Umplification: Refactoring to Incrementally Add Abstraction to a Program

Timothy C. Lethbridge, Andrew Forward, Omar Badreddin
School of Information Technology and Engineering, University of Ottawa, Canada
{tcl,aforward,obadr024}@site.uottawa.ca

Abstract

Umple adds UML abstractions to a base programming language. The resulting program can be seen as both model and code at the same time. Base languages currently supported include Java, PHP, and Ruby. Umplification is the process of converting a base language program to Umple by a series of refactorings. The result is a program that is semantically equivalent to the original, but which can be rendered and edited as UML diagrams by any modeling tool. Yet, it can still be manipulated as a textual program for those who prefer. In this paper we discuss the basic principles of Umple, the general process of umplification, and our experiences performing it in real contexts, including umplification of the Umple compiler itself.

1. Introduction

Our research has been motivated by our survey [1] indicating that adoption of modeling is limited by weaknesses in usability and code generation.

We have a broad vision for the technology described in this paper: Firstly, our approach makes it possible to incrementally refactor existing programs so that the code becomes more abstract, and also so that such programs can be directly maintained using visual modeling tools. Secondly, our approach eliminates the artificial distinction between modeling and programming, allowing both to be interchangeable for the purposes of new development and maintenance.

We call our technology *Umple*. This is a play on words, combining 'UML Programming Language', 'Simple' and 'Ample'. We call the process of incrementally converting a program to Umple, *umplification*. This plays on the words 'amplification' and 'simplification' since by umplification you are indeed amplifying the abstraction level of the program and simplifying the code too.

This paper is organized as follows. In Section 2, we describe Umple by example. In Section 3, we explain Umple's key philosophies. In Section 4, we describe the basics of umplification and give a small artificial example; and in Section 5, we discuss three experiences with umplification, including umpifying Umple itself. Section 6 concludes the paper.

2. Essence and Examples of Umple

Umple adds a set of UML-derived features to several object-oriented programming languages. The features enhance each language's level of abstraction, allowing the programmer to express common concepts more succinctly.

So far we have applied Umple to Java, PHP and Ruby, and we are working on C++. We refer to these as the 'base' languages, and when referring to one directly, we use the term language L. In this paper we normally refer to Umple in a pure sense, but if we need to refer to language L enhanced with Umple features, we use the notation Umple/L. An Umple/L compiler will generate a program in L, with L serving as an intermediate language. After generating the program in L, the Umple compiler will then call the L compiler.

We do not provide the full specification of Umple here. For full details, the reader should refer to [2-5]. However, the following are some simple examples.

A UML association can be represented in either of its end classes, or independently. Figure 1 shows a UML class diagram with a one-to-many association. The following are two semantically equivalent ways of writing this in Umple.

In this paper, Umple code has a grey background and base language code has a white background.

```
class A {
   1 -- * B;}
class B {}
```

```
class A {}
class B {}
association {
   1 A -- * B;}
```

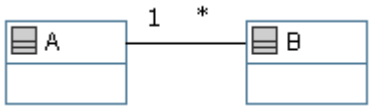

Figure 1: simple one-to-many association

Note that any UML multiplicity can be used at either end of an association, and navigability in only one direction can be indicated using -> or <- instead of --. Having specified the above, the programmer can then manipulate the association in the base language by calling the generated API. The following is be the API currently generated from Umple/Java in class A

```
public B getB(int index)
public List<B> getBs() /* unmodifiable */
public int numberOfBs()
public boolean hasBs()
public int indexOfB(B aB)
public B addB() /* creates new B */
public boolean addB(B aB)
public boolean removeB(B aB)
```

The following is the API in class B:

```
public A getA()
public boolean setA(A aA)
public void delete()
```

A constructor is also generated. Multiplicity and referential integrity constraints are always respected by the generated code. Note that different types of associations will generate different code.

A state machine with two states and events to transition back and forth between states can be modeled in Umple as follows.

```
class A {
  sm {
    S1 {
      e1 -> S2;}
    S2 {
      e2 -> S1;}
  }
}
```

The equivalent UML is shown in Figure 2.

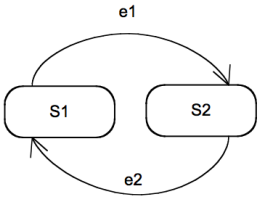

Figure 2: A simple state machine

The programmer can change and query the state using the following datatype:

```
enum Sm { S1, S2 }
```

and the following API:

```
public Sm getSm()
public boolean e1()
public boolean e2()
```

Umple state machines support features such as transition actions, entry actions, exit actions, interruptible do activities and nested states.

In addition to the associations and state machines illustrated above, Umple supports numerous other concepts. These include a rich semantics for attributes, including the notion of keys, read-only access, and defaults. Umple supports before and after code injections that allow specification of constraints and triggers in the base language; these are activated when attributes, associations and states are set or queried. Finally, Umple incorporates the notion of mixins, allowing features to be specified separately, and then combined into a final program.

3. Philosophy of Umple

The following are some key philosophies that have guided our design decisions regarding Umple:

P1. Modeling is programming and vice versa: UML concepts can be expressed textually in Umple; hence, one can model in UML using Umple. For a programmer, Umple looks like a programming language, therefore, to such a person they are just programming more abstractly.

P2. An Umple compiler can accept and generate code that uses nothing but UML abstractions. The resulting executable will be a module providing an API rather than a complete program, since it will lack a 'main' method and algorithmic methods. P2 is a corollary of P1.

P3. A program without Umple features can be compiled by an Umple compiler. This is the inverse of P2. In other words, any program P in base language L compiled by an Umple/L compiler will generate P. This provides a convenient starting point for a programmer who wants to begin using Umple incrementally: They can just change from using an L compiler to an Umple/L compiler.

P4. A programmer can incrementally add Umple features to an existing program. This allows for iterative conversion of a base-language L program into Umple/L, with each step being a straightforward refactoring. We call this process *umplification*. Discussion of this forms the core material of this paper, starting in Section 4.

P5. Umple features can be created and viewed diagrammatically or textually. One can use a UML diagramming tool to generate an Umple program that contains only UML abstractions, as in P2. Similarly, since Umple features map directly to UML, one can easily render any Umple program as a UML diagram. This can be done in real-time, as demonstrated in UmpleOnline [6]; a web-enabled Umple editor that supports Umple textual code, as well as UML class visualizations. Elements of code that are not Umple abstractions, such as the bodies of methods, are omitted from the diagram.

P6. Much of the base language code in an Umple/L program corresponds to UML's concept of an action language. To use UML for model-driven development, i.e. complete generation of an application from a UML model, UML calls for the use of an action language for the algorithmic details [7]. UML diagrammatic modeling tools currently allow snippets of action language to be specified for elements such as state machine actions. Users of these tools have to switch between the visual editor for manipulating the modeling elements, and the textual editor for manipulating the action language snippets. This context switching from diagram to text makes it difficult to understand an entire program.

P7. Umple extends the base language in a minimally invasive way. A programmer familiar with language L should see the addition of an Umple feature as just a natural extension of L. In other words, Umple appears harmonious with L. As an example, for C-family languages this is accomplished by co-opting the curly-bracket block idiom and adding a very small number of additional keywords.

P8. Umple goes beyond UML as needed to directly implement patterns and other common programming idioms. Umple can, for example, generate code for the singleton pattern [8]. Such capabilities further increase Umple's level of abstraction.

P9. An Umple programmer should never need to edit generated code to accomplish any task. The need for *round-tripping* (editing generated code and then reflecting the edits back into the model) common in the Model Driven Engineering world [9] is not needed.

4. Umplifying Base Language Code

The process for converting a base language program to an Umple program involves a set of refactorings. We have coined the term *umplification* to describe these refactorings as a set. The umplification process is similar to what any reverse engineering tool would perform when generating a UML diagram from a program. The key difference is that the end-product of umplification is not a separate model, but is an incremental change to the code/model.

The umplification process is illustrated in Figure 2. The source files with from language L code are initially just renamed as Umple files, with the extension .ump. At this point, the source code of L and Umple/L source text are identical.

The Umple code can be edited by any text editor, However, to gain the power of modeling it can be loaded into a modeling tool such as our Eclipse plugin or UmpleOnline.

The input Umple code is parsed by such a tool and at this point is represented internally in the tool in a manner that allows rendering and editing both in text and diagram form. Edits performed in one view are automatically reflected in the other view as they both represent the same underlying artifact.

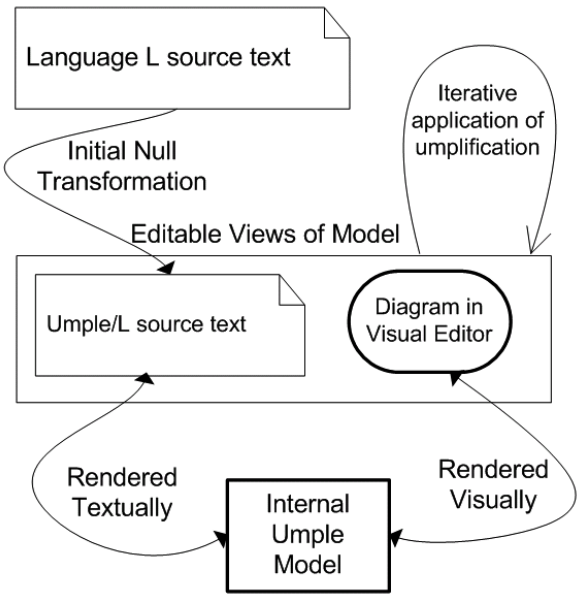

Figure 2: Refactoring Approach

To perform umplification, one iteratively makes small transformations that gradually add UML abstractions while preserving the semantics of the program. In other words, one performs a series of refactorings. Before making any of these transformations, the diagram in the visual editor will only show the names of the classes. As the Umple/L text is edited (currently manually) to convert sections of code into UML abstractions, such as associations, these appear in the visual editor too. Once the abstractions appear in the visual editor, they can then be edited in either editor. (New UML can be added in either textual or graphical editors, but that would change the system's semantics, so is beyond what we mean by umplification).

We recommend Umplification be performed in a test-driven environment. At each stage of refactoring, the resulting behavior of the system should remain semantically unchanged, and so the test suite acts as a litmus test to provide some assurance that the refactoring was done correctly.

Traditionally, the literature refers to transformations using terms such as M2M (model to model), M2T (model to target code or code generation), T2M (code to model or reverse engineering) and T2T (code to code, refactoring, restructuring, or changing the language). Since Umple should be seen as both model and code (P1 in Section 3), all of these terms can be applied to umplification. We suggest using the terms U2U for incremental umplification; T2U for automated complete umplification of a base language program in one step; M2U for rendering a particular modeling notation to Umple so it can also be shown textually; U2M for generation of a pure diagrammatic model from Umple, and U2T for generating base language code from Umple.

We have found that the process of manually umplifying code is reasonably straightforward for someone familiar with Umple, and with knowledge of UML modeling pragmatics. We have performed umplification of several significant systems, including Umple itself. Case studies are reported in Section 5.

To recap, he basic algorithm for umplification is:

```
1. Take a program in language L and consider
it an Umple/L program

2. While (there exist one or more key
umplifiable concepts in the Umple/L program)
    2.1 Select a key umplifiable concept
    2.2 Umplify the selected concept
```

After Step 1, if the Umple/L program is compiled, the Umple compiler will not encounter Umple keywords or constructs (with the exception of a basic class declaration), so will pass the whole program through to the L compiler unchanged. This is consistent with P3, as described in Section 2.

As various iterations of Step 2 occur, more and more Umple notation will be processed by the Umple compiler, emitting language L code generated by Umple.

As mentioned, each iteration of Step 2 should involve testing. The test cases may need to be adjusted to convert them from the API of the original program to the new Umple-generated API, but the underlying intention of the tests should remain unchanged.

At the very end of the process, a large percentage of all the language L code emitted will be generated by Umple to handle Umple constructs. The major exception will be method bodies containing algorithms, which are still passed through unchanged. Please refer to [3] for a more detailed code complexity comparison between base programs and their refactored into Umple programs.

4.1 Umplifiable Concepts

The set of key umplifiable concepts includes:

• **Instance variables:** These are variously called fields, data members, or properties in different programming languages. Some instance variables will become attributes, some will be further transformed into state machines (with the values being the states), and others will become association ends. In all these cases umplification requires not only transforming the instance variable itself, but also locating the code that accesses the instance variable and transforming that code as well.

• **Primitive data types**. Umple uses its own data types such as Integer instead of either "int" or "Integer" in Java. This makes Umple uniform among base languages.

• **The notation in the base language that specifies generalization**. Umple uses its own notation since base languages differ so much.

In this paper, we do not present every last detail required when umplifying a program. Instead, we present the key aspects of the process in a way sufficient for the reader to understand and apply the process in most cases. It is assumed the reader is familiar with modeling in UML.

4.2 Details of Umplifying Instance Variables

Every instance variable in the base program can be umplified. However there are three basic cases:

IV1: An instance variable is private and there are no public getters and setters for it. In this case the variable is used for local storage. The transformation is to mark it using the keyword 'internal'

IV2: The instance variable is public and is an attribute data type. An attribute data type is either:

• a primitive data type (including one of a selected subset of special data types such as Date or Time);

• a class defined outside the current program, that does not itself contain any reference to the current class;

• a collection of any of the above

In such cases:

• The instance variable becomes an Umple attribute.

• Accesses to the instance variable are identified and changed to use the Umple API (e.g. if the variable is x, then it would be set using setX()).

• If the instance variable has only a few discrete values, then it is further transformed into a state machine. This is discussed in more detail later.

IV3: The instance variable is public and is not one of the types handled in U2. In this case the instance variable likely represents one end of an association. To perform umplification, the code implementing the association must be detected, and transformations must be performed corresponding to which of the many possible association-implementation patterns is present.

4.3 Details of Umplifying to Create a State Machine

The process of iteratively umplifying the original code to create a state machine in a class can be summarized as follows:

SM1: Identify a state machine. If an attribute has a finite set of values, it becomes a candidate for umplification into a state machine. An attribute is particularly likely to be transformable to a state machine if there are clear transitions, i.e. functions (which may not be public) that change the attribute to one of the finite set of values. An additional clues that an attribute may be a state machine is that it has a name including words like 'state' or 'status'. Examples of such state machine attributes are given later.

SM2: Identify state machine elements. In this step, candidate state machine elements are identified such as events, states, actions, and do activities.

Table 1 summarizes some of the characteristics of state machine elements.

Table 1: Umplification to create a state machine

Element to create	Elements in the original source that will give rise to the element to create
Event	Typically public functions that modify the state machine attribute.
Action	Most commonly public functions, but also may be blocks of code in more complex functions
Do Activity	Indicated by existence of long-running calculations where an action is taken when they complete, or parallel threads of execution.

5. Examples of Umplification in Practice

In this section we describe three separate examples of Umplification. The first example we describe in its entirety, giving the original Java and the resulting Umple. The second example describes how we applied this technique to the Umple compiler itself. The final example focuses on umplifying an event-driven system to demonstrate umplifying attributes into state machines.

5.1 An Introductory Example

The following example illustrates a simple Java program, and the steps required to convert it into the corresponding Umple.

```
// File Person.java
class Person {
    public String name;
}
```

```
// File Student.java
class Student extends Person {
   private int stNum;

   private int status; /* 0=appled;
1=enrolled; 2=graduated; 3=quit */
   private Supervisor mySupervisor;

   public Student(int stNum) {
     this.stNum= stNum;
     status=0;
   }

   public int stNum() {return stNum;}

   public void enrol()
      {if(status ==0) status=1;}
   void graduate() {
      if(status==1) {
         removeSupervisor();
         status=2;
```

```
      }
   }

   public void quit()
     {removeSupervisor(); status=3;}

   public boolean setSupervisor(
      Supervisor newSupervisor) {
      if(mySupervisor != null || status!=1)
         return(false);
      mySupervisor = newSupervisor;
      newSupervisor.mentees.add(this);
      return(true);
   }
   public boolean removeSupervisor() {
      if(mySupervisor == null)
         return(false);
      mySupervisor.mentees.remove(this);
      mySupervisor = null;
      return(true);
   }

   public String toString() {
      return(
         (name==null ? " " : name) +
         " status="+status +
         " stNum="+stNum + " Supervisor=" +
         (mySupervisor==null ? "nobody" :
            mySupervisor.toString())
      );
   }
}
```

```
// File Supervisor.java
import java.util.*;

class Supervisor extends Person
{
   List<Student> mentees = new
ArrayList<Student>();

   Supervisor() {}

   public String toString() {
      return(
         (name==null ? " " : name) + " " +
         mentees.size()+ " mentees"
      );
   }
}
```

The above source code, plus a test driver routine can be found online at [10].

The following is one possible set of Umplification steps for the code above. Other sequences could be followed. After the list of steps, we present the resulting Umple code.

Step A: Simply rename the files as .ump files (to save space, we simply show all resulting Umple code in one continuous block)

Step B: Change the 'extends' notation for subclassing to Umple's 'isA' notation. Umple will now recognize the class hierarchy.

Step C: Umplify the instance variable 'name' in Person. It will become an attribute. This requires i. removing 'public'; and ii changing the toString() method and the test driver to call getName().

Step D: Umplify the instance variable 'stNum' in Student. It will also become an attribute as in Step C with Umple type Integer. Since there is no setter, we will also tag it as 'immutable'. We have to get rid of the referring stNum() method, and adjust callers to call getStNum().

Step E: Umplify the instance variable mySupervisor in class Student. This becomes one end of an association, so we need to also transform the other end, the instance variable 'mentees' in class Supervisor. To do this we simply declare in class Student an association

```
* -- 0..1 Supervisor
```

Step F. Umplify the instance variable 'status' in Student. This will become a state machine since it has a discrete set of values that are affected by calls to certain methods. We replace i. any methods that modify status with calls to events; ii. any methods that query status with calls to getStatus(); iii. any conditions on whether we should respond to an event with guards; and iv. any code triggered by events with actions (in this example, additional features like do-activities are not used).

Step G: Clean up the constructors, as well as accessor methods of the attributes and association ends using before and after code injections. In our example, we add a before clause when setting a student's supervisor to ensure the Student is not already enrolled.

The resulting Umple code is as follows:

```
class Person {
  name;
}

class Student {
  isA Person;

  immutable Integer stNum;

  status {
    Applied {
      quit -> Quit;
      enrol -> Enrolled;
    }
    Enrolled {
      quit -> /{setSupervisor(null);}
        Quit;
```

```
      graduate -> /{setSupervisor(null);}
        Graduated;
    }
    Graduated {}
    Quit {}
  }

  * -- 0..1 Supervisor;

  before setSupervisor {
    if(aSupervisor != null
      && (supervisor != null ||
        status != Status.Enrolled)) {
      return false;
    }
  }

  // Base language method
  public String toString() {
    return(
      (getName()==null ?
        " " : getName()) +
      " status="+getStatus() +
      " stNum="+getStNum() +
      " Supervisor=" +
      (getSupervisor()==null ? "nobody" :
      getSupervisor().toString())
    );
  }
}

class Supervisor {
  isA Person;

  // Base language method
  public String toString() {
    return(
      (getName()==null ?
        " " : getName()) + " "
         +
      numberOfStudents()+ " mentees"
    );
  }
}
```

Note that only two base language methods remain, the printString() methods in Student and Supervisor.

The reader can take any versions of the code, at each step of umplification (obtained from [10]), and paste it into UmpleOnline [6] or our Eclipse-based Umple environment (also available for download at [6]). Note that the test driver at [10] for the umplified code was adjusted to account for the changes in the generated API, but the changes needed were minor.

5.2 Refactoring Umple into Itself

The Umple tools (parser, code generator, meta-model, analytics) was initially written entirely in pure Java code. As new Umple features were developed, we were able to iteratively refactor Umple into itself. For example, Umple's internal model representation was incrementally refactored to make use of Umple attributes and associations.

The Umple meta-model contains two main components; the model representing classes, attributes and associations, and the model representing states, transitions and actions. The attribute/association component was developed for the first release of Umple and was initially written entirely in Java. It was then refactored into Umple, using the umplification process described in this paper. The second component was developed from the start using Umple, making use of existing modeling capability such as attributes and associations.

5.2.1 Analysis of Our Experiences Umplifying Umple into Itself

The Umple meta-model is comprised of about 15 core classes, with another 25 subsidiary classes that navigate, analyze, and in general make use of those core classes.

The effort to refactor Umple into itself provided two main benefits to Umple. First, it provided a practical test-bed for identifying language and code-generation features required to build real systems. As we encountered code blocks in the Java code that could not be readily transformed into Umple; we enhanced Umple to support the necessary (and missing) features. Second, it provided a realistic migration scenario demonstrating that existing systems could, in fact, be migrated towards Umple in a stepwise (as opposed to big-bang) approach. This therefore became our first experience with umplification.

During the refactoring of Java into Umple, several less-than-ideal modeling decisions were uncovered in the Java code; resulting in a less-than-ideal model once our initial refactoring was complete. This can be somewhat explained by the fact that the design of the first model for Umple (i.e. the Umple meta-model) was separate from its implementation, with few facilities beyond reverse engineering tools to visually inspect the dependencies and relationships added to the system that were not part of the original design.

These less-than-ideal modeling decisions were not uncovered until we attempted to model the Umple system itself in Umple. At this point, further improvements to the underlying meta-model could be achieved. Now that Umple is represented in a model-oriented language, analysis and model inspections are used to help improve our models, which in turn improves our code.

It was discovered that Umple provided an excellent environment for model-level *code* inspections. As enhancements to the model are made, the results are instantaneously achieved in the code itself; as they are one-in-the-same with no disconnect between modeler and coder.

5.3 Umplification of an Event-Driven System

In this section, we illustrate Umplification of an open source elevator simulator that was written almost entirely manually by a team unaffiliated with us [11]. The system developers did not use state machine code generation to build any parts of the original system. At the same time, the system size (85 classes and 1396 functions) was big enough to exhibit interactions between system components, making it a good example for demonstrating umplification into state machines.

5.3.1 Identifying Candidate Elements for Umplification

In this example, we limit our scope to presenting umplification of attributes that are controlled by a state machine. Other aspects of this case study are out of the scope of this paper.

As discussed earlier, attributes exhibiting certain characteristics indicate candidacy for state machine umplification. The following Java code snippet shows three objects: UP, DOWN, and NONE.

```
Direction UP = new Direction("UP");
Direction DOWN = new Direction("DOWN");
Direction NONE = new Direction("NONE");
```

These three objects can immediately be refactored into a single Umple attribute controlled by a state machine. Umple supports state machines without any transitions or events (simple state machines). Therefore, the refactored Umple code is as follows:

```
Direction {
  Up { }
  Down { }
  None { }
}
```

5,3 2 Umplification to Create Events

The elevator simulator included a number of public functions that were easily identified as events. For example, the simulator implements a clock that sends signals at defined intervals.

Functions invoked from the user interface were also candidates for umplification as state machine events. Table 2 illustrates some of functions invoked from the UI that were Umplified as part of the Umple state machine.

Table 2: Some UI Functions That Became Events

Function Name	Description
actionPerformed	Calls a number of appropriate functions when the user performs an action during simulation (i.e, add people, add elevator)
setFloatValue	Sets the value of a simulation parameter, like the randomness of floor requests, the speed of the simulation, etc.
parametersApplied	The user applies new parameters to the simulation.

Umple state machine events return a Boolean value (true if the event triggers any transition, false otherwise). This is needed in scenarios where the system needs to track if the event resulted in a state transition or not. We observed that a number of events identified in the simulator system return no value. It was easy to simply ignore the return value, and update the test cases accordingly.

The following listing illustrates the state machine *Direction* partially updated with events of the state *Up*.

```
Direction {
  Up {
    floorRequestDown -> Down;
    floorRequestUp -> Up;
    topFloorReached -> None;
 }
  Down { }
  None { }
}
```

Umple events are language-independent. Since Java was the base language, the generated API for this example is the following. We adjusted the places where events are triggered to use these method names.

```
public boolean floorRequestUp()
public boolean floorRequestDown()
public boolean topFloorReached()
```

5.3.3 Umplifications to Create State Machine Actions

Umple actions are implemented as calls to functions directly implemented in the base language source code. The following listing illustrates our example enhanced

with an entry action in the None state that executes the operation of exiting and entering people out of and into an elevator. As the listing illustrates, the entry action calls a function that in turn calls two other functions in the base language. This is a powerful feature of Umple because it allows the refactoring of actions without having to change the existing source functions.

```
Direction {
  None {
    entry /{exitPeople();}
  }

  private void exitPeople() {
    chooseSomeoneFromList();
    updatePeopleWaitingSize();
    .. ..
  }
}
```

5.3.4 Umplification to Create Guards

Umple supports guards that represent a Boolean value, a Boolean expression, or any function that returns a Boolean result.

The following listing illustrates the state *None* umplified with a simple guard *arePeopleWaitingToExit*.

```
Direction {
  None {
    entry /{exitPeople();}
    [arePeopleWaitingToExit] -> None;
  }
}
```

5.3.5 Analysis of Our Experiences Umplifying the Event-Driven System

The elevator simulator system is comprised of 86 classes, and 40 test cases. The system had 420 attributes and about 1396 functions. The initial null transformation was applied to the entire system. Our umplification process was then limited to only one component of the system that resulted in umplifying 35 attributes, 5 guards, 65 events, and 40 actions. The 40 test cases provided with the system were run automatically after each umplification iteration.

Even though the elevator simulator was developed by a separate development team, and there was hardly any code documentation provided with the system, the umplification process was relatively simple. Refactorings were applied in small steps and system sanity was checked systematically by running the test cases. As we umplified parts of the targeted components, our understanding of the system was improved because we were able to visually inspect modeling elements embedded in the refactored Umple/Java.

6. Conclusion

We presented an approach called *umplification* whereby source code in a base language such as Java or PHP can be incrementally refactored to add modeling abstractions.

The result of a series of umplification steps is the conversion of a base language into our Umple language. At the same time, umplification results in the reverse engineering of a UML model.

Umple currently supports the most important features of class diagrams and state diagrams. Umple essentially adds UML abstractions to base language code. The Umple compiler generates the needed base-language code to implement the abstractions, and then passes the result to the base language compiler.

Umplification can conceptually be performed automatically by any reverse engineering technology that can extract a UML model from source code. However, existing technologies have a key weakness: they generate a model that represents the code, but is not actually the code itself. The Umple language can be seen by the programmer as just modified source code, still containing much of the syntax they are familiar with. On the other hand, the modeller can see the modeling constructs in textual form or graphical form. The tension between model-centric and code-centric software engineering is thus reduced.

In addition to helping recover a UML model, Umplification can be used to simplify a system. The resulting Umple code base tends to have many fewer lines of code than the original base language. The umplification process can also enable inspections that are concurrently model-based and code-based.

We demonstrated that the process works in practice: we have a functional compiler that we have used to umplify Umple itself, as well as several other systems. We have also used it for new development of both ‘pure’ models, and complete systems. Umple has been used in the classroom and in several industrial projects.

7. References

[1] Forward, A. and Lethbridge, T. C. "Problems and opportunities for model-centric versus code-centric software development: A survey of software professionals," in *MiSE '08: Proceedings of the 2008 International Workshop on Models in Software Engineering,* 2008, pp. 27-32.

[2] Forward, A., Badreddin, O., Lethbridge, T.C.. "Umple: Towards Combining Model Driven with Prototype Driven System Development", *proc. IEEE International Symposium on Rapid System Prototyping*, Fairfax Va, 2010

[3] Forward, A., Lethbridge, T. C. and Brestovansky, D. "Improving program comprehension by enhancing program constructs: An analysis of the Umple Language," *proc. International Conference on Program Comprehension*, 2009, pp. 311-312.

[4] "Umple Language," accessed June 2009, http://cruise.site.uottawa.ca/umple/

[5] Badreddin, O. "Umple: A model-oriented programming language," in *proc. 32nd ACM/IEEE International Conference on Software Engineering-Volume 2,* 2010, pp. 337-338.

[6] "UmpleOnline," accessed June 2009, http://cruise.site.uottawa.ca/umpleonline

[7] Object Management Group, "Concrete Syntax for a UML Action Language RFP," Tech. Rep. ad/2008-09-09, 2008.

[8] Gamma, E., Helm, R., Johnson, R. and Vlisides, J., "Design Patterns: Elements of Reusable Object-Oriented Software". New Jersey: Addison-Wesley Reading, MA, 1995

[9] France, R. and Rumpe, B. "Model-driven development of complex software: A research roadmap," in *FOSE '07: 2007 Future of Software Engineering,* 2007, pp. 37-54.

[10] "Umplification Example Person / Student," accessed 2010, http://cruise.site.uottawa.ca/umple/umplificationExample-PersonStudent/

[11] Chris Dailey, N. M. (2004-2005, "Elevator simulator". v. 0.4, 2005, http://sourceforge.net/projects/elevatorsim/files/elevatorsim/.