Model Oriented Programming: Bridging the Code-Model Divide

Omar Badreddin
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada
obadr024@uottawa.ca

Timothy C. Lethbridge
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada
tcl@site.uottawa.ca

***Abstract*—Model Driven Engineering proposes the use of models as the main development artifacts. This methodology involves generating code from models and then perhaps adding some details to the generated code. Frequently, it is required to also reverse-engineer code to generate models. Despite the wide acceptance of modeling benefits, the use of Model Driven Engineering practices remains limited. We present model oriented programming as a new paradigm to reduce the ever-present tension between model-centric and code-centric development styles. The core of this approach is to add UML abstractions such as associations and state machines directly into a high-level programming language code. In this approach, model diagrams become just another abstract view of the code. The need for reverse engineering is eliminated, since everything in the diagram is represented directly in the code. Model orientation is illustrated using Umple, a working model oriented programming platform. A functional layer of an airline reservation system is presented as a case study.**

I. INTRODUCTION

Model Driven Engineering [1] promises significant benefits to the software engineering community. Models are more compact and abstract than code and are typically rendered as diagrams. Therefore, models are generally easier to comprehend and serve as a great medium for communication and specifications. However, and despite the broad availability of modeling tools, modeling adoption remains very low [2].

Software engineering is a fast changing and evolving domain. The community has proved many times its willingness to, not only adapt to change, but adopt new technologies, even when supporting evidence of that technology's superiority is absent. Surprisingly, and even though empirical evidence supporting modeling is extensive, modeling has not witnessed a wide broad adoption. This has attracted a number of researchers to investigate why not [3].

We start by an analysis of the state of the art. We examine some of the prevalent practices in software engineering of modeling and coding. Our motivating questions are: to what extent do software engineers use modeling in practice? What are the factors behind this level of modeling infiltration? After this analysis, we present a working research prototype inspired by the findings of our analysis.

II. STATE OF THE ART MODELING PRACTICES

Modeling enables developers to work at a higher level of abstraction. This view seems to be well accepted in both academia and industry. However, the overwhelming majority of developers still treat source code as their primary work medium [2]. Software engineers may draw diagrams when explaining concepts in design meetings, and include some of them in design documents, but relatively few use models to actually drive development effort. Some communities, like the open source developers, remain entirely code-centric.

A. Literature Survey

There has been some research into why modeling is not more enthusiastically adopted. Afonso et al [4] state that although modeling is standard in the database design community (where use of Entity Relationship Diagrams is the norm), "there is little practical evidence of the impact" of model-driven approaches among the broader software engineering community.

There are clearly a number of common obstacles to modeling. Berenbach et al [5] suggest that these include a belief among developers that modeling is only about drawing "pretty pictures", and not understanding well enough how to model in the prevalent object-oriented paradigm.

Modeling is particularly important for safety-critical systems. Anda et al. [6] studied practitioners in this domain and note that they had good results when they adopted a more model-oriented approach. Difficulties using modeling tools and the costs of training were the biggest obstacles. Another obstacle was tendency of management to remain oriented towards the production of source code.

The Object Management Group (OMG) is the main industry association interested in promoting modeling. They supported Dobing and Parsons [7] in a survey in which 171 practitioners were asked how they use UML in practice. Most respondents felt that UML is indeed useful in software development, but half said they did not fully understand class diagrams, the most widely used diagraming notation in UML. The study concluded that complexity of UML is one of the main obstacles to its use. In another study [8], there was evidence that those who felt they know UML, often were not good at creating correct UML models.

One issue to consider is whether performing modeling has sufficient return on investment. Arisholm et al [9] concluded from a controlled study that the costs of maintaining UML documentation may sometimes outweigh the benefits of modeling.

Another issue is the usability of modeling tools. Agerwal et al [10, 11] studied this issue and found that poor usability contributes to higher than necessary costs associated with modeling.

Adoption is, of course, an issue with many tools and technologies. Iivari [3] explored why modeling tools are not used in general in software projects. In a study, he reports, that 70% of modeling tools are never used after being available for one year, 25% are used by only a single group, and only 5% are widely used. Sultan and Chan [12] provide an in-depth discussion of object-oriented technology adoption. They conclude that lack of adoption is likely not due to intrinsic weaknesses in the technology, but has more to do with culture and management.

B. Surveying Practitioners

Our survey received responses that seem more positive that what is reported in the literature cited above [2]. Well over half of our respondents do in fact perform some type of modeling, with about 52% using UML often or always. 60% use visual notations to document their code prior to design, although only a third always do this. However, only 17.5% often or always generate code from models and 36.5% never do this. Most of the value of models is therefore to document and communicate designs. Eighty percent in fact said that modeling tools are poor or awful at the task of generating all the code for a system.

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

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

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

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

C. Summary of Findings

We conclude from the above that the reasons for lack of more wholehearted adoption of modeling seem to be as follows:

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

b) Modeling tools are too difficult to use;

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

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

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

The remainder of the paper is organized as follows. First, we present model oriented programming, a new paradigm where modeling abstractions are part of the code. We then discuss how the new paradigm addresses the challenge of limited modeling adoption. Finally, we present a flight reservation system as a case study.

III. Model Oriented Paradigm

Model orientation refers to a development style where objects, their relationships and behavior are abstracted within the code. Object orientation, a powerful and widely adopted programming paradigm, abstracts entities and their attributes and methods as objects. This paradigm is widely available in the majority of modern programming languages, like Java and C++.

There has been a continuous trend in increasing levels of abstractions. Prior to object orientation, programming languages were procedural, where computational tasks are grouped under procedures that can be referenced. Computational concepts like the while loop and the for loop were, similarly, abstractions added to earlier generations of programming languages.

Model orientation is the natural next step in the evolution of abstraction in programming languages, where objects' behavior and relationships are also abstracted in the code. To demonstrate this paradigm, we introduce Umple, a model oriented programming language.

A. The Umple Model Oriented Programming Language

UML models describe, among other things, objects and their relationships (class diagrams), behavior and object states (state machine diagrams). The model-oriented programming technology we are presenting adds these two key modeling abstractions within the code. This means that the programming language not only contains classes (types of objects), but also associations (relationships) between classes, and state machines (behavior of the objects).

Umple is an implementation of model orientation available for Java, PhP, and Ruby, with C++ coming soon. Umple also supports constraints similar to OCL, model level tracing and debugging. In this paper, we limit ourselves to Java, and to associations and state machines.

A key philosophy behind Umple is that modeling and coding are essentially the same activity, with no clear boundary between them and with the only differences being the level of abstraction and the completeness of the artifact.

If one just writes Java code in an Umple file, without any UML abstractions, then one is doing traditional coding, but one could also consider oneself to be doing low-level modeling, especially if one is creating an incomplete prototype of code sketch.

On the other hand, if one just sketches classes, associations and state machines in an Umple file, without any main program or custom algorithm, then one is undoubtedly doing modeling. But at the same time, one is also using exactly the same process as a programmer: Writing textual code.

Umple encourages textual development, but also allows development using diagrams. For example, one can draw a class diagram using Umple tools: As one does this, the textual code is dynamically edited. In essence, Umple enables users to model textually and code visually.

The best way to demonstrate the above ideas is by an example, presented in the next section.

B. Motivating Example

Consider the sample system in Listing 1. At first glance, the lines of code may look familiar to a Java developer, but lines 4, 6 to 17, 18, and 21 represent modeling elements embedded within the Java code.

Line 18 indicates that class Student has a many-to-optional one relationship with class Advisor. Similarly, lines 4 and 21 indicate that the class Student and Advisor, respectively, inherit from class Person. These modeling elements are typically represented visually in a UML class diagram, such as the one in Figure 1.

Lines 6 to 17 describe the behavior of the Student class with a state machine following UML semantics. The equivalent UML state machine model for these lines of Umple code is depicted in Figure 2. When a student object is created, it starts at the Admitted state. The state machine then specifies different states it goes to in response to events. For example, when the event quit occurs, the student object goes into state Quit.

This modeling/coding paradigm is not intended to replace the need for visual models, but rather it compliments it. Umple online [13], an online based editor and compiler, demonstrates the relationship between code and model. A developer may start to draw the system visually by adding classes and associations. At any point, the developer may also wish to edit the system textually; in this case, if the change is purely in the code, then no change happens to the visual representation. However, if the developer changes a modeling element textually, like adding or editing an association, the change is automatically reflected on the visual model. Similarly, visual editing is reflected automatically on the code.

```
1    class Person { }
2
3    class Student {
4      isA Person;
5      Integer stNum;
6      status {
7        Admitted {
8          quit -> Quit;
9          enroll[!hold]->Enrolled;
10       }
11       Enrolled {
12        quit -> Quit;
13        graduate-> Graduated;
14       }
15       Graduated {}
16       Quit {}
17     }
18     * -- 0..1 Advisor; }
19
20   class Advisor {
21     isA Person; }
```

Listing 1: Sample Umple code/model [14]

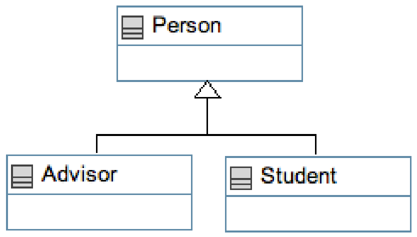

Figure 1: A class diagram in UML

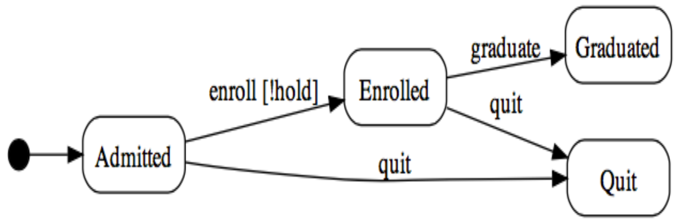

Figure 2: State machine model

This approach has the following key benefits. First, it eliminates the need for the generation of user-editable code, and reverse engineering when such code is edited. These steps are together called 'round-trip engineering'. Umple generates code in the same manner that any compiler generates code. It happens to be the case that the generated code is highly readable, because we want to enable educational use of Umple, and certification of safety-critical code, where every artifact must be inspected. But the cardinal rule for Umple modelers/programmers is to never modify the generated code.

The second key benefit is that software engineers that are comfortable with the textual paradigm can continue with their style of coding. We expect that such a paradigm can enhance the level of modeling adoption.

As part of evaluating this paradigm, we have conducted a controlled experiment to measure the comprehensibility of Umple as compared with Java and UML [14]. The key findings are that model oriented code is easier to comprehend that the equivalent Java code. The experiment also concludes that model oriented code is able to retain the comprehensibility benefits of a visual modeling notation like UML.

IV. Open Source Examples of Umple in Use

We have created a significant body of Umple code to act as a test suite. Samples of Umple code is available online [13]. The largest test case is Umple system itself. Umple is developed entirely using Umple and is open source [15]. Contributors to the Umple system commit Umple code. The build process automatically compiles Umple code and generates executable artifacts.

A. Airline System

To demonstrate usage of Umple, we present a short sample airline system built as a case study.

To make the system conform to real airlines, we downloaded the entire Air Canada's flight schedule which includes code-shared flights from Star Alliance and affiliated airlines. This schedule is made public by Air Canada on its website. We processed the schedule to extract the essential data (using Excel Macros); we then reverse-engineered what the schema must look like.

A result of our reverse engineering effort was the creation of the following Umple code. The reader can generate the diagram and executables from the following using UmpleOnline [13].

Next we wrote a simple Java program to load all 10,000 objects into Umple-generated system and perform some queries that navigated the associations to search for flights matching various criteria. This case study demonstrates that Umple can be used rapidly and in a realistic context. The resulting Umple-generated system could be put to serious use for creating an open-source tool for browsing airline schedules.

B. Umple System

The Umple system was entirely developed in Umple itself. The first versions of the system were developed in Java, and then it was iteratively ported to Umple. We refer to the process of reverse engineering a system from a modern high level programming language to Umple as Umplification. Umplified systems tend to have fewer lines of code. Modeling abstractions that are embedded in the code enhance comprehensibility and reduces maintenance effort.

The entire Umple code base is published on Google Code [15]. For purposes of this paper, we illustrate a part of the Umple metamodel for state machine modeling.

As shown in **Listing 3**, Umple's state machine metamodel is implemented in Umple. The metamodel is similar to OMG's UML metamodel for state machines. A key difference is that the metamodel developed in Umple is both a model and code at the same time. For example, attributes in the metamodel are initialized. The actual full metamodel also contains methods written in Java.

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

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

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

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

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

class AirplaneType {
   String typeCode;
   1 -- * RegularFlightSchedule;
}

class Airport {
   String code;
   String name;
}

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

Listing 2: Airline system

```
class State
{
  name;
  Boolean isConcurrent =
 { numberOfNestedStateMachines() > 1 }

  1 -- 0..1 Activity;
  0..1 -> * Action;
  * -- 1 StateMachine;

  Boolean isStartState = false;
  Boolean isInternal = false;
  Boolean isHistoryState = false;
  Boolean isDeepHistoryState = false;
  Boolean finalState = false;
}

class Activity
{
  activityCode;
  * -> 0..1 Event onCompletionEvent;
  CodeBlock codeblock = null;
}

class Transition
{
  Boolean isInternal = false;
  Boolean autoTransition = false;

  * -> 0..1 Event;
  * -- 1 State fromState;
  * nextTransition -- 1 State nextState;
  * -> 0..1 Guard;
  0..1 -> 0..1 Action;
}

class Action
{
  actionType = null;
  lazy Position position;
  actionCode;
  Boolean isInternal = false;
  CodeBlock codeblock = null;
}

class Event
{
  name;
  String args = null;
  Boolean isTimer = false;
  Boolean autoTransition = false;
  timerInSeconds = "0";
  Boolean isInternal = false;
}
```

Listing 3: Umple state machine metamodel

V. UML Action Languages

Action languages emerged to fill in the gap between the highly abstract and visual model notations to manage structure and relationships, with the more algorithmic manipulation of the model's structure (i.e. programming language-like-statements). This gap, commonly referred to as 'execution semantics', has not yet been completely formalized. In that sense, both Umple and UML action languages attempt to bridge the model/code gap. Umple attempts to bridge this gap by introducing additional abstract constructs that matches the abstraction level of models, while at the same time reusing the syntax of current object oriented languages. On the other side, Action Languages attempts to bridge this gap by introducing yet another high level syntax to bridge this gap.

A UML action language (UAL) is geared towards describing elements of a system, such as actions, algorithms, and navigation paths, which are not readily described by typical UML diagramming notation. Snippets of languages like C++ and Java can be used as a UAL, but such languages are unaware of UML abstractions, resulting in mixed levels of abstractions and 'boilerplate' code.

Current directions in standardizing action languages for UML take a top-down approach, where a new language and new constructs are defined forming an additional layer of abstraction. Umple adopts an alternative approach; namely, adding modeling abstractions to existing object oriented languages.

The reasons for not using an existing programming language can be summarized in the following four points. These mirror the points expressed by Mellor et al [16].

A. Programming Languages Provide Much More than What an Action Language Needs.

The java console I/O statements, and the variety of UI frameworks for Java are examples where the programming language is too powerful for what is needed from an action language. A programming language (such as Java) provides a large number of statements and libraries to accomplish the same or similar effect, which is to display output. Similarly, programming languages provide considerable freedom regarding how instance variables and methods can be used to represent and manipulate properties and relationships. The abstract UML concepts of attributes and associations therefore have many concrete mappings; when presented with implementation code, the software developer has a hard time seeing the abstractions. A UML action language can hence abstract the most commonly used concepts and make the algorithmic elements in models easier to understand.

B. Commitment to Implementation

When programming an abstraction such as an association in a language like Java, one is forced to choose the low-level details, such as the names of methods and the algorithms. It is hard to change these later. As another example, when implementing a state machine, one may choose to use a string attribute, but one may later on decide to change to an *enum* and hence have to change the code considerably. On the other hand, if using a UAL, this decision would be made by the compiler or code generator, and could be changed simply by changing the some configuration options.

C. Programming Languages Do not Directly Support UML Concepts such as Association or States.

A language like Java does not have constructs for the representation of associations or state machines, and consequently does not promote abstract thinking on the part of programmers. Instead, programmers have to implement the behavior specified in a state machine model using a case statement or enum data structure.

D. Declarative Constraint Languages, such as OCL, Lack the Support for Algorithmic Level Specification.

OCL-like languages do a good job in navigating associations and defining pre and post conditions, but generally, do not support effective implementation of algorithms.

VI. CONCLUSIONS

It is evident that the software engineering community relies much less on modeling than it ought to. There are multiple factors behind this low level of modeling adoption. These factors may include risk aversion, reliance on code centric approaches that work good enough, and the fact that modeling tools are complex and require significant investment in time and effort.

We have presented model oriented programming, a new programming/modeling paradigm where modeling elements are also present in the code. This paradigm enables software developers to model in the same manner they would write code. This paradigm is particularly suitable for software developers who are used to textual editing paradigms. At the same time, developers can benefit from the visual models that tend to be a better medium for communication and documentation.

Umple is an example of a model-oriented language where abstractions such as associations and state machines are abstracted in the code. We have used Umple to write the Umple compiler itself. We have also demonstrated in this paper two case studies; one using Umple for implementing the core of an airline reservation system, the other is a case study of Umple system itself.

Model oriented programming has some similarities with UML action languages. Both are an attempt to fill the abstraction gap between model and code. Action languages attempt to do so by defining a new language and new constructs that are at the same abstraction level as the models. Model-oriented programs attempt to do so by adding modeling abstractions within existing object oriented programming languages.

Model orientation can help bridge the gap between model-centric developers and code-centric developers because it allows both to continue to do what they prefer, while also giving them the benefits of the alternative approach.

REFERENCES

[1] Object Management Group (OMG). " OMG Model Driven Architecture", accessed 2010, http://www.omg.org/mda/.

[2] Forward, A., Badreddin, O. and Lethbridge, T. C. "Perceptions of Software Modeling: A Survey of Software Practitioners," in *5th Workshop from Code Centric to Model Centric: Evaluating the Effectiveness of MDD (C2M:EEMDD),* 2010. Available: http://www.esi.es/modelplex/c2m/papers.php

[3] Iivari, J. "Why are CASE Tools Not used? Communications of the ACM". 1996. *Communications of the ACM,* vol 39, pp. 94-103.

[4] Afonso, M., Vogel, R. and Teixeira, J. "From Code Centric to Model Centric Software Engineering: Practical Case Study of MDD Infusion in a Systems Integration Company," in *MBD-MOMPES '06: Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software,* 2006. pp. 125-134.

[5] Berenbach, B. and Konrad, S. "Putting the "Engineering" into Software Engineering with Models," in *Modeling in Software Engineering, 2007. MISE '07: ICSE Workshop 2007. International Workshop on,* 2007. pp. 4-4.

[6] Anda, B., Hansen, K., Gullesen, I. and Thorsen, H. "Experiences from Introducing UML-Based Development in a Large Safety-Critical Project". 2006. *Empirical Software Engineering,* vol 11, pp. 555-581.

[7] Dobing, B. and Parsons, J. "How UML is used". 2006. *Commun ACM,* vol 49, ACM Press. pp. 109-113.

[8] Farah, H. and Lethbridge, T. C. "Temporal Exploration of Software Models: A Tool Feature to Enhance Software Understanding," in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on,* 2007. pp. 41-49.

[9] Arisholm, E., Briand, L. C., Hove, S. E. and Labiche, Y. "The Impact of UML Documentation on Software Maintenance: An Experimental Evaluation". 2006. *IEEE Trans. Software Eng.,* vol 32, pp. 365-381.

[10] Agarwal, R., De, P., Sinha, A. P. and Tanniru, M. "On the Usability of OO Representations". 2000. *Comm ACM,* vol 43, ACM Press. pp. 83-89.

[11] Agarwal, R. and Sinha, A. P. "Object-Oriented Modeling with UML: A Study of Developers' Perceptions". 2003. *Commun ACM,* vol 46, ACM Press. pp. 248-256.

[12] Sultan, F. and Chan, L. "The Adoption of New Technology: The Case of Object-Oriented Computing in Software Companies". 2000. *IEEE Trans. on Engineering Management,* vol 47, pp. 106-126.

[13] Lethbridge T.C., Forward, A. and Badreddin, O. " Umple Language Online.", accessed 2012, http://try.umple.org.

[14] Badreddin, O., Forward, A. and Lethbridge, T. C. "Model Oriented Programming: An Empirical Study of Comprehension". 2012. *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research,* pp. 73-86.

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

[16] Mellor, S. J., Tockey, S. R., Arthaud, R. and Leblanc, P. "An Action Language for UML: Proposal for a Precise Execution Semantics". 1999. Lecture notes in computer science, Springer. pp. 307-318.