Preprint of Lethbridge, TC and Algablan, A, **Umple: An Executable UML-Based Technology for Agile Model-Driven Development**, *Advancements in Model-Driven Architecture in Software Engineering*, Page Range: 1-25, *IGI Global*
2021, https://www.igi-global.com/gateway/chapter/261451, doi: 10.4018/978-1-7998-3661-2.ch001

Umple:
An Executable UML-Based Technology for Agile Model-Driven Development

INTRODUCTION

When software engineers refer to *models,* they generally are referring to abstract representations of some part of a software system. Since the late 1990's UML has been the most widely used language to express such models, rendering them as several different types of diagrams, such as class diagrams and state machines.

Models are typically contrasted with *source code*, generally written in a textual programming language. Almost all software is currently written by humans is in the form of such source code, which is the master formal description from which the system is built. By *formal*, we mean machine interpretable in this context.

The main contribution of the technology discussed in this chapter is a demonstration of how source-code based approaches and diagrammatic modeling approaches can be brought together as one, enabling greater adoption of model-based agile software engineering processes. Added benefits include improvements in separation of concerns, feature-based development, and product-line based developmen.

Modeling Adoption Challenges

Petre (2013) showed that modeling in UML is not as widely practiced as its proponents might have expected. UML diagrams are commonly just drawn on whiteboards to help developers better understand what source code will be needed. UML diagrams, often created using a simple drawing tool, are reasonably common in documentation, but often become out of date. A key observation is that models are not widely used to generate code, except in certain niches such as some kinds of safety-critical software.

The grand vision of many modeling proponents is that models should *become* the master formal source of the system, or at least of parts of systems. Models can currently be used to verify or check designs for *planned* source code, with the source code then being handwritten. But comprehensive code generation seems the most reasonable path forward. Code generation avoids both duplication of development effort, and errors caused by incorrect human transformation of models into code.

Unfortunately, our own research (Agner, Lethbridge & Soares, 2019; Forward & Lethbridge, 2008) has shown that existing modeling tools have numerous weaknesses, particularly regarding code generation. Expensive, proprietary and complex tools are often able to generate reasonable code, but open source tools accessible to average developers tend to create only stubs or no code at all. A key goal for the work reported in this chapter was to create a highly usable tool that can do comprehensive and reliable code generation and build complete systems.

Beneficial Properties of a Source-Code-Like Format

Creating a tool that manipulates models primarily in the form of diagrams will not, we realized, satisfy the vast majority of developers, even if such a tool was fantastic at code generation. Developers value many other properties of their textual source code:

1. **Compact and readable**: Source code tends to be compact, whereas diagrams consume a lot of space visually. By analogy, there was a fashion in the 1960's to create flowcharts, but good structured code was later found to take much less space and be as readable.
2. **Documentable**: Source code can be easily commented, with few constraints as to where the comments are placed, and without adding as much visual complexity as would be required on a diagram.
3. **Comparable and mergeable**: Versions of source code can be easily compared as *deltas* with widely used 'diff' tools; furthermore, version-control tools like Git store these deltas compactly, and allow easy merging of proposed changes. Code review tools allow reviewers to readily evaluate and comment on the precise changes being proposed in deltas. We explored this topic in earlier research (Badreddin, Lethbridge and Forward, 2014).
4. **Focusable**: Source code can be easily searched, with search results organized so as to focus on lists of relevant lines. Similarly, results of analysis (such as a compiler presenting errors and warnings), or the results of tests can be organized as lists of source code lines that need attention.
5. **Parsable and transformable**: Source code can be easily parsed, so third-party tools can be created with little effort to perform tasks such as transformation. This facilitates exchange among tools. Search-and-replace can be done easily to facilitate refactoring.
6. **Sophisticatedly editable**: Source code can be edited in any text editor, so useful features of text editors can be used (e.g. block collapsing), auto-indenting, and syntax assistance.
7. **Composable to specify variants**: Capabilities such as macros, file inclusion, and so on, can be layered onto textual forms easily.

Many of these properties of source code are particularly important for supporting the trend towards *agility*, the center of which is the ability to make frequent, small and properly tested changes to systems. In particular, agility demands good version control and the ability comment on deltas (property 3); it demands ease of debugging through search, analysis and tool assistance (properties 4-6); it requires automatic testing, with test failures pointing to the right place (property 4); it requires easy refactoring (property 5); and it encourages lightweight in-code documentation (property 2).

Properties 3 and 7 also facilitate the ability to organize software into features (feature-oriented development) or variants in a product line (product line engineering).

The software engineering community has created textual languages such as XMI and other XML schemas to allow rendering of models as text for exchange. But only aspects of property 5 really apply to such languages. In particular, such formats are far from human readable (other than toy examples), and cannot be usefully edited by humans.

Table 1 summarizes how the seven benefits of a textual format tend to be challenged or lacking when UML modeling is done diagrammatically with reliance on XMI to store and exchange the models. Later on, we will see that Umple has done of the drawbacks.

Table 1. Challenges faced by relying on models to be primarily conveyed as diagrams

Advantage of a textual form	Drawbacks of reliance on diagram forms of models
1. Compact and readable	Diagrams are not compact; XMI is not human-readable

2. Documentable	Comments are either hidden (requiring clicking/hovering to see them), or cause clutter if shown as 'callouts'.
3. Comparable and Mergeable	This has been a challenging research problem, and tools are complex (Brunet et al, 2006; Bendix et al, 2008). Small changes to diagrams can lead to large changes in the saved XMI, meaning that ordinary text diffs do not work.
4. Focusable	Search results or warning messages that point to places on a variety of diagrams can be unwieldy.
5. Parsable and transformable	Tools require deeper semantic knowledge than required for textual formats. Also, XMI has long had problems with subtly incompatible versions of its schemas (Lundell, Lings, Persson & Mattsson, 2006).
8. Sophisticatedly editable	Requires specific editors, so a market for many editors does not develop easily.
9. Composable to create variants	This has been a challenging research problem for models (Jayaraman, Whittle, Elkhodary & Gomaa 2007).

Despite Table 1, we do not mean to imply that diagrams are bad or to be avoided. They can help humans understand certain aspects of software much better than a textual format. As we will see in the next section, it is key to Umple that both text *and* diagram forms must exist simultaneously.

Code-Diagram Duality and Mutual Benefit

Early in our research, we realized that the above properties of textual languages are so compelling that textual 'code' is here to stay for the foreseeable future as a dominant way to represent most software.

But that certainly does not mean that diagrammatic models are irrelevant. Being able to see a diagram of how classes and their instances are organized (a class diagram), how a system behaves (a state diagram), or many other system views, is also extremely useful.

It became clear to us that some software engineering tasks are easier to do with diagrams and some are better done by working with a textual form. Reverse engineering tools can allow extraction of diagrams from arbitrary source, but such tools tend to break if the source code does not quite conform to expected conventions. Additionally, editing the diagram in order to update the source code – *round-trip engineering* (Medvidovic, Egyed & Rosenblum, 1999) – is a source of fragility. Engineers we have worked with report, "it just doesn't work in practice."

We became convinced therefore that what is needed is a modeling language that allows diagram and code forms to be simultaneously available, representing different views of the same model. We came to use the term *code-diagram duality* to describe this.

Desired Characteristics of a Textual Modeling Technology

In 2007 we set out to design Umple as a language and accompanying toolset with the following characteristics:

- It has comprehensive and robust code generation for multiple languages.
- It has a simple-to-use readable textual representation of common UML constructs, that looks like what one expects in a good programming language.
- It preserves all the benefits of textual source code listed earlier. Indeed, it goes beyond UML in ways enabled by its textual form, such as allowing various ways to separate concerns textually (aspects, traits, mixins, and mixsets, all discussed later).
- Diagram *editing* is also available, enabling updating of the text without round trip engineering.

- It allows real-time diagram generation in several target languages so developers can see the code from the textual and diagram perspective at all times.
- It blends in with traditional code written in the same target languages, allowing parts of systems that cannot be represented using the modeling constructs to be incorporated parsimoniously in the same textual language. Such code is sometimes called 'action language' code.
- It is agnostic to the model driven software engineering approach being used, and can be used to represent models ranging from abstract technology-independent business models, all the way to concrete executable systems. This will be discussed more in the section on Model-Driven Engineering later.
- It allows comprehensive analysis of models, pointing out errors and warnings just like a programming language compiler.
- It free to use and open source (Github 2020a).

Umple, has been under development since 2007 and is now widely used, particularly in educational institutions, where it has been shown to help improve student grades (Lethbridge, Mussbacher, Forward & Badreddin, 2011), and has been found to be usable by students (Agner, Lethbridge & Soares, 2019). Umple's online server (UmpleOnline, 2020), which allows users to work with Umple in the cloud, receives over 200,000 user sessions, and over 2 million user interactions per year. Use of Umple on private computers (in Docker, in Eclipse and Visual Studio Code plugins, or on the command line) is not tracked, for privacy reasons.

Other Approaches to Textual Modeling

Umple is certainly not the only textual modeling language. TextUML (Abstratt, 2020), USE (Gogola, Büttner, & Richters, 2007) and txtUML (Dévai, Kovács & An, 2014) are other examples of textual syntaxes for UML. Nor is Umple the only tool that enables executability of UML: Executable UML (Mellor & Balcer, 2002) is an example. Indeed, UML is not the only modeling language: SDL (Rockstrom & Saracco, 1982) has been around for over 35 years and has a textual form. Additionally, many 'formal methods' languages are textual and can be used for modeling parts of systems. Umple is the only current tool that combines all the above characteristics, or even most of them.

Research Questions Addressed

The main research gap addressed by Umple is to a gain a fuller understanding of the benefits of textual vs. diagrammatic modeling and code-diagram duality, especially with regard to agility. We also want to build a technology that can scale, to build very large models with this technology (such as Umple itself) and to get it into widespread use in the open-source community.

By doing the above will we be able to delve deep into new modeling problems that emerge 'at scale'. Existing large-scale models tend to be hidden in proprietary technologies that are only minimally accessible to researchers.

In the coming section we give an outline of key Umple features, showing how Umple's textual representation aligns with UML diagrams, and goes beyond UML.

OVERVIEW OF UMPLE

In this section we outline key features of Umple, along with some of the design rationale relating to agility, textual modeling and product-line or feature-based development. We give examples from a case study. Complete details of all Umple features can be found in the Umple user Manual (Umple, 2020a) and in the various papers cited in the following sections.

The user manual is a live document, with over 450 examples, all of which can be loaded into UmpleOnline for analysis. The examples include cases that trigger all of the over 200 potential modeling problems that Umple's analysis engine can detect, along with their potential solutions.

Umple can be used in different environments: In addition to UmpleOnline, there are bindings for Microsoft Visual Studio and Eclipse, as well as the ability to use any text editor. A Docker image is also available, allowing offline use of UmpleOnline.

An Umple system is organized as a set of files with the .ump suffix. Within the files are various top-level entities (classes, associations, traits, state machines, mixsets and so on) all described later. Some elements can contain others (for example, a state machine has states, and a class has attributes and associations). An important feature of Umple is the *mixin*. If two top-level elements are repeated with the same name, then the contents of the top-level elements are 'mixed in' together to create a single top-level element. We will see examples of this later on.

UML Class Diagram Support

Classes

Classes are a core construct any object-oriented technology. Java, C++, PHP and other programming languages all support ways to declare classes that contain methods and instance variables (often called fields, and called attributes in UML), as well as generalizations (extends, or subclass relationships). Umple classes are declared in a very similar way as in other textual object-oriented languages.

The case study Umple code in the next section has three classes, starting on lines 7, 13 and 20.

Umple supports *interfaces*, but to simplify this chapter we will not discuss them.

Traits

In addition to classes, Umple supports the notion of *traits* (Abdelzad & Lethbridge, 2015). A trait can be considered a class fragment, containing any elements that can be present in classes. One or more traits can be incorporated into a given class by a *copying* process, as opposed to inheritance. The concept of traits goes beyond UML and enables support for multiple inheritance, product-line development and feature-based development. For example, each trait incorporated into a class can serve as part of the implementation of a feature. Changing a trait could also be a way to create a variant product in product-line development.

Traits can be arranged in generalization hierarchies, and Umple performs extensive semantic analysis to ensure that traits are used to build classes in an error-free way.

The case study code below shows a trait starting at line 3. The trait is invoked using the *isA* keyword in the three classes, starting at lines 8, 14 and 21, ensuring that the classes all have the features that come with being devices. Generalization could have been used to achieve the same effect, but later on in the case study we will need to add a superclass to the SmartLight class; the use of traits helps avoid the complexity of multiple inheritance that would otherwise be required, and enables generation of code in languages that don't support multiple inheritance.

Attributes

Umple attributes, just like in Java or C++, are specified using a type name followed by the attribute name. However, as in UML, Umple attributes are more than just variables. They are used to generate accessor methods (get and set methods), and are subject to constraints (discussed later).

Umple generates code to ensure that attributes are properly initialized in the constructor. If the developer does not want this initialization, they can switch it off with the stereotype *lazy*.

The case study code below shows attributes at lines 4, 15, 16 and 17. Note that line 4 has no specified type: In Umple, *String* is used as the default type. Badreddin, Forward and Lethbridge (2013a) and Forward (2010) give full details of Umple attributes.

Generalization

Umple generalizations follow UML semantics and are represented using the 'isA' keyword in the same way that traits are included in classes. Generalizations are translated into the different needed syntaxes of Java, PhP and C++ during code generation. There is an example of generalization later in the case study.

Methods

Umple generates an API from each model with many methods generated from the modeling constructs (associations, attributes, etc.) present. Examples include methods to add a link to an association, or to set an attribute. Umple can generate a Javadoc-like set of webpages describing this API, with pointers back to the original Umple files.

The developer may also write *user-defined methods* to perform arbitrary actions; these are written in Umple classes very much as they would be in the target language (Java, C++, Php etc.). The Umple compiler outputs them relatively unchanged when generating code. Umple does, however:

- Enable injection of preconditions into methods (see the section on constraints below)
- Enable injection of code into the start, end and labeled locations in methods (see the section on aspects below)
- Allow a method to have multiple bodies, one for each target language to be generated.

Methods can be included or excluded from being visible in class diagrams, by specifying an argument to the diagram generators.

Even in diagrammatic modeling approaches, the bodies of methods are written textually, so they are a natural fit for Umple's textual form.

Associations

UML and Umple go further than popular programming languages and allows specification the key concept *association*. Associations describe the relationships (links) that will exist at run-time between instances of the associated classes. The case study code below shows associations on lines 9, 10 and 22. When code is generated from this Umple sample, there will be methods to add and delete links of the associations, always maintaining referential integrity as in a database. Full details of Umple associations are described by Forward (2010) and Badreddin, Forward and Lethbridge (2013b).

Mixsets

An additional feature of Umple shown in the sample code below is the *mixset*, as appears in lines 9 and 22. This builds on the notion of the mixin. A mixset block has a name and allows conditional modeling. The associations in lines 9 and 22 only appear if the 'basic' mixset is activated. The *use* statement in Line 1 performs this activation, but activation could also be performed by a build script passing the mixset name to the Umple compiler.

Multiple statements or groups of statements can be tagged with the same mixset label, to group them as a feature. Like traits, mixsets are yet another capability that takes advantage of Umple's textual programming-language-like form and goes beyond UML. Mixsets are discussed by Lethbridge and Algablan (2018).

Case Study: Smart Light System

In order to explain various features of Umple, we present the following small case study, showing how it can be modeled in two steps using Umple's features.

A company is developing a smart light system. The hardware consists of a controller and various switches and lights connected by WiFi or Bluetooth. The user interface of the controller is a smartphone app; this is used to configure the system, and can be used in place of switches.

The system is to be developed in an agile manner, in a series of sprints. New versions will be released when each sprint is complete. Here we show two sprints only.

Sprint 1: Basic Features

Each light has a serial number that is used to connect it to the system and a maximum lumens. All lights can have adjustable brightness between zero and 100%. Lights can be configured with a default percentage of the maximum brightness. A basic switch can be set to control a light. Switches also have a serial number.

Sprint 2: Grouping and Modes

The second release of the case study system adds the ability for lights to have a name, and for switches to have dimmers. Lights can now be in named groups to allow several to be controlled as if they were a single light. A switch can be set to control a light or group.

An additional capability added in the second release is that the system can be set to random "away safety" mode, to simulate a residence being occupied, and "alarm" mode where all lights flash between maximum brightness and 75% brightness to alert others.

Umple Model Code and Generated Outputs for Sprint 1 of the Case Study

The model code below might be put in a file called SmartLightV1.ump. We invite the readers to load this code and try it out; it is available as an example in UmpleOnline. The reader can try commenting out line 1 to see the associations at lines 9 and 22 disappear.

When processed by Umple, these 23 lines can be used to produce 765 lines of Java, as well as the diagrams shown in Figure 1 and Figure 2, and many other possible artifacts.

Figure 1 is a standard UML class diagram. Its 2-dimensional layout, with classes as nodes and associations as arcs, can greatly help the developer understand the system. UmpleOnline even allows editing of class diagrams, with edits instantly reflected in the textual code. But as we have been arguing, the textual notation below allows many complementary benefits: The text has comments, can be searched, errors can be easily highlighted, editing the text tends to be faster than editing a diagram, and git commits and merges work very well with it.

Figure 2 is a UML class diagram generated from the same code, but with the option to explicitly show traits, rather than merging them into the code. This illustrates how generating multiple diagrams (instantly) from a given textual source can be very useful.

```
1   use basic;
2
3   trait Device {
4     serialNumber;   // attribute; String by default
5   }
6
7   class SmartLightConsole {
8     isA Device;
9     mixset basic { 1 -- * SmartLight; }
10    1 -- * Switch;
```

```
11  }
12
13  class SmartLight {
14    isA Device;
15    Integer maxLumens;
16    Integer defaultBrightness; // value set to when simply 'on'
17    Integer currentBrightness;
17  }
19
20  class Switch {
21    isA Device;
22    mixset basic { * controller -- * SmartLight; }
23  }
```

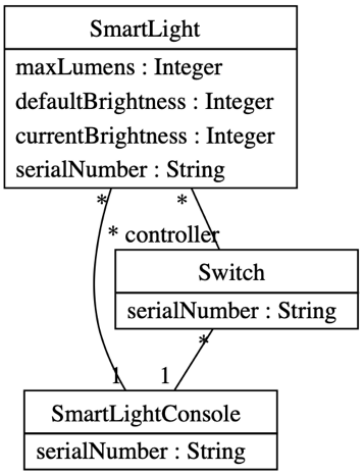

Figure 1: Class diagram of the basic version of the Smart Light system, as generated from the Umple code. The trait is invisible since its elements are folded into the classes.

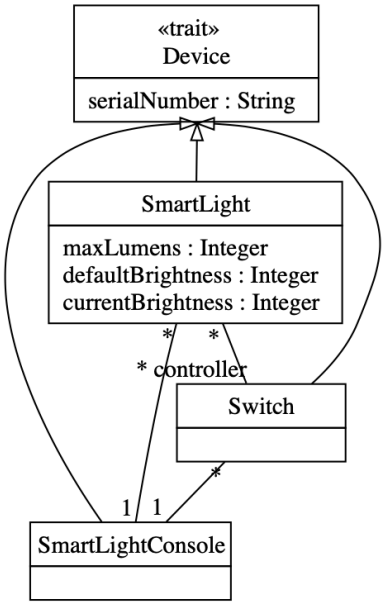

Figure 2: Class diagram showing the trait explicitly.

Umple Model Code for Sprint 2 of the Case Study

Below, we show relevant modeling code for Sprint 2 of the case study. We have added the following.

- New classes LightOrGroup (line 4), LightGroup (line 12), and DimmableSwitch (line 26).
- Generalizations (lines 9, 13 and 27).
- Redefinitions of the associations from the original version, so that switches and the console are now connected to LightOrGroup. This is in the mixset starting in line 17, which is activated on line 1.
- Mixins adding extra functionality to classes SmartLight (line 8), SmartLightConsole (lines 17 and 31) and Switch (line 21), This ability to add new features to a class via mixins is a key feature of Umple, and is yet another way that Umple leverages its textual format.
- A state machine (lines 32-45), that will be explained in the next section.

The following code might be put in SmartLightV2.ump; when processed along with SmartLightV1.ump this generates 1299 lines of Java. Note that this case study shows one of many ways of organizing the source code. We have chosen to create a series of version-specific files. The system could instead be organized by class, or the two files could be merged.

```
1   use controlGroups;
2
3   // A named element that can be controlled as a unit, e.g. 'Living Room'
4   class LightOrGroup {
5     name;
6   }
7
8   class SmartLight {
9     isA LightOrGroup;
10  }
11
12  class LightGroup {
13    isA LightOrGroup;
14    0..1 -- * SmartLight;
15  }
16
17  mixset controlGroups {
17    class SmartLightConsole {
19      1 -- * LightOrGroup;
20    }
21    class Switch {
22      * controller -- * LightOrGroup;
23    }
24  }
25
26  class DimmableSwitch {
27    isA Switch;
28    Integer dimmerSetting;
29  }
30
31  class SmartLightConsole {
32    mode {
33      normal {
34        goAway -> awaySafety;
```

```
35        panicButton -> alarm;
35      }
37      awaySafety {
38        do {cycleLights();}
39        backHome -> normal;
40      }
41      alarm {
42        do {flashLights();}
43        panicButton -> Normal;
44      }
45    }
46  }
```

Figure 3 shows the class diagram generated from the above.

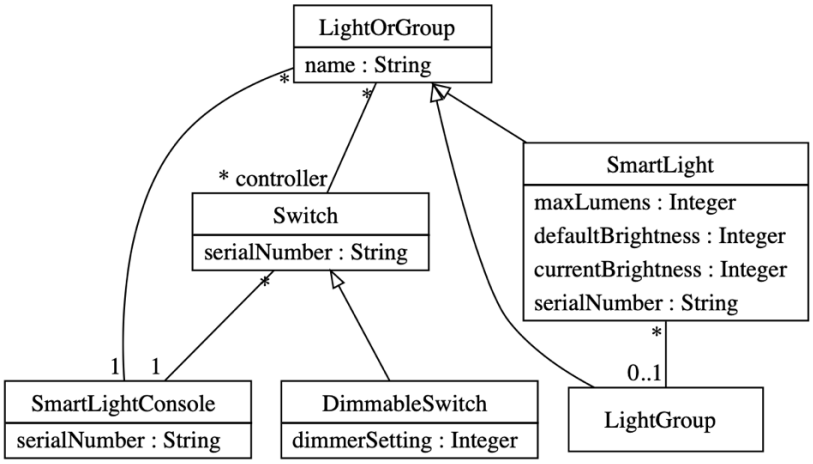

Figure 3: Class diagram of the second version of the Smart Light system.

UML State Machine Support

A state machine can be used to define the behavior of an entity such as an object, user interface, subsystem or entire system; Umple follows UML state machine semantics. The entity controlled by the state machine is said to be in a given *state* at each point in time; the initialization of the entity places it in a defined *start state*. When in a particular state, the entity has certain behaviour. This includes *actions* on entry and exit from the state, ongoing *activities* while in the state, and receptiveness to a certain set of *events* that can cause it to *transition* to another state (where it behaves differently). Constraints called *guards* can prevent this; guards are discussed more later in the section about constraints.

Although to save space we have not shown it in this chapter, states can be grouped hierarchically, allowing transitions to be defined that can take the entity from one state or *superstate* (group of states) to another state or to the start state of another superstate. The states defined within a group are called *substates*. Several state machines operating concurrently can be defined for the same entity or within a superstate.

Umple allows developers to embed all the above state machine concepts in its textual form.

In the Umple case study given above, a state machine called ‘mode’ is defined starting at line 32. This has three states. The start state is ‘normal’ (line 33). When event goAway occurs (via a message call to a generated goAway() method from a button, perhaps on the user interface) the system changes to ‘awaySafety’ state (line 37). While in this state, the cycleLights() activity method is executed in a

separate thread, which would be interrupted when the 'backHome' event is called. Similarly, if the user presses the panicButton, the system transitions to alarm state (line 41), which runs the flashLights() method in a thread until interrupted by a second press of the button.

The state machine diagram for this appears in Figure 4. As with Figures 1, 2 and 3, this diagram is kept in constant sync as the code is edited.

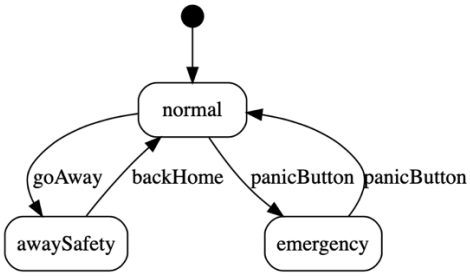

Figure 4: State machine diagram of the second version of the Smart Light system.

Like class diagrams, it is clear that state machines are naturally diagrammatic in nature, since they are two-dimensional. An Umple developer can constantly see the diagram change as they edit the textual code. The textual form provides all the advantages described earlier including compactness, documentability, comparability, composability and editing in any tool.

Umple also provides additional outputs from state models, aside from standard state diagrams and generated code. Figure 5 shows the state tables, and Figure 6 shows a random sequence generated for the state machine. Both of these are ways of helping the developer understand whether or not their model is correct. Not shown here are the ability of Umple to generate formal method code to allow model checking in languages such as NuXmv (Adesina et al, 2018). All of these can be generated at any time from the model code.

State-event table

	backHome	goAway	panicButton
normal		awaySafety	emergency
awaySafety	normal		
emergency			normal

State-state table

	normal	awaySafety	emergency
normal		goAway	panicButton
awaySafety	backHome		
emergency	panicButton		

Figure 5: State tables generated by UmpleOnline from the state machine in the case study.

States	START	1. goAway	2. backHome	3. panicButton
normal	X		X	
awaySafety		X		
emergency				X

Event Number	Event	Next State
START	-	normal
1	goAway	awaySafety
2	backHome	normal
3	panicButton	emergency

Figure 6: Randomly generated event sequence generated from the state machine in the case study.

Some Other Umple Features that Benefit from a Textual Environment

This chapter does not have the space to cover all the features of Umple. However, the following are brief descriptions and examples of some of the additional features that derive synergies from being present in a textual modeling language, as opposed to a predominantly diagrammatic one.

Constraints

Umple has four types of constraints, three of which are derived from UML. Even in UML, which is otherwise diagrammatic, constraints are expressed textually.

- **Multiplicity of associations**. These specifying bounds on the number of links that can occur at run time. Several examples are given in the earlier code.
- **OCL-like constraints in classes**. Umple supports a subset of OCL to represent invariants, method preconditions and method postconditions; they are Boolean expressions written in square brackets.
- **Guards on state machine transitions**. These also appear in square brackets, and are syntactically the same as the OCL constraints that are used for method preconditions.
- **Various built-in stereotypes**. These are tags added to Umple elements to constrain the behavior of generated code. For example, the stereotype ‘singleton’ indicates that the generated code is to allow only one instance of a class to be created; the stereotype ‘immutable’ indicates that an attribute or entire class instance must not be allowed to be changed after instantiation.

The following simple mixin shows a use of constraints that could be added to the case study. This will limit the range of the attribute MaxLumens. We invite the reader to paste this into the example in UmpleOnline and generate code to see its effect.

```
class SmartLight { [maxLumens > 0] [maxLumens < 5000] }
```

Aspects

An aspect contains a block of code (typically called *advice*) that can be injected into code according to the instructions found in a pattern (called a *pointcut*). Umple comes with an aspect capability that allows injection of code into both generated and user-defined methods. For example, the following Java code would ensure adjustment of lights after a dimmer is set to a particular percentage:

```
class DimmableSwitch {
  after setDimmerSetting {
    for (aLight SmartLight: getSmartLights()) {
      aLight.setCurrentBrightness(dimmerSetting);
    }
  }
}
```

The reader is invited to paste this into the case study code in UmpleOnline to see its effect on the generated code.

Umple aspects have the ability to inject code *before* the start of any method matching a pattern, *after* methods (i.e. before all returns), and after any textual *label* found in the interior of matching methods. The latter capability allows construction of methods with capabilities that differ depending on which features are active.

Filters

There are several situations when a user wants to query or *slice* part of a model. The full set of Umple files of a system represents the complete system. Umple's *filters* capability, which can be activated individually or in groups using mixsets, allows some of the following:

- **Specification of diagrams:** Some tools use a separate language to specify diagrams, but in Umple, filters select part of the model to be used for a diagram. Different filters are used to create separate diagrams, with activation of a mixset to cause display of the desired diagram. The multiple Umple metamodel diagrams (Umple 2020b) are generated this way.
- **Generation of part of the system:** It can be useful to split a system into two executables, each of which is specified by a filter.

The following mixin can be used to draw a diagram containing SmartLight, SmartLight Console, the association linking them, and any of their superclasses. There are options, not shown here, to control the number of 'hops' from listed items (so as to add connected classes), as well as the types of arcs to be displayed.

```
filter { include SmartLight, SmartLightConsole;}
```

Text Generation

Generation of textual output is needed in almost all application types. Examples include generating html or xml for webpages, code from compilers, messages to be transmitted to other systems, and so on.

Several languages have built-in capabilities for this, PHP being a well-known example where this capability is widely used to generate html. Umple has a text-generation capability that is integrated with the language in the same manner as PHP, but it has several unique features that distinguish it, such as being able use knowledge about associations or state machines and to synergistically work with all the other Umple features mentioned in this chapter.

Full details of text generation can be found in Husseini Orabi, Husseini Orabi and Lethbridge (2020). In brief, various kinds of nested blocks bracketed by << and >> can be specified to build templates. If the << is followed by ! then it indicates text to be output. If the << is followed by #, it indicates target language code used inside the template to computationally build output; this can call the Umple-generated API that refers to model elements. The 'emit' keyword is used to construct methods that use the templates.

The following example mixin would add in our case study a display() method to class SmartLightConsole that would show the name and brightness of each light.

```
class SmartLightConsole {
  rows <<!<<# for  (aLight : smartLights) {#>>
    <<=aLight.getName()>> <<=aLight.getcurrentBrightness()>><<#}#>>!>>
  emit display()(rows);
}
```

Synergies among all the above

We have described above many features of Umple that benefit from being rendered textually. These operate synergistically with each other, and facilitate agility as well as product-line development. Table 2 shows some of the characteristics of these features.

All of the features shown in column 1 of Table 2 benefit from the properties listed in Table 1 such as being documentable with code comments, focusable so warning messages can be displayed (as per the analysis described in the right column of Table 2), and composable using Traits.

In Table 2, Umple features marked ** are in UML as diagram elements. Those marked * are in UML textually. Those marked + are value-added textual features in Umple. Those marked & facilitate separation of concerns, such as for better modularity or feature-driven or product-line development.

UMPLE AS A SYSTEM WRITTEN IN ITSELF

As a second case study in this chapter, we will discuss how Umple has been written in itself.

The Umple compiler and code generators are written completely in Umple, with embedded code in Java for the bodies of methods and state machine actions. As far as we are aware, Umple is the only modeling tool that can be said to be *self-hosted* or *boostrapped* in this manner. Umple was originally written in Java, with Jet for code emission and ANTLR for parsing. However, once the first version was in place, the compiler was *umplified* (Garzon et al, 2012), eliminating Jet and ANTLR. Umple has no dependencies on other technologies except Java and the previous version of Umple (needed for self-compilation).

The core compiler has 152 .ump files, comprising 69 KLOC of Umple, with additional code in several domain-specific languages. There are a total of 587 classes. Extensive use is made of mixins and aspects. Feature driven development is used to some extent, so the codebase is organized with a set of files per feature. Mixins for a class may appear in numerous files.

There are another 655 .ump files and 37 KLOC in the various generators (Java, Php, C++, Ruby, various diagrams forms, SQL, unit test languages, etc.) that are organized and processed separately when building Umple. Extensive use is made of Umple's template generation capability.

The process used to develop Umple combines the best of both test-driven and model-driven techniques: There are over 300 test files, with 47 KLOC of tests, and over 5400 tests. These operate on over 99K lines of test data in over 1000 files. This does not include the compiler itself which serves as its own giant testcase as it has to be built and run on itself by every new version of the compiler. A new compiler version is not created unless 100% of the test cases pass.

It is fair to say that a system of Umple's size could not have been managed effectively if its model was rendered strictly as a set of diagrams. Most of the seven characteristics of textual forms we discussed earlier have helped Umple to be developed into a large and reliable system.

Umple's internal design is self-documenting: Various class diagrams are generated whenever a new version of the compiler is built. These can be seen online (Umple 2020a). Mixsets and filters are used to describe each diagram.

Umple's Javadoc plugin is used to keep Umple's documentation webpage up to date with respect to the Umple code. The result can be seen online (Umple 2020b).

Table 2. Characteristics and effects of a subset of Umple features.

	Can be top level	Can be in classes, traits	Can be built in pieces by	Diagram Effect	Effect on generated API	Analysis performed by compiler beyond basic syntactic analysis
Class **	Yes		Trait, Mixin, Mixset	In default Class Diag.	Constructor and other methods	Extensive semantic analysis (e.g. avoiding circular inheritance)
Association **	Yes	Yes	Aspect (for related behaviour)	In default Class Diag.	Methods to manipulate	Presence of referred elements, and extensive semantic analysis
Trait + &	Yes		Mixin, Mixset	In special class Diag.; Default class Diag. merges elements into classes	API appears in classes that use it	Extensive semantic analysis such as required methods, relationships among traits
User Defined Method *		Yes	Aspect	In class Diag. with option to show it or not	Output largely as is	(Syntactic analysis for signature only)
Mixin (repeated top-level element) + &	Yes			Diagram shows mixin contents all combined	Adds methods related to contents	
Mixset (set of similarly named mixins; a file is a special case) + &	Yes	Yes	Mixin	Class or state D. shows selected variant; Feature diagram		Conformity to require statements, which impose rules on which mixsets can coexist
State Machine **	Yes	Yes	Trait, Mixin, Mixset, Aspect	State Diag.	Adds events and other methods	Extensive semantic analysis (e.g. reachability)
Constraint *		Yes			Modifies many methods	Presence of referenced elements, type conformity
Aspect + &	Yes	Yes			Modifies any selected methods	Presence of referenced elements
Filter + &	Yes		Mixin	Diagram shows selected subset		Presence of referenced elements
Text Generation Element +		Yes			Generates emission methods	Presence of referenced elements

Rigorous attention is paid to version control and code review. This would be more difficult if only diagrams were available of modeling elements. Pull requests are merged only when all tests pass, and a code review indicates there are no issues.

When a developer is modifying the compiler, the previous version of the compiler points out problems and localizes them to lines of Umple. Grep searches, IDE searches and failing testcases do the same, facilitating rapid correction of errors.

Refactoring has proved easy, due to the textual form of the language. Some unique refactoring techniques have been developed during the creation of Umple in itself, such as:

- **Extract mixin**: to pull out parts of a file to create a simpler file.
- **Separate pure model from methods**: Takes 'extract mixin' one step further by making the code for class diagrams stand out more clearly from the methods that operate on the various elements.

In addition to test-driven development, other agile techniques that are enforced include a product backlog: All changes are made in response to issues. Also, a continuous integration and release process is followed: Updates to the master version of Umple (which appears in UmpleOnline) occur whenever issues are closed, sometimes several times in a day.

Umple has been developed largely by students (PhD, masters and many 4th year capstone students). Yet its core compiler can run for months online without crashing. Infrequent crashes are rapidly solved with regressions of them prevented through test-driven development. This has only been possible due to

- The rigors imposed by the quality assurance processes (test-driven development, code review) we have discussed;
- Model-driven development, consisting of both the use of abstract modeling elements such as associations, and the presence of model diagrams generated from the text, to enable understanding the system;
- The benefits of the textual form (as per Table 1);
- Umple's value-added features such as the text-emission template capability and mixins to help organize the codebase;
- Extensive built-in model analysis that prevents many kinds of errors; and
- Complete code generation that both dramatically reduces errors that would appear if code were written by hand and also reduces code volume to about 10% of what would be required if the model code was written manually.

UMPLE IN THE CONTEXT OF MODEL-DRIVEN ENGINEERING

One of the important features of model-driven software engineering is that models should not only represent various aspects of a system, such as the data (e.g. class diagrams) and behaviour (e.g. state diagrams), but that there also should be models at various levels of abstraction and platform commitment.

The OMG's Model Driven Architecture approach in particular (Meservy and Fenstermacher, 2005), suggests that there should be several levels of modeling. MDA's most abstract level is Computation Independent Models (CIMs), where modelers can define business processes and general requirements before any commitment is made to design. MDA's next level is Platform Independent Models (PIMs); these incorporate design decisions (such as the specific data as represented in class diagrams) without commitment to particular frameworks, programming languages, databases, and so on. Finally, MDA incorporates Platform Specific Models (PSMs) that have the most concrete levels of detail.

MDA is centred on the development of *transformations* that allow generation of more concrete levels of modeling from more abstract levels. For example, Rhazali, Hadi, and Mouloudi (2016) describe transformations from CIMs to PIMs; Rhazali et al (2020) describe transformations through all three

levels, starting with business processes in BPMN and using transformations to ultimately produce concrete code.

There have been some attempts, such as that of Essebaa and Chantit (2018) to bring together agility and model-driven approaches, because both have the objective of enabling small, frequent and disciplined changes. Source-code-focused agile techniques do this by focusing on testing; MDA does this by focusing on automated transformation. For example, in MDA one might add a new use case to a CIM, derive new needed data from this in a corresponding PIM and from that generate a new PSM, and eventually new code. In a code-oriented agile approach one might write a user story in an issue, then write some new unit tests, then write the code in a small delta, finally committing tests and code together while referencing the issue.

A key feature of the Umple approach is that it is designed to be agnostic to any particular model-driven methodology, and can be used in a manner borrowing ideas from MDA, while also embracing code-oriented agile techniques. In particular, pure models (at the PIM level) can be maintained as artifacts that are separate from algorithmic code, as mentioned in our discussion of the *separate pure model from methods* refactoring above. Umple's ability to generate multiple targets, manage multiple target-language method bodies, and perform conditional modeling with mixsets facilitates the PIM-to-PSM and PSM-to-code transformations. Yet Umple's textual form facilitates creating small deltas containing model constructs along with tests in the standard agile manner.

Umple does have a prototype capability for automated test generation and test-driven modeling. This is in preparation for release and will strengthen its integration of agility and model-driven development.

Umple currently does not manage some of the models found primarily at the CIM level, such as use cases, but such improvements are planned, as discussed in the section on future research directions, below.

EVIDENCE FOR THE EFFECTIVENESS OF THE APPROACH

There are several lines of evidence supporting our proposition that the concepts embodied in Umple are an advance in the practice of software engineering:

- **A study of model comprehension**. Baddredin and Lethbridge (2012) conducted experiments comparing understandability of three versions of three systems: Each system had a UML diagram version, an Umple textual version and a Java version. Comprehension was measured by asking developers questions about the systems. The results showed that UML and Umple versions were equally understandable, and were considerably more understandable than the Java versions. Additional details can be found in Badreddin, Forward, and Lethbridge (2012).
- **A study of student grades**: Lethbridge, Mussbacher, Forward and Badreddin (2011) conducted a study with students in the classroom. Grades of 332 students prior to the introduction of Umple as the teaching tool were compared to those of 122 students after its introduction. Average grades increased by 9% on similar UML modeling questions. Surveys of the students also indicated they very much liked Umple's textual form, code-diagram duality and the code generation.
- **Surveys of tool use**: We conducted international surveys to study modeling tool use. The first survey was directed at 150 professors teaching software modeling with 32 modeling tools in 30 countries (Agner and Lethbridge, 2017). Umple was one of the few tools rated by the professors as being easy to use. The second survey (Agner, Lethbridge & Soares, 2019) covered 117 students in 7 countries. The students had used 14 tools, and 20 of the students had used Umple. The tools were compared according to numerous criteria. Umple scored higher than any other tool in appreciation of its code generation, and also scored well in being easy to use and not being complex. Umple received relatively low scores for technical support, slowness of the website and

bugginess – problems which have all since been addressed. All other tools had low scores in a greater number of criteria than Umple did.

Additional evidence of Umple's usefulness can be seen in the uptake of UmpleOnline, which sees over 2 million transactions per year in over 200,000 user sessions.

Our first item of future work, as discussed in the next section, is to leverage the UmpleOnline platform to conduct additional experiments on the effectiveness of Umple, and textual modeling in general.

FUTURE RESEARCH DIRECTIONS

We are planning several major future research directions for Umple.

The first is to add an integrated educational and experimental platform into UmpleOnline. This would allow experimenters or educators to present modeling problems to users and gather their answers as well as data about their experiences. There is a great need for more empirical studies of modeling and this platform should allow the Umple team to obtain help in this regard from its many thousands of users. In particular, we intend to create a mechanism that can be embedded in multiple tools, to allow comparison of Umple with competing tools such as Papyrus. This is described in Lethbridge (2019), and also in Umple issue 1490 (Github 2020b)

The second direction is to add the ability to embed textual requirements languages into Umple, so that the link between CIM and PIM can be made. One idea is that Umple's strength at drawing diagrams from textual models would be further leveraged. Secondly, any Umple element would be taggable with an 'implements requirement' clause, adding traceability. These capabilities would synergistically combine with Umple's separation-of-concern mechanisms: For example, it would be possible to show the requirements for a particular feature or variant constructed from certain mixsets.

The third direction is to continue to expand Umple's modeling and code-generation capabilities as driven by customer demand. For example, where projects demand new generation targets (and the ability to embed methods from new languages), we will attempt to enable them. There have been requests for Python, for example. Also, there have been requests for generation of example instance diagrams. There have also been requests to enable Umple to generate code that would work with sources of 'Big Data' corresponding to UML models.

CONCLUSION

Textual modeling languages like Umple allow developers to use powerful modeling constructs, while also maintaining the numerous advantages of human-readable textual languages such as the ability to comment effectively, use textual toolchains and do effective version control.

Umple shares some features with other UML-based textual languages, such as having a syntax for associations and state machines. Some competing tools are also open source. However, Umple's key features that distinguish it from other modeling technology, including other textual modeling languages are:

- It transparently blends modeling constructs with code in multiple target languages, resulting in a single codebase.
- It generates comprehensive code sufficient for the development of diverse and complex systems, including of itself. It is the only modeling tool we are aware of that was developed in itself.
- It incorporates special features that only work effectively in textual languages to allow separation of concerns, feature-oriented development and product-line development such as mixins, mixsets, traits, text-emission templates and aspects.

- Its codebase can be organized in numerous ways, such as by class, by subsystem, by feature, and/or by separating pure model from user-defined methods.

Whether or not Umple becomes a major player in the commercial modeling market, we strongly suggest that other others developing modeling tools and languages should consider incorporating similar features.

Umple is also developed in a test-driven manner and has been hardened by widespread use in the education market.

The core lessons from our research and the success of Umple are:

- Modeling in UML can be successfully undertaken in a user-friendly, tool-agnostic way in order to build systems of all sizes. Research and experience shows that mainstream developers (other than those working in safety-critical domains) are reluctant to use UML except in an informal way because of weak or complicated tools. But Umple shows such limitations can be overcome.
- It is highly beneficial for a modeling language to have a human-readable textual form to make use of models simpler, to facilitate agility, and so that text-based tools and language features can be used. This does not preclude the simultaneous use of diagrams viewing and allowing editing of the same model: In fact, code and diagram forms of a model work synergistically with each other.

The next steps for Umple include 1) integrating the ability to use it to conduct modeling experiments; 2) incorporating requirements and other computation-independent modeling constructs, and 3) adding other customer-driven capabilities such as new transformation targets.

ACKNOWLEDGMENT

The authors acknowledge the 60 students and other open source developers who have contributed to Umple and are listed in the Umple License file on Github.

This research was supported by the Natural Sciences and Engineering Research Council of Canada grant numbers 453224, 569913 and 634504.

REFERENCES

Adesina, O., Lethbridge, T.C., Somé, S., Abdelzad, V., & Boaye Belle, A. (2018). Improving Formal Analysis of State Machines with Particular Emphasis on And-Cross Transitions. *Computer Languages, Systems and Structures*, 54, 544-585.

Abdelzad V., & Lethbridge T.C. (2015). Promoting Traits into Model-Driven Development, *Software & Systems Modeling,* 16(4) 997-1017.

Abstratt (2020). TextUML, Retrieved from http://abstratt.github.io/textuml/readme.html

Agner, L.T.W., & Lethbridge, T. C. (2017). A Survey of Tool Use in Modeling Education. *20th International Conference on Model Driven Engineering Languages and Systems* (MODELS) (pp. 303-311). IEEE.

Agner, L.T.W., Lethbridge, T.C. & Soares, I.W. (2019). Student Experience with Software Modeling Tools, *Software & Systems Modeling*, 18 (5), 3025-3047.

Badreddin, O., & Lethbridge, T. C. (2012). Combining Experiments and Grounded Theory to Evaluate a Research Prototype: Lessons from the Umple Model-Oriented Programming

Technology. *First International Workshop on User Evaluation for Software Engineering Researchers* (USER) (pp. 1-4). IEEE.

Badreddin, O., Forward, A., & Lethbridge, T. C. (2012). Model Oriented Programming: An Empirical Study of Comprehension. *Conference of the Center for Advanced Studies on Collaborative Research* (pp. 73-86). IBM Corp and ACM.

Badreddin, O., Forward, A., & Lethbridge, T.C. (2013a). Exploring a Model-Oriented and Executable Syntax for UML Attributes, *Software Engineering Research, Management and Applications*, (pp. 33-53) Prague, Springer.

Badreddin, O., Forward, A., & Lethbridge, T.C. (2013b). Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity, *Software Engineering Research, Management and Applications* (pp. 129-149), Prague, Springer.

Badreddin, O., Lethbridge, T.C., & Forward, A. (2014). A Novel Approach to Versioning and Merging Model and Code Uniformly, *2nd International Conference on Model-Driven Engineering and Software Development* (*MODELSWARD),* Portugal*, (*pp. 254-263*)*, INSTICC and IEEE.

Bendix, L. & Emanuelsson, P. (2008). Diff and Merge Support for Model Based Development. *International workshop on Comparison and Versioning of Software Models (CVSM '08)* (pp. 31–34) ACM.

Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., & Sabetzadeh, M. (2006). A Manifesto for Model Merging. *International Workshop on Global Integrated Model Management (GaMMa '06)* (pp. 5-12), ACM.

Dévai, G., Kovács, G.F. & An, A. (2014). Textual, Executable, Translatable UML. *OCL@ MoDELS*, (pp. 3-12).

Essebaa, I., & Chantit, S. (2018). Model Driven Architecture and Agile Methodologies: Reflexion and discussion of their combination. *Federated Conference on Computer Science and Information Systems* (FedCSIS) (pp. 939-948). IEEE.

Forward, A. (2010). The Convergence of Modeling and Programming: Facilitating the Representation of Attributes and Associations in the Umple Model-Oriented Programming Language (Doctoral Dissertation, University of Ottawa, Canada).

Forward, A., & Lethbridge, T.C. (2008). Problems and Opportunities for Model-Centric Versus Code-Centric Software Development: A Survey of Software Professionals, *International Workshop on Models in Software Engineering*, (pp. 27-32), ACM.

Garzon, M, & Lethbridge, T.C., (2012). Exploring how to Develop Transformations and Tools for Automated Umplification, *Working Conference on Reverse Engineering (WCRE), (pp. 491-494), IEEE*.

Github (2020a). Umple, Retrieved from http://code.umple.org

Github (2020b). UmpleOnline Module to Allow Running Experiments, Retrieved from https://github.com/umple/umple/issues/1490

Gogolla, M., Büttner, F. & Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming* 69, 27-34.

Husseini Orabi, M, Husseini Orabi, A., & Lethbridge T.C. (2020). Umple-TL: A Model-Oriented, Dependency-Free Text Emission Tool, *Communications in Computer and Information Science*, 1161, Springer, 127-155.

Jayaraman P., Whittle, J., Elkhodary, A.M., & Gomaa, H. (2007). Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. *Model Driven Engineering Languages and Systems. MODELS 2007. Lecture Notes in Computer Science,* 4735. Springer, 151-165.

Lethbridge, T.C, & Algablan, A., (2018). Using Umple to Synergistically Process Features, Variants, UML Models and Classic Code, *International Symposium on Leveraging Applications of Formal Methods*, Cyprus, (pp. 69-88),Springer.

Lethbridge. T.C., (2019). UmpleOnline as a Testbed for Modeling Empirical Studies: A Position Paper, *Fourth International Workshop on Human Factors in Modeling (HuFaMo)*, Munich, (pp. 412-413) IEEE

Lethbridge, T.C., Mussbacher, G., Forward, A., & Badreddin, O. (2011). Teaching UML using Umple: Applying Model-Oriented Programming in the Classroom. *Software Engineering Education and Training (CSEE&T), (pp. 421-428*) IEEE.

Lundell, B., Lings B., Persson A., & Mattsson A. (2006). UML Model Interchange in Heterogeneous Tool Environments: An Analysis of Adoptions of XMI 2. *Model Driven Engineering Languages and Systems (MODELS). Lecture Notes in Computer Science,* 4199. Springer, 619-630.

Medvidovic, N., Egyed, A. & Rosenblum, D.S. (1999). Round-trip Software Engineering Using UML: From Architecture to Design and Back. *Second International Workshop on Object-Oriented Reengineering (WOOR'99),* Toulouse, France (pp. 1-8)

Mellor, S.J., & Balcer, M. (2002). *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman.

Meservy, T. O., & Fenstermacher, K. D. (2005). Transforming software development: an MDA road map. *Computer*, 38(9), 52-58.

Petre, M. (2013). UML In Practice. *35th International Conference on Software Engineering (ICSE)* (pp. 722-731), IEEE.

Rhazali, Y., Hadi, Y., & Mouloudi, A. (2016). CIM to PIM Transformation in MDA: From Service-Oriented Business Models to Web-Based Design Models. *International Journal of Software Engineering and Its Applications*, 10(4), 125-142.

Rhazali, Y., El Hachimi, A., Chana, I., Lahmer, M., & Rhattoy, A. (2020). Automate Model Transformation From CIM to PIM up to PSM in Model-Driven Architecture. *Modern Principles, Practices, and Algorithms for Cloud Security* (pp. 262-283). IGI Global.

Rockstrom, A., & Saracco R. (1982). SDL-CCITT specification and description language. *IEEE Transactions on Communications*, 30(6), 1310-1318.

Umple (2020a). Umple User Manual. Retrieved from http://manual.umple.org

Umple (2020b). Class Diagram of the Umple Compiler Generated by Umple. Retrieved from http://metamodel.umple.org

Umple (2020c). Umple Compiler API of Generated Java Files. Retrieved from http://javadoc.umple.org

UmpleOnline (2020), Retrieved From http://try.umple.org

KEY TERMS AND DEFINITIONS

Aspect: The representation of some part of software that describes how certain source code is to be blended into other source code, via a pattern-matching process.

Association: A UML and Umple modeling abstraction that connects classes and describes the links between instances that can be created at runtime. Associations in Umple result in generation of methods to manage links and maintain referential integrity.

Attribute: A data item listed in a class description; each instance of the class will have this data. In Umple, attributes are subject to constraints and result in the generation of accessor methods.

Mixin: A lightweight block of code that can be blended into a textual program or model to add features. In Umple, multiple mixins declaring the same class can be combined to build a class. The multiple mixins can be in separate files or a single file. Mixins are a syntactic feature, as no analysis is done prior to the combination; this in contrast with Traits.

Mixset: A set of mixins sharing the same name that can be made active by an Umple use statement in order to add a feature to a model in Umple.

Trait: A class-like modeling entity that groups various class features, such as associations, attributes, state machines and methods, to facilitate separation of concerns, feature-driven development, product lines and multiple inheritance. In contrast to a mixin, it has strong semantics so various errors can be detected.

UML: A widely used language using many types of diagrams to model software, including state machines and class diagrams.

Umple: A textual language for modeling software that incorporates many UML constructs as well as textual constructs such as mixins, mixsets, traits and aspects.

List of Responses to Reviewers

We thank the reviewers for their positive comments, which we have not repeated here. Below we list the various suggestions or criticisms, and indicate the changes we have made to respond to each.

REVIEWER 1

It lacks experimentation setup which can be added to strengthen the paper.

> **Done. We have added a section called EVIDENCE FOR THE EFFECTIVENESS OF THE APPROACH to respond to this**

…experimentation setup needs to be added in order to show that the proposed method is better than other existing methods thus making it unclear whether the new approach is accurate, effective or efficient.

> **As mentioned, we have added a section on evidence. However, we have also shown Umple Developed in itself as a case study as a large tool and pointed out that no other tool has been developed in itself.**

1. The paper lacks experimentation setup which must be added to strengthen the paper.

> **Done. See the comment above**

2. The authors are advised to include main contribution in introduction

> **Done. See the new third paragraph in the introduction.**

… and research gaps in literature section.

> **Done. We have added a section entitled Research Questions Addressed**

REVIEWER 2

at least one paragraph must be presented which introduces the MDA approach, thus transforming them from CIM to PIM "https://www.semanticscholar.org/paper/CIM-to-PIM-Transformation-in-MDA%3A-from-Business-to-Rhazali-Hadi/cbdcab8aa471434d53d69e9695c6ce43a35ba862" and from PIM to PSM "https://www.igi-global.com/chapter/automate-model-transformation-from-cim-to-pim-up-to-psm-in-model-driven-architecture/238912".

> **Done. A section entitled UMPLE IN THE CONTEXT OF MODEL-DRIVEN ENGINEERING has been added discussing MDA and its relation to Umple, and citing these references.**

We also added a mention of MDA in the 'Desired Characteristics' section to ensure the paper flows better to the new section.

Title, authors/affiliations, abstract, and keywords must be removed from your document

Done. This was not entirely clear since the template included these.

Each table must have a corresponding callout

Each table does indeed have a callout. Same for figures.

In the conclusion you must clearly describe the next works.

Done. We have added a new last paragraph to summarize the next steps (which had been described in more detail in the Future Research Directions section).