipulation of Umple generated objects at runtime. Semantic defects affect the behaviour of a system, and in the lowest levels, the behaviour of objects. The UmpleRuntime tool allows us to test the behaviour of these objects.

6.4 Umplepad

We have introduced Umplepad in order to maximize the usability of our tools. Umplepad is a combination of both Runtime and UmpleCore (components which are currently open-source). This stand-alone version of these components allows a user to pick up Umple very quickly without having to possess any knowledge about the delivery platform (Eclipse).

Umplepad is implemented as a rich client Eclipse platform. This means that a single distribution of Umplepad contains all components and plug-ins necessary to run UmpleCore and Runtime. Having an all-in-one distribution like this makes Umple much more appealing to those looking to quickly try the tools, because it does not require a lengthy set-up.

6.5 Evaluation of Umple and Umple Software – Case Study

As mentioned previously, Appendix 3 provides a set of example systems implemented using Umple. These examples, however, do not represent industry-grade applications, and are used more as Umple test cases. To further assess the quality of Umple, we used it to model a system of a much larger scale. For this purpose, we have developed an Airline example (which is larger than the one in Appendix 2) by using data obtained from Air Canada.

As there are many different types of systems being developed today, one needs a taxonomy of software which is used to categorize our case study. If we use Forward's software taxonomy [31], then this example would fall in the Data-dominant software domain. Characteristics of this domain are a fairly simple control flow over a large dataset.

This system was developed by reverse engineering the Air Canada flight schedule, available on the web in a pdf document. From this schedule we were able to determine the metamodel of classes and their relationships. This metamodel is shown in Figure 30.

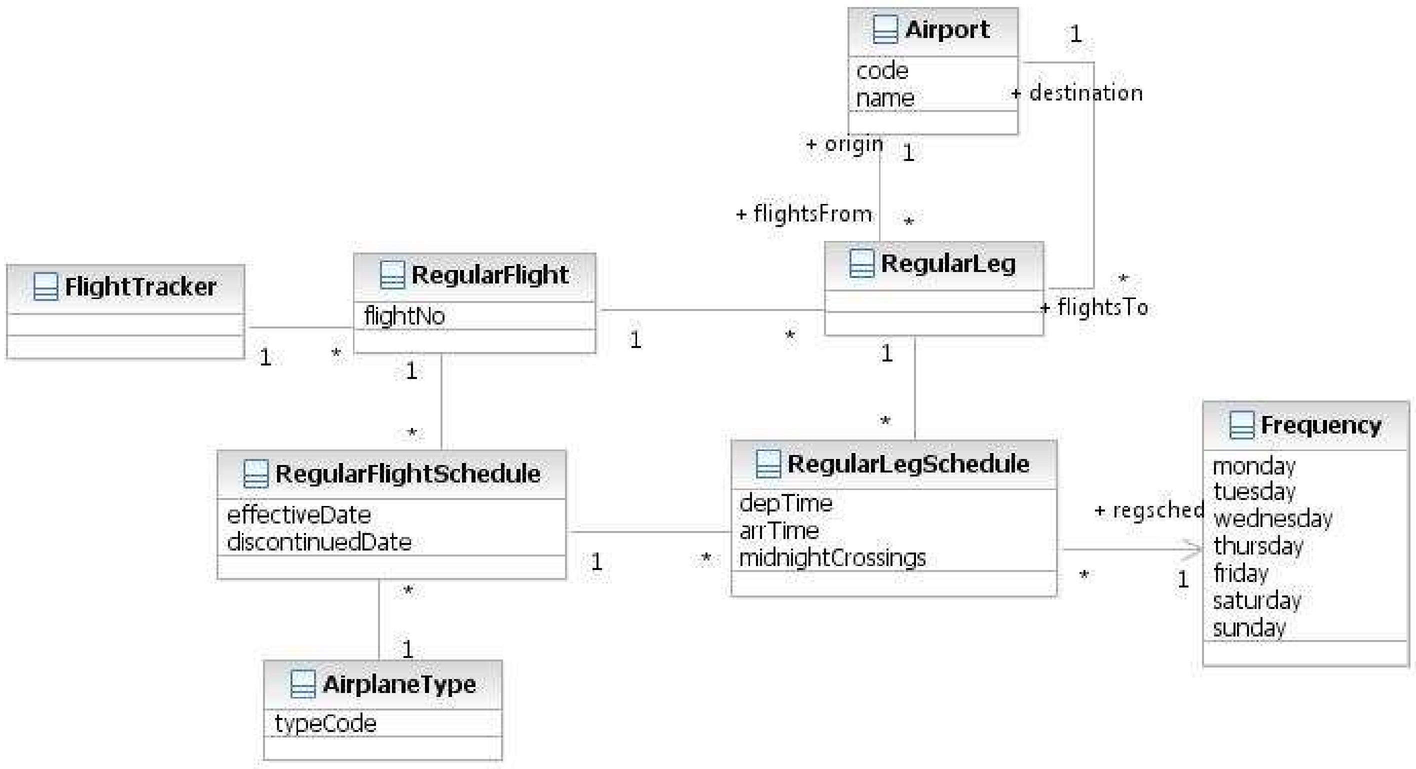


Figure 30: Airline case study metamodel.

This metamodel was coded in Umple and a system was generated accordingly. Most of Umple features were used in this case study, including the use of patterns and application logic. At this point, we have an airline system which is able to model the Air Canada schedule of flights.

UmpleRuntime was used to populate this model with real objects from the schedule. A simple script traversed the flights of the schedule and created corresponding UmpleRuntime commands. In this way, we were able to recreate over 9000 regular flights using our Umple generated airline system.

After the data is loaded using our UmpleRuntime tool, we are then able to manipulate the data as we wish. One of these ways is to recall and alter data of the created objects, effectively modifying the airline schedule. Another way is to query the system, in much the same way the real Air Canada airline system would be queried. One of such queries is to find all the flights from an origin to a destination airport which occur on specified days of the week. Another query is to view all flights between two airports, which make a stop in another airport. In fact, these queries, and several others were input to the system, as part of the class FlightTracker, using the application logic feature of Umple. These queries can be executed through UmpleRuntime or can be natively called by any system which interacts with our Umple generated airline.

There are, of course, differences between the real Air Canada airline system and our case study, most notably in the fact that Air Canada undoubtedly uses a database to store all the data, while UmpleRuntime created an in-memory representation of it. Also, our airline system is merely a representation of the schedule, and does not contain any algorithmic functionality for assigning and creating new flights based on the availability of resources. However, this example helps to prove the validity of Umple on a data-intensive system. For the full code, the original Air Canada schedule, and all other accompanying data please see Appendix 4 of this thesis.

7 Conclusions

The ideas and concepts presented in this thesis are part of an on going effort to unify modeling and implementation of a system. Even in the initial stages, it is easy to see that Umple has large potential which builds on top of the MDE and UML methodology. This is why there are many advantages which stem from the abstraction merge proposed by Umple.

7.1 Research Contributions

We have proposed several questions in Section 3. These were the research questions our thesis hoped to answer, or provide solutions to. This section briefly outlines our answers to these research questions.

RQ1: What are the advantages and disadvantages of adding abstractions found in UML to a Java-like programming language? This research question was only partially answered. We have mentioned the potential advantages in many parts of this thesis. The potential advantages include decrease in development time, and developers being able to produce, understand and maintain systems quicker due to the higher level of abstraction. However, these remain hypothetical advantages which need to be tested through empirical study. One of the answers offered to this question is that the Umple code is shorter than the corresponding Java code, which can be seen as an advantage. Some of the disadvantages include the lack of a formal standard for implementing these UML abstractions in object-oriented languages such as Java. Another disadvantage is that, at the current time, Umple imposes a relatively simple model on developers. This means that not every system which can be developed using Java can also be developed using Umple. However, planned future work will expand Umple and the supporting Umple tools to reduce, and eventually substantially diminish, this limitation.

RQ2: How can we bridge the gap between model and implementation, by making the Umple code represent both? The answer to this question lies in our design of the Umple language. Our language is mimetically compatible [17] with Java, which makes Umple feel familiar to users skilled with modern programming languages. Even the high level constructs inherited from UML that were added to Umple resemble Java constructs for better familiarity. Umple software also helps to bridge the gap between modeling and implementation by offering familiar syntax highlighting to users of UmpleCore. Lastly, integration into RSM allows for the user to work with both the diagrammatic UML and the Umple at the same time, thus bridging the gap. The ability to edit both diagram and Umple to manipulate the model will be part of future research.

RQ3: What are the advantages and disadvantages of text-based modeling? We have highlighted the advantages of text-based modeling in many areas of this thesis. One of the most prominent advantages is that a model of a system can be created much quicker than a UML drawing tool. Understanding such a model is also often easier in a textual environment, because of the inherent structure of text and code. One of the disadvantages we have found is that, much like in UML diagrams, information may become too cluttered if it is compacted too densely, which may cause loss of cohesion. Umple combats this through its many cohesion mechanisms. Once again, these points are mostly hypothetical and need to be tested empirically as part of future work.

RQ4: To what extent does the Umple approach simplify development? RQ5: To what extent can the Umple approach speed development? Even though we offer no formal empirical proof of the claims that Umple simplifies and hastens development, we theorize this based on the properties we have designed Umple to have, and by our case studies. Umple provides a simpler view of the system than the view offered from the point of view of the implementation code. This is because of the smaller number of lines of code and the fact that Umple operates on the level of UML. This higher level of abstraction, fewer lines of code and the general simplicity of the code make a system easier to maintain and develop. This ease of development then translates to the speed. Lastly, the fact that Umple code produces both implementation and UML models means that updating one results in changes in both, therefore saving time by combining these two tasks which would otherwise be done in sequence, or one of these would not be done at all.

RQ6: To what extent can the Umple approach assist with maintenance? In this context, the task of maintenance refers to understanding existing code, and making changes to it (further development) and accompanying documentation. If we think of UML class diagrams as being part of the accompanying documentation, then we have answered this question already. RQ1, RQ2, and RQ3 discussed above give us an answer as to how Umple helps with the understanding of existing code and it can be extended to include understanding code in terms of maintenance. Answers to questions RQ4 and RQ5 give us an idea of how Umple helps with the development part of maintenance.

RQ7: To what extent can we create a language that has the full power of both a programming and modelling language, such that it would be sufficient in order to produce a system? To achieve the full power of a programming language, we introduced the concept of application logic. This is logic which cannot yet be generated automatically and has to be specified by the user. Even though we provide an answer to this question in this thesis, we recognize that our solution is not the best one and is used only temporarily. The context of specifying action semantics and constraints is subject to much research, since many research groups are looking for ways to implement both OCL[26,27] and UML Action Semantics into programming languages. However, the current state of UML Action Semantics has been deemed as ”not the most practical approach” to specifying actions in a language such as Umple [28]. This research could be incorporated into Umple, as is touched on in Section 7.3.3.

7.2 Improving modeling languages

As Umple works so closely with UML, findings about Umple can introduce improvements and resolve ambiguities in UML, or could even spark the creation of new and better graphical modeling languages. Furthermore, the use of Umple promotes the use of modeling and UML which also promises further progress and improvements made in these areas.

One improvement which becomes apparent quickly is that UML should incorporate both a usable textual version, as well as its existing graphical version, taking advantage of mapping ideas put forth in this work. The textual notation has to be easy to use, learn, and understand for both novice and expert users, however this is difficult to accomplish using a generated language such as the ones created by HUTN, discussed in 2.8.2. Umple would be a much better candidate as textual representation of UML because it is more intuitive and custom tailored to UML, as opposed to generically generated from MOF.

7.3 Future work and possibilities

This thesis is the first section of the Umple effort. There is much work and possible research to be done in order to improve Umple and its functionality.

7.3.1 Expanding the Umple concept

To truly harness the power of Umple, it needs to be extended into other commonly used model types of UML such as state diagrams. Class diagrams cannot capture every detail one might want to model in a system. Umple will be much more powerful when it incorporates the ability to textually describe a wider range of UML diagrams, because it will allow one to describe and implement a system more completely using the modeling artefacts utilized by UML and Umple.

Modeling of state diagrams in code

One of the most common diagrams used by UML practitioners is the State diagram. State diagrams are very useful representations of a system, and can be incorporated easily with class diagrams both conceptually and technically. A quick and meaningful Umple representation of state diagrams combined with class diagrams can simplify creation, understanding, management and even maintenance of systems. This is due to the fact that even object oriented systems are already very state-based – the instance variables represent the types of state that can be found in each object.

Business process modeling with Umple

Another area where Umple would prove useful is in business process modeling. Bumple currently being developed by Andrew Forward in our research group, is an effort to bring the Umple approach into the business process world. This is facilitated by the observation that Business Process Modeling Notation (BPMN) and Business Process Execution Language (BPEL) are similar in concept to UML activity diagrams.

Activity diagrams could be modeled using Umple, which would result in a more complete system specification, where Bumple is used at the very high level to define processes and their interactions, while Umple, with class and state diagrams is used to implement these processes at a lower level. Each component of the system would be modeled using a branch of Umple, each of which offers the advantages outlined in this thesis. The system would then also inherit all the advantages one expects from Umple.

Concurrency modeling with Umple

Currently, there are no concurrency considerations built into Umple, but this is a feature which needs to be made part of Umple in the future. Concurrency could be modeled through activity diagrams or collaboration diagram notations. Furthermore, barring further research, sequence diagrams could also be used, even though their usability and understandability has been questioned [4, 24] for development of complete systems.

Incorporation of OCL to Umple

OCL is a language for describing constraints on UML models so a translation of this into Java-like notation would make a natural part of Umple. It would be used to specify invariants, pre-conditions, post-conditions, and other types of logic such as navigation expressions, the same way it is used by practitioners of UML.

7.3.2 Improving the current state of Umple software

As the Umple software is intended to be prototype giving us the proof of concept, we had to limit the scope of our work. One of the areas where Umple could be improved is the translation and mapping between the different types of multiplicities available in class diagram associations.

Umple focuses on the most common types of associations used, the ones which make use of 1-to-many or many-to-many multiplicities. UmpleCore was tested extensively using these two configurations. However, there are many other possibilities available to UML modelers, and so Umple needs to be able to handle these cases flawlessly. These possibilities include multiplicities of n and m cardinality, and the 1-to-1 multiplicity, which is often problematic and difficult to implement. The work of Andrew Forward includes a look at the different types of multiplicities and how they could be translated to working generated code, and so this work will be used to enhance Umple software and its capabilities.

7.3.3 Validating our approach

Umple needs to be tested on realistic large-scale applications, and by experienced developers. This is the best way for new languages and new paradigms to improve and evolve over time. Umple builds on time-tested ideas, and appears to be a logical advancement in programming methodology. Even so, Umple still needs developer buy-in to test both ideas and goals of Umple, as well as the software we have developed. There is much opportunity to test Umple in these ways as part of future research.

More robust business cases

Using Umple in real systems creates more robust business cases, and makes it easier to promote Umple through its past success. These business cases could be used to both improve Umple and introduce other developers, including ones who do not yet use modeling, to the MDE methodology. These business cases need to include systems from all areas of Forward’s taxonomy [31] if software systems.

Validation through the Open Source community

The open source community (OSC) has had many successes and many failures. We believe that the OSC would be very valuable when evaluating Umple, because of the large user base Umple could accumulate, as well as the technical expertise offered by the community.

Combating industry resistance (in applying this new paradigm)

Even though Umple builds on ideas and paradigms already in use, there is still an inherent developer resistance when it comes to new technology and methodology. The ultimate goal is to make users aware of the potential advantages of our approach so that they get excited about it and seek to both use and contribute to the project.

Using Umple as teaching tool

Lastly, Umple presents opportunities as a teaching tool. At the very basic level, Umple could be used to show students how UML concepts map to code. Umple generated systems are designed with simplicity in mind, and achieve the desired functionality while following good programming practices.

Furthermore, Umple could be used to demonstrate patterns. We already incorporated a few patterns into both the Umple language and resulting generated code, and there is opportunity to incorporate many more as part of future research. Besides the advantages Umple provides to students, student exposure to Umple would make it more possible that Umple will gain needed research and developer support in the future.

References

[1] Chris P. Gane, Trish Sarson, *Structured Systems Analysis: Tools and Techniques*. Prentice Hall Professional Technical Reference, 1979.

[2] JavaScript Object Notation, http://www.json.org/.

[3] Terence Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages.* ISBN:978-0-9787392-5-6, May 2007.

[4] Ritu Agarwal and Atish P. Sinha, O*bject-Oriented Modeling with UML: A Study of Developers' Perceptions*. Communications of the ACM, 248 September 2003/Vol. 46, No. 9.

[5] Edsger W. Dijkstra, *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, ISBN 0–387–90652–5, 1982

[6] Green UML, http://www.eclipseplugincentral.com/Web_Links-index-req-viewlink-cid-626.html

[7] UmlLet, http://www.eclipseplugincentral.com/Web_Links-index-req-viewlink-cid-492.html

[8] T. Dwayer, *Three dimensional UML using force directed layout.* ACM International Conference Proceeding Series, Vol. 16,2001

[9] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, Patrick Valduriez, *Model-based DLS Frameworks.* Dynamic Languages Symposium, 2006

[10] Jean Bézivin, *On the Unification Power of Models.* Software and System Modeling, SoSym Journal, 4(2):171-188, 2005

[11] OMG Model Driven Architecture Specification, http://www.omg.org/mda/specs.htm

[12] OMG Unified Modeling Language Specification V2.1.2, http://www.omg.org/technology/documents/formal/uml.htm

[13] OMG MetaObject Facility, http://www.omg.org/mof/

[14] OMG Human-Usable Textual Notation, http://www.omg.org/technology/documents/formal/hutn.htm

[15] James Skene, Wolfgang Emmerich, *Specifications, not meta-models*. International Conference on Software Engineering, 2006. http://doi.acm.org/10.1145/1138304.1138315

[16] Frederic Richard and Henry F. Ledgard, *A Reminder for Language Designers*. ACM SIGPLAN Notices, Vol. 12 No. 12 Pp 73-82. (December 1977)

[17] Linda McIver and Damian Conway, *Seven Deadly Sins of Introductory Programming Language Design*. In Proceedings, 1996 Conference on Software Engineering: Education and Practice, IEEE Computing Society Press, Los Alamitos, CA, USA. Pp 309-316. 1996

[18] Forward, A., and Lethbridge T. C., *Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals*. In Proceedings of MiSE'08 pages 27 - 32. Leipzig, Germany. May 2008

[19] R. France and B. Rumpe, *Model-driven Development of Complex Software: A Research Roadmap*. FoSE, pp. 27-54, ICSE 2007.

[20] Emfatic Wiki, EMFT, http://wiki.eclipse.org/Emfatic

[21] Emfatic Language Reference. Obtained from Emfatic Help of the Emfatic Eclipse plug-in, June 2008.

[22] Edsger W Dijkstra, *On the role of scientific thought*, *Selected writings on Computing: A Personal Perspective.* New York, NY, USA: Springer-Verlag New York, Inc., pp. 60–66, ISBN 0-387-90652-5. 1982

[23] Ed Merks and Dave Steinberg, *Models to Code with Eclipse Modeling Framework*. Eclipse CON 2005. http://www.eclipsecon.org/2005/tutorials.php

[24] Brian Dobing, Jeffrey Parsons, *How UML is used*. Communications of the ACM Volume 49, Issue 5 (May 2006). http://doi.acm.org/10.1145/1125944.1125949

[25] Timothy Lethbridge and Robert Laganiere, *Object-oriented Software Engineering: Practical Software Development Using Uml And Java*. ISBN 0-07-710908-2

[26] Bernhard Reus, Martin Wirsing, Rolf Hennicker, *A Hoare Calculus for Verifying Java Realizations of OCL-Constrained Design Models*. Fundamental Approaches to Software Engineering, Volume 2029/2001.

[27] Wojciech J. Dzidek, Lionel C. Briand, Yvan Labiche, *Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java*. Satellite Events at the MoDELS 2005 Conference, Volume 3844/2006.

[28] Claudius Heitz, Peter Thiemann, and Thomas Wölfle, *Integration of an Action Language Via UML Action Semantics*. Trends in Enterprise Application Architecture, Volume 4473/2007

[29] OMG Unified Modeling Language Specification (Action Semantics), http://www.omg.org/docs/ptc/02-01-09.pdf

[30] Kennedy Carter Ltd., http://www.kc.com/

[31] T. Clark, A. Evans, A. Moore, R. Venkatesh, T. Weigert, *Review of the Response to OMG RFP ad/98-11-01Action Semantics for the UML, Revised Submission.* Ad/01-03-01, dated: March 24, 2001

[32] Forward, A. and Lethbridge, *A Taxonomy of Software Types to Facilitate Search and Evidence-Based Software Engineering*. In Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, Toronto, Ontario, Canada, October 27, 2008, IBM Press, Toronto, Canada.

Appendices

Appendix 1: Umple syntax specification

(For complete ANTLR specification, please see Umple_ASTGrammar.g)

```
/*------------------------------------------------------------------
 * SYNTAX RULES
 *------------------------------------------------------------------*/

prog    :         (useStatement?)(item )* ;

useStatement:         'use' namespaceExpr;

item:   langStruct | namespaceDecl;

namespaceDecl:      'namespace' namespaceExpr;

namespaceExpr:      IDENTIFIER('.' IDENTIFIER)*;

langStruct
        : classStruct  | associationClass |association ;

classStruct
        :         'class' IDENTIFIER '{' (classContent)* '}';

classContent
        :         classItem | classStruct | appLogic;

appLogic:       codeFromUmpleFile;

classItem
        :         isARule
        |         varDecRule
        |         singletonRule
        |         implicitAssociationDeclaration
        ;

implicitAssociationDeclaration
        :         multiplicity implicitAssociationDirectionality multiplicity IDENTIFIER
(IDENTIFIER)? EOL;
```

```
implicitAssociationDirectionality
        :       ('--'|'->') ;

association
        :       'association' '{' (associationItem)* '}' ;

associationClass
        :       'association' IDENTIFIER '{' (associationClassItem)* '}';

associationClassItem
        :       classItem | associationItem;

associationItem
        :       associationLine ;
isARule :       'isA' IDENTIFIER EOL

singletonRule
        :       'singleton' EOL;

varDecRule
        :       regVarDec |  autoVarDec;

regVarDec
        :       (uniqueDec)?(attributeModifier)? (attributeType)? IDENTIFIER
(EQUALS value)? EOL;

uniqueDec    :       'unique';

attributeModifier
        :       'immutable'|'settable'|'internal';


autoVarDec
        :       'autoUnique' IDENTIFIER EOL;

attributeType
        :       'String'|'Time'|'Integer'|'Float'|'Date'|'Double'|'Boolean';

associationLine
        :       multiplicity IDENTIFIER (IDENTIFIER)?
implicitAssociationDirectionality multiplicity IDENTIFIER (IDENTIFIER)? EOL;

multiplicity:   (NUMBER ('..'!( NUMBER | '*'))?)|'*';

associationModifier::       'immutable'|'nonNavigable'|'internal'|'settable';
```

```
roleName:     IDENTIFIER;

value  :  (StringLiteral) | '""' | (NUMBER);

codeFromUmpleFile:        methodDeclaration;

methodDeclaration
        :         (modifier)* type IDENTIFIER '(' formalParameters ')'('throws'
IDENTIFIER (',' IDENTIFIER)*)? methodDeclaratorRest;

modifier:   'public'
    |  'protected'
    |  'private'
    |  'static'
    |  'abstract'
    |  'final'
    |  'native'
    |  'synchronized'
    |  'transient'
    |  'volatile'
    |  'strictfp'
    ;

type    :         primitiveType ('[' ']')*;

primitiveType
    :  'boolean'
    |  'char'
    |  'byte'
    |  'short'
    |  'int'
    |  'long'
    |  'float'
    |  'double'
    |     'void'
    |  IDENTIFIER
    |     attributeType;

formalParameters
    :   formalParameterDecls?;

formalParameterDecls
    :  ('final')* type formalParameterDeclsRest;

formalParameterDeclsRest
    :  variableDeclaratorId (',' formalParameterDecls)? |   '...' variableDeclaratorId;
```

```
variableDeclaratorId
    :   IDENTIFIER ('[' ']')* ;

methodDeclaratorRest
    :   formalParameters ('[' ']')* ( methodBody  |   ';' )  ;

methodBody:   block;

block: ('{'    (~('{'|'}') | block )* '}') ;


/*------------------------------------------------------------------
 * LEXER RULES
 *------------------------------------------------------------------*/

NUMBER      :          (DIGIT)+;

IDENTIFIER  :  LETTER (LETTER|DIGIT|'_')*;

StringLiteral
    :  '"'( EscapeSequence | ~('\\'|'"') )* '"'    ;

fragment EscapeSequence
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\') ;

fragment LETTER : ('a'..'z'|'A'..'Z');

fragment DIGIT: ('0'..'9');

WS     :  (' '|'\r'|'\t'|'\n');

COMMENT:   '/*' .* '*/';

LINE_COMMENT:    '//' ~('\n'|'\r')* '\r'? '\n';
```

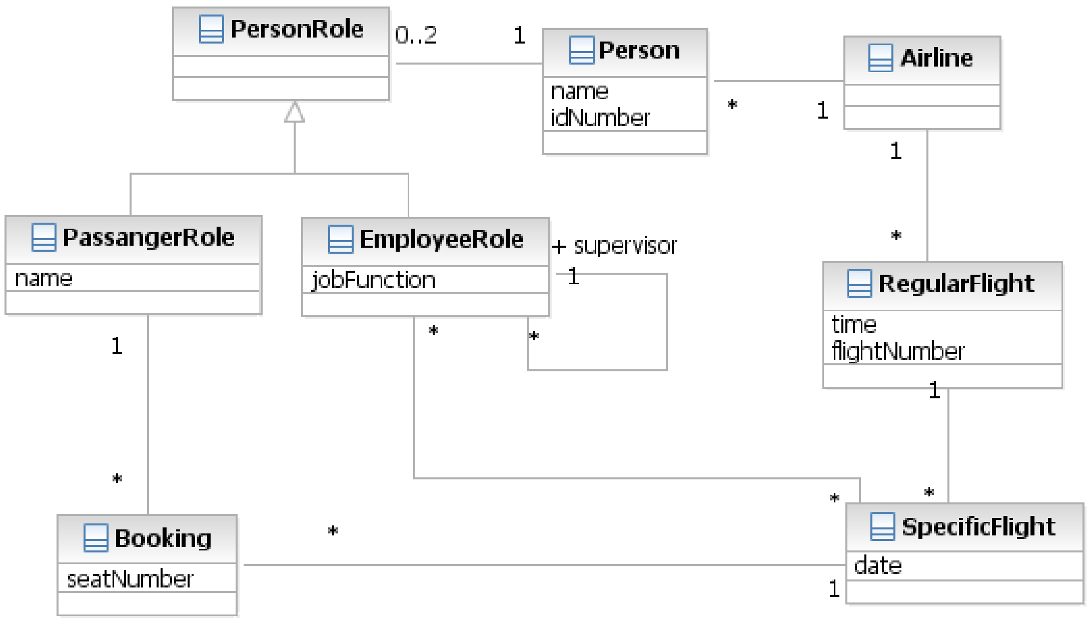
Appendix 2: Airline Example

This is an example showing much of Umple features and is used as a running example for this thesis. We refer to this example in many sections as it is useful in demonstrating the intended use of Umple. This example originates from Dr. Lethbridge's text book [25]. We have since augmented the example to fit our purpose.

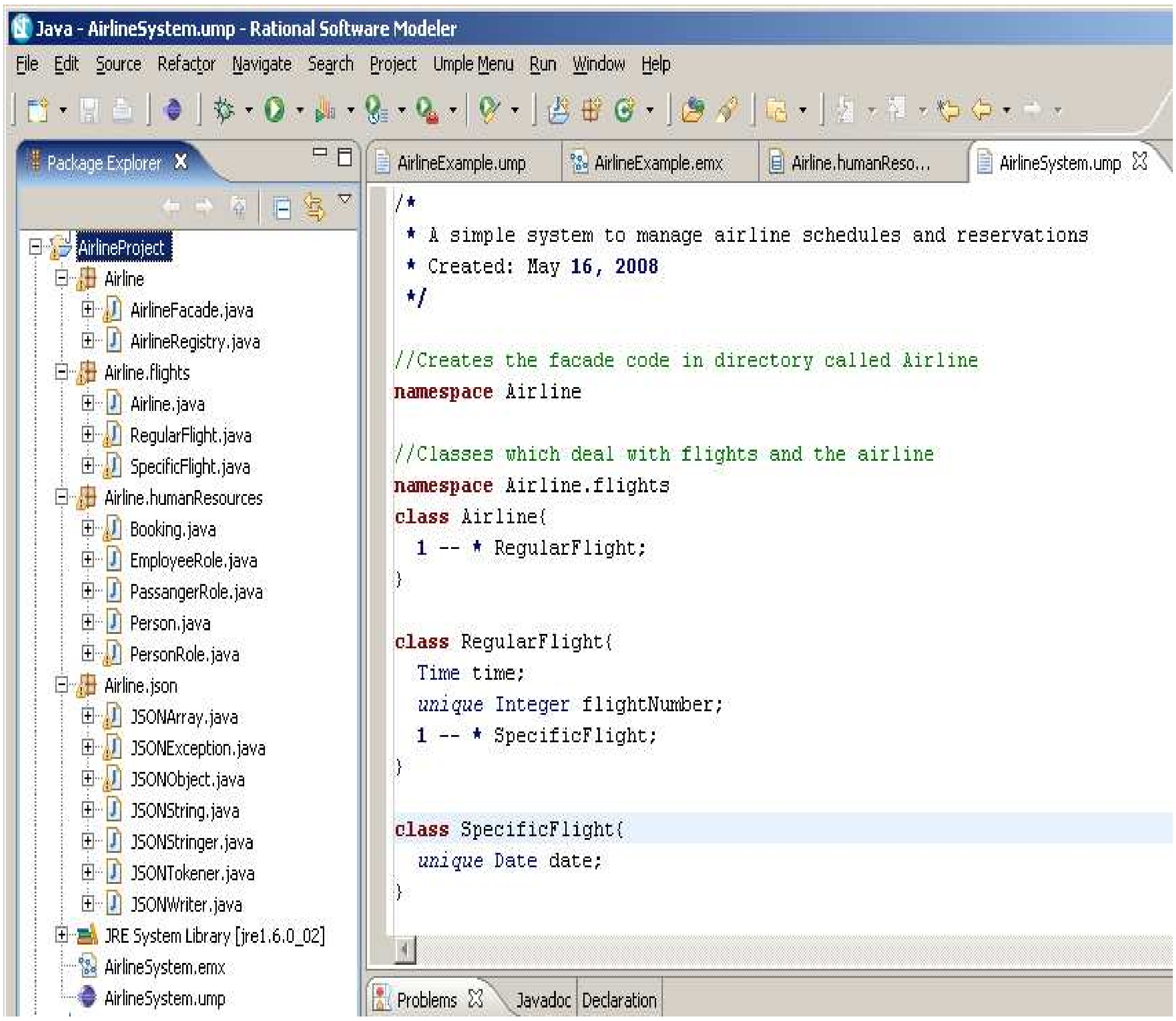
Umple code:

```
/*
 * A simple system to manage airline schedules and reservations
 * Created: May 16, 2008
 */

//Creates the facade code in directory called Airline
namespace Airline

//Classes which deal with flights and the airline
namespace Airline.flights
class Airline{
  1 -- * RegularFlight;
}

class RegularFlight{
  Time time;
  unique Integer flightNumber;
  1 -- * SpecificFlight;
}

class SpecificFlight{
  unique Date date;
}

//Classes which deal with people
namespace Airline.humanResources
class PassangerRole
{
  isA PersonRole;
  immutable String name ;
  1 -- * Booking;
}


class EmployeeRole
{
  String jobFunction ;
  isA PersonRole;
  * -- 1 EmployeeRole supervisor;
}

class Person
{
  settable String name;
  Integer idNumber;
  1 -- 0..2 PersonRole;
}

class PersonRole{}

class Booking{
  String seatNumber;
}

//Associations between subcomponents
association {
  * EmployeeRole -- * SpecificFlight;
}

association {
  * Booking -- 1 SpecificFlight;
}

association {
  1 Airline -- * Person;
}
```

Generated UML Class diagram:



Generated package structure and files:


Java - AirlineSystem.ump - Rational Software Modeler
File Edit Source Refactor Navigate Search Project Umple Menu Run Window Help
Package Explorer
AirlineExample.ump
AirlineExample.emx
Airline.humanReso...
AirlineSystem.ump
Problems
Javadoc
Declaration

```
AirlineProject
  Airline
    AirlineFacade.java
    AirlineRegistry.java
  Airline.flights
    Airline.java
    RegularFlight.java
    SpecificFlight.java
  Airline.humanResources
    Booking.java
    EmployeeRole.java
    PassangerRole.java
    Person.java
    PersonRole.java
  Airline.json
    JSONArray.java
    JSONException.java
    JSONObject.java
    JSONString.java
    JSONStringer.java
    JSONTokener.java
    JSONWriter.java
  JRE System Library [jre1.6.0_02]
  AirlineSystem.emx
  AirlineSystem.ump
```

```
/*
 * A simple system to manage airline schedules and reservations
 * Created: May 16, 2008
 */

//Creates the facade code in directory called Airline
namespace Airline

//Classes which deal with flights and the airline
namespace Airline.flights
class Airline{
  1 -- * RegularFlight;
}

class RegularFlight{
  Time time;
  unique Integer flightNumber;
  1 -- * SpecificFlight;
}

class SpecificFlight{
  unique Date date;
}
```

The code generated by UmpleCore will be available on

http://www.site.uottawa.ca/~tcl/gradtheses/dbrestovansky/.