Umple: Bridging the Code-Model Divide

Timothy C. Lethbridge, Andrew Forward, Dusan Brestovansky
School of Information Technology and Engineering, University of Ottawa
{tcl,aforward}@site.uottawa.ca, dbres042@gmail.com

Abstract

We present an approach to object-oriented software engineering intended to reduce the ever-present tension between model-centric and code-centric developers. The former prefer to model visually and generate code, while the latter see source code as the only important software artifact. The core of the Umple approach is to add UML abstractions such as associations directly into a high-level programming language – our current implementation works with Java. With Umple, the developer can work interchangeably with UML diagrams and Umple code; these are just views of the same thing. We describe the basics of Umple as well as the tools used to work with it; these include plugins for Eclipse and Rational Software Modeler. To demonstrate the benefits of the approach, we present as case studies a few systems we have developed in Umple, in particular the functional layers of airline reservation, elevator control and timesheet management systems.

1. Introduction

In this paper, we present Umple. Umple can be viewed from several perspectives. It can be seen as a programming language based on Java that incorporates UML constructs to raise the level of abstraction. It can be viewed as a tool for rapidly creating UML diagrams. Or it can be viewed as a tool to help broaden the appeal of modeling by allowing models to be created textually. In fact, it is all three of these.

In the next section we discuss our motivations for developing Umple. These include the lack of adoption of modeling technology caused in part by difficulties using modeling tools, and a desire to reduce the amount of code needed in certain object-oriented programming tasks.

Then we present various aspects of Umple: These include the overall philosophy, the concrete syntax, an overview of the semantics and the tools we have developed.

Finally, we present some case studies that demonstrate the value of the Umple approach.

2. Motivations

Our desire to develop Umple arose for two main reasons. We address each of these in the following subsections. Together, these motivations naturally led us to want to develop Umple.

2.1 Prevalence of the code-centric approach in software engineering: Lack of adoption of modeling

Many software engineering researchers and opinion leaders describe what they see as the obvious benefits of modeling. Principal among these is enabling developers to work at a higher level of abstraction [1]. However, when one interacts with developers in real companies, is clear that the overwhelming majority of developers see source code as their primary work medium, Most of them will draw diagrams when explaining concepts in design meetings, and include some of them in design documents, but relatively few use models to actually drive development. This is particularly true in the agile and open source communities [2].

2.1.1 A brief literature survey: There has been some research into why modeling is not more enthusiastically adopted. Afonso et al [3] state that although modeling is standard in the database design community (where use of Entity Relationship Diagrams is the norm), "there is little practical evidence of the impact" of model-driven approaches among the broader software engineering community.

There are clearly a number of common obstacles to modeling. Berenbach et al [4] suggest that these include a belief among developers that modeling is only about drawing "pretty pictures", and not understanding well enough how to model in the prevalent object-oriented paradigm.

Modeling is seemingly particularly important for safety-critical systems. Anda et al. [5] studied practitioners in this domain and note that they had good results when they adopted a more model-oriented approach. Difficulties using modeling tools and the costs of training were the biggest obstacles. Another obstacle was tendency of management to remain be oriented towards the production of source code.

The Object Management Group is the main industry association interested in promoting modeling. They supported Dobing and Parsons [6] in a survey in which 171 practitioners were asked how they use UML in practice. Most respondents felt that UML is indeed useful in software development, but half the said they did not fully understand class diagrams. The study concluded that complexity of UML is one of the main obstacles to its use.

One issue to consider is whether performing modeling has sufficient return on investment. Arisholm et al. [7] concluded from a controlled study that the costs of maintaining UML documentation may sometimes outweigh the benefits of modeling.

Another issue is the usability of modeling tools. Agerwal et al, [8], [9] studied this issue and found that poor usability contributes to higher than necessary costs associated with modeling.

Adoption is, of course, an issue with many tools and technologies. Sultan and Chan [10] provide and in-depth discussion of object-oriented technology adoption. They conclude that lack of adoption is likely not due to intrinsic weaknesses in the technology, but has more to do with culture and management.

2.1.2 Our own study of participants: We conducted our own study [11] of 113 software developers in a wide variety of industries and countries. We received responses that seem more positive that what is reported in the literature cited above.

Well over half of our respondents do in fact perform some type of modeling, with about 52% using UML often or always. Sixty percent use visual notations to document their code prior to design, although only a third *always* do this. However, only 17.5% often or always generate code from models and 36.5% never do this. Most of the value of models is therefore to document and communicate designs. Eighty percent in fact said that modeling tools are poor or awful at the task of generating all the code for a system.

We presented the respondents with a list development styles as follows a) *Model-only*: Approaches where the model is effectively all there is, except for small amounts of code. b) *Model-centric*: Approaches where modeling is performed first and code is generated from the model, for possible subsequent manual manipulation. c) *Model-as-design-aid*: Approaches where modeling is done for design purposes, but then code is written mostly by hand. d) *Model-as-documentation*: Approaches where modeling is done to outline or describe the system, largely after the code is written; and e) *Code-centric*: Approaches where modeling is almost entirely absent.

The respondents in our study felt that for corrective maintenance and developing *efficient* software the code-centric end of the spectrum would be better, however, they agreed that for almost all other tasks, including new development, adaptive maintenance, and program comprehension, model-centric approaches work best.

The respondents had three main criticisms of the model centric approach: 68% felt that it is a bad or terrible problem that models become out of date or inconsistent with the code; 52% complained about incompatibility among tools, and 39% complained about tools being too heavyweight.

On the other hand the respondents also had complaints about code-centric approaches: Two thirds complained about difficulties understanding the design or behaviour of the system based on code, and over half complained about code being difficult to change in general, as compared to models.

2.1.3 Personal experiences. We do a considerable amount of modeling and would like to be able to generate and modify UML diagrams very rapidly. Whether it be for teaching UML, illustrating books or papers, or developing actual systems, we have found both the commercial and open source tools slower at modeling than we would like.

2.1.4 Summary and links to Umple. We conclude from the above that the reasons for lack of more wholehearted adoption of modeling seem to be as follows:

a) Code generation doesn't work as well as it needs to;

b) Modeling tools are too difficult to use;

c) A culture of coding prevails and is hard to overcome;

d) There is a lack of understanding of modeling notations and technologies;

e) The code-centric approach works well enough, such that the return on investment of changing is marginal, yet the risks high.

The Umple approach addresses all these reasons. Point a) is addressed by the fact that Umple is a programming language, and a simple one. One of the main difficulties in generating code from a language like UML is that the semantics of UML were in fact explicitly designed to be for abstract modeling – it is

difficult to translate a model into code. Umple adopts key modeling features, but in its design we have chosen to err on the side of making it simple and usable for programming. It adopts Java semantics when these differ from UML semantics.

Umple addresses point b) by allowing modeling to be done using a simple text editor. Umple has a Java-like syntax so a syntax-directed editor can be used to help produce error-free code or models. At the same time, however, the Umple tools provide the capability to rendering Umple code as UML diagrams directly in IBM's Rational Software Modeler. The user therefore has the 'best of both worlds'.

Umple addresses point c) simply because it does not try to go against the prevalent coding culture. Umple allows you to keep coding, even though you are actually developing using some features that are at a modeling level of abstraction. If you are in the modeling culture, and want to work with full UML models in diagrammatic form, Umple does not stop you from take that approach either: You can use Umple code to generate or edit your diagrams.

Umple addresses point d) by only introducing the very simplest modeling concepts in its initial release. These include associations (with multiplicity), attributes, association classes and a few basic design patterns. This is intended to ease the transition for users of Umple.

Finally, Umple addresses point e) by providing a path to adoption that doesn't require a major investment.

2.2 Reducing the need to program boilerplate code.

In the last section we explained our first motivation for developing Umple: A desire to ease people's ability to model.

However our second major motivation is to ease people's ability to write object-oriented code.

Long before the advent of UML it was a commonplace object-oriented programming idiom for two classes to contain instance variables that reference the opposite class. The programming challenges included maintaining referential integrity, and deciding which class would take the prime responsibility for adding and deleting the references (links) between objects of the two classes.

With the coming of object-oriented modeling languages (e.g., OMT and later UML) these between-class references were modeled as *associations*. We will not describe the syntax or semantics of UML associations here, referring the reader to the UML specification [14], or one of the many books on the subject. e.g. [13].

Ultimately, however, association abstractions still have to be rendered into programming language code.

Unless one can use a UML tool that can generate the code for you, the first thing you need to do is to declare instance variables. In Java and other OO languages, the class opposite a '1' or '0..1' bidirectional association end would typically contain an instance variable with its type declared to be the other class. For a many association end (*), one of the collection classes would be used as in the instance variable's type. In C++ and in Java since version 1.5, genericity can be used to constrain the contents of this collection to be only objects of the 'other' class.

Next, an appropriate set of methods must be written to instantiate these variables, and allow bidirectional links to be established and changed as needed. Developing bug-free code to do this involves considerable work. However, the code ends up being very similar from association to association. It is called 'boilerplate' code because it is standard in nature, yet needs to be replicated in many parts of a system.

With Umple, this boilerplate code for associations is completely eliminated. Instead one declares associations and lets the compiler take care of the rest.

3. Description of Umple

The word 'Umple' is a play on words, meaning 'Simple', 'UML programming Language' and 'Ample'. Let us expand on these concepts a little:

3.1 Simple

Umple is intended to be simple from the programmer's perspective because, a) there is less code to write, and b) there are fewer degrees of freedom than either java or UML. Code that is eliminated includes boilerplate code for adding, deleting and modifying associations, as well as constructors and methods for accessing variables. In all these cases, and many others, Umple provides sensible default implementations.

An Umple program deliberately enforces a highly layered style of software. In particular it provides no user interface facilities other than a debugging mechanism. The result of compiling Umple code is the generation of a Java Archive file (JAR) containing an API that can be accessed by other layers, such as a user interface layer.

The generated API can be accessed either by Java code or through web services. Data passed back and forth through web services uses the JavaScript Object

Notation (JSON) format. JSON is considerably simpler than XML, and allows easy integration of Umple-based business logic with AJAX-style web applications.

3.2 UML programming language

Umple adds key features of UML to Java. In this paper we will focus on associations and attributes. As mentioned earlier, Umple can be used to generate UML class diagrams, or alternatively a class diagram can be rendered straightforwardly into Umple. We are also adding state machines to Umple, but we will not discuss that feature in the current paper since it is less mature, and because space is limited.

3.3 Ample

Despite the restrictions imposed by the deliberate simplicity of Umple, it is intended to have sufficient power to program the functional layer of most kinds of system. As we will describe later, we have used it for several moderately-sized applications including an airline scheduling system and an elevator control system.

3.4 Motivating example

We will provide the following motivating example. This example shows how one would declare attributes and associations, if one were just starting the process of modeling a system.

```
class Student {}

class Registration {
  String grade;
  * -- 1 Student;
  * -- 1 CourseSection;
}

class CourseSection {}
```

Immediately after writing the above Umple code, we can generate a class diagram of this tiny system. This is shown in Figure 1. We can also compile the system; the result would be an executable Java archive (JAR) file with an API that allows us to create instances of the classes, as well as add and delete links of the associations. We could also plug the result into a web server, and the API would be available as web services.

As the above example shows, the basic declaration of a class follows the syntactic style of Java or C++.

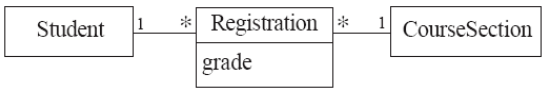

Figure 1: UML class diagram for part of a student registration system (from [13])

Let us now look at the other syntactic elements illustrated above.

3.5 Attributes

The declaration of attributes looks very much like the declaration of instance variables in Java. However, there are some important differences. Firstly the set of primitive data types is different. In the initial version we allow Integer, Float, Date, Time, String, Boolean and Enum. This set was chosen to cover most basic modeling needs, without having to deal with the complexities of 'primitive' vs. 'class' datatypes. The Integer type will in fact generate 'int' code when compiled in Java, but we want to insulate the user from that. The set of attribute types is also inspired by those available in relational databases.

It is possible to leave an attribute untyped. This can be useful when you simply want to use Umple to quickly generate a class diagram. The default datatype is String, so you still can compile an Umple file that has untyped attributes. This concept of allowing information to be omitted follows the UML convention.

The presence of an attribute generally means the following:

- The generated API will have methods getX() and setX(), where X is the attribute name. So in the above example, there would be setGrade() and getGrade() methods.
- Code in Umple methods (discussed later) will be able to use the attributes on the left and right of an assignment, and in method arguments.

There are additional notations relating to attributes that an Umple programmer can use for common programming situations. We will only outline these briefly here. Each of these keywords is placed before the data type.

The following four are mutually exclusive

immutable: The attribute must be set in the constructor and cannot then be set again, so there is no setX method. An entire class can be declared immutable, as discussed later.

key: Indicates that this attribute is to be part of the unique key. Uniqueness of keys is enforced, and an exception is thrown of the uniqueness is violated in an

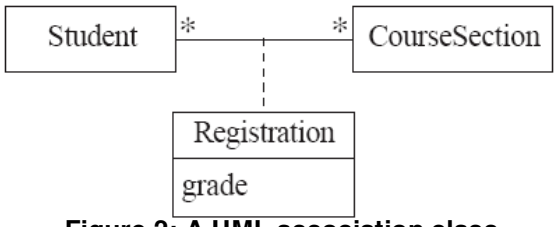

Figure 2: A UML association class

attempt to instantiate an object. Key implies immutable.

autounique specified on an Integer attribute indicates that a new integer unique within the class will be issued each time an object is instantiated; no argument for the attribute will appear in the constructor.

settable: The attribute can be set by the setX method, but only once.

As in java, an attribute can be given an initial value. For example

String grade = "INC";

Finally, by default, access to attributes is public in the Java sense. The keyword **private** prevents the attribute from appearing on the API for external systems to use. Private attributes are also not transmitted over the API.

3.6 Associations

Associations are the key feature that makes the first version of Umple interesting. The declaration of an association is designed to look visually similar to how it would look in a UML class diagram.

In the above example, and in Figure 1, we see that there is a many-to-one association between Registration and Student.

Declarations of associations can appear in two places in Umple. They can appear inside one of the two associated classes, or else as standalone 'first class' associations.

The basic syntax for an inline association is

<mult1> {<role1>} <direct> {<asname>} <mult2> <class2> {<role2>};

<mult1> and <mult2> follow the standard UML syntax for multiplicity, so typical values are 0..1, 1, 1..* and *.

<role1> and <role2> are role names following the UML convention. They should be nouns that represent the class in a certain context. For example the code for Figure 1 could be enhanced thus:

* -- 1 registrant Student

Registrant is a role name that would appear on the association when the diagram is rendered visually.

<asname> is an optional association name following the UML convention. We find that we use this relatively rarely, preferring to use role names.

<direct> is one of '--' for a bidirectional association, '->' or '<-', the latter two meaning that navigation is limited to the direction given by the arrow. The above code uses only bidirectional associations.

Class Registration above has two inline associations. Both of these associations could have instead been placed in the second class, so for example in Student we could have had

1 -- * Registration;

This association could have been placed on its own as a standalone association as follows:

```
association {
   1 Student -- * Registration; }
```

The general syntax for standalone associations is:

<mult1> <class1> {<role1>} <direct> {<asname>} <mult2> <class2> {<role2>};

The only difference between this and the syntax for inline associations that the additional class must appear. For inline associations, <class1> is implicitly the containing class.

Finally Umple supports UML's notion of association classes. These use the notation shown in Figure 2.

The code to generate Figure 2 is as follows:

```
class Student {}

association Registration {
  String grade;
  * Student -- * CourseSection;
}

class CourseSection {}
```

Those familiar with UML will recognize that the functionality embodied in Figures 1 and 2 is essentially the same.

3.7 Generalizations

There are two approaches to create a generalization relationship in Umple. One can simple add an expression in the subclass following the syntax:

```
isa <superclass>;
```

Or one can embed the subclass definition inside the definition of the superclass. For example, the following are equivalent:

```
class Manager {
  isa Employee;}
class employee {}

class Manager {
  class Employee {}
}
```

The embedded form is nice in that the inheritance hierarchy appears visually as increasing levels of indentation.

3.8 Other annotations that can be added to Umple code.

We will not attempt to describe all the features of Umple, but the following are some of the more interesting.

A class can be declared a singleton. This ensures that only a single instance can be created, and eliminates the need for the standard boilerplate code to implement the singleton design pattern.

A class can also be declared immutable, meaning that all attributes must be provided in the constructor, and there will be no setX methods.

A class can be declared as a façade. This means that the only methods available through the generated API will be the ones that appear in this class.

The syntax namespace <name> can be used to organize code into packages.

3.9 Java-like code for methods

Methods in Umple look very much like Java methods. In fact, the compiler relies on the Java parser to convert them into bytecode. However there are certain restrictions placed on the code in a pre-processing step:

Umple methods can only do the following:

a) Read and write attributes. Direct attribute access is converted to calls to the appropriate setX and getX methods. Rules about immutability are enforced.

b) Navigate associations where the other end is '1' in the same manner as accessing attributes.

b) Navigate 'many' associations by calling a built-in method to obtain a List iterator.

c) Instantiate objects, destroy objects, and add, delete and update links of associations by calling methods generated for this purpose. Referential integrity is automatically maintained.

d) Call methods in this and immediately neighboring classes. We enforce the Law of Demeter [15] to improve the maintainability of the resulting system.

e) Perform normal Java computations with local variables declared using limited set of data types.

Umple methods can be placed inline in the class, or can be written in separate files that can be merged into several classes. This provides a Ruby-like mixin capability.

A number of important rules are enforced when Umple works with associations. The full set of rules flows naturally from the semantics of associations. For example:

• Creating a new object of a class (called a 'driver' class) that has '1' associations to other classes (called 'subordinate' classes), implies that instances of the subordinate classes must be created simultaneously. The constructor will ensure this takes place. This effect can cascade to further subordinates.

• Deleting an object of such a driver class will result in destroying the subordinate objects. This effect can also cascade.

• Exceptions are thrown if attempts are made to violate multiplicity.

3.10 Umple as a language family

In the above discussion, we have presented the version of Umple that incorporates Java methods and follows Umple syntax.

However the important thing about Umple is the concept of adding UML associations and attributes to a programming language. We are working on doing the same thing with Ruby. When we discussing the Java-specific flavor of Umple, we use the term Jumple, and when we are discussing the Ruby-specific flavor, we use the term Rumple.

We have also created a member of the Umple family called Bumple. This applies many of the same concepts to Business Process Execution Language (BPEL) instead of UML.

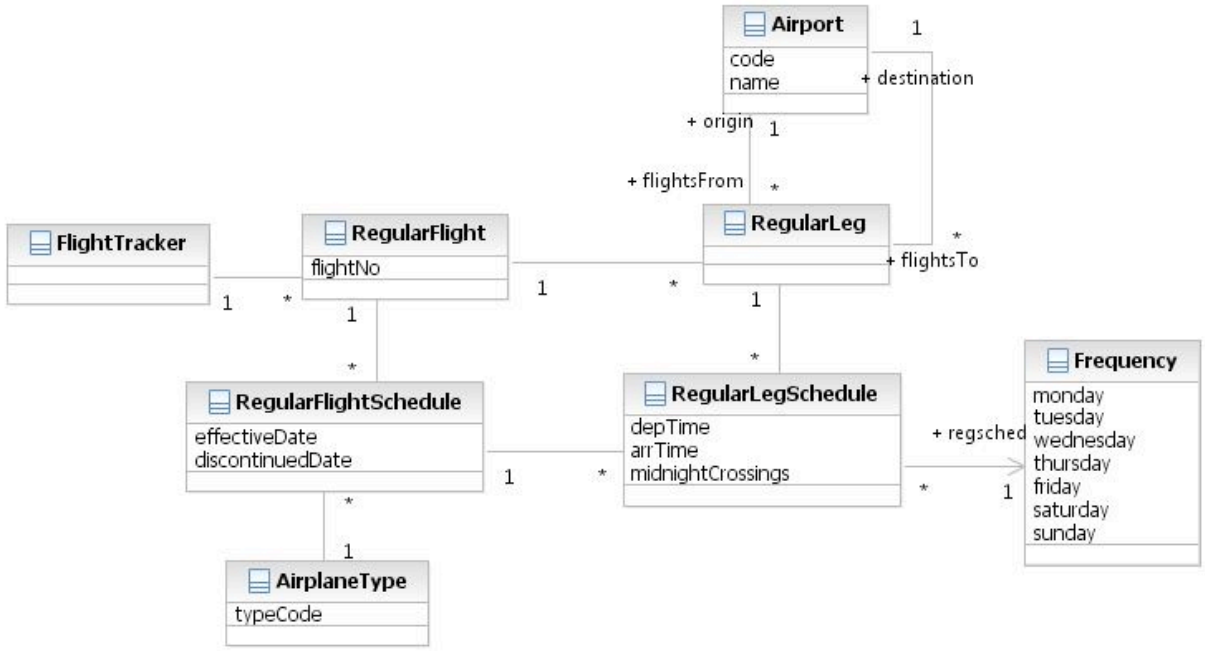

Figure 3: UML class diagram for the reverse engineered airline scheduling system

In the remainder of the paper our references to Umple imply Jumple.

4. Using Umple

The tooling we have built for Umple consists of the following:

• A standalone compiler (UmpleCore).

• A plugin to IBM Rational Software Modeler that allows one to generate and work with UML class diagrams.

• A runtime environment that allows one to navigate Umple and Java objects, and to call the methods in the API.

• A simple environment that combines the compiler and the runtime environment.

The ensemble of components is illustrated in Figure 4.

5. Case studies

We have created a significant body of Umple code to act as a test suite. In this section, we want to present three interesting cases.

5.1 Airline system

The first case study is of an airline system. We went to Air Canada's website and downloaded the pdf file of their schedule, which includes code-shared flights from star alliance and affiliated airlines. We processed the pdf file to extract the essential data (using Excel Macros); we then reverse-engineered what the schema must look like.

The result of our reverse engineering effort was the creation of the following code. Then we ran the code through the Umple plugin of RSM to generate the diagram, which appears in Figure 3.

```
class FlightTracker {
  singleton;
  1 -- * RegularFlight;
}

class RegularFlight {
  Integer flightNo;
  1 -- * RegularLeg;
  1 -- * RegularFlightSchedule;
}

class RegularLeg {
  * flightsTo -- 1 Airport destination;
  * flightsFrom -- 1 Airport origin;
  1 -- * RegularLegSchedule;
}

class RegularFlightSchedule {
  Date effectiveDate;
  Date discontinuedDate;
  1 -- * RegularLegSchedule;
}
```

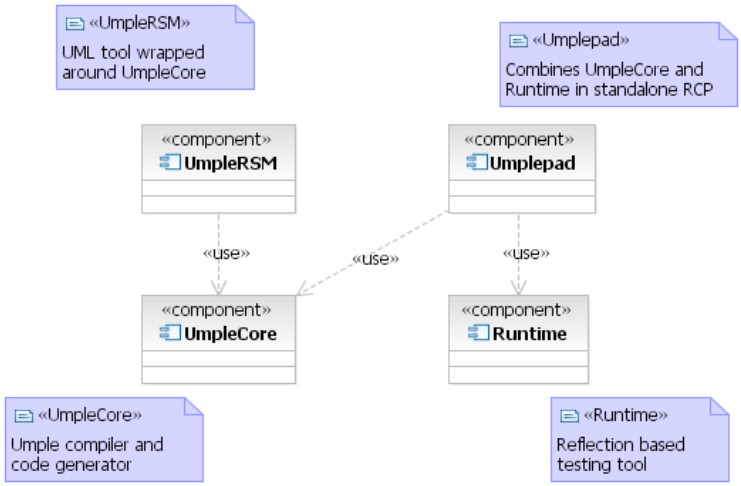

Figure 4: Umple components

```
class RegularLegSchedule {
  Time depTime;
  Time arrTime;
  Integer midnightCrossings;
  * -> 1 Frequency regsched;
}

class AirplaneType {
  // e.g. "747"
  String typeCode;
  1 -- * RegularFlightSchedule;
}

class Airport {
  String code;
  String name;
}

class Frequency {
  // Days it operates
  Boolean Monday;
  Boolean Tuesday;
  Boolean Wednesday;
  Boolean Thursday;
  Boolean Friday;
  Boolean Saturday;
  Boolean Sunday;
}
```

Next we wrote a simple Java program to load all 10,000 objects into Umple-generated system and perform some queries that navigated the associations to search for flights matching various criteria.

The purpose of this case study was to demonstrate that Umple can be used in a data reverse engineering context and that the resulting Umple-generated system can be put to serious use for rapid prototyping and testing of business logic.

5.2 Timesheet management system

This case study involved re-creating the business logic of commercial timesheet management system. In a matter of a couple of hours, we were able to write 329 lines of Umple code for the business logic of the system merely by examining through its user interface. The resulting system contains over 30 classes, over 30 associations, several generalizations and numerous attributes.

We were able to write virtually error-free code, demonstrating that Umple is easy use for programming. The only errors we encountered were that we needed to specify role names in one situation where there were two otherwise-identical associations between two classes.

The resulting Java code generated by the Umple compiler consists of 9224 lines! Most of that is boilerplate code needed for manipulating and navigating the associations.

5.3 Elevator simulation

In order to demonstrate that Umple is not just for data processing, we created a simulation for multiple banks of elevators in a complex building.

We used Umple code to model the classes and associations, and then added methods to complete what is necessary to allow the elevators to respond to events such as buttons being pressed, arriving at a floor, a door opening etc.

We connected the generated system to a separately-written user interface that showed animated elevators.

6. Conclusions

Umple is a programming language that incorporates UML concepts. It is also an environment to assist with creation of UML models textually.

We created Umple to respond to two needs: The first is the resistance to modeling prevalent in industry. The second is the desire to eliminate boilerplate code and thus simplify some aspects of object-oriented programming.

We believe that a language like Umple can help bridge the gap between model-centric developers and code-centric developers because it allows both to continue to do what they prefer, while also giving them the benefits of the alternative approach.

In the current version of Umple we have focused on implementing associations, attributes and a few design patterns. We intend to expand Umple by also adding state machines to the language.

We have created numerous testcases as part of the test-driven development process we used to create Umple. However, in this paper we highlighted three of them. These case studies suggest that Umple is easy to program with both rapidity and in an error-free manner. The case studies also show that it can be used to in tasks such as generating complex class diagrams, reverse engineering systems, and creating new systems. The Umple code can have in an extreme case less than 4% of the number of lines of code that corresponding Java code needs to have.

10. References

[1] L. Lavaggno, G. Martin, G., and B. Selic, eds., *UML for Real: Design of Embedded Real-Time Systems*, Springer, 2004

[2] G. Hertel, S. Niedner, and S. Herrmann, "Motivation of software developers in Open Source projects: an Internet based survey of contributors to the Linux kernel", *Research Policy* 32, 2003, pp. 1159-1177.

[3] M. Afonso, R. Vogel, and J. Teixeira, "From code centric to model centric software engineering: practical case study of MDD infusion in a systems integration company", *Int. Wkshps. on Model-Based Development of Computer-Based Systems and Model-Based Methodologies for Pervasive and Embedded Software*, 2006, IEEE Computer Society, 10 pp.

[4] B. Berenbach, and S. Konrad, "Putting the "Engineering" into Software Engineering with Models", *Int. Workshop on Modeling in Software Engineering (MISE'07),* IEEE Computer Society, 2007, pp 4-4

[5] B. Anda, K. Hansen, I. Gullesen, and H.K. Thorsen, "Experiences from introducing UML-based development in a large safety-critical project", *Empirical Software Engineering*, 11, 4, Dec. 2006 pp 555-581

[6] B. Dobing, and J. Parsons, "How UML is Used", *CACM*, 49, 5, May 2006, pp. 109-114.

[7] E. Arisholm, L.C. Briand, S.E. Hove, and Y. LaBiche, "The impact of UML documentation on software maintenance: an experimental evaluation", *IEEE Trans. Softw. Engg.,* 32, 6, June 2006, pp. 365-381.

[8] R. Agarwal, P. De, A.P. Sinha, and M. Tanniru, "On the usability of OO representations". *CACM* 43, 10, Oct. 2000, pp. 83-89.

[9] R. Agarwal and A.P. Sinha, "Object-oriented modeling with UML: a study of developers' perceptions", *CACM* 46, 9 Sep. 2003, pp. 248-256.

[10] F. Sultan and L. Chan, "The adoption of new technology: the case of object-oriented computing in software companies", IEEE Trans. Engg. Mgmt. 47, 1, 2000, pp. 106-126.

[11] A. Forward, *Perceptions of Software Modeling: A Survey of Software Practitioners*, technical report, University of Ottawa, 2007. http://www.site.uottawa.ca/~tcl/gradtheses/aforwardphd/

[12] D. Brestovansky, "Exploring Textual Modeling using the Umple Language", Masters thesis in Computer Science, University of Ottawa, http://www.site.uottawa.ca/~tcl/gradtheses/dbrestovansky/

[13] T.C. Lethbridge and R. Laganière, *Object-Oriented Software Engineering: Practical Software Development using UML and Java*, McGraw Hill, 2001.

[14] Object Management Group, Unified Modeling Language (UML), version 2.1.2, visited Sept 5, 2008, http://www.omg.org/technology/documents/formal/uml.htm

[15] K. J. Lienberherr, "Formulations and benefits of the law of Demeter", *ACM SIGPLAN Notices*, 24, 3, March 1989, pp. 67-78.