The Convergence of Modeling and Programming:

Facilitating the Representation of Attributes and Associations in the Umple Model-Oriented Programming Language

by

Andrew Forward

PhD Thesis

Presented to the Faculty of Graduate and Postdoctoral Studies in
partial fulfillment of the requirements for the degree
Doctor of Philosophy (Computer Science[1])

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, K1N 6N5
Canada

[1] The Ph.D. program in Computer Science is a joint program with Carleton University, administered by the Ottawa Carleton Institute for Computer Science

Acknowledgements

A very special, and well-deserved, thank you to the following:

a) Dr. Timothy C. Lethbridge. Tim has been a mentor of mine for several years, first as one of my undergraduate professors, later as my Master's supervisor. Tim has again helped to shape my approach to software engineering, research and academics during my journey as a PhD candidate.

b) The Complexity Reduction in Software Engineering (CRUISE) group and in particular Omar Badreddin and Julie Filion. Our weekly meetings, work with IBM, and the collaboration with the development of Umple were of great help.

c) My family and friends. Thank you and much love Ayana; your support during this endeavor was much appreciated despite the occasional teasing about me *still* being in school. To my mom (and editor) Jayne, my dad Bill, my sister Allison and her husband Dennis. And, to my friends Neil, Roy, Van, Rob, Pat, and Ernesto – your help will be forever recorded in my work. Finally a special note to Ryan Lowe, a fellow Software Engineer that helped to keep my work grounded during our lengthy discussion about software development – I will miss you greatly.

d) Software professionals around the world. Sincere thanks to the individuals that participated in my research, published valuable references for my writing, as well as to those in the various news-groups about software engineering that I follow. Your knowledge and insight helped provide the necessary substance for my work.

Abstract

This thesis investigates approaches to model-driven development (MDD) in which developers can keep using their familiar textual programming languages, but with additional model-oriented concepts. The added concepts include associations and attributes as found in the Unified Modeling Language (UML), as well as concepts from software patterns and other common programming abstractions.

By keeping text at the forefront of development, we maintain all of the advantages of text, without having to sacrifice the benefits of diagrams. By allowing a model to be equally expressed in either diagrammatic or textual form, we enable what we have termed *text-diagram duality*, a duality that benefits programmers and modelers alike.

We explore why software developers in some situations prefer diagrams, but tend to revert to textual means to write and maintain software systems. To explore the capabilities of modeling in code, we developed a model-oriented programming language called Umple. At its core, Umple is a family of object-oriented languages enhanced with additional abstractions. Umple supports both platform-independent models (PIM), as well as platform specific models (PSM). Umple currently integrates with Java, PHP and Ruby; referred to as base languages throughout this thesis. Our research focuses on investigating the opportunities and obstacles we discovered in the course of implementing and using UML-like associations and attributes in Umple.

It is our hypothesis that current features available in object-oriented languages can be enhanced with a more model-oriented approach, providing a textual form for modeling concepts that have been primarily available diagrammatically. By providing modeling abstractions in a programming language, the complexity and size of the resulting systems, we argue, is reduced and more developers, particularly those who are used to writing code, will be more eager to adopt modeling practices. At the same time, our approach maintains the benefits of diagrammatic approaches to software development, since Umple can be rendered and edited as a UML-like diagram.

Our primary contributions to the field of computer science are as follows: First, we provide an empirical investigation on the nature of modeling practices. Second, we present the design, implementation and analysis of a model-oriented language, Umple. The language is presented as enhancements to existing programming languages including Java, PHP and Ruby.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

In our research, we investigate to what extent software modeling can be achieved in textual form, without hindering a diagrammatic representation. By understanding the dual nature of a model (in textual and diagrammatic form), we hypothesize that software engineering could be improved by exploiting the efficiencies of text without losing the communicational power of diagrams.

In this thesis we present the design and analysis of a technology called Umple. Umple allows textual modeling, and integration of code and model, all without losing the ability to model graphically. We discuss the following: The research into modeling in practice that led us to develop Umple; the syntax, semantics and code generation of Umple, particularly with regard to associations and attributes; and, our analysis of the use of Umple in several real systems. We also compare Umple to other technologies that have related goals.

For our work, we consider a *software model* to be an artifact that represents an abstraction of the software system being built. A model can typically be viewed as a set of diagrams and/or pieces of text. It can be recorded on a white board, paper, or using a software tool. A model can use formal syntax and semantics, but this is not necessary.

Software engineering currently adopts a range of approaches from code-centric to model-centric. In a pure code-centric approach, software engineers write the code for a system entirely in a textual programming language. The code is then compiled and linked with libraries to create a working system. In a model-centric approach, the system is instead largely generated from more abstract models created using modeling languages. Modeling languages, the most prominent of which is UML, tend to be diagrammatic with some text inserted as necessary. There seems to be a strong opinion among software engineering experts that model-centric approaches are now considered a *best practice* for software development, since they allow developers to apply powerful abstractions, increase productivity, improve quality, standardization and formalization [11, 12]. However, code-centric approaches still appear to be dominant in practice [12]. Developers seem divided into de facto *camps* ranging from those that believe in a model-centric (see glossary for a definition of this term as well as many others used throughout this thesis) approach to those that are quite opposed to it and who prefer to simply *write code* in a code-centric fashion. In what follows, we refer to this divergence of both practice and attitude as the *model-code divide*.

A key hypothesis we investigated in the work leading to this thesis is that the most important concepts of a diagrammatic modeling language can be rendered into a usable and useful textual form that extends or is similar to a programming language. And, by enhancing programming languages with model-oriented syntax, we raise their level of abstraction to better align with the intentions of designers. By *usable* we are referring to enabling more efficient and effective creation and use by *developers*. By contrast, XML-based notations such as XMI, primarily serve to allow exchange of models among *tools*. Our language, Umple, has advantages over approaches such as EMF, xUML, and SDL, which we will explore in Sections 3.8 .

We used two main approaches throughout our research. The first was to conduct empirical studies into how modeling is practiced, or not practiced, as the case may be. The second approach was to design, implement and analyze a model-oriented programming language. By understanding how textual languages can be applied in addition to diagrammatic languages, we intend to help bridge the model-code divide. Our bridge in this context will be known as *text-diagram duality.* Text-diagram duality means that the underlying abstraction (i.e. model) of a system can and should be equally expressible both textually and diagrammatically, and should also be capable of being manipulated in both textual and diagram editors.

The goals of our research are: 1) To learn more about the model-code divide in software engineering; 2) To develop and extend tools to better exploit the text-diagram duality of software artifacts; 3) To analyze the effectiveness of languages that leverage modeling concepts as first-class entities, in order to investigate the extent to which software development can be improved. In practical terms, we are trying to help software coders to incorporate more and better modeling concepts in the code they write. Modeling by writing code may be a less-daunting task than trying to get developers to opt for creating diagrams; a view shared by Martin Fowler [13], a vocal proponent of textual modeling.

This is not the first thesis to work in this domain. We build on the work of Brestovansky [14], who worked with us to create the first version of Umple. In this thesis, we focus on deepening our understanding of text-diagram duality, with particular emphasis on UML associations and attributes; other researchers are already working on doing the same for state machines.

1.1 Research Questions

The guiding research questions that have motivated our research are presented below.

RQ1. Why do software practitioners resist the current style of software modeling and show a tendency to prefer to design directly in code?

Although model-centric design has been demonstrated to produce good results, most software engineering is still done using an approach where the model is either secondary to the code, or does not exist [12]. In other words, software engineers resist the use of models. Some of the reasons for this resistance may include:

i. **Habituation**: Software engineers may have become trained or habituated to code-centric thinking;

ii. **Efficacy**: In some contexts, code-centric thinking may in fact be better than model-centric thinking – the costs of modeling may exceed the benefits either as perceived or in practice.

iii. **Politics and Practices**: Political or management practices might lead developers to take a code-centric approach. For example, the business domain may enforce code-based deliverables, or teams may be evaluated using code-centric metrics such as lines of code.

iv. **Software Process**: Some processes advocate early implementation of prototypes or iterations that are then repeatedly refactored or re-implemented to produce the final system. These processes reinforce thinking about code and might make it difficult to integrate modeling.

v. **Tool Weaknesses**: Tools (both languages like UML and design tools) may not yet be capable of supporting model-centric thinking adequately. Modeling tools may also be too large and complex, resulting in obstacles to learning and to creating software with them.

vi. **Intrinsic Utility**: Software developers may find it easier to use the mechanisms that code-editors provide rather than the mechanisms provided by graphical diagram editors. Factors such as the number of 'clicks' or keystrokes to accomplish a task, the amount of data manipulation required, the amount of data that can be seen at one time, and the searchability of the document all may affect utility.

vii. **Software Engineering Education:** Modeling may be an under-emphasized component of software curricula, resulting in software practitioners that lack the necessary skills and knowledge to properly apply modeling concepts. Furthermore, the modeling education that is provided may simply impart textbook syntax and toy examples, leaving students somewhat skeptical, and giving too little guidance as to the pragmatics and practice of modeling in industrial projects.

viii. **Domain suitability:** Particular types of models and modeling might be more appropriate in certain domains. For example, class diagrams might be more suitable in data-dominant software applications, whereas in real-time applications state machines might be more useful. However developers may be most familiar with class diagrams, leading them to feel that modeling is not suitable for real-time software.

We investigate the extent to which some of the reasons above may contribute to the model-code divide. We approach these issues from the programming language perspective to uncover whether improvements to languages can help improve both the quality of software models and the extent to which modeling occurs. We contrast this with approaches that consider code-to-model transformations and reverse engineering.

A trend can be observed in programming languages wherein both semantics and syntax have slowly evolved by progressively increasing abstraction away from of the details of computer architecture and data structure, and towards expression of domain-level concepts. Assembly programming languages provided abstractions of numeric machine code, which was tied to the architecture of the CPU. Assembly was later superseded with early programming languages like COBOL and Fortran that abstracted away the details of the machine code. Böhm and Jacopini later formalized the use of branches and loops to create what is known as structured programming [15]. Ole-Johan Dahl introduced object-oriented programming concepts in Simula [16] which eventually gave rise to programming languages like Java, Smalltalk and C#. Object-oriented languages added several abstractions such as automatic memory management, polymorphism and encapsulation.

Interestingly, human languages have also evolved in similar fashion over thousands of years with a trend towards increasing the level of abstraction [17]. In addition to simply increasing the number of vocabulary items, human languages have developed new terminologies to describe more complex concepts, resulting in more efficient communication media.

If we look at the progression of software languages from the early days of assembler to fully object-oriented languages like Ruby, we can observe another trend in the process of evolution. There appears to be a trend whereby advancements in programming languages are first introduced as visual mechanisms for designing and modeling. For example, flow-charts gave rise to structured programs; and Entity-Relationship diagrams helped spawn object oriented class diagrams. If such a relationship were to exist, then it might be the case that the current trend towards highly diagram-oriented approaches to software development are acting as a precursor or prototype for features yet to be incorporated into textual programming languages.

In this thesis, we address this research question primarily through a survey of software engineers described in Chapter 2.

RQ2. What is the level of modeling adoption in industry, and what are the factors affecting this?

We seek to uncover the extent to which various types of developers perform modeling, and how they approach it. We use the following initial categorization of how developers use modeling:

i. **Model-only**: Approaches where the model is effectively all there is, except for small amounts of code for such things as calculations.

ii. **Model-centric**: Approaches where modeling is performed first, and code is generated from the model, for possible subsequent manual manipulation.

iii. **Model-as-design-aid**: Approaches where modeling is done for design purposes, but then code is written mostly by hand.

iv. **Model-as-documentation**: Approaches where modeling is done to outline or describe the system, largely after the code is written.

v. **Code-only**: Approaches where modeling is almost entirely absent.

To answer this question we conducted meta-analyses of the literature, and surveyed modeling practitioners using a variety of questions. The results of this research question are presented in Chapter 2.

RQ3. Can programming languages be enhanced with model-oriented constructs and provide benefit to software developers?

In the course of investigating RQ1 and RQ2, we uncovered a discrepancy between the actual usage of modeling versus coding and the perceived benefits of working in a model-oriented environment. In essence, there seems to be evidence that software developers believe in the benefits of modeling, but for various reasons they are not able, willing or satisfied to work in a model driven environment.

In answering this question, we set out to design, implement and analyze a language that looks, feels and behaves like current programming languages, but includes additional modeling constructs and software patterns to increase the abstraction of the programming language and to bring modeling into *code*. This allows developers to continue to work with code without sacrificing the ability to work with models.

1.2 Hypothesized Solutions

Our main hypothesis is stated as follows.

> H1: Software development can be enhanced by providing a textual format to write and maintain software constructs and abstractions that have previously been diagrammatic in nature.

Our hypothesis is more than simply providing a textual syntax for software modeling and model notations, it is more deeply rooted in integrating software models at the *code-level* to support the notion that the "code is the model" from a code-centric view, and the "model is the code" from a model-centric view.

Our investigation into RQ1, why software developers seem to revert back to textual behaviours for software development, suggests several possibilities. One is that the use of pure text is a symptom of being unable to change habits and embrace diagrammatic approaches. Alternatively, the inverse may be true: perhaps it is the key benefits of textual approaches that software engineers consciously or subconsciously embrace. Regardless of which of these is true, and they may both be true, our focus is on how textual forms of abstractions can be used to improve software development. Our second research question RQ2 helps to substantiate and clarify the assumptions about how modeling is or is not currently practiced with some initial attempts at explaining why. For RQ3, we set out to enhance object-oriented programming languages with modeling constructs and built-in software patterns to analyze the impacts on software complexity and assess the efficiency of building *real* systems with such a language.

Our research focuses on object oriented software development. We believe more abstract modeling constructs can be added to such languages, that would allow software developers to *model in code*. Our modeling language is called Umple. Umple embeds class-diagram concepts such as attributes and associations (with multiplicity, etc.), state machines, and software patterns. Umple also eliminates (and/or reduces) the need for the resulting systems to have implicit associations coded and duplicated as instance variables, resulting in what is clearly a lot of boilerplate code (see glossary for a definition). Similarly, state-diagram concepts and more advanced software patterns could also be designed into such a textual modeling language.

By providing modeling constructs in a model-oriented language like Umple, we hope to solve many of the issues identified in our research question RQ2. Namely:

- Code-centric thinkers would actually be modeling; there would be a smoother path from code to model.

- If code-centric thinking is better in some way, then one can continue to use it while continuing to model. In other words, any intrinsic benefit of textual approaches can be leveraged.
- If managers like a process that places code first, then the software team can continue to use it, while adopting modeling practices.
- The best features of both code-editing and graphical editing tools can be applied to the problem.

Any added complexity of the new constructs made available by Umple (such as associations and multiplicity) must be offset by the potential gains to be achieved by being able to communicate at a more abstract level. By focusing on text, we should be able to offer a process that allows software practitioners to model without drastically changing their work environments. The focus should be to allow developers to take advantage of all of the benefits of textual software development in addition to the full benefits of visualizing software models based on the *program's* textual equivalent.

Chapter 2 Attitudes Towards Software Modeling – a Survey

As a first step in our research, we sought to understand how modeling is performed in industry. This led us to develop a survey; the aim of the survey is to uncover the attitudes and experiences regarding software modeling and development approaches that avoid modeling. We are motivated by observations that modeling is not widely adopted; many developers continue to take a code-centric approach. We seek to understand the extent to which this is true and the reasons why. We also aim to understand how tools might be enhanced to improve adoption of modeling approaches to software development.

During the course of developing the survey questions, we realized that we needed to gather information about what types of applications our participants develop. Doing this would enable us to determine if modeling practices and application type are correlated; we suspected there may be a strong correlation. However, our research indicated that no one has published a definitive taxonomy of software types; we therefore set out to develop such a study, prior to completing our main survey. Development of the taxonomy is discussed in Section 2.2.

The main survey is discussed in Sections 2.3 to 2.6 . Key results are as follows: 1) UML is confirmed as the dominant modeling notation; 2) modeling tools are primarily used to create documentation and for up-front design with little code generation; 3) modeling tools are also used to transcribe models from other media including whiteboards; 4) participants believe that model-centric approaches to software engineering are easier, but are currently not very popular as most participants currently work in code-centric environments; and 5) the type and quality of generated code is one of the biggest reported problems.

Section 2.2 of this chapter is based on work published in [18] and sections 2.3 to 2.6 are based on [12, 19]. The complete results are available online at [20].

2.1 Attitudes towards Modeling

A significant area of variability in software engineering practice today is the extent to which a development team uses models. At one extreme, model-centric or model-driven development, software engineers use tools to describe the structure and behaviour of their system using a language like UML, and then generate source code for the software automatically. In many cases, they start by using even higher level models, such as business models, from which they generate UML and other models. France and Rump provide an excellent overview [21] of the current state of the art in modeling. At the other end of the spectrum are those developers who work entirely by editing code, i.e. using a code-centric approach.

One of our objectives has been to uncover why modeling is not more widely practiced. Proponents of modeling, and modeling languages, would have us believe it is entirely obvious that most of us should be modeling; they cite successes with modeling and the apparently obvious benefits of working at a higher level of abstraction [22]. However, "in the trenches" one often sees an entirely different perspective. In many organizations, developers primarily toil on vast volumes of source code, rarely looking at diagrams or other models, let alone creating or editing them. This is typified by the open-source community, where in most projects source code file patches are the key unit exchanged [23], and repositories consist almost entirely of code. The very names 'SourceForge' and 'GoogleCode' are telling in this regard. Agile developers also tend to be very code centric. If modeling is so great, why are these people not doing it more?

There has been only a small amount of research into why modeling is not as prevalent as it might be. Afonso et al. [24] point out that although in certain areas of software development modeling is the norm, most notably the design of databases, "there is little practical evidence of the impact" of model driven development in general software engineering. Berenbach et al. [25] describe a number of common modeling problems that lead to models being less effective, and hence less adopted. These include a belief that it is just about "pretty pictures", not understanding the underlying paradigm well enough (typically the object-oriented paradigm), and lack of attention to consistent style in modeling. Anda et al. [26] studied modeling in the safety-critical context, where modeling would seem to be particularly appropriate. They reported that modeling yielded positive results, but that the inadequacies of tools, and the cost of training were important obstacles. Also, it was difficult to apply modeling in a legacy environment, and when development teams are assigned specific blocks of source code to work on; in both these cases the code-centric approach is presupposed.

Dobing and Parsons [27] conducted a web-based survey on how UML is used in practice. They were supported by the Object Management Group and collected 171 responses. One of their results was that class diagrams, although being the most frequently used UML diagram type, are "not well understood" by 50% of analysts. Furthermore 42% felt that use case diagrams are not worth their cost, and 37% felt that use case narrative text is not worth the cost. On a more positive note, the majority of respondents found that all aspects of UML are useful for most projects. The authors suggest that complexity and lack of usage guidelines are the biggest concerns with UML.

Several researchers have conducted controlled experiments to measure the benefit that modeling may provide; however the results have not been encouraging for the modeling community. For example Arisholm et al. [28] concluded that the costs of maintaining UML documentation in the type of software they studied balance the benefits of the modeling. This sentiment is echoed by Agerwal et al., [29, 30] who conducted experiments to examine the usability of modeling tools. Sjoeberg et al. [31] provide a comprehensive survey of experiments, some of which relate to modeling.

Reasons for the lack of adoption of modeling therefore seem to boil down to the following: a) weaknesses in tools, modeling languages and processes; b) the lack of education or training of developers or their managers; and c) satisficing, wherein the benefits of modeling are judged, rightly or wrongly, to not warrant the costs, given the level of quality desired. The problems inherent in adoption of any technology also undoubtedly come into play; Sultan and Chan [32] provide a detailed study of technology adoption for object-oriented technology, concluding that management issues and organizational culture have a larger role to play than factors intrinsic to the technology itself.

Our work contributes to the field of software modeling by learning first hand from software practitioners from a wide variety of companies from several countries around the world, and with diverse experience about the state of the art. We set out to uncover what software professionals believe to be modeling, the tools they use, the activities they perform and what difficulties they perceive towards model-centric and code-centric approaches. Our survey of software professionals discussed in this chapter complements the survey results of Dobing and Parsons [27] in that we asked questions about modeling in general, as opposed to specifics of UML.

2.2 The Preliminary Study: Categorizing Software Applications

We are interested in modeling practices, and it seems reasonable that some types of software applications have a greater need for modeling than others. Prior to surveying software practitioners (the results of which are presented later in this chapter), we wanted to be able to distinguish responses based on software application type.

Empirical software research, including our own, could be improved if there were a systematic way to identify the types of software for which empirical evidence applies. This is because results are unlikely to be globally applicable, but are more likely to apply only in certain contexts such as the type of software on which the evidence has been tested. We therefore developed a software taxonomy to help apply our research systematically to particular types of software. The taxonomy was generated using existing partial taxonomies and input from survey participants. If a taxonomy such as ours gains acceptance, it will facilitate comparison and appropriate application of research.

The notion of taxonomy, a structured vocabulary, is widely familiar and is extensively used in modeling tools to organize information. Almost all newer conceptual models like attributes, tagging and axioms are all backed by a hierarchy [33], making a taxonomy a logical starting point to uncover software application types.

Maintaining a software taxonomy provides the following benefits for software engineering research:

- **Improved software research**. A software taxonomy is able to provide a context for empirical results in software engineering, as well as to facilitate exploring the applicability of those results. The taxonomy would be used to suggest the types of software on which results could be tested. For example, a survey of software practitioners could be subdivided into participants that work with data-dominant versus computational-dominant software. From that sub-sampling, the researcher might be able to demonstrate that a particular software process (or architecture, or framework, or testing technique, etc.) is indeed appropriate in both types of applications, or is in fact applicable only to one of the types.
- **Artefacts more readily reusable**. Artefacts such as libraries, plug-ins, software patterns, and algorithms could be more easily reused if mapped to categories within a software taxonomy. The taxonomy could help identify candidate artefacts for reuse, as well as identify gaps – i.e. application types where more reusable artefacts are required.
- **Increased use of reference models** (sketches of the architecture) **and frameworks**. If reference models and frameworks were mapped to a taxonomy of software types, developers may have a better starting point when building new software in a particular domain. Similar to the artefacts above, the taxonomy will also identify which application types lack frameworks and reference models.
- **Appropriate education coverage.** With a software taxonomy, educators and curriculum designers would be able to build courses and programs that provide exposure to a broad range of application types.

Below is a summary of the first two levels of the software taxonomy we developed. Our contribution aims to add closure to much of the effort already invested in this field. We used a systematic process so the taxonomy covers all types of software currently developed. Please refer to [18] for a further analysis of justification for our work, for the existing literature on the topic, for the formalized approach to developing the taxonomy, as well as to see *where* a particular software application fits within the structure below.

A. Data-dominant systems
- A.con Consumer-oriented software
- A.bus Business-oriented software
- A.des Design and engineering software
- A.inf Information display and transaction entry

B. Systems software
- B.os Operating systems
- B.net Networking / Communications
- B.dev Device / Peripheral drivers
- B.ut Support utilities
- B.mid Middleware and system components
- B.bp Software Backplanes (e.g. Eclipse)
- B.svr Servers
- B.mal Malware

C. Control-dominant software
- C.hw Hardware control
- C.em Embedded software
- C.rt Real time control software
- C.pc Process control software (e.g. air traffic control, industrial process, nuclear plants)

D. Computational-dominant software
- D.or Operations research
- D.im Information management and manipulation
- D.art Artistic creativity
- D.sci Scientific software
- D.ai Artificial intelligence

The taxonomy should be useful to both researchers and practitioners who need to perform such tasks as cataloguing, filing or searching for applications. The taxonomy will also facilitate tagging of components and techniques according to the application types for which they are suitable.

By explicitly defining a software taxonomy with a well-defined purpose, and formalized criteria for classification, we can start to apply *strong* approaches to application domains to solve specific problems. This taxonomy should help with software reuse; re-using the appropriate tools, methodologies, languages, paradigms (e.g. object-oriented, procedural), software patterns, and software components.

2.3 The Main Study: Modeling Survey

Once we had developed our software taxonomy to a satisfactory level of detail, we were able to proceed with our survey. The modeling survey was conducted online. Please refer to [20] for the questions posed and a complete analysis of the responses to the survey, since the official survey is no longer online. We sent targeted requests to personal contacts in a wide variety of organizations. We also asked for participation using a variety of Internet forums.

The survey consisted of 18 questions. Most questions involved several sub-questions answered using 5-point Likert scales. Responses were in ranges such as strongly disagree to strongly agree, or never to always.

The survey was divided into groups of questions as follows:

• Q1: What is or is not a model? Various options were presented ranging from class diagrams, use cases, to source code. Our objective was to see if participants had a preconceived notion about what they considered a model to be.

• Q2-5: How and when do you model, and using which notations? The objective of these questions was to understand the state of the practice.

• Q6: How do you approach a new task or feature with respect to requirements, design, modeling, testing and documentation?

• Q7-10: What tools, methods and platforms do you use, and what type of software do you develop?

• Q11-14: To what extent do you use modeling, and how good is it for various tasks?

• Q15-16: What are the principal difficulties you perceive with the model-centric and code-centric approaches?

• Q17: An open-ended free form question for comments about the survey and / or modeling in general.

• Q18: Demographics question with sub-questions about country of origin, education level, and years of experience of the participant.

Some randomization in the order of questions was applied to reduce bias towards either code-centric or model-centric questions. Questions 2 to 5 were presented in a random order. We then defined modeling as follows so that participants could consistently answer subsequent questions:

For the remainder of the survey, please assume that any reference to a software model refers to an artefact that represents an abstraction of the software you are building. A model can typically be viewed as a set of diagrams and/or pieces of structured text. It can be recorded on a white board, paper, or using a software tool. A model could use formal syntax and semantics but this is not necessary. We will consider the final source code of the system, and requirements written in natural language to not be models, although models can be embedded in a requirements document.

Questions 7 through 16 were then presented to the participant in a random order. The survey had 113 participants. Of those, at least 88 answered each question, and at least 63 answered the two modeling tools questions. Participants were able to ignore these modeling questions if they did not have substantial experience with modeling tools. The full technical report outlining the questions posed, method, threats to validity and the results obtained is available in [20].

2.4 Demographics of the Modeling Survey

Our survey was conducted between April and December 2007 and ultimately attracted 113 software practitioners. Participants averaged 14 years of experience, with 80% having more than 5 years of experience, and about 20% having more than 20 years of experience.

About two thirds of the respondents were from Canada or the United States. The rest of the world was fairly well represented with participation from the United Kingdom, the rest of Europe, India, and Pakistan, as well as a few participants from Australia, Mexico and Singapore.

The most prevalent type of software that participants work on is business software (identified as 'very often' by 46% of participants), followed by design and engineering software (25%), website content management and information display such as search, maps and news sites (23%). The least represented categories were malware (2% - good!), industrial control (10%), and system utilities (7%). At least one participant has 'always' worked in one of the available categories (except for malware). The categories of software applications from which to choose were based on work building a software taxonomy (described in Section 2.2). By using a comprehensive taxonomy, we are able to (1) validate that our results apply broadly, and (2) provide a means to filter our results. The software modeling needs of business application developers, for example, may not necessarily reflect the needs of real-time developers.

The dominant programming languages in use are Java, and C/C++ with about 30% using the language 'very often'. PHP / Perl was used 'very often' by 19%, ASP.Net 14%, and Ruby /

Python 9% by the participants. Other languages mentioned in the free-form answers include JavaScript, SAP, Documentum, ASP, Smalltalk, MatLab, and SOAP.

About 90% of participants have leadership roles (team leader, project manager) at least sometimes, and 53% lead very often or always. Design or modeling is performed at least sometimes by over 95% of participants, and 57% do this very often or always. On the other hand, 86% at least sometimes work with source code (developing new code, maintaining and/or bug fixing), and 49% very often or always work with code.

Almost half of the participants had a Masters degree (44%) – which is more than in the general software engineering population. However, an advantage of this over-representation is that our respondents should be among the more knowledgeable software engineers.

In our analysis, we not only studied the whole sample, but also sub-sampled according to various criteria to ascertain whether particular groups of developers had different opinions. For example, we separately analyzed the data for those who reported they programmed extensively, or used modeling tools extensively. We also sought any geographic differences, and differences based on the type of software the developers focused on. We used Student's t-test to determine statistical significance, with significance deemed to with at least 90% confidence ($p <= 0.1$). The appropriateness of using the t-test on Likert scale data is summarized in [34].

The sizes of the major sub-groups are shown in Table 1.

Table 1: Sub-Sample Sizes

Category	N	%
All Participants	113	100%
Participants in Canada/USA	63	56%
Participants Outside Canada/USA*	27	24%
Software Developers	53	47%
Software Modellers	46	41%
Participants that Generate Code	15	13%
Experienced Participants (≥ 12 years)	53	47%
Participants in Real-Time Projects	19	17%

*Of the 113 participants, 23 participants did not indicate what country they live in and were not included in the geographic sub-sampling

2.5 Modeling Survey Results

The tables presented in this section will provide the following data:

- N is the number of participants that actually responded to the question / sub-question
- Mean is the mean of the rating from 1 to 5

- S.d. is the standard deviation of the particular statistic
- The final four columns present the % of responses that answered at either extreme (1, 1+2, 4+5, and 5).

The format of the tables gives a sense of the polarization of the data, as the mean does not always present an accurate interpretation of the data. Notably high values are highlighted in bold in the tables, and notably low values are highlighted in italics. The data is also sorted based by the mean and does not reflect the order in which the sub-questions were asked.

2.5.1 What is a Software Model?

Our first question listed various types of artefacts that could conceivably be perceived as models and asked the participants to give us their opinions.

Participants almost completely agree that class diagrams (48% strongly agree, 88% agree or strongly agree), UML deployment diagrams (36% strongly agree, 78% agree), and use case diagrams (34% strongly agree, 82% agree) all represent models of a software system. In fact, hardly anyone felt that any of these artefacts are not models (3% and 6%, respectively). There was also good agreement that the following can be models: pictures by drawing tool (86% agree); textual use cases (79% agree); pictures drawn by hand (79% agree) and whiteboard drawings (57% agree). The source code of a system is perceived by about half of the participants to be a model (47% agree, 39% disagree). Similar data was observed for pictures drawn by hand (57% agree, 10% disagree). Most participants were neutral as to whether source code comments represent a model of a software system (34% agree, 41% disagree), but few strongly agreed either way.

A key conclusion from this question is that developers do not limit their perception of a model to diagrams in a modeling tool. Models can be represented textually (use cases, and source code), and in non-electronic formats (white boards, and hand drawn pictures), and in drawing tools.

2.5.2 Creating Versus Consuming Software Models

There is little agreement among participants regarding how software models are created and maintained. Similarly, there is also little agreement about among participants about how they learn about the design of software.

The most frequent way of creating models was by drawing or writing on a whiteboard, with 45% agreeing they do this, and 33% disagreeing. Next most frequent was using diagramming tools (37% agree, 42% disagree), and word-processing software (27% agree, 46% disagree). Other mechanisms to maintain software models like word of mouth, handwritten material, source code comments, and modeling tools/CASE had between 22% - 30% agreement that the participants used them. Finally, very few (13% agree, 72% disagree) used drawing software to maintain models.

It is interesting to reiterate that 27% of people feel they create models by word of mouth, although 42% of people disagree with this.

The most important source of design information was word of mouth, with 55% of people very often or always using information originating from this source, whereas 24% of people sometimes or never use this. The next most important source was material created in word processors (48% very often, 30% sometimes), and using diagramming tools (that can create structured diagrams, but not integrated models with 42% very-often and 32% sometimes). Diagramming tools and whiteboards were cited as being used very often by 42% of the participants.

The least important source of information was material in fully integrated modeling tools. Only 32% use material created with such tools very often, and conversely 33% never do. Handwritten material also ranked as an unimportant source of information (20% very-often, 24% never).

When comparing the results of what is a software model, there are several differences among the sub-samples. The most interesting differences are highlighted below:

- Software developers are less likely to use modeling tools compared to the entire sample ($p=0.014$).
- Software modellers that write/maintain software are less likely to use modeling tools compared to software modellers that do not write/maintain software systems ($p=0.06$)

The contrast between what mechanisms participants use to generate models versus consume them is interesting. Although our survey did not attempt to answer why there is a divide, a key conclusion is that the tools and data formats used by individuals responsible for creating software design information tend not to be the same as those used by the audience of that information. In other words, assimilation of data is done differently from its dissemination.

2.5.3 What Modeling Notations Do You Use?

About 52% of practitioners very often or always use UML – and more practitioners use UML 2.* (34%) versus UML 1.* (28%). UML is also indicated as being the most highly used notation. Our survey supports the idea that the UML is, in fact, now the universally accepted notation for software modeling as few people are using domain-specific languages for their projects, or other notations like Entity-Relationship Diagrams (ERD), SQL and Specification and Description Language (SDL).

2.5.4 How Are You Using Your Modeling Tools?

The main uses of modeling tools reported by participants were to develop the design of a software system (48% very often) or to simply transcribe a design into a digital format (39% very often) in somewhat of a data-entry fashion.

Conversely, tools to generate code from the model are not widely used (18% very often generate some code and only 14% very often generate all code for a software system); this may be because participants do not need code generated, or perhaps because the tools do not do it in the way they want.

Relatively few participants use software modeling tools to brainstorm about possible design ideas and alternatives (23% very-often, and 55% only sometimes), and it seems that collaboration amongst developers is not an important feature for software modellers. From earlier questions, it seems that whiteboards still are superior for this and the role of modeling tools in this instance is to help transcribe the design as opposed to actively develop it. This woud seem to be especially true when you consider that many modeling tools are not good for brainstorming – 15% believe they are awful at brainstorming, and 45% see them as poor for brainstorming.

As a large percentage of software practitioners use modeling tools to transcribe a design into a digital format, this might suggest that more (or better) text-driven features within software modeling tools would make such transcriptions more efficient; as the data is most likely derived from informal mediums such as whiteboards and diagramming tools where it may be too optimistic to suggest automatic transcription from the disparate sources.

When comparing the results of what is a software model, the following differences surfaced among the sub-samples:

- Real-time developers are more likely to use modeling tools for prototyping a design compared to entire sample (p=0.043)
- Software developers that model are more likely to use modeling tools to develop a design (p=0.0002) and generate some code (p=0.004) compared to software developers that do not model

2.5.5 How Good Are Your Modeling Tools?

As a follow up to the previous question we asked, “Based on past experience, how good (based on qualities like efficiency, accuracy and usability) are software design or modeling tools at accomplishing the following tasks.” The results are presented in Table 2.

Table 2: Responses for Question 12: How good are modeling tools at ...?

Available activities	N	mean	s.d.	% Awful (1)	% Poor (1 + 2)	% Good (4 + 5)	% Excellent (5)
Developing a design	71	3.4	1.0	2.8	16.9	47.9	12.7
Transcribing a design into digital format	69	3.2	1.0	2.9	24.6	42.0	7.2
Generating code (code is editable)	69	2.9	1.1	10.1	39.1	29.0	8.7
Prototyping a design	68	2.9	1.1	10.3	41.2	29.4	8.8
Brainstorming possible designs	71	2.8	1.2	15.5	45.1	32.4	4.2
Generating all code (no manual coding)	69	1.9	1.1	42.0	79.7	8.7	4.3

Note. Values range from Awful (1) to Excellent (5).

The order of items in this table is almost the same as in the previous question that asked how modeling tools are used, suggesting that participants use tools for what they are good at, and might use them differently if the capabilities of the tools changed.

Participants believe that design and modeling tools are adequate at developing and transcribing a design. Some participants believe modeling tools are okay at generating source code templates (code that will be then edited). Many believe that modeling tools are poor at facilitating brainstorming and generating the entire system.

This presents two possible scenarios: modeling tools are poor at generating entire systems and that is why software practitioners do not use them; or, developers do not really need to use modeling tools as full-system source code generators. The same scenario is true for brainstorming sessions (there is currently little use of modeling tools for the purpose of brainstorming, and modeling tools do not facilitate brainstorming that well).

2.5.6 Important Attributes of a Software Model

Participants seem to care most about how well a software model can effectively communicate to others and be readable. This result (ranking communication and readability as the top two qualities of a software model) is unsurprising. The third and fourth most important attributes of a software model are more interesting and are discussed below. The results are summarized below in Table 3 and are ranked in order of most to least important qualities of a software model.

Table 3: Responses for Question 13: Important Attributes of a Modeling Tool?

Attribute / Ability to	N	Rank	% Bottom 2	% Bottom 4	% Top 4	% Top 2
Communicate to others	89	1	10.1	16.9	78.7	68.5
Readability	89	2	10.1	20.2	68.5	51.7
Ease and speed to create	89	3	7.9	36.0	55.1	19.1
Ability to analyze	89	4	10.1	33.7	55.1	21.3
Collaborate amongst developers	89	5	12.4	38.2	43.8	15.7
Ability to view different aspects of a model	89	6	10.1	42.7	40.4	13.5
Generate code	89	7	52.8	70.8	23.6	11.2
Information density	88	8	51.1	72.7	17.0	3.4
Embed parts of model in documentation	89	9	55.1	82.0	13.5	4.5

Note. The % top 4 represents the percentage of participants that listed the attribute in their top four. Similarly for % bottom four. The same applies for % top2, and % bottom 2.

The third most important attribute is how quickly and how easily models can be created. There seems to be a trend towards supporting advanced users, as participants are fairly keen on ease and speed of use of a tool. Again, our survey did not expand on what aspects of tools that developers want made more easy and faster, but further investigation could consider:

- Modeling tools need to accommodate users by providing quick ability to edit information when the user thinks of a needed change or quick ability to access information when the developer needs to look something up.
- Power users may benefit from expert-user techniques that include more keyboard-focused activities: keyboard shortcuts, textual data entry and keyboard navigation.

The fourth most important attribute suggests that modeling tools should facilitate the analysis of the models they produce. Although our survey did not expand on the exact meaning of how a tool can best support the analysis of a model, the strong response (57% rated it within the top 4) warrants further investigation. Possible avenues of research include investigating features such as:

- Model feedback for correctness (syntax and semantics), best practices / pragmatism, and modeling style (potentially similar to coding style)
- Model search whereby tools provide a mechanism to answer questions about the model in a similar manner as CAD tools and development environments for source code. For example, “Where is this association being used?”, “How often is this method being requested” or “Which of these two designs has lower coupling?”.

Software modeling tools should allow a developer to create models quickly and easily. Perhaps the trend towards visual programming languages accommodates only part of this requirement: the ease-to-create part, not necessarily the speed-to-deliver.

2.5.7 Which Approach, Code Versus Model Works Best?

The following questions make reference to model-centric and code-centric approaches to developing software first presented in Chapter 1. To recall, in a model-centric approach, the developers look to the model to see the design, and change the model as the first step in performing any design change. Extensive modeling is performed, and the coding is either automated, or at least straightforwardly determined from the model. In a code-centric approach, the code is seen as the main artifact; developers understand the design by understanding the code, and the process of design change is equated with changing the code.

The results of our survey do include practitioners that fit each level of this model-code-centric spectrum – increasing the confidence in our sampling technique. In the following questions we will analyze how well modeling tools perform specific tasks, as well as uncover what approach (model versus code) is more suitable based on particular situations.

The participants' perceptions about which approach works best for various activities are presented below in Table 4.

Table 4: Responses for Question 14: Model versus Code-Centric Tasks.

Available activities	N	mean	s.d.	% Much easier in Models (1)	% Easier in Models (1 + 2)	% Easier in Code (4 + 5)	% Much easier in Code (5)
Fixing a bug	90	3.2	1.5	21.1	28.9	43.3	25.6
Creating efficient software	92	3.1	1.4	16.3	35.9	43.5	21.7
Creating a system as quickly as possible	92	3.0	1.5	23.9	46.7	42.4	23.9
Creating a prototype	92	2.9	1.5	26.7	43.0	32.6	22.8
Creating a usable system for end users	92	2.7	1.3	26.1	42.4	22.8	10.9
Modifying a system when requirements change	91	2.5	1.4	34.1	54.9	24.2	13.2
Creating a system that most accurately meets requirements	91	2.2	1.3	42.9	67.0	19.8	8.8
Creating a re-usable system	92	2.2	1.3	44.6	63.0	15.2	9.8
Creating a new system overall	92	2.2	1.3	43.5	68.5	20.7	7.6
Comprehending a system's behaviour	89	2.0	1.3	51.7	71.9	15.7	5.6
Explaining a system to others	92	1.7	1.1	61.1	81.8	7.6	6.5

Values range from much easier in a model-centric approach (1), to much easier in a code-centric approach (5).

The results suggest that most activities tend to be easier in a model-centric approach, in the opinion of the survey participants. Even the task judged to be the most code-centric (creating efficient software) was considered to be achievable with about the same amount of difficulty as in a model-centric approach. Overall, this question seems to show considerable polarization of the participants, as even the activities that are seen as highly model-centric still had 15% - 24% of the participants believe that the code-centric approach was the easier approach. Model-centric approaches appear to be particularly appropriate for higher-level activities including: explaining a system to others (82% at least somewhat easier); understanding a system's behaviour (72%); creating a new system (69%); and creating a re-usable system (63%).

The participants seem to really want to incorporate modeling into their process, but as earlier questions indicate, at present they are not doing so to any significant degree. This is demonstrated in that only half of the participants answered the question about how they use modeling tools and, of those participants, very few were using the tools to generate source code from their models.

2.5.8 Model-Centric Issues and Concerns

As expected, the biggest perceived problem of model-centric approaches is keeping the model up to date with the code (recall that participants did not want code to be generated from models) with 68% in agreement.

The results reporting the problems with a model-centric approach are presented in Table 5.

Participants did not identify that they had had past bad experiences with modeling (only 17% agree to have had a bad experience). This suggests little bias against model-centric approaches. Also, modeling languages (primarily UML, as identified as the dominant modeling language with 52% usage among the survey participants, whereas the next highest, structured design models, had only 22% usage) are not the limiting factor to adopting model-centric approaches, since languages are not considered that difficult to understand (only 10% agree modeling languages are difficult).

The issue that "code generated from a modeling tool is not of the kind I would like" also appears to be of high concern with 37% in agreement that this problem is real.

Other high ranking problems include:

- Difficulty exchanging models between tools (52% agree)
- Tools are too big and too hard to learn (39% agree)
- You cannot describe in modeling languages the kinds of detail required to be implemented in source code (36% agree)
- Models become obsolete as tools change (33% agree)
- Creating models is slow (23% agree)

These problems are mostly tool centric ranging from underlying storage formats, and tool usability (too big, too slow and insufficient data). It is interesting to note that 34% felt that a model's underlying storage format would become obsolete (7th most perceived problem) and this perhaps reflects an assumption (whether true or not) that tools conform differently to the underlying data standard (either officially via extensions, or unofficially deviating from the standard) creating tool-specific models. Text and source code seem to have a much longer shelf-life; this lends support to the main hypothesis of this thesis that perhaps it is time to explore models in textual notations that are human editable just like a programming language.

Table 5: Responses for Question 15: Problems with a model-centric approach.

Potential problems	N	mean	s.d.	% Not Problem (1)	% Slight Problem (1 + 2)	% Bad Problem (4 + 5)	% Terrible Problem (5)
Models become out of date and inconsistent with code	92	3.8	1.2	7.6	16.3	68.5	37.0
Models cannot be easily exchanged between tools	91	3.3	1.3	15.4	26.4	51.6	17.6
Modeling tools are 'heavyweight' (install, learn, configure, use)	92	3.1	1.2	10.9	31.5	39.1	12.0
Code generated from a modeling tool not of the kind I would like	91	3.0	1.4	18.7	39.6	38.5	16.5
Not enough detail to be implemented in code	89	2.8	1.3	23.6	43.8	36.0	7.9
Creating and editing a model is slow	92	2.7	1.2	17.4	43.5	22.8	12.0
Modeling tools change, models become obsolete	92	2.7	1.2	22.8	44.6	32.6	5.4
Modeling tools lack features I need or want	89	2.6	1.1	19.1	44.9	21.3	5.6
Modeling tools hide details (source code fully visible)	92	2.6	1.1	19.6	44.6	23.9	1.1
Modeling tools are too expensive	90	2.6	1.3	26.7	46.7	26.7	6.7
Modeling tools cannot be analyzed as intended	90	2.5	1.3	28.9	51.1	25.6	6.7
Organization culture does not like modeling	92	2.5	1.2	31.5	48.9	23.9	4.3
Semantics of models different from prog. language	90	2.4	1.3	31.1	56.7	23.3	8.9
Modeling languages are not expressive enough	91	2.4	1.1	28.6	54.9	17.6	2.2
Modeling language hard to understand	91	2.2	1.0	28.6	62.6	9.9	3.3
Have had bad experiences with modeling	91	2.2	1.2	39.6	63.7	16.5	6.6
Do not trust companies will continue to support their tools	89	2.0	1.0	44.9	67.4	10.1	0.0

Note. Values range from Not a problem (1), to Terrible problem (5).

2.5.9 Code-Centric Issues and Concerns

As expected, when the participants were asked about the potential difficulties with code-centric development most feel that code-centric approaches fail to deliver a high-level view of the system (66% agree) and entropy means that situation only gets worse over time (55% agree). The results are summarized in Table 6.

Table 6: Responses for Question 16: Problems with a code-centric approach.

Potential problems	N	mean	s.d.	% Not Problem (1)	% Slight Problem (1 + 2)	% Bad Problem (4 + 5)	% Terrible Problem (5)
Hard to see overall design	94	3.8	1.1	4.3	13.8	66.0	35.1
Hard to understand behaviour of system	94	3.6	1.1	4.3	19.1	60.6	21.3
Code becomes of poorer quality over time	92	3.4	1.3	9.8	28.3	55.4	25.0
Too difficult to restructure system when needed	93	3.4	1.2	8.6	22.6	51.6	17.2
Difficult to change code without adding bugs	93	3.4	1.2	9.7	22.6	50.5	18.3
Changing code takes too much time	94	2.8	1.2	20.2	39.4	27.7	8.5
Our prog. language leads to complex code	94	2.5	1.2	26.6	51.1	20.2	8.5
More skill than available to develop high quality code	91	2.5	1.2	29.7	53.8	22.0	6.6
Prog. Languages not expressive enough	91	2.1	1.2	46.2	64.8	14.3	5.5
Organization culture does not like code-centric	92	1.9	1.2	58.7	72.8	14.1	4.3
Our prog. language likely to become obsolete	93	1.9	1.1	51.6	75.3	9.7	3.2

Note. Values range from Not a problem (1) to Terrible problem (5).

Participants are of the opinion that programming languages they use do not result in complex code (20% see this issue as a bad problem), are expressive enough (14% bad problem) and are not likely to become obsolete (10% bad problem).

The participants were divided on whether or not changing the code takes too much time; 28% agree and 39% disagree.

Based on the results identifying the current problems with model-centric and code-centric approaches to software development; development tools may benefit from features that help to better:

- Synchronize code and models to reduce the inconsistencies.
- Provide better traceability between models and code to help identify relationships among code and model artefacts to help indicate aspects of models that may require maintenance should the code change (and vice-versa).

- Provide better modeling capabilities and expressions within the programming code to reduce the need for external and disjoint modeling artefacts.

The research presented in subsequent chapters helps to substantiate these claims with the design, implementation and analysis of the model-oriented programming language, Umple.

2.6 Threats to Validity

The main threats to validity of this part of our work are summarized below. We have also outlined the steps we have taken to help mitigate these threats.

Question interpretation. The survey was conducted over the Internet and respondents may have misunderstood the intended meaning of our questions. We took two steps to reduce the ambiguity of our questions by asking colleagues to first review the questions, and then having team members complete the survey during our trial run. Both activities helped improve the overall survey prior to go-live. We also separated the survey into two main parts: the first part to solicit the participants' personal thoughts towards "what is a model" and the second to answer modeling based question based on our explicit definition.

Researcher bias. The survey questions attempt to uncover problems with both model-centric and code-centric approaches to software development. A potential bias could be introduced if our survey appeared to be overly negative towards either modeling or software coding. To reduce the chance of bias we tried to be objective when referring to both code-centric and model-centric questions, as well as presenting the questions in a random order. Since at the time of this survey we had already conceived some of the notions later embedded in Umple, it may also be the case that the researchers only asked questions that would lead to answers supporting the features they were already considering. This may have led to omission of questions that may have shed different lights on this topic.

Non randomized sample. To help ensure that our sample was based on a representative collection of software practitioners, we approached both open and closed forums for participation. In particular, we submitted link articles to Digg.com, and Dzone.com - two popular technology and news sites. We submitted email requests to UML user groups, agile user groups, Java user groups, and RUP user groups. We also submitted personal requests to current and former colleagues. Our demographics results indicate that we do have representation from most regions of the world, most educational backgrounds, most software industries, and most types of developers. Prior to conducting the survey we also developed a software taxonomy

[18], as discussed earlier in this chapter, to categorize software applications; our results do include representation from each of the top-level application types.

Missing/inappropriate question options. Most questions were closed questions and did not allow additional comments from the participants. Conducting a closed questionnaire can improve the rate of participation, and it is easier provide quantitative analysis, but it can limit the kind of feedback that participants can provide. It is almost impossible to provide all possible scenarios to explain a certain occurrence, and providing too much freedom in the data collection process (e.g. free-form answers to each question) reduces the quantitative analysis potential of the survey as a whole and can add considerable length to the survey. To mitigate the threat that our survey does not learn anything new we put the following safeguards in place (some have already been mentioned above): the questionnaire was reviewed with colleagues to improve coverage of the answers; and, the questionnaire concluded with a free-form open question where participants could provide additional insights (and perhaps additional explanations to a closed question) on the subject of software modeling.

2.7 Contributions of the Modeling Survey

The following are the main contributions and findings from our study.

Firstly, it is clear that most participants take a broad view of what they consider to be a model, including informal material such as hand-drawn diagrams. The use of formal modeling tools is, however, not widespread, with over a third never using them. UML is now clearly the dominant modeling language for modeling – but tends to be used informally.

For those who use modeling tools, the main uses are to develop a design or transcribe a design into digital format. Use of a tool to generate *all* code is not common; this is seemingly because participants feel that the generated code is not of the type of quality they want, and presumably also because most developers are reluctant to use full-fledged modeling tools capable of generating code.

Overall, most developers clearly see the value of the modeling approach, even though they practice it to only a limited degree. However, there remains a strong core of code-centric practitioners who seem strongly opposed to modeling.

The following observations gathered from the modeling survey lend credibility to our research problems identified in Chapter 1. The results provide some insight into the validity of our hypothesis that the uptake of modeling might be higher if it were possible to create models textually, in the same manner as code-centric software developers currently program. Below is a summary of the observations from the survey data that have helped shape the direction of subsequent chapters:

- Models are more than just diagrams and can include a multitude of other media, including text
- Model-centric approaches seem to fit many situations, more so than code-centric approaches, but modeling is not yet a common practice among software practitioners
- Modeling tools are often used to transcribe a model into a digital format; they are used for this more than for brainstorming the design, or generating code.
- Models quickly become out of sync with the code, and the code often poorly reflects the design intent of the system
- Software practitioners want modeling tools that allow easy and quick creation of models; perhaps visual editors are easy to use but maybe they do not adequately facilitate quickness.

Software developers still appear to love to code, so instead of focusing our work on changing the development habits from textual to graphical (since most modeling tools are graphical), it may be possible to raise the level of abstraction (i.e. modeling capabilities) of programming languages to incorporate many of the concepts currently found in visual modeling languages like UML. Visual modeling tools could then be enhanced to support a textual equivalent, allowing both text and diagram to be renditions of the same underlying model. This duality of text and diagram (referred to as text-diagram duality) would eliminate the need for a *reverse engineering* process. If the above were possible, then the complaints our participants had about the quality of generated code may disappear, so too would the complaints regarding models being out of sync with the code, or the code poorly reflecting the design of the system. By learning more about the text-diagram duality of software modeling and by addressing the identified limitations of code-centric and model-centric programming, we may be able to increase the uptake of modeling as a whole.

In subsequent chapters we will design, implement and analyze a model-oriented programming language called Umple that allows developers to *model* in code. We are seeking to understand more about this phenomenon and to see if text can be successfully used for what has been a predominantly visual task.

Chapter 3 Umple: A Model Oriented Language

This chapter leverages the insights gained from previous chapters as we introduce our approach to address the model-code divide. Our objective is to build a textual modeling language, which we call Umple. This enables developers to continue to work with text but to adopt more model-centric approaches to software development.

Umple is a model-oriented programming language that can be used as a diagramming / documentation tool similar to Yuml [35]. Umple can be used as an evolutionary prototype platform as described in [36]. Finally, Umple can be used for full application development with example applications described in Section 7.3 .

The Umple approach is intended to reduce the ever-present tension between model-centric and code-centric developers. Model-centric developers prefer to model visually and generate code, while the code-centric developers see source code as the only important software artefact. There are significant challenges in model driven development of software. One major challenge is synchronizing generated code with source models, a practice generally referred to as round-tripping [37]. Our past research also indicates that modelers quite often create models that do not quite reflect how the system is intended to behave [38]. By eliminating the need for round-tripping and by reflecting the model directly in the *code*; these challenges can be reduced. On the other hand, by enabling coders to model, we raise the level of abstraction available to developers allowing them to focus more on the *what* instead of simply on the *how*.

The core of the Umple approach is to add UML abstractions such as attributes and associations directly into high-level programming languages. Umple is unique in that it is not simply another Domain Specific Language (DSL) among the sea of many languages; Umple adds facilities to general-purpose programming languages with a Java-like syntax; it enhances existing object-oriented languages instead of competing with them.

A benefit of using Umple (which enabled our team to refactor Umple's Java implementation into Umple), compared to other model specifications like xUML [39], and DSLs [40] is that a system need not be fully specified in Umple to use Umple (as is the case with many other model specifications). Umple can easily be refactored into an existing system by following certain patterns as discussed in [41].

Currently, Umple can work with Java, PHP and Ruby. Future versions of the Umple could also support additional languages such as C++, C# and Python.

In addition to being built on top of existing languages, developers that develop (and/or model) in Umple can work interchangeably with UML diagrams and Umple code; they are just views of the same model. In this chapter, we provide an overview of the basics of the Umple language as well as the supporting tools. In subsequent chapters we provide a deeper analysis of the implementation of attributes, and associations; as well as analyze the implications on program size and complexity of systems developed in Umple.

Umple's objective is to simplify software by incorporating higher-level abstractions in programming languages. Umple can be viewed from several perspectives. It can be seen as a programming language based on Java that incorporates UML constructs to raise the level of abstraction. It can be viewed as a tool for rapidly creating UML diagrams. Or it can be viewed as a tool to help broaden the appeal of modeling by allowing models to be created textually. In fact, it is all three of these.

In this chapter we discuss our motivations for developing Umple. These include the lack of adoption of modeling technology caused in part by difficulties using modeling tools, and a desire to reduce the amount of code needed in certain object-oriented programming tasks. Then we present various aspects of Umple including the overall philosophy, the concrete syntax, an overview of the semantics and the tools we have developed.

3.1 What Models and Modeling Languages Should Be

Miller believes that "If a model had a clear meaning, it would be an asset, rather than a cost" [42]. Miller's paper goes on to describe the need for a shared language with a solid foundation, that is well understood, suits the range of application types being built and does not impose restrictions on the user. Although we agree with the arguments presented for building an unambiguous modeling language, we are skeptical that such a language could serve all application types. In the same manner that source code can be an asset despite there being no shared language from which all application types can be built, we believe that modeling code can be equally valuable despite there being no universal language. To further that point, we intend to apply modeling-like constructs to a variety of application domains. In some cases we present a simple proof-of-concept to demonstrate the feasibility of our approach to modeling, and in others we look more closely at particular issues, concerns and implications that arise from introducing new constructs to a language.

Selic provides his perspective on the pragmatics of model-driven development in [43]. Selic describes the pragmatic qualities of a model-driven approach that, if available, might help improve the up-take of modeling. These qualities include:

- **Model-level observability**: Static and runtime reporting of models should be displayed at the modeling level; just like compiler errors are reported at the code level (not the *generated byte code* level)
- **Model executability**: Provides direct experience with the system being designed.
- **Efficiency of generated code:** Identified as an early concern to MDD adoption, the ability for model compilers to produce as (or more) efficient systems compares to early concerns about program compilers. This concern, in time, should become less and less of an issue.
- **Scalability**: MDD is intended for large-scale systems. Scalability refers to both the compilation time and the system size.
- **Integration with legacy environments and systems**: Not only must MDD software work with legacy system, it must also work with legacy software development processes and environments.

It is important to note that despite being model-oriented, there is very little discussion of text versus diagrams in the pragmatics above; also, the implications on integration with legacy environments (which is mostly textual) imply at least somewhat of a textual approach to MDD.

Pawson and Matthews in [44] describe a modeling technique called *naked objects*. The premise involves a one-to-one mapping between business model and user representation – somewhat of an object-oriented user interface. The goal of naked objects, as we understand it, is to provide reduced modality and greater expressiveness. There are no views, no controllers, just models. The view and controller functionality is provided by an object viewing model.

Naked Objects is an approach that discourages the separation of process and data, and instead the intent is to provide behaviourally complete objects [45]. Naked objects use a "convention-over-configuration" approach to achieve its goal (i.e. if you follow our rules, then the system works) and presents a compelling argument for straightforward application development. The framework does, however, admit its limitations in handling associations relative to a modeling language like UML. Naked objects provide a compelling prototype language whereby modeling objects can be translated into a tangible form. But its adoption will be limited due to the complexities that arise in enterprise level application user interfaces. Pawson has demonstrated that business models can be directly translatable into a functioning software system and our work tends to apply similar high-level objectives; albeit our focus is on describing models within an executable context without constraining either the persistence of model data, nor its presentation and interaction with end users (i.e. the GUI).

We agree with Spinellis [46] that design models can be composed textually, and graphs can then be automatically generated. A model is a simplification of reality and that simplification need not imply a purely graphical representation. Manipulating shapes on a canvas can be tedious and time consuming. The effort and coordination to create the diagrams is somewhat irrelevant to the end result. In addition, with diagrams it can be very difficult to determine if model diagrams are inconsistent or simply incomplete [47].

Drawings are also problematic for configuration and revision control as well as refactoring, code generation, and metrics extraction. Treude et al. [48] reports that medium- to large-size models can take five minutes to one hour of processing for merging. Some approaches result in significant overhead, for example, the use of Universal Unique ID (UUID) for all modeling elements, which cannot change once created, and whose uniqueness must be guaranteed at all times [49]. In addition to the limitations of merging models from different sources, such approaches are inevitably computationally complex. While the computational processing of XML-based documents may be efficient compared to semantic model merging, XML-based versioning of software models appears to not have an appropriate level of abstraction, which can result in falsely identifying model conflicts [50].

Spinellis [46] lists the following advantages to textual models:

- **High-level Skills**: Textual modeling takes advantage of a programmer's high-level skills at abstract formalization of concrete concepts.
- **Low-level Skills**: The model may be manipulated using an editor of the developer's choice, which may include advanced features for searching, editing, macro-preprocessing, merging. Etc.
- **Design versus Diagram**: Textual modeling allows developers to concentrate on the act of modeling, not the aesthetics of diagrams and documentation. For example, text may be much easier to automatically tidy, or even maintain in a clean manner from the start; whereas visual models may require developers to spend time *perfecting their pictures*.

Gronniger's position paper on text based modeling [51] also analyzed the benefits of text, and points out the following additional advantages.

- **Information Content**: Text can be more compact and concise than a typically spread-out diagram, resulting in graphical tools requiring a significant amount of zooming, scrolling or pop-ups to convey the same amount of textual information.
- **Speed of Creation:** There is less need for drag and drop, menus, pop-ups and switching between mouse and keyboard in a textual environment.
- **Integration of languages:** Not everything can be readily expressed in a diagram, and most graphic tools do a poor job of integration of elements that intrinsically need to be textual.
- **Version control:** Small changes in diagrams can result in large changes in its model serialization for version control, making model versioning compared to plain text very difficult.
- **Outlines and graphical overviews:** Text can be represented graphically if needed, so a textual form serves as a good main form.
- **Composition of modeling languages:** Text shares a common storage format with disparate modeling and programming languages, allowing for easier integration among these.

Both graphical and textual representations have their limitations and Gronniger [51] states that most papers investigating graphical modeling tools make no comparison to textual versions. The premise that *graphical tools and languages are better simply because they are graphical* has been questioned in [52]. Although he does not present empirical evidence in support of or refuting graphical versus textual notations, Gronniger argues anecdotally that because he observed software developers choosing a textual approach as opposed to a graphical approach, where both options were available, that there is merit to text and that at least in some situations text can be the preferred development approach. He made these observations in the context of the USE tool [53].

Jacobson and Cook [54] provide a short summary of the possible future direction of UML to "prepare [UML] for the future" as a response to the Request for Information on the Future Development of UML in 2009 [55] (issued by the OMG and led by Steve Cook). The results of the RFI showed a need for UML to be leaner, easier (to learn, to integrate with other languages), and to be more expressive.

One area that our research addresses is the ability for "models to be deeply integrated with modern programming languages without any semantic conflicts" [54]. A future objective of

UML is to create a kernel, called "Essential UML" which contains a compact set of elements that can be quickly learned. Our work looks to bring UMLs future objective to reality now, with an unambiguous implementation of a small but significant UML subset that when used in combination with an executable action language like Java, PHP, or Ruby can fully specify software systems.

Most MDA approaches provide different layers of software system descriptions from platform independent models (PIM) to platform specific models (PSM) for a particular implementation. It is likely the case that several languages will be used in a system [56]. Our approach to modeling looks to *reduce* the number of languages in use by extending current languages with modeling constructs. In [57], Skene argues that modeling languages do a poor job of preserving the link between artefacts (i.e. generated source code) and their language. If the model were the code then the need to preserve the link is a moot point because they are now one and the same.

OMG recently made available an RFP for a concrete syntax for a UML Action Language (UAL) [58]. The proposal is to seek a definition of a textual language for representing the UML subset defined in the Foundation Subset for executable UML Models (fUML) [59]. The OMG proposal requires that the UAL be suitable for use in executable UML models. A proposed UAL must meet a number of objectives including [60]:

- It must be computationally complete, meaning it must include standard arithmetic and logical capabilities supported natively or by the use of libraries.
- The UAL must allow the invocation of user-specified external code such as legacy code.
- It must allow embedding of native code. For example, if the target platform is Java, the UAL should allow the embedding of Java statements and constructs.

The approach requested by OMG with a UAL presents just a small aspect of textual modeling (in this case the action semantics of a model), but it does represent an increased movement towards text-based (and human usable) modeling languages. In November 2009, OMG published two proposals, one from IBM and one from Mentor Graphics. In mid-February 2010, the two proposals were consolidated into one to be called Action Language for Foundational UML (Alf) [61]. Development of Alf continues at the time of writing.

Umple allows developers to model the structure of a system (i.e. class diagram), and it currently delegates much of the system's behaviour (i.e. action semantics) to the underlying programming language such as Java, PHP, or Ruby. Some might take the perspective that all code is at some level a model – which might be even truer in the future if or when Umple extends support for Alf action semantic.

3.2 Motivations

Our desire to develop Umple arose for two main reasons. We address each of these in the following subsections.

3.2.1 Lack of Adoption of Modeling

It is apparent from our survey in Chapter 2 that although most developers model sometimes, the code-centric approach is more prevalent.

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 sync 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.

We believe it is possible, in the approach we take with Umple, to deal with both sets of criticisms and hence satisfy both groups.

Our own personal experience corresponds with what we have observed in our studies of others: We do a considerable amount of modeling and would like to be able to generate and modify UML diagrams very rapidly. Whether it is for teaching UML, illustrating books or papers, or developing actual systems, we have found both the commercial and open source tools to be slower at modeling than we would like.

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

a) Code generation does not 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 Umple approach addresses these reasons as follows.

Point a) is addressed by the fact that Umple is a programming language, and a simple one. One of the main difficulties in generating code from a language like UML is that the semantics of UML were in fact explicitly designed to be for abstract modeling – making it difficult to translate a model into code. Umple adopts key modeling features, but in its design we have chosen to err on the side of making it simple and usable for programming. It adopts Java semantics when these differ from UML semantics.

Umple addresses point b) by allowing models to be created using a simple text editor. Umple has a syntax-directed editor that is used to help produce error-free code or models. At the same time, however, the Umple tools provide the capability to rendering Umple code as UML diagrams directly in IBM’s Rational Software Modeler, as well as other modeling tools such as Papyrus and EclipseUML. In general, Umple supports Eclipse Modeling Framework (EMF) tools via Ecore. The user therefore has the ‘best of both worlds’.

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

Umple addresses point d) by only introducing the very simplest modeling concepts in its initial release. These include generalization, associations (with multiplicity), attributes, association classes and a few basic design patterns. Current work also includes support for state machines, the discussion of which is outside the scope of this work.

Finally, Umple addresses point e) by providing a path to adoption that does not require a significant investment.

3.2.2 Reducing the Need to Program Boilerplate Code

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

However, our second major motivation is to ease people's ability to write object-oriented code. Long before the advent of UML, it was a commonplace object-oriented programming idiom for two classes to contain instance variables that reference the opposite class. The programming challenges include maintaining referential integrity, and deciding which class would take the prime responsibility for adding and deleting the references (links) between objects of the two classes.

With the emergence of object-oriented modeling languages (e.g., OMT and later UML) these between-class references were modeled as associations. Detailed semantics can be found in the UML specification [62], or the many books on the subject (e.g. [2]).

Ultimately, however, association abstractions still have to be rendered into programming language code. Developing bug-free code to do this involves considerable work. However the code ends up being very similar from association to association. It is called boilerplate code because it is standard in nature, yet needs to be replicated in many parts of a system.

With Umple, this boilerplate code for associations is completely eliminated. Instead one declares associations and lets the compiler handle the details of referential integrity and link assignment.

In the next section we provide an overview of the language constructs available in Umple, followed by a review of Umple tooling and its underlying design.

3.3 Description of Umple

The word 'Umple' is a play on words, meaning 'Simple', 'UML programming Language' and 'Ample'.

3.3.1 Simple

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

An Umple program deliberately enforces a highly layered style of software. In particular it provides no user interface facilities other than a simulation environment and debugging mechanism. The result of compiling Umple code is the generation of Java, PHP or Ruby code implementing an API that can be accessed by other layers, such as a user interface or data access layer. This allows for the use of existing tools and frameworks like Java Server Faces [63], and Zend Framework [64] to build the user interface and a tool like Hibernate [65] for database access.

3.3.2 UML Programming Language

Umple adds key features of UML to Java, PHP and Ruby. The Umple language supports UML concepts like attributes, associations and state machines. As mentioned earlier, Umple can be used to generate UML class diagrams, or alternatively a class diagram can be rendered straightforwardly into Umple.

Umple is not the first attempt to provide a textual notation to UML. The XML Metadata Interchange XMI [66] is designed as an exchange specification between UML compliant tools, but its intent is to be consumed by computers and is not very human-readable; it also comes in different 'flavours' resulting in it actually not being very useful for interchange in practice. Umple can generate XMI as well as tool-specific model exchange formats including TextUML.

OMG proposed a Human-Usable Textual Notation HUTN [67]; but this representation is not a language but rather an approach for any model to be populated textually. The HUTN standard has remained stagnant since first being published in 2004. Should this standard be revived, our team could investigate model-to-model transformation from Umple into HUTN. In addition to those just mentioned, there are number of other textual modeling approaches and tools [35, 68-72]. For example, TextUML [73] is a tool that allows the modeler to create and edit UML models in text as opposed to in graphical form. TextUML is a UML editor whose output is compatible with other modeling tools so the user may obtain services such as code generation. Conversely, Umple provides the full end-to-end solution with a textual notation, action semantic and constraint notation as well as a model compiler (no round trip engineering required).

Although we believe that Umple's approach to easily facilitate the generation of running systems from textual UML models with embedded implementation code is unique. Umple's philosophy is to completely remove the need to round-trip between models and code, effectively eliminating the need for manual effort to synchronize model and code [74]. We successfully removed round-tripping by supporting a friendly human-readable (and textual) modeling notation seamlessly integrated with algorithmic code. With a model-is-the-code approach, developers are more likely to maintain and evolve the code (and modelers are more likely to maintain and evolve the model) as the system matures simply by the fact that both views represent the same underlying system.

3.3.3 Ample

Despite the restrictions imposed by the deliberate simplicity of Umple, it is intended to have sufficient power to program the functional layer of most kinds of systems. As we will describe later, we have used it for several moderately-sized applications including the Umple compiler itself, a private lender's mortgage company, a distance learning reporting tool, an online schedule manager, as well as 25 *example* systems including an airline scheduling system and an elevator control system.

3.4 Motivating Examples

Umple provides a family of programming languages that all incorporate UML constructs to raise the level of abstraction. Each Umple language instance (e.g. Java+Umple) supports the underlying action semantics of the target execution language. It can also be used as a tool for rapidly creating UML diagrams to help broaden the appeal of modeling by allowing models to be created textually.

Versions of Umple currently exist for Java, PHP and Ruby, and we intend to create versions for other object-oriented languages. For simplicity we will typically limit our discussion to Java.

An Umple program contains algorithmic methods that look more or less identical to their Java counterparts. The key differences lie in how classes are declared, the absence of explicit constructors, and the replacement of instance variables and many related methods by *attributes* and *associations*.

3.4.1 Course Registration

The following example shows how one would declare attributes and associations in the first steps on modeling a system using Umple. To help distinguish between Umple and Java code, the Umple examples use dashed borders in light-grey shading, and pure Java examples use solid-line borders with no shading.

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

As the above example shows, the basic declaration of a class follows the syntactic style of Java. The three lines in class Registration declare an attribute and two associations respectively. The class diagram to reflect the Umple code above is shown in Figure 1.

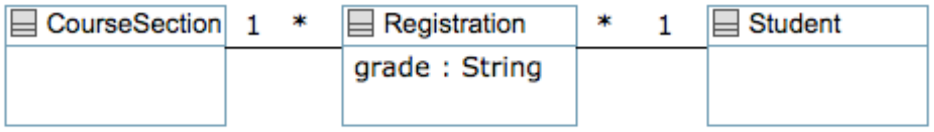


Figure 1: UML class diagram for part of the student registration system (from [2])

Immediately after writing the Umple code shown above, we can generate a class diagram of this tiny system. Alternatively, we could have drawn the diagram and the Umple code would have appeared in our tool. We can also generate an executable system in either Java, PHP or Ruby. The executable system implements an API that allows developers to create instances of the modeled classes, as well as manage attributes, and delete links of the associations. The few lines of Umple code above generate over 300 lines of Java (the details of the code generation is discussed in Chapter 4 and Chapter 5).

3.4.2 Airline Reservation System

Let us consider the UML class diagram outlining a simple Airline Reservation System [2] illustrated in Figure 2.

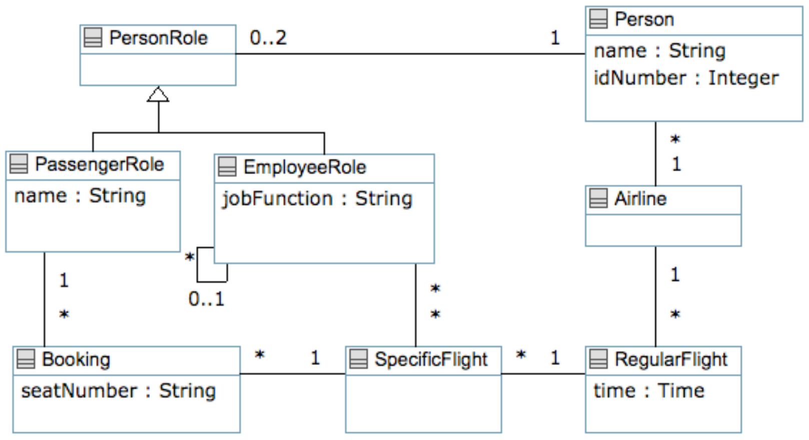


Figure 2: Airline Reservation System UML Class Diagram [2]

The system is comprised of eight classes, eight associations, and one generalization (with two subclasses). The airline system has regular flights that run on a particular schedule with daily occurrences (called specific flights) staffed by employees and filled with passengers, who make bookings.

In translating UML associations into executable languages like Java, it would be beneficial (but currently uncommon) to include access methods (get, set, add, remove) to manage links of associations. Ideally, these methods would maintain the multiplicity constraints of the association as well as preserve referential integrity – ensuring that both ends of an association are properly updated when adding or removing links.

With Umple, the model and the code (and the modeller and the coder) become one. There is no ‘code generation’ process that requires manual updating, since the Umple code represents the model and the code at the same time. Of course, there is still a compilation process, but because the full richness of object-oriented programming is available within Umple, there is no need for the developer to look at the compiled code, just as today’s programmers almost never look at the generated machine or byte code. By removing the necessity of editing the generated code we not only remove the need to reverse engineer the generated code; we also simplify the generated code as there is no longer a need to provide the infrastructure to *allow* hand-written code to be integrated into generated code. Indeed, supporting round-trip engineering in a code generator can be a nice feature to have, but not requiring one is even better.

Umple’s executable code is not intended to be read, as all the features of the underlying language would be available in Umple. Despite the fact that the generated system need not be further edited to be run, we believe that it is still a good practice to *generate code* as if it were written by hand. Undoubtedly, and at least until Umple becomes widely accepted, end-users will look at the generated Java/PHP/Ruby code. From an educational perspective, having well written generated code acts to demonstrate how modeling is translated into executable code.

The Umple syntax for describing the airline system above is shown below in Figure 3.

As seen below, the Umple language provides a Java-like syntax to describe software models and includes basic constructs such as classes, packages and primitives types such as Integer, Boolean, String, Date, and Time. Unlike Java, Umple also provides mechanisms to model associations separate from attributes.

Umple treats associations as first-class entities and associations can be expressed separately from the classes they link together. For example, to code the association that several employees can be assigned many specific flights, and many specific flights can have several employee roles assigned to them can be written as follows:

```
association  {
  * EmployeeRole -- * SpecificFlight;
}
```

A collection of additional Umple examples (currently 34 examples ranging from Banking Systems to Warehouse control systems) is available for review online at [3].

```
namespace Airline;

class Airline{
  1 -- * RegularFlight;
  1 -- * Person;
}

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

class PassengerRole {
  isA PersonRole;
  immutable String name ;
  1 -- * Booking;
}

class PersonRole{}
```

```
class EmployeeRole {
  String jobFunction;
  isA PersonRole;
  * -- 0..1 EmployeeRole supervisor;
  * -- * SpecificFlight;
}

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

class Booking {
  String seatNumber;
  * -- 1 SpecificFlight;
}

class SpecificFlight{
  unique Date date;
}
```

Figure 3: Airline Reservation System in Umple

The textual version of Umple shown in Figure 3 above includes some additional details such as role names and attribute modifiers (like immutable and unique) that are not displayed in the diagrammatic view that was used to generate Figure 2. The same is true for most other UML diagrams presented in this thesis.

Let us now look at the other syntactic elements illustrated above. Please note that attributes and associations are further analyzed in subsequent chapters. The description appearing below is meant to provide a high-level understanding of the Umple language.

3.5 Overview of Umple Entities

3.5.1 Attributes

The declaration of attributes in Umple looks very much like the declaration of instance variables in Java. However, there are some important differences. First, the set of primitive data types is different. Currently Umple supports the following attribute types: Integer, Double, Date, Time, String, and Boolean. The above set of primitives was chosen to cover most basic modeling and programming needs, without having to deal with the complexities of *primitive* vs. *class* data types. The Integer type will in fact generate 'int' code when compiled in Java, but we want to insulate the user from that constraint on the Java language. The set of attribute types is also inspired by those available in relational databases.

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

Chapter 4 will present Umple Attributes in more detail, including a discussion of the syntax and semantics of modeling attributes as well as their underlying code generation.

3.5.2 Associations

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

In Figure 1, there is a many-to-one association between Registration and Student. Declarations of associations can appear in two places in Umple. They can appear inside one of the two associated classes, or else as standalone 'first class' associations.

For example, we could have placed one of the association declarations inside the Student class (instead of keeping all of the associations declared in the Registration class) as shown below.

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

An association can also be defined independently of either association end as shown below:

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

This notation can be useful when you want to name the association (e.g. Enrolment as shown above), or when you want to use Umple's mix-in capabilities to enhance an existing model without modifying that model directly (e.g. re-using existing model with your own enhancements).

Another difference between the syntax for standalone versus inline associations is that the additional class name must appear in the syntax (e.g. *Student* above). For inline associations declared within a class referenced in one of the association ends, the first association end is implicitly defined as the containing class.

Finally Umple supports UML's notion of association classes. The notation and diagram of association classes is shown in Figure 4. The Umple code to generate Figure 4 is as follows:

```
class Student {}
class CourseSection {}
associationClass Registration {
  * Student;
  * CourseSection;
  String grade;
}
```

Those familiar with UML will recognize that the functionality embodied in Figure 1 and Figure 4 is essentially the same.

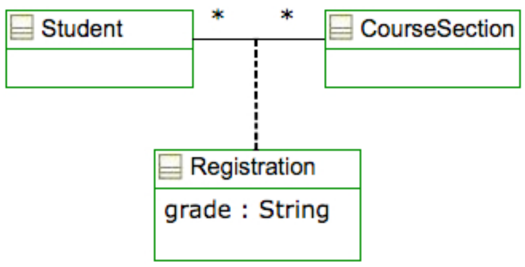


Figure 4: UML association class

The full Umple grammar, which also includes the various ways to declare associations, is presented in more detail in Section 3.7 . Chapter 5 will elaborate in more detail the syntax, semantics and implications on code generation of associations.

3.5.3 Generalizations

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

```
isA <superclass>;
```

For example, an Employee could be a subclass of a Manager.

```
class Manager
{
   isA Employee;
}

class Employee {}
```

Or, one can embed the subclass definition inside the definition of the superclass. For example, the following is equivalent to the previous example.

```
class Manager
{
   class Employee {}
}
```

The embedded form is appealing in that the inheritance hierarchy appears visually as increasing levels of indentation. There are no scope restrictions (such is the case with inheritance) using this indentation syntax, and it should be noted that inner classes are not supported in Umple (conceptually inner classes are not really needed, but they can help to make code easier to understand).

3.5.4 Action Semantics Using Java-Like Methods

Methods in Umple look very much like Java, PHP or Ruby methods. In fact, the Umple compiler relies on the action semantics of the base language to process them. However there are certain restrictions placed on the code in a pre-processing step:

Umple methods should be confined to the following:

- Read and write attributes. References should be made via the setX and getX methods. Rules about immutability and multiplicity are enforced when managed via these methods.
- Navigate associations where the other end is ‘1’ in the same manner as accessing attributes.
- Navigate ‘many’ associations by calling built-in methods to obtain a read-only version of the list of objects participating in the association end.
- Instantiate objects, destroy objects, and add, delete and update links of associations by calling methods generated for this purpose. Referential integrity is automatically maintained when using navigation and population methods.
- Perform normal computations with local variables, and reference external libraries as required.

Umple methods can be placed inline in the class, or can be separated (either within a single file, or several). This allows for mix-in capabilities to enhance (as opposed to extend) a class as well as allows an Umple model to be separated from the methods written in the base language (i.e. Java, PHP, or Ruby).

There is presently no standard notation for action semantics in UML. The approach to action semantics in Umple is built to support any and all object oriented syntaxes. As mentioned, Umple currently supports the Java, PHP, and Ruby syntaxes. Work is ongoing to support C++. Additional work is being proposed to support the soon-to-be-approved UML action language Alf that was discussed earlier.

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 diagrams. 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, like the current RFP submission Alf, take a top-down approach, where a new language and new constructs are defined forming an additional layer of abstraction, with the objective of formalizing model execution. In developing Umple, we adopted an alternative approach; iteratively discovering what is necessary in an action language starting with a pre-defined object-oriented language (in our case Java, PHP and Ruby) and adapting it by adding abstractions to fit the action language requirements.

Umple provides an excellent complement to UAL by providing a textual syntax for the aspects of models such as classes, associations, attributes and state machines without committing to any one action language. Just like Umple currently supports Java, PHP and Ruby, once a UAL standard is defined, Umple could easily support the concrete UAL syntax.

Chapter 6 discusses additional patterns and features available within the Umple language.

3.5.5 Layout and Positioning in Umple

Umple tracks diagrammatic positioning of classes and association links. The syntax to position a class is as follows:

```
position <top-left-x-point> <top-left-y-point> <width> <height>;
```

And, associations between classes are positioned with the following syntax:

```
position <qualified-role-name>__<qualified-role-name>
         <x-offset-1>,<y-offset-1> <x-offset-2><y-offset-2>;
```

The qualified-role-names refer to the combination of class name, as well as optionally the role name should it be explicitly provided. The offsets refer to the positioning of the link end-point on a class relative to the top-left quadrant. And example is shown below:

```
class Student
{
  position 50 30 109 45;
}

class Mentor
{
  1 -- 0..1 Student;

  position 50 130 109 45;
  position.association Mentor__Student 59,0 30,45;
}
```

The syntax presented above is sufficient to store the layout of a class diagram. Additional research regarding layout is deferred for future work.

3.6 Umple Design and Tooling

In this section, we provide an overview of the tools currently available to support the creation and analysis of Umple systems; as well as discuss some of the technical details of the underlying design and storage mechanism of Umple.

3.6.1 Umple's IDE Tooling

At its core, Umple is a language interpreter and code generator that parses *Umple* code, populates the Umple metamodel and then generates several output artefacts as shown below in Figure 5. Apart from the target language platform (e.g. Java JRE or PHP/Ruby interpreter) there are no other external dependencies, enabling Umple to work on a variety of platforms including Windows, Mac and Linux.

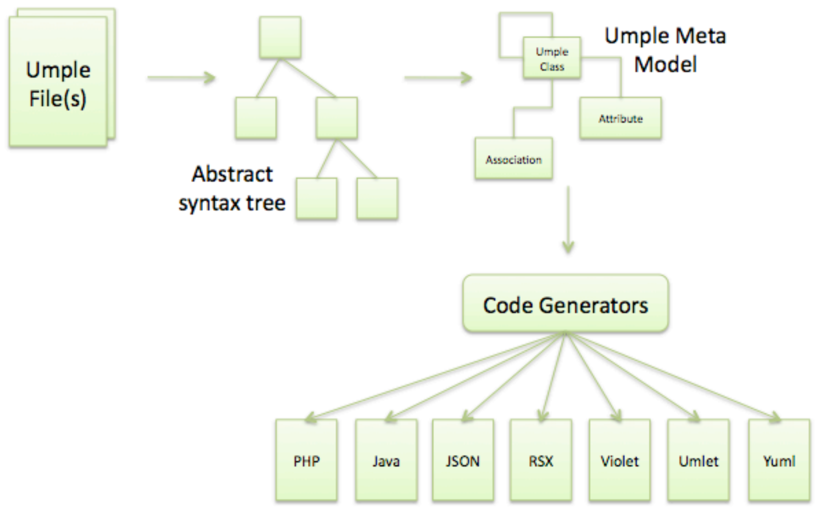


Figure 5: Umple Process

The Umple grammar and metamodel are discussed in later sections in greater detail. Below, we focus on the Umple tooling, and in particular we see that Umple currently generates a variety of artefact types include working Java, PHP and Ruby systems, JavaScript notation (JSON) for web visualization, Violet [75] / Umlet [76] visualizations that plug into Eclipse, Yuml [35] visualizations, as well as several other modeling notations including valid RSx [77], Papyrus [78], and TextUML [73] models. Umple relies on Java Emitter Templates (JET) [79] to generate Java, PHP and Ruby.

Umple also includes subsidiary and internal tools that include:

- Umple Statistics – A metric gathering tool to analyze certain aspects of an Umple system such as the types of associations present in a model.

- Umple Delta – A synchronization tool that accepts delta inputs (e.g. add or edit) in JSON format and updates the underling Umple code (and vice versa). This is used for the Umple Online tool that provides synchronization between an Umple diagram and the corresponding Umple text.
- VML – Processing of variability models whose target applications are written in Umple.

Umple is available as an IDE and works within Eclipse, Papyrus, Xtext and RSx (both Rational Software Modeler and Rational Software Architect). In these tools, Umple incorporates model visualization but does not yet provide synchronization between the code and the diagram. Below is an RSx model generated by Umple.

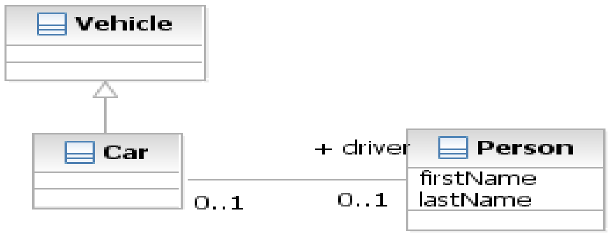


Figure 6: Rendering an Umple model in RSM

Umple is also integrated into light-weight model visualization tools including Violet, Umlet and Yuml. No synchronization between text and diagram is available for these tools. Below in Figure 7 is an Umlet diagram generated by Umple.

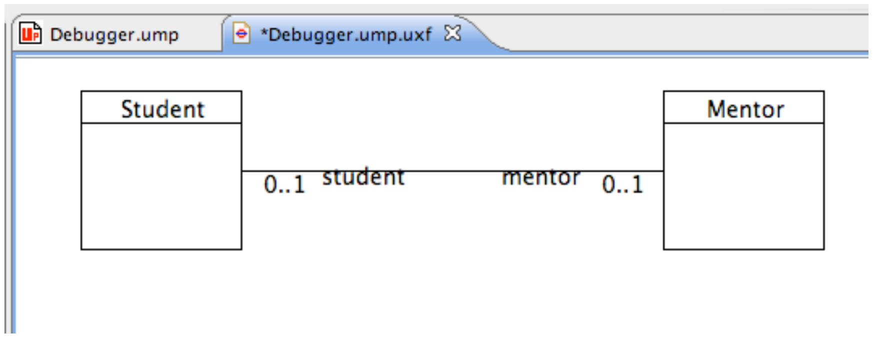


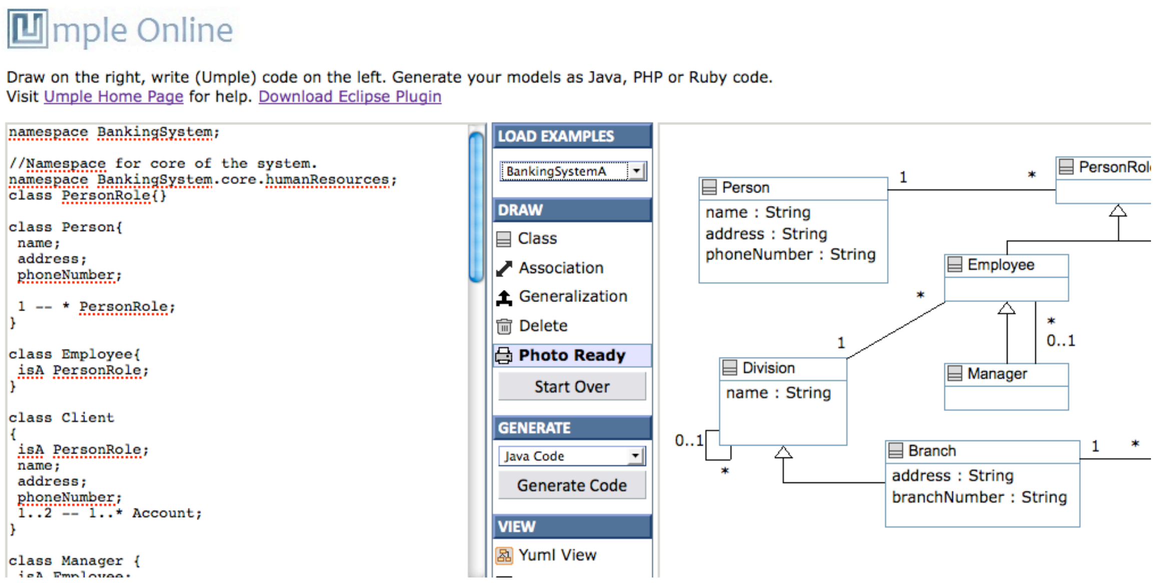
Figure 7: Rending an Umple model in Umlet

3.6.2 Modeling in the Browser

Umple is available as an online editor, called UmpleOnline [3]. The online project was created for several purposes. A web application provides a zero-footprint modeling environment. Casual users are able to experiment with the latest version of Umple with little more than a browser and

Internet connection. This access enables curious developers to try out the language, and simulate basic models (e.g. object creation, link management and attribute updates). It can act (and has acted) as a tool for software educators to quickly demonstrate various UML design alternatives as well as to teach the implications on the generated code of those alternatives. Umple provides a close link between a modeled system and its underlying implementation. Our online editor was also used to investigate our claim that text and diagrams can represent the same underlying model, allowing both to be manipulated interchangeably.

UmpleOnline works well as a demonstration tool. It allows the latest features to be used and to quickly analyze both the modeling and code generation capabilities of the language. Below is an example model generated using UmpleOnline.

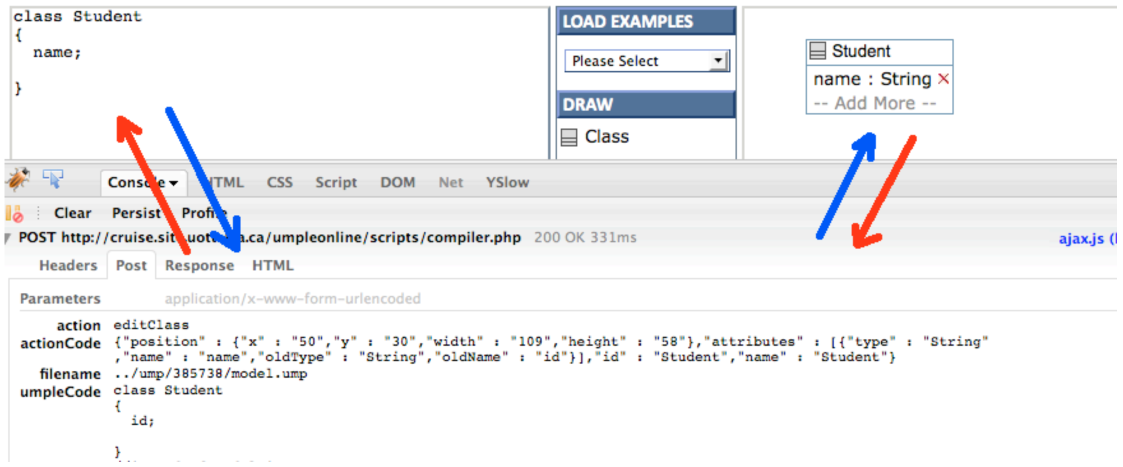

Draw on the right, write (Umple) code on the left. Generate your models as Java, PHP or Ruby code.
Visit Umple Home Page for help. Download Eclipse Plugin

```
namespace BankingSystem;

//Namespace for core of the system.
namespace BankingSystem.core.humanResources;
class PersonRole{}

class Person{
 name;
 address;
 phoneNumber;

 1 -- * PersonRole;
}

class Employee{
 isA PersonRole;
}

class Client
{
 isA PersonRole;
 name;
 address;
 phoneNumber;
 1..2 -- 1..* Account;
}

class Manager {
```

Figure 8: UmpleOnline Screenshot

UmpleOnline also demonstrates the successful synchronization of core modeling components including classes, associations and attributes. Figure 8 is a view of both visual and textual representations of the same model as seen from UmpleOnline. Figure 9 illustrates the synchronization mechanism of the textual and visual editors in the online environment.

The delta mechanism implemented in UmpleOnline synchronizes both the visual and textual views by synchronizing only the change deltas, making it suitable for the online modeling environment even when the available bandwidth is limited. Umple is presented and stored in the same textual representation (as opposed to visual editors whose presentation is diagrammatic but storage is textual, usually XML based).



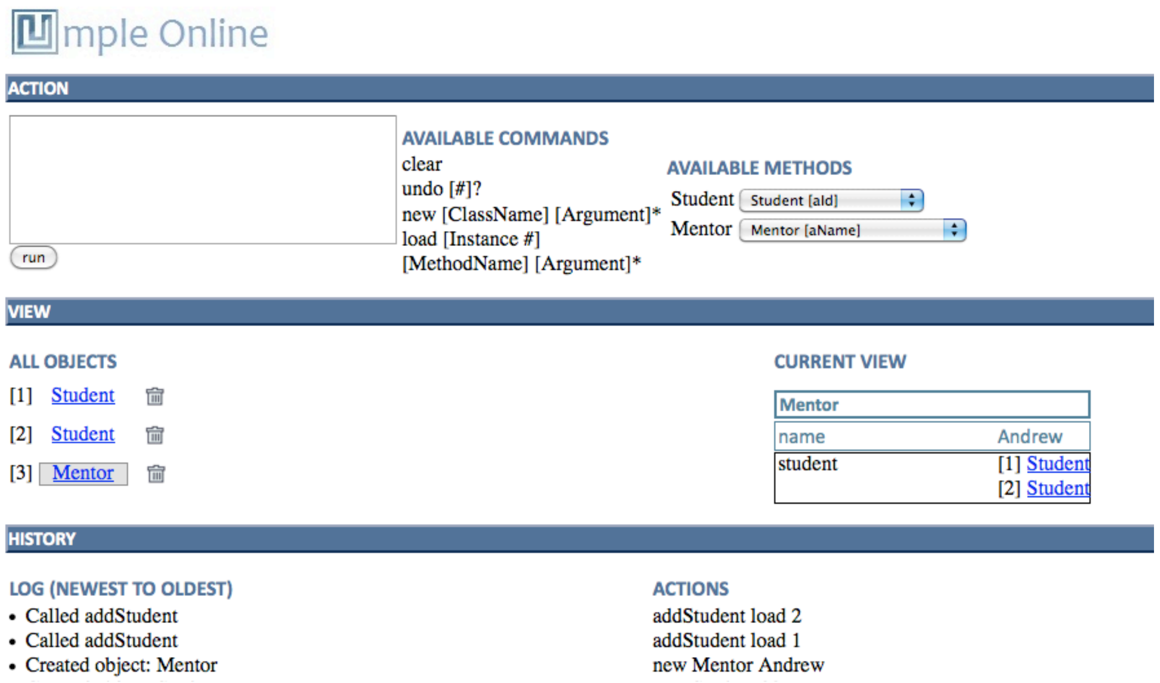
```
class Student
{
  name;

}
```

Student
name : String
-- Add More --

```
POST http://cruise.sit uot a.ca/umpleonline/scripts/compiler.php 200 OK 331ms

Parameters   application/x-www-form-urlencoded

    action editClass
actionCode {"position" : {"x" : "50","y" : "30","width" : "109","height" : "58"},"attributes" : [{"type" : "String"
           ,"name" : "name","oldType" : "String","oldName" : "id"}],"id" : "Student","name" : "Student"}
  filename ../ump/385738/model.ump
umpleCode class Student
           {
             id;

           }
```

Figure 9: Synchronization of Visual and Textual Representations of UmpleOnline

UmpleOnline features a simulator that allows an object-level manipulation of a model. This runtime execution allows early testing of the design and data model. A screenshot of a simple relationship between a student and a mentor is shown below in Figure 10.



AVAILABLE COMMANDS
clear
undo [#]?
new [ClassName] [Argument]*
load [Instance #]
[MethodName] [Argument]*

AVAILABLE METHODS

Student	Student [aId]
Mentor	Mentor [aName]

ALL OBJECTS
[1] Student
[2] Student
[3] Mentor

CURRENT VIEW

Mentor	
name	Andrew
student	[1] Student [2] Student

LOG (NEWEST TO OLDEST)
- Called addStudent
- Called addStudent
- Created object: Mentor

ACTIONS
addStudent load 2
addStudent load 1
new Mentor Andrew

Figure 10: Simulating a Simple Umple Model

The simulator allows you to create new instances, set/get class attributes, manipulate association links and call arbitrary methods defined in the model. The simulator provides navigation between objects via association links, and well as simple *undo*. The action language for the simulation is purposely light; the simulator may later be changed to adopt OMG's yet-to-be-finalized Alf UAL standard discussed earlier.

The Umple tools are developed in Umple itself, using Java as its underlying action language for data manipulation and algorithmic operations for the stand-alone application such as the Umple IDE, and support tooling. UmpleOnline is built in Umple, using Umple+PHP for the front end interaction with the end-user including the simulator, whereas the back-end parsing, processing, merging and synchronizing diagram and text uses Umple+Java.

The various tools mentioned throughout this section are built and deployed using the ANT scripting language; resulting in several executable jars for the various stand-alone tools as well as for the RSx and Eclipse plugins. These executable jars are also used in the UmpleOnline web application.

The development team of Umple (several graduate students including the author) follows a test-driven approach to provide confidence that future enhancements will not regress previously functioning (and tested) aspects of the Umple systems as well as to ensure that defects are less likely to re-surface. More on our approach to verification and validation is presented in 7.1 .

3.7 Defining the Umple Language

The Umple language is specified using an interpretable EBNF format with small enhancements to support domain specific languages where code blocks need not be fully parsed.

3.7.1 Grammar Notation

The notation we present for the Umple grammar is identical to that interpreted by the Umple system when *compiling* an Umple program. The benefit is that we ensure our documentation is always up to date, but at a cost of using a slightly non-standard EBNF syntax.

Our decision to abandon using Antlr [80] was due to its strictness – all tokens have to be strictly defined, yet Umple, needs to support arbitrary code blocks allowing for algorithmic code to be written in the native execution language (e.g. Java, PHP, Ruby or any other future language to which Umple is added). Similar limitations have been uncovered, and remain unresolved, when working with the xText IDE for Umple.

We strive to keep our notation in line with the EBNF format. Our syntax offers a very simple mechanism to define a new language, as well as extend an existing one. We will be using examples to help explain the syntax. Let us start with a simple assignment rule:

```
assignment : [name] = [value] ;
```

The rule name is "assignment". An assignment is comprised of a non-terminal called "name", then the equals symbol ("="), a non-terminal "value" and finally a semi-colon symbol (";").

The grammar includes two non-terminal notations, the first of which is shown above. A sequence non-terminals is a sequence of characters that is non-whitespace and is delimited by the next symbol as defined in the grammar. In our case above, the non-terminal "name" will be defined by the characters leading up to either a space, tab, newline, or an equals ("=").

Our grammar syntax allows for rapid language creation. The language author need not worry about the complex, repetitive and somewhat error prone regular expressions used to define common structures such as string sequences, decimal numbers, alphanumeric strings, and arbitrary code blocks as would be required when using a parser like Antlr [80]. This simplicity allows for rapid prototyping, which can later be refactored into rule based non-terminals as discussed below.

Here are a few examples that satisfy the assignment rule above:

```
key = "one";
wasSet=true;
numberOfItems =7;
```

Let us now consider examples using rule-based non-terminals (i.e. nesting rules within a rule).

```
directive- : [[facadeStatement]] | [[useStatement]]
facadeStatement- : facade [=facade:on|off] ;
useStatement : use [=type:file|url] [location] ;
```

Above, we have three rules, *directive*, *facadeStatement*, and *useStatement*. A *directive* is either a *facadeStatement* or a *useStatement* (the *or* expression is defined by the vertical bar "|"). To differentiate between the two types of non-terminals, we use single square brackets ("[" and "]") for sequence non-terminals, and double square brackets ("[[" and "]]") for rule based non-terminals.

By default, rule names are added to the tokenization sequence. But, some rules act more like placeholders to help modularize the grammar and to promote reuse. To exclude a rule name from the token sequence (and just the name, the rule itself will still be evaluated and tokenized as required), simply add a minus ("-") at the end of its name.

Above, we see that the rule names *directive*, and *facadeStatement* are not added to the tokenization string. Based on the example grammar above, parsing the statement:

```
facade off;
```

based on the *facadeStatement* rule shown above would result in the following tokenization:

```
[facade:off]
```

Note that because of the minus ("-"), the *facadeStatement* is not added to the tokenization string and is simply used internally by the grammar to group and re-use rules.

Symbols (i.e. terminals), such as "=" and ";" are used in the analysis phase of the parsing (to decide which parsing rule to invoke), but they are also not added to the resulting tokenization string for later processing. If we want to tokenize symbols, we can create a constant using the [=name] notation.

Above, we see that a *facadeStatement* is represented by the sequence of characters "facade" (i.e. a constant). To support lists of potential matches we use a similar notation [=name:list|separated|by|bars]. Above, we see that the *type* non-terminal can be the constant string sequence *file* or *url*.

Here are a few examples that satisfy the assignment rule above:

```
facade;
use file Parser.ump;
use url http://cruise.site.uottawa.ca/Parser.ump;
```

Our grammar syntax supports a simple mechanism for non-terminals that can include whitespace (e.g. comments). Consider the following rules to define inline and multi-line comments.

```
inlineComment- : // [*inlineComment]
multilineComment- : /* [**multilineComment] */
```

The [*name] (e.g. [*inlineComment]) non-terminal will match everything until a newline character. The [**name] (e.g. [**multilineComment]) non-terminal will match everything (including newlines) until the next character sequence is matched. In the case above, a *multilineComment* will match everything between "/*" and "*/".

Here are a few examples that satisfy the assignment rule above:

```
// remove all references to "x" once complete

/* This class will help calculate
      your overdue library fees */
```

The grammar language provides additional internal features well suited for the creation of programming languages / extensions. The above examples should be sufficient to allow the reader to review the Umple language syntax.

The current Umple grammar is shown below; the latest version is maintained online at [81] and is prepared automatically to ensure it is always accurate. For clarity, we omit the grammar pertaining to state machines as it is outside the scope of this work but currently available in the Umple language.

```
program- : ( [[comment]] | [[directive]] )*

directive- : [[glossary]] | [[generate]] | [[useStatement]] |
             [[namespace]] | [[entity]]
```

```
glossary : glossary { [[word]]* }

word : [singular] : [plural] ;

generate- : generate [=generate:Java|Php|Ruby|Json|Yuml|Violet|Umlet] ;

useStatement- : use [use] ;

namespace- : namespace [namespace] ;

entity- : [[classDefinition]] | [[interfaceDefinition]] | [[externalDefinition]] |
          [[associationDefinition]] | [[associationClassDefinition]]

classDefinition : class [name] { [[classContent]]* }

externalDefinition : external [name] { [[classContent]]* }

interfaceDefinition : interface [name] { [[depend]]* [[extraCode]]? }

associationDefinition : association [name]? { [[association]]* }

associationClassDefinition : associationClass [name] { [[associationClassContent]]* }

classContent- : [[comment]] | [[classDefinition]] | [[position]] |
    [[softwarePattern]] | [[depend]] | [[symmetricReflexiveAssociation]] |
    [[attribute]] | [[inlineAssociation]] | [[extraCode]]

associationClassContent- :  [[comment]] | [[classDefinition]] | [[softwarePattern]] |
    [[depend]] | [[singleAssociationEnd]] [[singleAssociationEnd]] |
    [[attribute]] | [[inlineAssociation]] | [[extraCode]]

association : [[associationEnd]] [=arrow:--|->|<-|><] [[associationEnd]] ;

symmetricReflexiveAssociation : [[multiplicity]] self [roleName] ;

inlineAssociation : [[inlineAssociationEnd]] [=arrow:--|->|<-|><] [[associationEnd]] ;

inlineAssociationEnd : [[multiplicity]] [roleName]?

singleAssociationEnd : [[multiplicity]] [type,roleName] ;

associationEnd : [[multiplicity]] [type,roleName]

multiplicity- : [=bound:*] | [lowerBound] .. [upperBound] | [bound]

softwarePattern- : [[isA]] | [[singleton]] | [[keyDefinition]] | [[codeInjection]]

isA- : isA [extendsName] ;

singleton- : [=singleton] ;

keyDefinition- : [[defaultKey]] | [[key]]

codeInjection- : [[beforeCode]] | [[afterCode]]

attribute : [=autounique] [name] ; |
            [=unique]? [=modifier:immutable|settable|internal|defaulted|const]? ([type]
            [=list:[]] [name] | [type,name>1,0]) (= [**value])? ;

beforeCode : before [operationName] { [**code] }

afterCode : after [operationName] { [**code] }

defaultKey : key { }

key : key { [keyId] ( , [keyId] )* }

depend- : depend [depend] ;

extraCode- : [**extraCode]
```

```
comment- : [[inlineComment]] | [[multilineComment]]

inlineComment- : // [*inlineComment]

multilineComment- : /* [**multilineComment] */

position- : [[associationPosition]] | [[classPosition]]

classPosition : position [x] [y] [width] [height] ;

associationPosition : position.association [name] [[coordinate]] [[coordinate]] ;

coordinate : [x] , [y]
```

The tools that enforce the syntactic rules of Umple programs use the grammar above; feedback is available using the Umple plugin to Eclipse, using Xtext. The enforcement of semantic rules, such as ensuring single inheritance, upper bounds >= lower bounds, no cyclic constructors, etc. is fundamentally in the Umple tools, but it is neither user friendly nor comprehensively defined at the current time.

Additional processing of the underlying base languages would be useful to not only identify syntactic violations of the action semantic code (refer to Section 3.5.4), but also to identify whether any of the action language methods violate modeled constraints by doing operations such as directly accessing a member variable. Detailed analysis, documentation and tool support to improve on these semantic and syntactic checks is left as future work.

The intent of discussing the underlying grammar is to help provide a context and completeness of the modeling syntax of Umple, but a true appreciation for the capabilities of Umple are better described by analyzing the semantics alongside the syntax. To that end, a discussion of the use and declaration of the more interesting aspects of the language are presented in subsequent chapters; attributes are discussed in Chapter 4, associations are discussed in Chapter 5, and software patterns are discussed in Chapter 6.

3.7.2 Umple Metamodel

The Umple metamodel lies at the core of the suite of technologies to manage systems built in Umple. It is mature, full featured, and is developed in Umple itself. In other words, future versions of Umple are developed and managed in current versions (with the first few versions being written entirely in Java).

Figure 11 gives an outline of the Umple metamodel. This class diagram was generated using Umple and it should be noted that role names are not displayed in the diagram view of the model but are provided in textual view that follows.

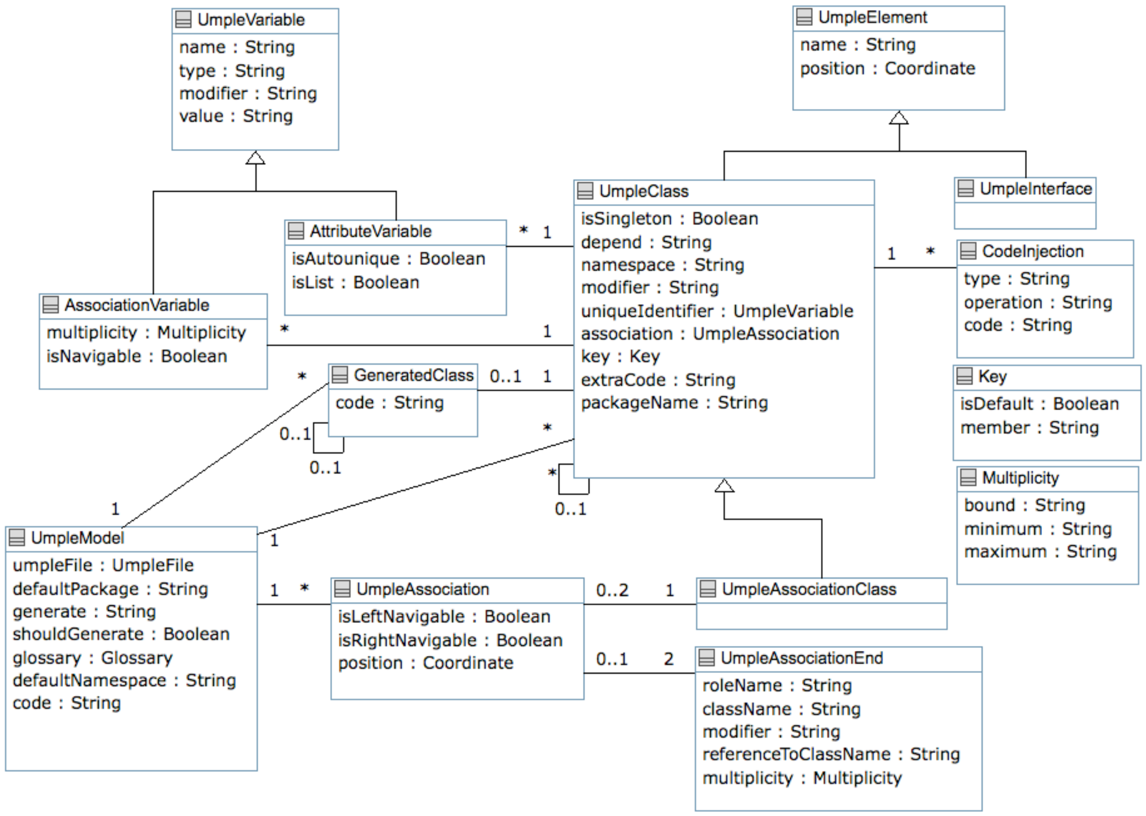


Figure 11: Umple Metamodel (described in Umple)

In the Umple code below we have omitted the algorithmic manipulation of the model for simplicity. This separation of concern is itself a feature and benefit of developing in Umple. Umple allows algorithmic code to be seamlessly integrated with model code without enforcing constraints on the how the resulting code should be implemented. Below is the actual Umple meta-model written in Umple itself.

```
class UmpleModel
{
  depend cruise.umple.util.*;
  depend cruise.umple.compiler.exceptions.*;

  UmpleFile umpleFile;
  defaultPackage = null;
  String[] generate;
  Boolean shouldGenerate = true;
  Glossary glossary = new Glossary();
  String defaultNamespace = null;
  String code = null;

  1 -> * UmpleAssociation association;
  1 -> * UmpleClass;
}

class UmpleElement
{
  name;
  Coordinate position = new Coordinate(100,100,0,0);
```

```
}

class UmpleInterface
{
  isA UmpleElement;
}

class CodeInjection
{
  type;
  operation;
  code;
}

class Key
{
  Boolean isDefault = false;
  String[] member;
}

class UmpleClass
{
  isA UmpleElement;
  Boolean isSingleton = false;
  String[] depend;
  String[] namespace;
  modifier = "class";
  UmpleVariable uniqueIdentifier = null;
  UmpleAssociation[] association;
  Key key = new Key();
  extraCode = "";
  packageName = "";

  1 -> * CodeInjection;
  * -> 0..1 UmpleClass extendsClass;
  1 -> * AttributeVariable;
  1 -> * AssociationVariable;

  before setPackageName { if (aPackageName == null) { return false; } }

 }

class UmpleAssociationClass
{
  isA UmpleClass;
  1 -> 0..2 UmpleAssociation associatedTo;
}

class UmpleVariable
{
  name;
  type;
  modifier; // potential enum, 'settable'
  value;

  before getName { if (name == null) { return defaultName(); } }
  before getModifier { if (modifier == null) { return "settable"; } }
}

class AttributeVariable
{
  isA UmpleVariable;
  Boolean isAutounique;
  Boolean isList = false;
}

class UmpleAssociation
{
  Boolean isLeftNavigable;
  Boolean isRightNavigable;
  Coordinate[] position;
  Integer index = -1;
  0..1 -- 2 UmpleAssociationEnd end;
}

class UmpleAssociationEnd
{
  const Integer MULT_MANY = -1;
  const Integer DEFAULT_MINIMUM = 0;
  const Integer DEFAULT_MAXIMUM = -1;

  roleName;
  className;
  modifier; // potential enum 'internal'
  referenceToClassName;
  Multiplicity multiplicity;

  key { multiplicity, roleName, className, modifier, referenceToClassName }
  //modifier { Settable, Immutable, Internal, Defaulted, Constant }

  before getRoleName { if (roleName == null) { return ""; } }
  before getClassName { if (className == null) { return ""; } }
  before getModifier { if (modifier == null || "".equals(modifier))
    { return "internal"; } }
  before getReferenceToClassName
    { if (referenceToClassName == null) { return ""; } }
}

class AssociationVariable
{
  isA UmpleVariable;
  Multiplicity multiplicity;
  Boolean isNavigable;
  0..1 self relatedAssociation;
}

class Multiplicity
{
  bound = null;
  minimum = null;
  maximum = null;

  key { bound, minimum, maximum }
}

class GeneratedClass
{
  depend java.util.*;

  code = null;
  * -> 1 UmpleModel model;
  0..1 -> 1 UmpleClass uClass;
  0..1 -> 0..1 GeneratedClass parentClass;
}
```

As it currently stands, Umple is used to model Umple, but Umple should not be considered a meta-metamodeling language, a major difference between Umple and EMF. For example, Umple is probably better described as a model-oriented programming language. Just as you can develop Scheme interpreters in Scheme, our team was able to model an Umple modeler completely in Umple.

3.8 Alternative Approaches and Representations to Modeling

The early focus of our work with Umple was to develop the syntax and semantics for the model-oriented language, ensuring the code generation was of high quality and reflected a style of code similar to that *written by hand*.

The following sub-sections will discuss other modeling languages, frameworks, tools and approaches that have been used to represent models, both textually and diagrammatically. Throughout the section we will make reference to an optional-one-to-many relationship between a Mentor and a Student, shown below in Umple.

```
class Student {}

class Mentor {
  0..1 -- * Student;
}
```

3.8.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [82] is a modeling framework and code generation facility for building tools and applications based on a structured data model. Models can be specified in Java, XML, or within a modeling tool like EclipseUML and Rational Software Modeller/Architect. The core tooling provides runtime support to produce a set of Java classes for a model, including adapter classes to manage the display and editing of that model in a basic editor. The metamodel used to store EMF models is called Ecore and it includes persistence (via XMI serialization) as well as a reflection API and notification mechanism to allow programmatic manipulation of EMF objects [83].

EMF includes additional support tools such as EMF.Edit and EMF.Codegen. EMF.Edit can be used for building visual editors based on EMF models. EMF.Codegen provides the tooling for code generation using Java Development Tooling (JDT). The code generation facility uses Java Emitter Templates (JET) [79] to create Java interface and implementation classes, factory and package implementation classes.

EMF can be seen as a modeling and data integration framework for storing meta-data and metamodels. Or, it can be seen as a code-generation platform for building plugins for Eclipse (i.e. Eclipse editors). Umple, in contrast, is a model-oriented programming language for building complete working systems. Umple is represented using a specific metamodel for building software systems and is not a meta-metamodeling framework for building models to build models. Umple uses code generation as a means to create executable systems, reduce boilerplate code, and create systems based on all kinds of languages and platforms, not simply Java and Eclipse.

EMF evolved from the experiences of building editors for WebSphere, whereas Umple evolved as an enhancement to existing programming languages. The Umple authors wanted to raise the level of abstraction of existing languages to help reduce the syntactic noise required to implement model-oriented entities such as attributes, associations, multiplicity and state machines.

Below in Figure 12 is an outline of the EMF approach recommended by the authors of Mastering EMF [84]

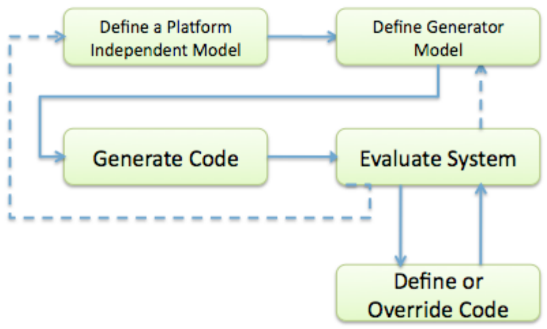


Figure 12: Software Development Workflow using EMF

The Umple approach as shown in Figure 13 follows more closely the software development style of defining a system including both model-level and code-level abstractions. This allows the process to easily fit within existing software development processes including waterfall, iterative, incremental, agile as well as code-and-debug. As mentioned, Umple also includes a simulation engine to allow early exploration of models before committing to a certain design.

Umple integrates with EMF using a model transformation into Ecore. By supporting EMF via model-to-model transformation we get all of the tooling support available to Ecore models without having to sacrifice the somewhat divergent goals of Umple's approach to model-driven development compared to those of EMF.

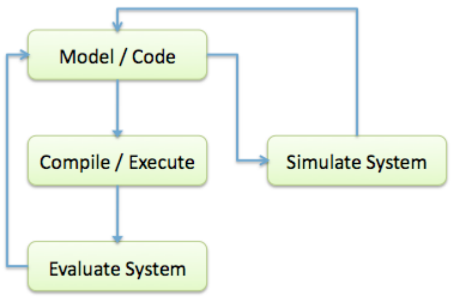


Figure 13: Software Development Workflow using Umple

In order to illustrate the difference in the positioning of Umple as compared to EMF, consider the following Ecore meta-model.

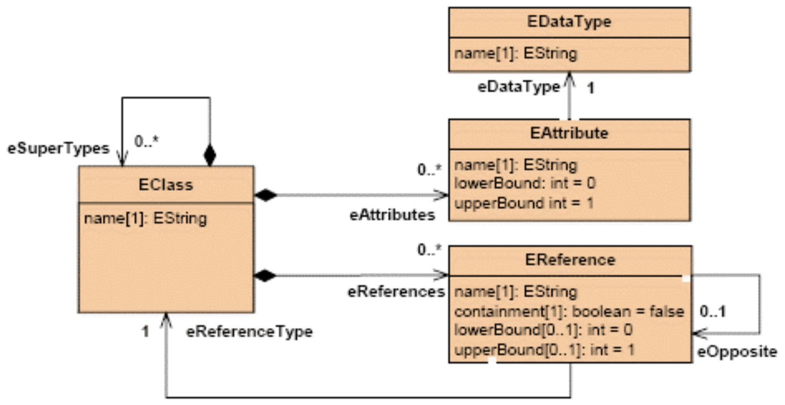


Figure 14: Snippet of Ecore Meta Model [4])

As evident in Figure 14, EMF captures only the class diagram subset of UML (page 14 of [4]). EMF Attributes are seen as pairs of methods. References are similar to associations (one end of an association) with support for referential integrity in the generated code. EMF does not support association classes, or state machines. In contrast, Umple allows associations to be defined as first class entities, in addition to managing the actual references amongst classes. By including state machines as entities, Umple allows simple enumerations in addition to more complex full state machines with events, actions, etc. Umple delegates the behavioural aspects of the system to an action language like Java, PHP and Ruby.

EMF also differs in its support for reverse engineering. Code generated with EMF allows for re-generation of code while preserving user modifications. As stated previously, Umple has set out to provide a model-oriented language that requires no end-user modifications of generated source code in a similar manner that a developer would not modify byte code generated from Java (i.e. no round trip engineering required).

We agree with the premise that one can both model and code within the same artefact (page 13 of [4]). EMF claims that it provides a gentle introduction to modeling (where only state machines are more effectively written in code). The desired approach for creating EMF models is to edit the Ecore model directly (page 17 of [4]), but editing a syntax tree to model can hardly be seen as efficient, and it cannot be expected that a modeler (or even a developer) would model/code in XMI. In fact, EMF was complex enough that Emfatic [85] was developed as a text-based editor built on top of EMF, meaning that the 'gentle' introduction now required the developer to write Emfatic code to generate an EMF model, which could then be used by other systems to generate code (or other editors) for that model.

Umple also prescribes the gentle approach to introducing modeling within software, but without limiting the act of modeling to just class diagrams, as well as ensuring that the process is just as straightforward as that currently available in programming languages (i.e. write, compile, run).

Below is a UML diagram based on the *Music* example from (page 20 of [84]).

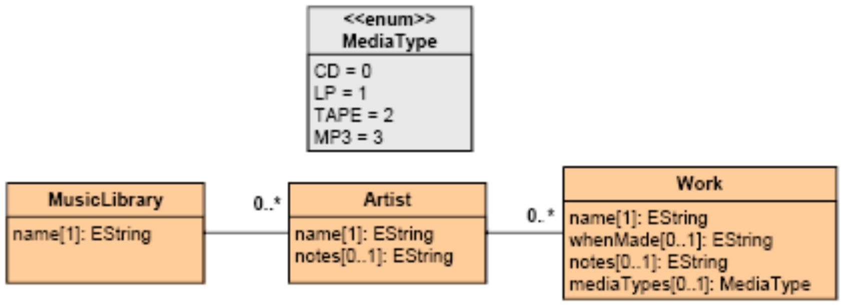


Figure 15: Music Library Example from [4]

The Umple code to describe the model above is written below.

```
class MusicLibrary {
  name;
  * -- 0..* Artist;
}

class Artist {
  name;
  notes = null;
  * -- 0..* Work;
}

class Work {
  name;
  whenMade = null;
  notes = null;
  mediaType { CD, LP, Tape, MP3 };
}
```

The Ecore model for the same model is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="model">
  <eClassifiers xsi:type="ecore:EDataType" name="Time"
instanceClassName="java.sql.Time"/>
  <eClassifiers xsi:type="ecore:EClass" name="MusicLibrary">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="artist"
lowerBound="0" upperBound="-1" eType="#//Artist"
eOpposite="#//Artist/musicLibrary"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Artist">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="note"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="work"
lowerBound="0" upperBound="-1" eType="#//Work" eOpposite="#//Work/artist"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="musicLibrary"
lowerBound="0" upperBound="-1" eType="#//MusicLibrary"
eOpposite="#//MusicLibrary/artist"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Work">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="whenDate"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="note"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="artist"
lowerBound="0" upperBound="-1" eType="#//Artist" eOpposite="#//Artist/work"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="mediaType"
eType="#//MediaType" />
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EEnum" name="MediaType">
    <eLiterals name="CD" value="0"/>
    <eLiterals name="LP" value="1"/>
    <eLiterals name="Tape" value="2"/>
    <eLiterals name="MP3" value="3"/>
  </eClassifiers>
</ecore:EPackage>
```

For comparison purposes, here is the code for our running optional-one-to-many Mentor / Student example.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="model">
  <eClassifiers xsi:type="ecore:EDataType" name="Time"
instanceClassName="java.sql.Time"/>
  <eClassifiers xsi:type="ecore:EClass" name="Mentor">
    <eStructuralFeatures xsi:type="ecore:EReference" name="student"
lowerBound="0" upperBound="-1" eType="#//Student"
eOpposite="#//Student/mentor"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Student">
```

```
    <eStructuralFeatures xsi:type="ecore:EReference" name="mentor"
lowerBound="0" upperBound="1" eType="#//Mentor"
eOpposite="#//Mentor/student"/>
  </eClassifiers>
</ecore:EPackage>
```

EMF models represent an incomplete picture of an executable system, and require that the generated code be edited and maintained after initial generation. EMF generators are expected to be merged with hand-written code, and include sophisticated mechanism to support this reverse engineering feature. EMF and Umple both use Java Emitter Templates (JET) for code generation. EMF focuses on Java (and Eclipse) based systems, whereas Umple supports a multitude of programming (Java, PHP, Ruby) and modeling languages (XMI, TextUML and Ecore). There is a separate EMF4CPP project [86] that provides a C++ implementation of the Ecore metamodel and supports C++ code generation.

Another area of contention with code generated from EMF is the use of interface / implementation separation. EMF mandates that all classes be generated as interfaces with a concrete implementation class. In Umple, the design decision for the most appropriate approach is left up to the modeller / developer.

Code generated from EMF also cannot run without EMF run-time libraries, a limitation that limits EMF modeling to both the Java language and the Eclipse environment. Umple is able to generate code that can run on Java (or PHP or Ruby) without the need to reference or access any special libraries or frameworks. Although there is some value with having an EObject as available in EMF (including the notifier interface), this can easily be modeled within a system as opposed to enforcing it for all scenarios.

```
class Student
{
  isA EObject;
}
```

It should be noted that in the beginning of developing any code generator, it is inevitable that the generated code will be scrutinized and debugged; therefore we adopted a strategy where the generated code is of similar (if not better) quality to that which would be written by an experienced developer.

A main message in favour of EMF is that the generated code is clean, simple and efficient, where "the idea is that the code that's generated should look pretty much like what you would have written, had you done it by hand." (page 23 in [4]). We disagree, mostly due to the coupling to EMF itself; including code geared specifically for Eclipse such as plug-in manifests, adapter classes and factory classes, all of which are forced on the developer, helping to bloat simple models and systems as well as lock the developer into the Eclipse technology both as a development platform but also as a target execution platform.

The focus of EMF-based systems and tooling is on model transformations. One such transformation is code generation for Java, but the overall process is relatively complex when compared to *writing* and *compiling* source code to byte code. The focus of Umple-based systems is on programming and modeling at the same time, whereas the process to *compile* the system is straightforward. As such, in Umple we decided to reuse only those aspects of EMF that focus on code generation (using JET [79] templates), and adopt our own process for building models.

Other industrial and research projects working with programming languages and DSLs have opted away from EMF as well. For example, Aschauer and others [87] have relied on EMF for the first phase of their project, and later realized similar limitations and restarted their Domain Specific Language project without EMF.

As Umple evolves, we continue to investigate other emerging and maturing modeling frameworks and tools, as well as the potential for interoperability. Eugenia [88] is a promising notation that might allow increased support of EMF based tooling. Eugenia is a tagging mechanism built into Emfatic that provides tooling hints for Eclipse's Graphical Modeling Framework (GMF) [89]. GMF helps transform an EMF model into a graphical editor and Eugenia helps to simplify the process of transforming an EMF model into a GMF model that can then be used to generate a custom editor (in our case an Umple visual editor). GMF tooling would simplify the development effort of building an Eclipse-based visual editor and would complement our online platform (built using HTML and JavaScript).

3.8.2 Executable UML (xUML or xtUML)

Model Driven Architecture (MDA) [90] is an OMG standard that "provides a solid framework that frees system infrastructures to evolve in response to a never-ending parade of platforms, while preserving and leveraging existing technology investments." MDA and other Model-Driven approaches place significant emphasis on software models, rather than on source code, as the main development artefacts. MDA is built on other standards such as UML, MOF, XMI and CORBA.

The general process starts with a platform independent model (PIM), which encompasses UML models without any platform dependency. MDA tools then transform these models into platform specific models (PSM) that exploit the facilities provided by the platform. A PSM can then be transformed into executable artefact. In practice, the transformation from PIM to PSM is not fully automated and does require manual intervention to maintain the PIM and PSM in synch as they evolve.

Executable UML (xUML or xtUML) [91] was introduced by Steve Mellor, but instead of transforming a PIM into a PSM, a model compiler can be used to skip the need for a PSM. To achieve such a model compiler, only an unambiguous subset of UML can be used. The general approach to building a system with xUML is to divide a system into areas of experience and within each domain to provide:

1. Classes to represent the data and processing requirements
2. State Machines / State Charts to process signals between the classes identified in Step 1.
3. Class operations to perform the necessary processing based on the state machine actions

xUML is implemented as a UML profile with sufficient details to allow modeling. xUML originally relied on Precise Action Semantics for UML as its action language [58]. Many pUML concepts were later integrated into OMGs Foundational UML (fUML) [59] which is an upcoming standard that specifies the precise semantics for an executable subset of UML; this can be seen as describing a virtual machine for UML. In addition to formal semantics, xUML also supports an $INLINE directive to provide access to platform specific languages like Java or C++ (a current necessity to ensure the resulting generated code requires no additional *editing*). Umple provides a similar mechanism by relying on the action semantics of the target language (e.g. Java or PHP or Ruby). Migration of Umple towards support for UAL will depend on the approval of the UAL standard and adequate compiler / code generators being made available to support the translation of UAL into some executable language.

xUML requires a model compiler with sufficient capabilities to build the entire system, whereas Umple *is* the compiler. In both cases, the generated code should be considered as read-only with no need for further editing or manual changes. Umple relies on JET templates to build an executable system, whereas an xUML compiler relies on what are known as archetypes.

Below is a snippet of code to define an Archetype for a Java based implementation [91].

```
for each object in O_OBJ
public class ${obj.name} extends StateMachine
  private StateMachineState currentState;
  .select many attributes related by object->O_ATTR[R105]
  .for each attribute in attributes
    private ${attribute.implType}${attribute.name};
  .end for

  .select many signals related by object->SM_SM[R301]->SM_EVT[R303]
  .for each signal in signals
    protected void ${signal,name} throws ooaException;
  .end for
}
.emit to file ${obj.name}.java
.end for
```

The archetype above would generate an Employee class similar to the following (assuming no signals for simplicity):

```
public class Employee extends StateMachine
{
  private Integer id;
  private String firstName;
  private String lastName
}
```

BridgePoint [92], OOA Tool [93] and xuml-compiler [94] are three implementations of an xUML compiler.

xUML supports only a subset of the binary relationships available in Umple. xUML supports the following multiplicities for association ends: 1..1, 1..*, 0..1 and 0..*. xUML and Umple both support generalization. Our decision to support all multiplicity combinations was because we identified modeling situations where constraints such as fixed lower and upper bounds are useful (e.g. a joint bank account having two customers). Furthermore, we identified that it was technically less challenging to provide full multiplicity support rather than relying on layering constraints on top of a particular subset of multiplicities.

xUML is a UML profile that defines a sufficient subset of UML to be executable. xUML requires an xUML compiler and typically also an editor to be used. xUML tools analyzed to date seem to all provide visual environments whereas Umple itself requires no specific tool for editing, but has available both textual and visual environments. xUML is not standardized and compatibility issues may arise when changing development environments. In fact, xUML currently supports two versions (one for UML1.x [91] and another for UML2.x [95]).

In a similar manner that Umple can be translated into Java, PHP, Ruby, Papyrus and TextUML; it stands to reason that Umple could be translated into other implementation of xUML and thus allow Umple to support the xUML standard.

3.8.3 Specification Description Language (SDL)

The aim of SDL is to provide an unambiguous specification of software systems with a focus on the telecommunication domain [96]. When modeling in SDL [97], SDL's *variables* provide a similar functionality as an Umple attribute, and SDL channels to pass messages between elements is similar to representing an Umple association. The representation of SDL from UML is discussed in detail by Bourduas' master's thesis [1] and is summarized below in Table 7.

Table 7. UML to SDL mapping rules [1].

UML	SDL
Subsystem	Block Type
Implementation Class	Process Type
Abstract Subsystem	Abstract Block Type
Abstract Implementation Type	Abstract Process Type
Class	New Type
Type	Gate
Associations Between Type Classes	Channels
Operations Described by Type Classes	Signal List
Interface	Interface
Inheritance Between Implementation Classes	Inheritance Between Process Types
Inheritance Between Subsystems	Inheritance Between Block Types

Our Student / Mentor example, represented in SDL "text" is shown below; using the SDL newtype and two channels to define the bi-directional association between Student and Mentor.

```
newtype Student struct
endnewtype Student;

newtype Mentor struct
endnewtype Mentor;

channel oneMentor
  from Student to Mentor with getMentor, setMentor;
endchannel oneMentor;

channel manyStudent
  from Mentor to Student
  with getStudent, getStudents, numberOfStudents, hasStudents,
       indexOfStudents, addStudent, removeStudent;
endchannel manyStudent;
```

Multiplicity constraints are not explicitly defined when declaring channels and instead could be defined within the state machines of a process agent. It should be noted that signals are typically used to represent messaging in distributed systems, which is different from a class's API as available in Umple (and UML).

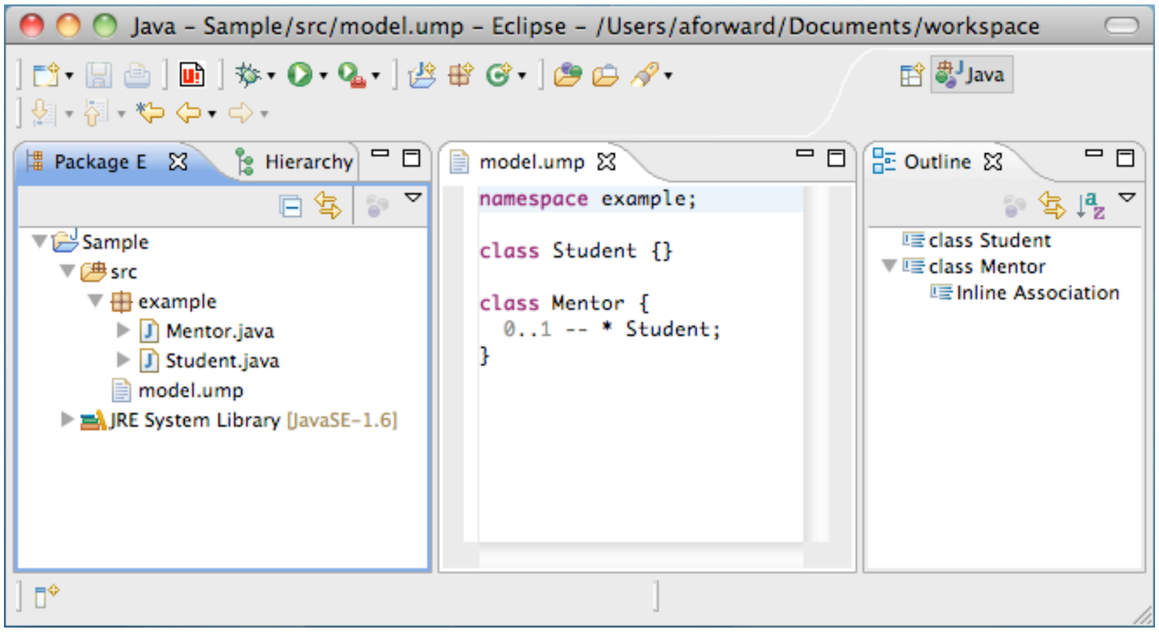
UML was used as the basis for Umple instead of SDL, as UML became a more widely used standard; this was affirmed in our study of software practitioners [12] where fewer than 4% of the participants very often to always use SDL (compared to 52% for UML).

SDL was last updated with version SDL-2000 (first published in 1999 and updated with minor corrections in 2007) [98]. The focus on SDL-2000 was alignment with UML including the eventual creation of an SDL UML profile (Z.109) [99]. There are plans for an SDL-2010 release (originally scheduled for SDL-2008) [100]. An upcoming feature proposed for SDL-2010 is to define a way of providing a binding to other languages such as Java, C and C++ [100].

The approach to binding to a base language differs from Umple's approach of relatively seamless integration into a base language. Umple relies on the underlying base language for action semantics; a benefit of this is that Umple is *not really* a new language, but rather is an extension of existing base languages with model-oriented features. Conversely, an approach of binding to a language such as is proposed in SDL implies it is a new language; an implication that differs from the *spirit* of Umple to appeal to *code-centric* base language programmers.

3.8.4 Xtext

Xtext [101] is an Eclipse-based DSL generator that was released in 2006 and is built on top of Antlr [80]. Xtext provides textual tooling support including a parser, an EMF metamodel (AST based) as well as a rich text editor in Eclipse. The tooling support available for Xtext includes advanced IDE features like intellisense (code suggestions), and syntax highlighting. Umple is currently integrated with Xtext to enhance the IDE experience within Eclipse, but to avoid the strong coupling with Eclipse, the underlying Umple parser and code generator remains separate from the plug-in. A screenshot of the Xtext plugin is shown below in Figure 16.



```
namespace example;

class Student {}

class Mentor {
  0..1 -- * Student;
}
```

Figure 16: Umple IDE (An Eclipse Plugin written in Xtext)

The spirit of Umple is to *write* your model in text and then execute it in the same manner that a developer would write code and execute it with no hidden external dependencies (apart for the runtime environment of the deployment platform). From an end-user perspective, our technology choices are irrelevant as the *developer* writes Umple code, and generates either Java, PHP or Ruby to run in the browser or on a JVM. From a compatibility and tooling standpoint, Umple is looking to better integrate with meta-meta modeling tools, again without losing Umple's simple nature.

Umple is a programming language with support for modeling built right into the language; whereas the tooling and frameworks above provide a mechanism to create model-oriented languages. One typically does not use tools such as Xtext and EMF as described earlier to directly develop software. Instead one relies on tools that use these technologies such as Papyrus [78] and TextUML [101] (which will be discussed further in subsequent chapters). Our decision to integrate with EMF and Xtext, treating them as *peers*, as opposed to building directly on top has allowed our team to embrace the benefits of those technologies without being tightly coupled to the philosophical differences about how best to bridge the model-code divide.

3.8.5 Microsoft's SQL Server Modeling CTP (formely Oslo)

Microsoft is working on a modeling platform called SQL Server Modeling CTP, which was originally codenamed Oslo [102]. This platform provides a modeling language M, a grammar notation language MGrammar, a visual representation of the language called Quadrant and a data store called Repository (which is backed by MSSQL). M is a declarative language used for working with data and building domain models with a focus on querying their data. SQL Service Modeling CTP appears to also follow the text-diagram duality that Umple provides; whereby the model is equally represented in text or in diagrammatic form. M seems geared towards declarative DSL (e.g. MURL [103] for command line URL manipulations or RESTful applications), but it has been shown to also work for code generation [104]. The syntax of a simple M model is shown below.

```
module School
{
  type Student {
    Id : Integer;
  }

  type Mentor {
    Name : Text;
  }
}
```

The intention of SQL Service Modeling CTP seems primarily geared towards building small data-oriented DSLs, allowing developers to *model* their data in text. Conversely, Umple is a general-purpose language for building a variety of systems.

Integration of Umple and Oslo could occur at several levels including: a model-to-model transformation of Umple code into M code; and building an Umple parser in MGrammar. Oslo seems adequate as a rapid DSL generator and consumer (creating your own programming language and then doing something of value with it); while Umple concentrates solely on providing a more abstract and model-oriented approach to developing software systems; whereas Oslo is more geared towards building a system like Umple (as opposed to building systems with Umple).

3.8.6 USE - A UML-based Specification Environment

USE [53] is a subset of UML similar to Umple that contains a textual description of model. Expressions are written in OCL and are used to specify integrity constraints against the model. The process of using USE is shown below in Figure 17.

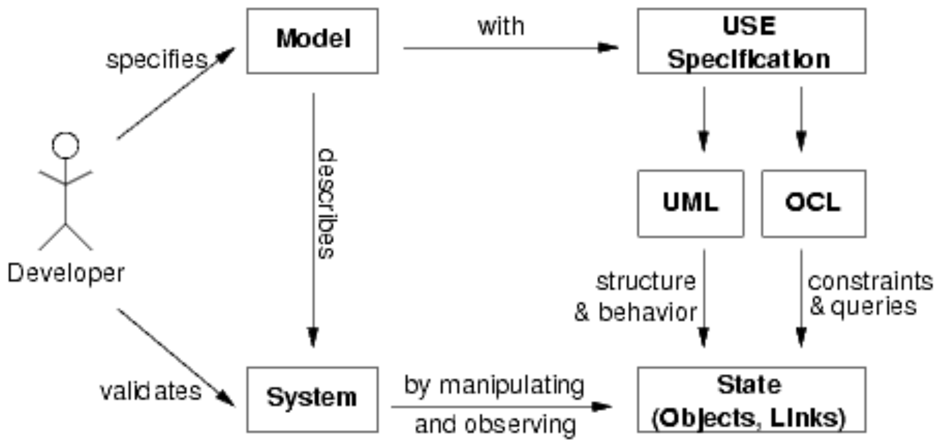


Figure 17: USE approach to software modeling

The USE specification is very similar to Umple, and a model-to-model transformation could be made available much like Umple can currently be transformed to other languages like Umlet and Violet. The USE code to specify the Mentor / Student relationship is shown below.

```
model Example

class Mentor
end

class Student
end

association teaches between
  Mentor[0..1] role mentor
  Student[*] role student
end
```

The approach adopted by USE is geared towards constraint verification and model checking, and not towards model execution. The action semantics of USE is based on OCL and that of Umple is based on whatever base language is chosen (Java, PHP or Ruby). USE models can be simulated using a textual or graphical syntax, but it is not used to build systems. Umple is primarily used to generate systems, in addition to model simulation available at [3] and [105].

3.8.7 Slime UML

Slime UML [106] is an Eclipse plugin that can create UML use case, package and class diagrams with dependencies on EMF and GEF. At the time of writing, code generation is not available.

Figure 18 displays a UML diagram based on the Mentor-Student relationship created using Slime UML

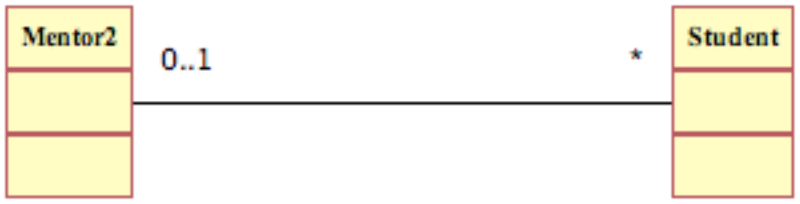


Figure 18: UML diagram created using Slime UML

The underlying XMI is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<slime:DocumentModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:slime="http://de.mvmsoft.slimeuml.model.slimemodel" x="0" y="0">
  <children xsi:type="slime:ClassModel" x="58" y="115" width="50" height="80"
classText="Mentor">
    <outgoingConnections xsi:type="slime:ClassConnectionModel"
target="//@children.1" fromMultiplicity="MULT_0_TO_1"
toMultiplicity="MULT_STAR"/>
  </children>
  <children xsi:type="slime:ClassModel" x="183" y="115" width="50"
height="80" incomingConnections="//@children.0/@outgoingConnections.0"
classText="Student"/>
</slime:DocumentModel>
```

Model transformation from Umple to Slime's XMI schema would be possible using Umple's model generator much like Umple currently supports other XMI based languages like Papyrus and Ecore.

3.8.8 PlantUML

PlantUML [107] is an Eclipse plugin that lets you quickly build several UML diagrams by embedding specialized comments within Java code, similar to the embedded approach described in the previous section. PlantUML supports sequence, use case, class, activity, and component diagrams.

PlantUML is enabled by embedding Javadoc-like comments within Java. The same UML diagram presented in Figure 18 would be represented in PlantUML as follows.

```
/**
 *
 * @startuml Model.png
 * Mentor "1" --- "*" Student
 * @enduml
 *
 */
```

3.8.9 Embedding Models in Source Code

Balz's approach to modeling is to embed the models directly in source code [5]. By enhancing programming code with modeling constructs, one reduces the need for a separate modeling language without losing the design, verification, execution and monitoring views of the system. The motivation for their work is to reduce the number of notations present during software development. A summary of the types of notations encountered during software development is shown in Figure 19.

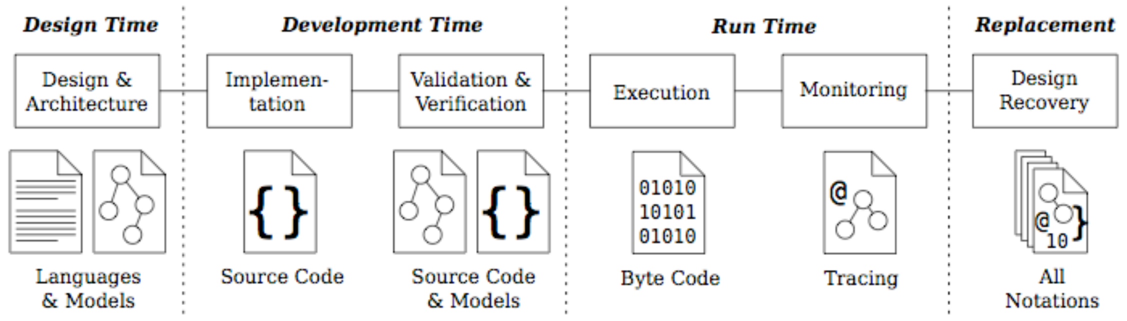


Figure 19: Software Development Notations identified from [5]

They argue that a system cannot be *modeled* completely, resulting in changes to the generated code [108]; and therefore, you have not reduced the number of languages, you have only moved *some* design into the modeling abstraction without completely removing the programming level abstraction. And, those modeling languages that do provide sufficient specificity to describe a complete system, they argue, are too complex. Their approach succeeds where others have failed by removing the need for a separate (and all encompassing) modeling language and instead embedding model abstractions in program code.

Umple's approach is similar in that we have abandoned the need for *conventional* modeling languages and instead create a new language built on top of the action semantics of existing languages. The technical approach to Umple, versus embedding modeling as described above is where Umple stands out.

Umple supports three action language semantics yet all three build on the same modeling syntax; maintaining the ability for modeler to model in Umple and then developers to write code in Umple.

The embedded approach suffers from low cohesion; as "Embedded models are not self-contained, but part of arbitrary program code so that well-defined interfaces realize abstractions between model specifications and other program code." [5]. In contrast, Umple models are self-contained and the language can be used solely for modeling (and doing so in no way hinders the ability to later add *implementation* code).

Umple's approach can be compared to a language preprocessor, something that has been used in the past as it not a foreign concept. We believe this approach provides a more suitable solution to *modeling-in-code* compared to techniques such as embedding model constructs in comments, which can be awkward.

Umple's runtime environment is identical to the runtime of the chosen action language (i.e. Java, PHP, Ruby), whereas the embedded approach requires a special runtime environment and is based heavily on reflection. However, enforcing a specific runtime does provide a convenient mechanism for monitoring and tracing, which is currently outside the scope of Umple and left for future work.

Umple supports attributes, associations, state machines and software patterns, whereas the embedded approach focuses on state machines and process models.

3.8.10 Active Record

The alternative approaches to representing models presented in the preceding subsections are clearly approaches to modeling; whether they be in XMI, in more readable text, or as mark-up tags embedded in existing programming languages. This subsection on Active Record, as well as the subsequent subsections on Data Mappers and Aspect-Oriented Programming (AOP) are further ways of adding modeling abstractions to a program.

ActiveRecord [109] is an object-relationship pattern and has several concrete implementations. The ActiveRecord pattern is implemented in many languages such as Ruby (on Rails) [110], PHP (symfony), and Python (Elixir). Our analysis will consider the implementation available in Ruby on Rails.

Ruby on Rails is a web framework that popularlized the notion of convention over configuration (see glossary for a definition) in modern day web application development. Ruby on Rails includes the following language structures to manage association multiplicities using: *has_one*, *has_many*, and *belongs_to*. Below is an outline of how each supported association multiplicity type can be achieved.

Table 8: Association Notation for Active Record Implementation in RoR

Association Multiplicity	ClassA Structure	ClassB Structure
0..1 – 0..1	has_one	belongs_to
0..1 – *	has_many	belongs_to
* – *	has_many	has_many

ActiveRecord uses a *validate* mechanism to verify constraints; therefore, those constraints can be violated during an interim state, but must be satisfied prior to persisting the model data. This validate-before-safe approach combined with custom coding can be added to an ActiveRecord implementation to enforce additional multiplicity constraints (i.e. 3..4 -- *)

Additional hooks are available when dealing with collections (i.e. many multiplicities) and implementations like Ruby On Rails include specialized hooks to manage association relationships including: before_add, after_add, before_remove, and after_remove.

When dealing with a one-end of an association relationship, several methods are available to manipulate the association end. For example, let us assume that a Post has an Author, the following methods would be available when using ActiveRecord:

Table 9: Active Record API when dealing a multiplicity end of 1 (or 0..1)

Class#method	Description	Available in Umple
Post#author	Get the current author	Yes
Post#author=	Assign the author	Yes
Post#author?	Check for equality to another author	Indirectly
Post#author.nil?	Check to see that the author is set	Yes
Post#build_author	Create a new author	No
Post#create_author	Create and save a new author	No

For a many-end of an association, the following methods would be available. For example, let us assume that a Developer can be a part of several projects. The methods available to the Developer class are shown below in Table 10.

The API available in Umple is similar to the API provided by Active Record. Active Record systems are typically associated with data storage and make available APIs to load, find and create relationships based on primary keys. Umple instead provides a *key* mechanism (discussed in subsequent chapters) to enable such searching without enforcing it.

Below is an example association written in Ruby using the ActiveRecord framework.

```
class Student < ActiveRecord::Base
  has_one :mentor
end

class Mentor < ActiveRecord::Base
  has_many :student
end
```

Table 10: Active Record API when dealing a multiplicity end > 1 (e.g. *)

Class#method	**Description**	**Available in Umple**
Developer#projects	Retrieve all projects for the developer	Yes
Developer#projects<<	Add a project to the developer	Yes
Developer#projects.delete	Remove a project from a developer	Yes
Developer#projects=	Set all projects for the developer	Indirectly supported
Developer#project_ids	Retrieve all project ids for the developer	No
Developer#project_ids=	Set all project ids	No
Developer#projects.clear	Remove all projects	Indirectly supported
Developer#projects.empty?	Answers: does the developer have any projects?	Yes
Developer#projects.size	Answers: how many projects does the developer have?	Yes
Developer#projects.find(id)	Retrieve the project based on its id	No
Developer#projects.exists?	Answers: does this developer belong to the provided project?	Indirectly
Developer#projects.build	Create a new project for this developer	No
Developer#projects.create	Create and save a new project for this developer	No

ActiveRecord provides many convenient language structures to efficiently manage associations. The additional interface provided by ActiveRecord makes available additional convenience methods; otherwise both Umple and ActiveRecord APIs are the same. ActiveRecord distributes the definition of an association between both classes of the association. In contrast, Umple provides a more succinct syntax whereby you can define both ends of an association in one place (either in one of the relationship classes, or in a separate declaration).

The ActiveRecord pattern is available in other languages such as Python (Elixir) and PHP (Symfony). The semantics of these implementations are very similar to that available in Ruby on Rails (using has_one, belongs_to style syntax). In addition, Elixir provides syntactic synonyms more closely related to UML terminology including oneToMany and manyToOne. These implementation provide various approach to the generic ActiveRecord pattern.

Umple currently supports the Ruby language. Due to the ability to *mix-in* frameworks like Rails directly into the Ruby language, it is often difficult to distinguish where the language ends and the framework begins. Future work is being considered to provide additional support geared specifically towards Ruby on Rails (which can be seen as language separate from Ruby itself), and in particular making use of the conventions shown above to align the generated code with "the Rails way".

3.8.11 Data Mapper Pattern

The Active Record pattern could be seen as a simple *convention over configuration* data mapper where the object model is mapped to a persistence layer using specific conventions (e.g. an Author business model would automatically map to an Authors database table). The data mapper pattern extends that convention to allow data to be loaded from and written to a database in an independent manner. A common implementation of this pattern is Hibernate [65].

Hibernate requires a more explicit configuration to map business objects to the persistence layer (from here on, we will refer to this as a database). Below is an example Employee map.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="example">
  <class name="Employee" table="EMPLOYEES">
    <id name="id" column="EMPLOYEE_ID">
      <generator class="native"/>
    </id>
    <property name="firstName"/>
    <property name="lastName"/>
  </class>
</hibernate-mapping>
```

Hibernate provides finer-grained control over object relational mapping and could be integrated seamlessly with Umple: Umple would provide the data model and Hibernate the mapper between the data model and the database.

Although the approach above provides excellent decoupling between the data and business tiers of an application, the separation also reduces the natural cohesion that exists between business and data models. To allow for tighter cohesion, Hibernate also supports the use of annotations to provide in-line directives to map business to data models. The example below is equivalent to the mapper shown above (in addition to the Java code of the Employee class itself).

```
@Entity
@Table(name="EMPLOYEES")
public class Employee
{
  private int id;
  private String firstName;
  private String lastName;

  public Employee(int aId, String aFirstName, String aLastName)
  {
    id = aId;
    firstName = aFirstName;
    lastName = aLastName;
  }

  public void setId(int aId)
  {
    id = aId;
  }

  public void setFirstName(String aFirstName)
  {
    firstName = aFirstName;
  }

  public void setLastName(String aLastName)
  {
    lastName = aLastName;
  }

  @Id
  @Column(name="EVENT_ID", length=5)
  public int getId()
  {
    return id;
  }

  @Column(name = "firstName")
  public String getFirstName()
  {
    return firstName;
  }

  @Column(name = "lastName")
  public String getLastName()
  {
    return lastName;
  }
}
```

Umple purposely does not provide explicit guidance regarding persistence, deferring the choice to the application developer. This freedom to choose from existing persistence mechanisms such as plain text files, XML, databases, or in-memory data stores, provides maximum flexibility to Umple developers. Persistence could be achieved by modeling data access objects to bridge with existing frameworks, or one could simply write one's own. Future work could investigate integration with common persistence mechanisms, perhaps integrated as *drivers* to the Umple language.

3.8.12 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a programming style that attempts to address the problem of scattered and tangled code, typically resulting from how existing languages implement crosscutting concerns [111]. A concern is a property or area of interest of a system. A concern is *crosscutting* if the technique to address the property leads to scattered and tangled code [112] (i.e. the desired functionality does not align with the composition of the solution). Example crosscutting concerns include tracing, debugging and synchronization; typical implementations of these properties involve low-cohesive solutions where the functionality is implemented throughout the application. Aspects can also be used to implement model-oriented concepts such as association relationships.

Using aspects to implement relationships between classes was proposed by Pearce and Noble [113]. Vaccare Braga and Rosana [6] discuss their approach to using aspects to introduce associations between software patterns. Figure 20 shows a theoretical system that integrates the implementation of three software patterns.

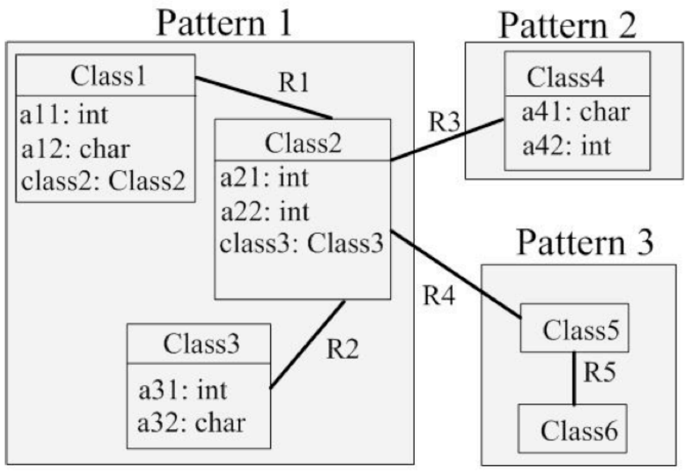


Figure 20: Example system integrating three software patterns [6]

A typical object-oriented approach would be to hard-code the R3 and R4 associations amongst the three software patterns. This approach is problematic due to the increased coupling between the three patterns; making it difficult to isolate the pattern implementations for re-use (or for incremental development). The approach in [6] is to use generics to weave the R3 association into Pattern 1 and Pattern 2 (and the R4 association into Pattern 1 and Pattern 3).

Below is the implementation of the R3 association based on [6].

```
public abstract aspect R3<X extends Class2, Y extends Class4>
{
  interface InterfaceR3 <T>     {}
  declare parents : X implements IntefaceR3 <Y>;
  private T InterfaceR3.class4;
  public T getClass4() { return class4; }
  public void setClass4(T c4) { class4 = c4; }
}
```

This aspect-oriented approach leaves the patterns decoupled and does not *mangle* the purity of the patterns implementation. The implications of such a process is that potential for greater re-use of software pattern implementations such as those defined by the GOF [114] because custom relationships can now be integrated into the pattern without altering the pattern itself.

Umple provides a similar mechanism and the syntax to weave in the R3 association in Umple is shown below.

```
association { * Class2 -- * Class4 }
```

The Umple approach is more concise, and has richer multiplicity semantics compared to the aspect approach above. In addition to weaving associations between classes, Umple also supports weaving attributes, as well as additional action semantic code (i.e. additional methods).

Rajan and Sullivan [112] designed and evaluated a modeling language to unify aspects and classes. AOP languages [115] typically support aspects separate from classes and advice separate from methods. Rajan and Sullivan argue AOP becomes more difficult to understand and use, and can harm modularity if you maintain this separation between classes and aspects. Below in Figure 21, they outline how classes versus aspects and methods versus advice are both nonorthogonal and asymmetric.

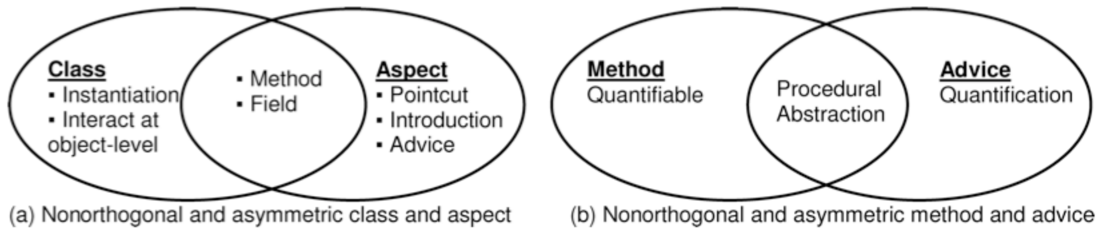


Figure 21: Nonorthogonality and asymmetry in AO languages [112]

Their unified language is called Eos and represents an extension of C#. Eos supports a new ***classpects*** module construct, and supports implicit invocations using before and after bindings as well as overriding using around bindings.

Rajan and Sullivan [112] found that it is both possible and useful to combine the notion of classes and aspects using a single model construct supporting features of both classes and aspects. Umple's approach to dealing with cross cutting concerns is similar. As opposed to introducing separate aspect oriented constructs, Umple instead provides a mechanism to deal with cross-cutting concerns including:

1) Class mix-ins to weave in changes to attributes, and associations;
2) Defining association relationships externally to the class definition;
3) Applying before and after advice based on method join points.

Chapter 4 discusses Umple attributes in greater detail; Chapter 5 discusses associations in greater detail; and Chapter 6 discusses software patterns and the use of before and after advice.

AO languages, such as AspectJ [116] include a join model which implement crosscutting concerns. Join points are either a defined location in a program (static) or predicated on runtime properties (dynamic). Dynamic join points represent the core of AOP languages like AspectJ [117].

Below is a simple example aspect written in AspectJ that inject debugging code into the *getMentor* method of a Student class, based on our running Student/Mentor system.

```
public aspect MyAspect
{
   public pointcut getMentorMethodCall (): call
     (public String Student.getMentor());
   before(): getMentorMethodCall()
     { System.out.println("Student.getMentor start..." ); }
   after(): getIdMethodCall()
     { System.out.println("Student.getMentor end."); }
}
```

In Umple, the same code injection would be defined as follows.

```
class Student
{
  before getMentor { System.out.println("Student.getMentor start..." );}
  after getMentor { System.out.println("Student.getMentor end.”);}
}
```

Note that the before and after advice in Umple resides within the class to which it applies and Umple currently only supports before and after advice based on exact predicate matches of generated API methods and constructors. More detailed analysis of Umple’s support of before and after is discussed in detail in Section 6.3 .

Kiczales provides some evidence in [115] that AOP can improve performance and reduce code volume, helping to improve program comprehensive. One principle of Umple is the reduction in boilerplate code by adding more abstract concepts to the language. The approach has resulted in some features exhibiting the qualities addressed by aspect-oriented programming such as dealing with crosscutting concerns in a cohesive manner.

3.9 Summary

Umple is a programming language that incorporates UML concepts. It is also an environment to create and edit UML models textually.

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

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

It should be emphasized that although Umple contains modeling constructs which were previously not available in other languages, it was purposely built to support a multitude of languages and platforms. For example, at the time of writing, Umple can generate working Java, PHP and Ruby systems, Umple can generate Papyrus, Ecore/EMF and TextUML models as well as visualizations in Yuml, Violet, and Umlet.

Support for languages like C/C++, and C# are forthcoming, and future work is being considered to integrate with existing object-relational patterns to support Ruby on Rails (Active Record) and Hibernate (Data Mapper).

In the current version of Umple, we have focused on implementing associations, attributes and a few design patterns. A discussion of expanding Umple with capabilities such as state machines is left as future work.

Please refer to [81] for full details about Umple. Our online application (UmpleOnline [3]) will allow you to *try out* Umple without having to install any software on your local computer. Simply type in Umple code and generate the Java (or, PHP, Ruby, Papyrus, etc.) implementation.

In the following chapters we analyze the syntax, semantics and resulting generated code for attributes and associations.

Chapter 4 Syntax and Semantics of Attributes

Implementing UML attributes directly in an object-oriented language may not appear to be complex since these languages already provide syntax for defining member variables. At first glance, these member variables may appear equivalent to UML attributes. However, the distinction arises when you consider the differences between modeling a class and implementing it in the underlying language. Member variables can represent not only attributes, but also associations, state machine variables, and internal data such as counters, caching, or sharing of local data. There is also a need to properly define a syntax for characteristics of attributes such as being unique, immutable, or subject to lazy instantiation. In this chapter, we analyze the modeling characteristics of attributes from first principles. We also study the attributes present in seven open-source projects. We describe a model-oriented syntax for attributes that is part of our Umple language. We analyze how existing modeling tools generate code for attributes and finally we demonstrate our own code generation patterns for attributes using Java as the target language.

4.1 UML Attributes

A UML attribute is a simple property of a class. For example, a Student class might have a studentNumber and a name. Attributes should be contrasted with associations, which represent *relationships* among classes and will be discussed in the next chapter. UML 2.0 defines an attribute as a directed association, but we believe that it is a good modeling policy to distinguish between the two as it can help create better models. Our view of an attribute is one that is a primitive type (String, Integer, Double, Date, Time), an enumeration or is a class that only contains attributes (e.g. an Address class that might have street, city, state, and country attributes, all of which only contain strings).

Various constraints can be applied to attributes; for example, they can be immutable or they can be constrained to a certain set of values or a certain range. In translating UML attributes into executable languages like Java, it is common to include accessor (get and set) methods to manage each attribute.

In this chapter, we study the use of attributes in several representative systems, and analyze their semantics. We also provide syntax to represent attributes within the Umple language, which we introduced in the previous chapter. We present code-generation patterns for attributes as used by Umple for the Java language.

4.2 Related Work

A number of approaches and studies have been presented in the literature on code generation from UML [7, 118-120]. In [7], Harrison, Barton, and Raghavachari present a novel approach to code generation of attributes; for each UML class, their approach results in one Java interface class, one abstract class and one regular class. An example is shown below in Figure 22.

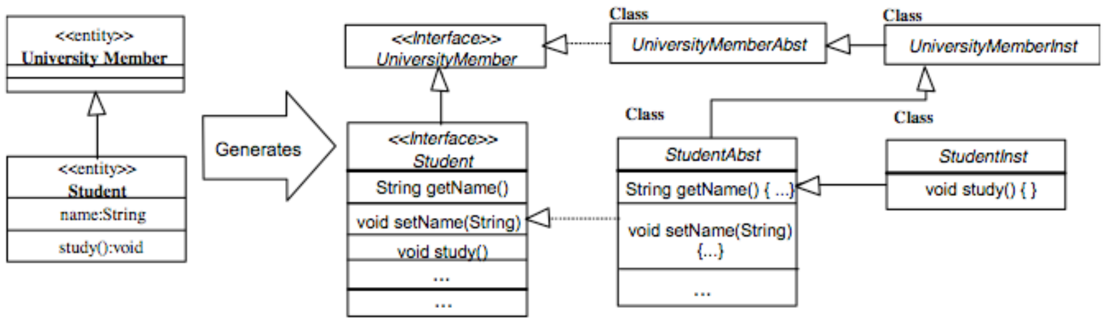


Figure 22: Class Hierarchy generated from UML from [7]

We see that the Student UML class creates a Student interface, an abstract class (StudentAbst) that implements the UML class attributes set and get methods, and finally an instantiable class (StudentInst) that provides an extension mechanism to implement the Student operations. The need for both an abstract class and an instantiable class is to deal with the limitations of UML where it is not practical to express algorithmic logic in a UML class diagram. The StudentInst acts as a skeleton class that must be edited by the developer. This adds a layer of complexity to subsequent code generation and allows for the generated code to become out of synchronization with the model. Umple provides a more direct approach, and the generated code more closely resembles that which would be written by hand, whereas the approach above seems guided more by the limitations of using UML as a programming language and dealing with quasi-generated systems where some of the generated code must edited by hand after generation.

Jifeng, Liu, and Qin [8] present an object-oriented language that supports a number of features like subtypes, visibility, inheritance, and dynamic binding. Their textual object-oriented language is an extension of standard predicate logic [121]. An example class declaration is shown below.

```
class Student extends Person
{
  private String name = "Andrew"
  method helloWorld()
  {
     //supported commands
  }
}
```

The language is similar to Java, where attributes resemble member variables and can be declared private, protected, or public. Methods are explicitly labeled using the keyword ‘method’; they can take a list of parameters and can execute a set of commands. The command set allows for OO-like operations but the syntax more closely resembles a formal Hoare proof than a programming language. The approach to Umple was not to create a *new language*, but rather to enhance existing ones with a more model-oriented syntax and behaviour.

Below in Figure 23 is an example Bank system based on Jifeng’s mathematical model for object-oriented designs.

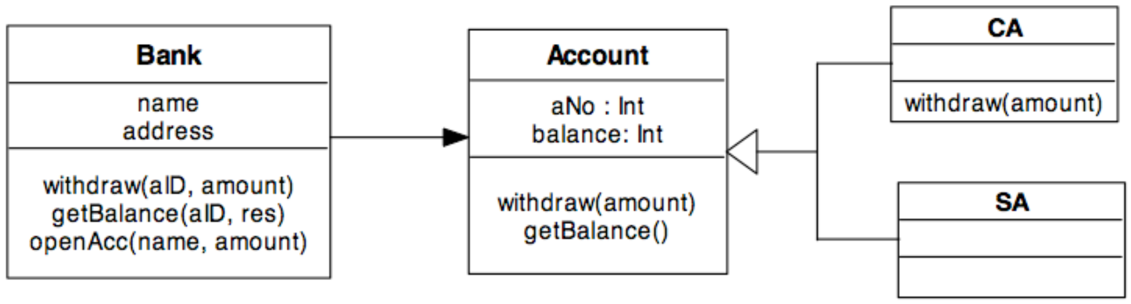


Figure 23: Simple Bank Account examples from [8]

The code in Jifeng’s language to describe the Account class is as follows:

```
class Account {
  protected : Int aNo, Int balance;
  method : getBalance(ϑ,Int b,ϑ) { b := balance } ;
           withdraw(Int x, ϑ, ϑ) { balance ≥ x |- balance’ = balance – x };
}
```

The Umple+Java implementation would resemble the following:

```
class Account {
  Integer aNo;
  Integer balance;

  public void withdraw(Integer amount){
    if (balance >= amount){
      balance = balance – amount;
    }
  }
}
```

The Umple approach reads more like a programming language, whereas the Jifeng et al. approach is more suited to formal proof about model composition and behaviour.

The Umple language can be used to generate entire systems without the need for reverse engineering or round-tripping. This is a desirable quality of a modeling language as reverse engineering tools tend to blindly generate a UML attribute when they encounter a member variable. But, not all member variables are attributes; some represent associations, others private data. Sutton and Maletic [122] advocate that an attribute should reflect a facet of the class interface. These *attributes* can be read or written rather than representing the implementation details of a member variable. They present their findings on the number of class entities, attributes and relationships that were recovered using several reverse engineering tools, revealing the inconsistencies in the reverse engineering approaches. They present their prototype tool, *pilfer*, that creates UML models that reflect the abstract design rather than recreating the structure of the program.

Gueheneuc [123] has analyzed existing technology and tools in reverse engineering of Java programs, and highlights their inability to abstract relationships that must be inferred from both the static and dynamic models of the Java programs. They developed PADL (Pattern and Abstract-level Description Language) to describe programs using class diagrams. However, their proposed approach requires the availability and analysis of both static and dynamic models to build the class diagrams. In another study [124], two commercial reverse engineering tools (Together and Rose) are compared to two research prototypes tools (Fujaba and Idea) and they noted that different tools resulted in significantly different elements recovered from the source code.

Lange and Chaudron [125] conducted an empirical analysis of three software systems and identified violations of a number of well-formedness rules. In one of the systems under analysis, they found 67% of attributes were declared as public without the use of setters and getters.

In the next section we will present our own findings regarding how attributes are used in software systems. Our approach to analyzing attributes attempts to provide a structured method to review, categorize and understand how attributes are used in practice. The approach is subjective at times and susceptible to human error as there are several manual steps throughout the process. But, as shown in most of the cases above, automated analysis done by reverse engineering tools resulted in vastly different perceptions about the systems being studied, despite being automated.

4.3 Attributes In Practice

To ground our work in the pragmatics of industrial software development practices, we set out to analyze how real projects implement and manage attributes. An understanding of existing projects will help us identify code-generation patterns, as well as identify areas where Umple could be improved.

Two goals of our empirical analysis of software attributes are as follows:

- To determine how attributes are defined, accessed and used in practice.
- To find attribute patterns that can enhance the vocabulary with which we describe attributes.

For our research, we considered seven open-source software projects. The criteria by which the projects were selected are described below, followed by a review of the results and the implications for building a model-oriented syntax to describe attributes.

4.3.1 Method

The first step in our case study of attributes in practice was to identify which open source repositories to consider. Our criteria for selecting a repository were that it had at least 1000 full projects in Java or C#. We analyzed 33 repositories, and selected three that met our criteria: GoogleCode (code.google.com), Freshmeat (freshmeat.net) and SourceForge (sourceforge.net).

The second step was to select suitable projects to analyze. The projects were selected by first randomly picking the repository, then randomly selecting the target language (Java, or C#), and finally randomly selecting one of the first 1000 most recently updated projects. Because the number of projects in each repository varied, this approach provided a consistent criteria from which projects could be selected from. The seven projects selected for analysis were: from GoogleCode *fizzbuzz*, *ExcelLibrary*, *ndependencyinjection* and *Java Bug Reporting Tool*; from SourceForge, *jEdit* and *Freemaker*; and from Freshmeat, *Java Financial Library*.

Table 11. Categorizing member variables.

Category	Values	Description
Set in Constructor	No, Yes	Is the member variable set in the object's constructor?
Set Method	None, Simple, Custom	Is the variable private without a set method (i.e. '*None*'), or does the variable have a *'Simple'* setter method / public access or does it have a setter method with '*Custom'* behaviour apart from setting the variable (such as validating constraints, managing a cache, filtering the input, or managing referential integrity)?
Get Method	None, Simple, Custom	Is the variable public ('*None*'), or does the variable have a 'S*imple'* getter method / public access, or does it have any '*Custom'* behaviour like returning defaulted values, using cached values or filtering the output?
Notes	Free Text	Other characteristics such as whether the variable is static, read-only, or derived.

The third step of the case study was to document all member variables. For each variable we recorded the project, namespace, object type, and variable name, as well as certain characteristics presented in Table 11.

Please note that publically available instance variables would fall under the '*Simple*' set and get method, as the *simple* mechanism is then to use the assignment (set) or access (get) of the member variable.

4.3.2 Categorizing Variables (Attributes, Associations, Internals)

By manually inspecting the code in the seven projects, we found 1831 member/instance variables in 469 classes. Instance variables might correspond to attributes or associations, or else they might simply be used for internal data. Our approach to categorizing these member variables as attributes, associations, and internal data is outlined in the following sub-sections.

4.3.2.1Grouping Static Versus Instance Variables

Table 12 gives a distribution of the types of static variables. Of the member variables identified, 620 were static (class variables) and 1211 were instance variables.

Table 12. Distribution of static (class) variables.

Object Type	Frequency	%	Description
Integer	431	69%	All whole number types including primitive integers, unsigned, and signed numbers.
String	53	9%	All string and string builder objects.
Boolean	29	5%	All True/False object types.
Other	107	17%	All other object and data types
	620	100%	

Of the 620 static members analyzed, 90% were read-only constants, 69% were publically visible, and 83% were written in ALL_CAPS. From this point onwards, we will focus on the instance variables. The remaining 1211 instance variables might correspond to attributes or associations, or else they might simply be used for internal data.

4.3.2.2Grouping Instance Variables Based on Constructor/Set/Get Methods

We then looked at how the member variables were accessed (and/or available as part of the constructor). We filtered out any private variables that were also not part of the constructor. Table 13 gives a distribution of the availability of a member variable as part of a constructor's arguments and whether get/set methods are provided (or whether the variable is public and hence has get/set qualities).

To filter out potential internal data, we removed all private variables that did not have get methods from our list of potential attributes (public variables are categorized as having both a set and get method). Prior to doing so, we visually inspected the list and observed that most no-getter variables appeared to be cached objects and results (i.e. size or length), and user-interface controls. Due to the subjectivity in determining the intention of a member variable, we make no quantifiable observation regarding these private-without-access variables. In total, 637 member variables were removed during this process. We also filtered out five member variables with the word *cache*, or *internal* in their name; as they most likely also refer to internal data.

Table 13. Analyzing variables for presence in the constructor and get/set methods.

Constructor	Setter	Getter	Frequency	%	Likelihood of being an attribute, or association (High, Medium, Low)
Yes	Yes	Yes	32	3%	High, full variable access
Yes	Yes	No	8	1%	Low, no access to variable
Yes	No	Yes	44	4%	High, potential immutable variable
Yes	No	No	160	13%	Low, more likely an internal configuration
No	Yes	Yes	318	26%	High, postpone setting variable
No	Yes	No	41	3%	Low, no access to variable
No	No	Yes	179	15%	Medium, no access to set the variable
No	No	No	429	35%	Low, no access at all to set/get variable
			1211	100%	

Once again, public variables are considered to have both setters and getters.

Only 3% of the variables were initialized during construction, could be overwritten via a set method, and could be accessed via a get method, as shown below in Table 13. The most common occurrence of member variables in the systems under study was that they were private variables without external access (i.e. no setter or getter) and not included in the object's constructor. The second most common occurrence was a variable whose value was set only after construction.

4.3.2.3Grouping Primitive versus All Other Object Types

We then grouped *primitive* types (Integer, String, Boolean, Double, Date/Time) as either attributes, or internal data; all other types cannot yet be grouped. Table 14 gives a distribution of all instance members (i.e. non-static variables) for the five most common attribute types, as well as for other types.

Variables for these five types (637 instance variables) either represent attributes or internal data, but not associations. The remaining 574 variables of *other* types were further studied using a recursive approach as described next.

Table 14. Distribution of instance variable types.

Object Type	Number of Variables	%	Description (if required)
Integer	326	27%	All whole number types including primitive integers, unsigned, and signed numbers.
String	169	14%	All string and string builder objects.
Boolean	121	10%	All True/False object types.
Double	12	1%	All decimal object types like doubles, and floats.
Date/Time	9	1%	All date, time, calendar object types.
Other	574	47%	All other data types
	1211	100%	

4.3.2.4Grouping *Other* Types as Attributes or Associations

To find variables representing true attributes as opposed to associations, we used a recursive approach. An attribute is considered to have as its type either: a) a simple data type identified in the first five rows of Table 16, or b) a class that only itself contains instance variables meeting conditions a and b, with the proviso that in this recursive search process; if a cycle is found, then the variable is deemed an association. This approach was partially automated (identifying and removing 12 association member variables) where both ends of the association were defined within the system (so we could automate condition b above). The remaining variables were inspected by hand; subjective judgments were made to categorize the variable type as *entity* or *complex* classes. An entity class was heuristically comprised of primitive data types such as an address, a key, or a measurement. A complex class was heuristically comprised of more complex structures and associations such as actions, containers and nodes. Table 15 was used to help distinguish class categories.

Table 15. Entity versus complex object type criteria hints.

Entity Class	Complex Class
Properties, Formats, Types and Data	Nodes, Worksheets
Files, Records, and Directories	Writers, Readers
Colors, Fonts, and Measurements	Engines, Factories and Strategies
Indices, Offsets, Keys and Names	Proxies, Wrappers, and Generic Objects
	Actions, Listeners, and Handlers
	Views, Panes, and Containers

Node objects (e.g. TreeNode) were marked as complex due to the fact that they are potentially composite (i.e. a Node might have many Nodes).

4.3.3 Results After Filtering Out Associations and Internals

The process described in the previous section was long and arduous, but also a motivation to continue our efforts to better identify internal versus attribute versus association variables. Once the filtering process was complete, we were left with 457 potential attributes and the distribution of attribute types is shown in Table 16. As expected, most potential attributes are integers, strings and booleans.

Table 16. Distribution of attribute types.

Object Type	Frequency	%	Description (if required)
Integer	200	44%	All whole number types (e.g. integers, signed, and unsigned).
String	102	22%	All string and string builder objects.
Boolean	67	15%	All True/False object types.
Double	6	1%	All decimal object types like doubles, and floats
DateTime	5	1%	All date, time, calendar object types.
Other	77	17%	All other data types such as char, Color, File, and Guid
	457	100%	

Table 17 shows how attributes are set and get throughout an object's lifecycle. Only 29 attributes (6%) had immutable-like qualities (available in the constructor, with no setter). About 31% of the attributes were managed internally with no setter and not available in the constructor. Finally, only about 11% of the attributes were available in the object's constructor.

Table 17. Attribute Constructor and Access Method Patterns.

Constructor	Setter	Frequency	%	Probable Intention
Yes	Yes	23	5%	Fully editable
Yes	No	29	6%	Immutable
No	Yes	262	57%	Lazy / postponed initialization
No	No	143	31%	Derived or calculated attribute
		457	100%	

Note that all attributes have a get method (or are publically available).

Next, we investigated the implementation of the set and get methods as described in Table 11. To recall, a set or get method, if present, can be simple or custom. Table 18 illustrates the frequency of the various combinations of attribute set and get methods.

Table 18. Distribution of attribute properties based on type of setters and getters.

Setter	Getter	Frequency	%
Simple	Simple	250	55%
Simple	Custom	1	0%
Custom	Simple	9	2%
Custom	Custom	25	5%
None	Simple	46	10%
None	Custom	126	28%
		457	100%

Over 55% of the attributes used a simple set and get mechanism to manage the attributes, 10% used a simple get method with no set method, and the remainder had at least some custom set or get method.

Next, we investigated the attribute multiplicities. We were able to distinguish between 'one' (0..1 or 1) and 'many' (*) based on the attribute type. List structures and classes with a plural noun (e.g. Properties) were identified as 'many', all other structures were identified as 'one'.

Overall 93% of the attributes had a multiplicity of 'one', leaving only 7% with a 'many' multiplicity. To more finely categorize the multiplicity types would be too subjective, as the multiplicity constraints are programmed in diverse ways.

Next, we studied the characteristics of custom access methods.

For custom *set* method implementations we observed: (a) having a caching mechanism, (b) lazy loading, (c) updating multiple member variables at once, and (d) deriving the underlying member variable's value based on the provided input.

For custom *get* method implementations we observed: (a) constant values returned, (d) default values returned if the attribute had not been set yet, (c) lazy loading of attribute data, (d) attribute values derived from other member variable(s), and (e) the attribute value returned from a previously cached value.

A summary of the major implementation types for set and get methods is shown in Table 19. The frequency shown in Table 19 is based on the total number of attributes and not simply those attributes with custom set or get methods. An interesting observation from this table is that almost a quarter of all attributes were derived from other data of the class.

Table 19. Distribution of attribute set and get method implementations.

Method Implementation	Description	Frequency	%
Derived Set	Input filtered prior to setting variable's value	4	1%
Other Custom Set	Caching values, updating multiple members at once	30	7%
Derived Get	Based on a cache, or other member variables	105	23%
Other Custom Get	Custom constraints applied prior to returning the value	28	6%
Constant Get	Always returns the same value	19	4%

4.3.4 Analysis and Observations

Key findings based on the results from the previous section include:

- Many attributes follow a simple member variable get and set approach, suggesting that such behaviour could be the default, helping to reduce the need for boilerplate set and get methods.
- Few attributes are set during construction, implying a separation between building objects and populating their attributes. Despite this, we believe it is still important to allow attributes to be immutable, and hence generally set in the constructor (if not, then immediately after construction). Immutableness is important to help ensure the proper implementation of hash codes and equality, which, for example, allow consistent storage and retrieval from hash tables. It is also important for asynchronous and distributed processing where the system needs to test equality among instances that are supposed to represent the same thing, but, for example, reside on different processors.
- Attribute multiplicities are almost always 'one' (93%). Based on the overwhelming number of 'one' multiplicity ends, there seems to be little need for syntactic support the vast array of possible multiplicities available in UML such as m, m..n, m..*, etc beyond simple support for the generic 'many' (*).
- Class level attributes (i.e. static) were mostly written in ALL_CAPS (83%), a convention that some languages use instead of a keyword like *static*.

By analyzing existing projects we were able to align our model-oriented language Umple with the observed trends in actual projects. This will be expanded upon in the next section. We were also able to provide code generation that aligns with industry practices – in order to help make the quality of the generated code similar to code that a software developer would write himself or herself.

4.3.5 Umple Syntax for Attributes

In this section we show how the Umple language allows a programmer to specify attributes, with common characteristics found in practice as presented in the last section. In UML, attributes represent a special subset of semantics of UML associations, although pragmatically we have found it more useful in Umple to consider them entirely separately.

The main features of Umple's syntax for attributes, and its code generation, result from answering the following three questions.

Q1: Is the attribute value required upon construction?

Q2: Can the attribute value change throughout the lifecycle of the object?

Q3: What traits / constraints limit the value and accessibility of the attribute?

As we will discuss in subsequent sections, most current code generators provide the most liberal answers to the questions above: no, the value is not required upon construction; yes the attribute value can change; and no there are no constraints on or special traits of the attribute. In UML you can add OCL constraints to answer Q3, but there is no straightforward way to specify answers to Q1 and Q2.

As observed in the previous section the answer to Q1 is usually 'no' (89%), and the answer to Q2 is split between 'yes' (62%) and 'no' (38%).

The answer to Q3 is 'none' about half the time (55%) – in other words most attributes have straightforward set and get behaviour. The other half of the time there are a large number of possible characteristics to consider, since each project has unique constraints to which an attribute much conform. Two of the characteristics we observe reasonably frequently are uniqueness and default values.

In the work below, we look at how the answers to the questions above could be reflected in a model-oriented syntax, with consideration of the affects on code generation. We also determine which scenarios do not make semantic (or pragmatic) sense in order to further simplify the attribute syntax.

4.3.6 Is the Attribute Specified in the Constructor (Q1)?

First, let us consider attributes that are available in the constructor (Q1.Yes). By default in Umple, an attribute's value is required at construction, and the syntax to describe this scenario is to declare the attribute with no extra adornment. E.g.

```
String x;
Integer y;
```

For attributes that are not to be set during construction (Q1.No), the Umple syntax is to provide an initial value (which can be null) to the attribute, as shown below.

```
String x = "Open";
Integer y = 1;
String z = null;
String p = nextValue();
```

The initialized value follows the semantics of the base language (e.g. Java, PHP or Ruby). It can either be a constant as we see for x and y, uninitialized as we see for z or an arbitrary method call (that the developer must define) as in the case of p.

4.3.7 Can the Attribute Change After Construction (Q2)?

By default in Umple, an attribute's value can change after construction (Q2.Yes), requiring no additional syntax to describe this scenario. A set method is generated in this case.

Attributes that cannot change after construction (Q2.No) are marked 'immutable'; the value set in the constructor cannot then be changed. No set method is generated.

```
immutable String x;
```

As discussed above, immutability is very useful to provide consistent semantics for equality and hashing, although not many attributes in our study exhibited the immutable property. Part of the issue is that in current languages it is difficult to specify.

There are also instances where an attribute should be immutable, but it might be the case that the value is not available at the instant of construction, only very shortly thereafter. Examples of this include application frameworks where the creation of an object is controlled by the framework and is outside the developer's control. In these cases, an initially empty object is provided to the application, to be immediately populated with the attribute data that cannot then be changed. Therefore, to support this case in Umple, we allow immutable attributes to delay instantiation by initializing the value to null as shown below.

```
lazy immutable y;
```

The use of the lazy syntax means that the attribute is not initialized in the constructor (i.e. it is not part of the constructor's signature). The generated code will contain a flag to track whether the object has been set yet, allowing only a single set to occur. We elaborate on immutability and the underlying executable implementation in following sections.

4.3.8 What Other Characteristics Does the Attribute Possess (Q3)?

The potential characteristics are somewhat limitless. In our analysis of existing software we found three somewhat common patterns that we have incorporated into Umple and will elaborate on them below.

Before we consider these special cases explicitly supported by the Umple language, we should first recognize that many attributes have no *explicit* constraints. In general, a property like a name or jobTitle has no constraints apart from those enforced by the underlying implementation language (i.e. type checking).

4.3.8.1Attribute Uniqueness

The first characteristic we will consider is *uniqueness*. In databases, being able to guarantee uniqueness allows for efficient searching and equality assertions; many domains also have data that is unique by design (e.g. flight numbers in an airline). In some cases, objects are automatically assigned a unique identifier upon creation, whereas in others uniqueness is checked whenever the attribute is set.

In UML, an attribute's uniqueness can be specified using a *qualifier*, which is really a special type of attribute. Let us consider an example of an Airline that has many RegularFlights.

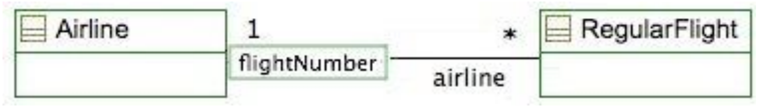


Figure 24: Unique flightNumber on the airline association

Two RegularFlights of the same Airline should not have the same flightNumber, representing a uniqueness constraint on the system. It is also possible to allow for global uniqueness within a system, for example an ipAddress attribute should perhaps be unique throughout the entire application.

In the cases above, it is the responsibility of the end-user developer to define unique attributes. The example below provides a mechanism to allow the underlying system to manage the generation of valid and unique identifiers, within or outside the context of an association. The Umple syntax to describe the constraints mentioned above is shown below.

```
unique String ipAddress;
unique Integer flightNumber on airline;
```

The second example above for *flightNumber* includes an additional qualifier to limit uniqueness of that attribute to be unique for a particular airline. The qualifiers available for the uniqueness modifier are limited to association ends of the containing class with a multiplicity of 0..1 or 1.

Uniqueness for integer attributes can also be managed automatically in Umple using the 'autounique' keyword. Autounique includes basic support for an Integer (1,2,3,…) and a String (a,b,c,…) attribute id generator. The syntax for supported autounique attributes is shown below.

```
autounique Integer flightNumber;
```

For more complex autounique attributes, such as generating a license number, user name or email address, the developer must also provide a custom function to manage uniqueness. The syntax in Umple is as follows.

```
autounique String userId = { nextUserId(); }
```

The syntax uses a lambda function notation (similar to derived attributes presented below in 4.3.8.4).

4.3.8.2Default Values

Next, let us consider *defaulted* values. A default value ensures an attribute is never *unspecified*. Any time the internal value of the attribute is unspecified, the accessor method (i.e. get method) would return a default value. The generated API includes methods to reset the attribute to the default and determine what the default would be. Note that defaulted attributes also do not appear on the constructor's argument list.

```
defaulted type = "Long";
```

4.3.8.3Internal Members

Next, let us consider all other custom constraints and internal data. Not all attributes conform to the standard simple set/get semantics as described in the previous sections. In addition, many member variables are not attributes, but are support variables used internally [126]. To specify these in Umple, the syntax is:

```
internal Integer cachedSize = 0;
```

Internal attributes do not form part of the constructor and do not have accessor methods, allowing developers to manage this data in the way they see fit. However, the variables are private.

4.3.8.4Derived Values

A derived value is a value that is based on possibly several values. For example, the area of a circle is a derived value based on the circle's diameter. A derived value differs from an initialized value in that an initialized value is set once upon construction and can then later be overwritten; whereas a derived value is always set based on a predetermined calculation such as the area of a circle, or perimeter of a rectangle.

The syntax for derived attributes uses a lambda-function like notation and is shown below; this function must make reference to the attribute or attributes upon which the derived value is based.

```
Integer length;
Integer width;
Integer perimeter = { 2 * getLength() + 2 * getWidth() }
Integer area = { getLength() * getWidth() }
```

Derived attributes only generate get methods, as generating a set method would make no sense.

Caching mechanisms for derived values is currently outside the scope of the Umple language, due the many design factors to take into consideration to ensure correctness. Below is an example showing how an Umple developer could implement his/her own caching mechanism.

```
Integer length;
Integer width;

internal Boolean isPerimeterCached = false;
internal Integer cachedPerimeter = -1;
Integer perimeter = { 2 * getLength() + 2 * getWidth(); };

before getPerimeter {if (isPerimeterCached) { return cachedPerimeter; } }
after getPermimeter {cachedPerimeter = aPerimeter; isPerimeterCached = true;}
after setLength { if (wasSet) { isPerimeterCached = false; } }
after setWidth { if (wasSet) { isPerimeterCached = false; } }
```

In the example above, we use our code injection syntax described in Section 6.3.1 . The code attaches a dirty bit to all attributes involved in calculating the derived value.

4.3.8.5Constants (const)

Although technically not an object attribute, Umple also supports a *const* keyword to denote read-only class level attribute. An example is shown below.

```
const MAX = 10
```

4.3.8.6List (Many) Attributes

Finally, let us consider a ‘many’ multiplicity. Using the square brackets [] syntax, attributes can also be represented as multiple instances of the attribute type.

```
String[] names;
String[0..3] addressLines;
```

By default, list attribute multiplicities are many (*) as illustrated by the *names* attribute above. The second example, *addressLines*, explicitly defines the multiplicity constraint as 0..3.

4.4 Existing Tools that Generate Code for Attributes

In the previous sections, we investigated how attributes are used in practice. We then presented the Umple syntax to describe attributes at a more abstract level. The next step in our research has been to understand the code generation patterns of existing tools to see how they deal with the complexities of implementing attributes.

The UML modeling tools considered were identified by two sources: Gartner [127] and an online list of UML tools from [128]. We selected four open source projects and one closed source application to analyze.

ArgoUML and StarUML are two of the most active open source UML modeling tools and RSA has the largest market share, helping ensure that our review of existing tools is relevant and popular [127, 129].

Table 20. UML code generation tools.

Tool	Version	Source
ArgoUML	0.26.2	argouml.tigris.org
StarUML	5.0.2.1570	staruml.sourceforge.net
BOUML	4.11	bouml.free.fr
Green	3.1.0	green.sourceforge.net
RSA	7.5	ibm.com/software/awdtools/architect/swarchitect
Eclipse EMF	2.5	www.eclipse.org/modeling/emf/

The above are not the only modeling tools that were considered. Table 21 lists additional candidate tools for completeness of our research. Most of these tools either did not provide code generation capabilities, did not provide code generation of class diagrams, did not provide sufficient documentation to achieve code generation, or did not run on the platforms we had available.

Table 21. Additional UML tools not considered for our case study.

Tool	Version	Source
Acceleo	2.5.1	acceleo.org
Jink UML	0.745	code.google.com/p/jink-uml
Hugo	0.51	pst.ifi.lmu.de/projekte/hugo
Umbrello	2.0	uml.sourceforge.net
Umlet	9.1	umlet.com
Fujaba	5.0.1	wwwcs.upb.de/cs/fujaba/
Modelio	1.0.0	modeliosoft.com
Topcased	1.2.0	topcased.org
NetBeans UML Modeling	6.7	netbeans.org
Papyrus	1.11.0	papyrusuml.org

For the tools listed in Table 20, we used a Student class with two attributes, an integer representing an id, and a list of names (represented as simple strings).

```
class Student
{
  Integer id;
  String[] names;
}
```

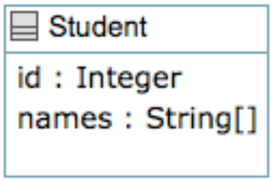


Figure 25: Student class with a simple id attribute and a list attribute

4.4.1 ArgoUML

ArgoUML is an open source modeling platform that provides code generation for Java, C++, C#, PHP4 and PHP5. Below is the code generated from Figure 25.

```
import java.util.Vector;
public class Student {
  public Integer id;
  public Vector names;
}
```

The generated code provides public access to set and get the attributes; a pattern not often used in practice as it breaks encapsulation by fully exposing the internals of the class. The attributes are made directly available without the inclusion of accessor methods.

4.4.2 StarUML

StarUML is an open source modeling tool. Its generated code is almost identical to that of ArgoUML, except that the import clause was missing, and *names* is declared erroneously as a String. StarUML's generated code hence does not account for the *many* multiplicity, resulting in unusable generated code.

4.4.3 Bouml

BOUML is another free tool based on UML 2 that provides source code generation for C++, Java, Idl, PHP and Python. Its code is very similar to the two systems above, with the primary difference being that the attributes are declared as private and genericity is used:

```
class Student {
  private int id;
  private List<String> names;
}
```

The Bouml source code does not provide any mechanism to set or get the attributes; this code must be written by hand after code generation.

4.4.4 Green Code Generator

Green UML is another UML editor that can generate source code from a class diagram.

```
import java.util.List;
public class Student {
  int id;
  List<String> names;
  public Student() {}
}
```

Green's implementation contains similar flaws as BOUML, where access to the attributes must be written manually after generation.

Although Green UML does attempt to provide some additional source code generation to manage the various types of association multiplicities available; the results provide little if any added benefit in representing the model's intentions.

4.4.5 Rational Software Architect (RSA) and RSA Real-Time

IBM's Rational Software Architect is a full-fledged development environment that supports model-driven development including source code generation from UML diagrams. Its code for Figure 25 is as follows:

```
import java.util.Set;
public class Student {
  public Integer id;
  public Integer getId() { return id; }
  public void setId(Integer id) { this.id = id; }
  private Set<String> names;
  public Set<Student> getNames() { return names; }
  public void setNames(Set<Student> names) {
    this.names = names;
  }
}
```

RSA's model transformation into Java code provided some flexibility regarding the template patterns including (a) which Java collection to use, and (b) whether or not to include get/set methods for the attributes. RSA Real Time generated the same code as RSA for class diagrams.

4.4.6 Eclipse Modeling Framework (EMF)

EMF is described in detail in Section 3.8.1 . As shown below, classes are generated as interfaces.

```
import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.ecore.EObject;
public interface Student extends EObject {
  int getId();
  void setId(int value);
  EList<String> getNames();
}
```

The implementation of the Student class (StudentImpl) is shown below.

```
import example.ExamplePackage;
import example.Student;
import java.util.Collection;
import org.eclipse.emf.common.notify.Notification;
import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.ecore.EClass;
import org.eclipse.emf.ecore.impl.ENotificationImpl;
import org.eclipse.emf.ecore.impl.EObjectImpl;
import org.eclipse.emf.ecore.util.EDataTypeUniqueEList;

public class StudentImpl extends EObjectImpl implements Student {
  protected static final int ID_EDEFAULT = 0;
  protected int id = ID_EDEFAULT;
  protected EList<String> names;

  protected StudentImpl() { super(); }

  public int getId() { return id; }

  public void setId(int newId) {
    int oldId = id;
    id = newId;
    if (eNotificationRequired())
      eNotify(new ENotificationImpl(this, Notification.SET,
                     ExamplePackage.STUDENT__ID, oldId, id));
  }

  public EList<String> getNames() {
    if (names == null) {
      names = new EDataTypeUniqueEList<String>(String.class, this,
                     ExamplePackage.STUDENT__NAMES);
    }
    return names;
  }

  public Object eGet(int featureID, boolean resolve, boolean coreType) {
    switch (featureID) {
      case ExamplePackage.STUDENT__ID:
        return getId();
      case ExamplePackage.STUDENT__NAMES:
        return getNames();
    }
    return super.eGet(featureID, resolve, coreType);
  }

  public void eSet(int featureID, Object newValue) {
    switch (featureID) {
      case ExamplePackage.STUDENT__ID:
        setId((Integer)newValue);
        return;
      case ExamplePackage.STUDENT__NAMES:
        getNames().clear();
        getNames().addAll((Collection<? extends String>)newValue);
        return;
    }
    super.eSet(featureID, newValue);
  }

  public void eUnset(int featureID) {
    switch (featureID) {
      case ExamplePackage.STUDENT__ID:
        setId(ID_EDEFAULT);
        return;
      case ExamplePackage.STUDENT__NAMES:
        getNames().clear();
        return;
    }
    super.eUnset(featureID);
  }

  public boolean eIsSet(int featureID) {
    switch (featureID) {
      case ExamplePackage.STUDENT__ID:
        return id != ID_EDEFAULT;
      case ExamplePackage.STUDENT__NAMES:
        return names != null && !names.isEmpty();
    }
    return super.eIsSet(featureID);
  }
}
```

The EMF implementation of the Student class has the following qualities.

- The get implementation when the multiplicity is *one* returns the member variable (i.e. a simple get method).
- The set implementation when the multiplicity is *one* assigns the member variable as well as notifies other interested instances using the Observable pattern.
- Attributes with multiplicity *many* are accessed directly via a get method that returns an EList interface (EList extends Java's java.util.List interface with two additional *move* methods).
- The *many* attributes are lazily instantiated (i.e. when first requested).
- The generated classes have only one constructor with no parameters.
- Additional methods provide generic access to all attributes using an eGet, eSet, eIsSet and eUnset, using an argument to specify the attribute.

The simple attributes follow a similar pattern as would be written by hand; with the addition of an observable mechanism to notify other objects of changes to this instance. The object maintains little control over the *many* attributes, as the *list* is passed directly to the caller for manipulation (and unfortunately the possibility of mangling it, too, thus making it impossible to maintain constraints). And, because of the generic EList being used, it is unlikely to contain model specific constraints, or validation. The additional generic access methods (eGet, eSet, etc) expose the internals of the implementation and are presumably used by Eclipse to provide a generic mechanism to interact with EMF generated code with less reliance on reflection.

EMF does not directly support attribute constraint checking or validation. EMF does provide some *hints* about how to extend the EMF code generator to support custom behaviour, like validation. Below is a sample validation interface generated by EMF for the Student class.

```
/**
 * A sample validator interface for {@link example.Student}.
 * This doesn't really do anything, and it's not a real EMF artifact.
 * It was generated by the org.eclipse.emf.examples.generator.validator plug-
in to illustrate how EMF's code generator can be extended.
 * This can be disabled with -vmargs -
Dorg.eclipse.emf.examples.generator.validator=false.
 */
public interface StudentValidator
{
  boolean validate();
  boolean validateId(int value);
  boolean validateNames(EList<String> value);
}
```

Please note that we left the EMF comment in place to highlight that this mechanism is not integrated within EMF. Also note that only an interface is provided with no concrete implementation.

Of all the code generators analyzed, EMF provided the most comprehensive and complex implementation. As such, we investigated additional code generation features of EMF. The following observations were found:

- EMF treats multiplicities uniformly as either optional-one, or many. No additional constraint checking is provided if the relationship is mandatory (e.g. 1..1), or limited (e.g. 3..5).
- EMF does not support immutable attributes. All object constructors are empty. If you set the *changeable* property to false, the attribute will always be null and setting a *default* value will (in essence) make the attribute a constant; neither of which represent a meaningful implementation of immutability.
- EMF ignores the *unique* properties for one multiplicity attributes and only applies to many-multiplicities.
- EMF ignores the *ordered* property.

As discussed in Section 3.8.1 , EMF is geared specifically towards the Java language run within the Eclipse environment. As such, the generated code is tightly coupled to the EMF framework and includes additional methods (eSet, eGet, eUnset) and classes (PoFactory, PoPackage, PoSwitch) and most likely would not extend well to other languages like PHP or Ruby, unless the EMF framework was also ported to those languages or a bridge was created (such as JRuby) allowing both languages to work in unison.

4.5 Generating Code for Attributes using Umple

As seen in the previous example, existing code generating tools do not consider the many complicated facets of implementing attributes in a target language like Java. Section 4.3.5 described the syntax of Umple attributes and Section 4.4 highlighted the simplistic implementation of attributes in several code generators and software modeling tools. In this section, we demonstrate code generation patterns based on Umple attributes and generated into Java code.

Umple provides the full breadth of capabilities of the underlying runtime language (currently Java, PHP or Ruby), without the need to manually edit the underlying generated code. Our approach can be compared to compiling Java into byte code; just as a developer should not have to analyze the generated byte code the same should be true of the underlying generated code of Umple.

Regardless, it is unrealistic to believe that a language such as Umple will allow software developers to simply ignore the generated code and work entirely in the Umple language (especially in the beginning). It is more likely that early adopters will want to (or worse, need to) explore the underlying implementation. And, it is because of early adopters that we strive to provide high-quality and well-styled generated code.

The following example shows how one would declare attributes in the first steps of modeling a system using Umple. To help distinguish between Umple and Java code, the Umple examples use dashed borders in light-grey shading, and pure Java examples use solid-line borders with no shading. For conciseness, we have omitted the code comments and some additional methods not related to the attributes in the generated Java.

4.5.1 Basic Attributes

At the core of an Umple attribute is a name. The implications on code generation include a parameter in the constructor, a default type of String and a simple set and get method to manage access to the attribute. Consider the attribute code shown below:

```
class Student {
  name;
}
```

This would result in the following Java implementation.

```
public class Student {
  private String name;
  public Student(String aName) { name = aName; }
  public boolean setName(String aName) {
    name = aName;
    return true;
  }
  public String getName() {
    return name;
  }
}
```

The style is similar to that generated from RSA, and similar to the *simple* cases observed in the open source projects. As we discussed in Section 2, relatively few attributes are set in the constructor. In Umple, this can be achieved by specifying an initial value as shown below.

```
class Student {
  name = “Unknown”;
}
```

The generated code would differ only in the constructor, which is shown below:

```
public class Student {
  ...
  public Student() {
    name = "Unknown";
  }
  ...
}
```

Please note the initial value can be null, or some user defined function written in the underlying target language (i.e. Java).

4.5.2 Immutable Attributes

If a Student’s name variable was declared immutable, as shown in Section 4.3.7 the resulting Java code would be the same as the basic attribute implementation from the previous section, except that there would be no setName method.

In the default case, immutable attributes must be specified on the constructor, and no setter method is provided. But, Umple also supports lazy instantiation of immutable objects (also discussed in Section 4.3.7).

Lazy immutable attributes are neither initialized nor provided as arguments in the constructor. A set method is provided, but can only be called once as shown with the example below.

```
public class Student {
  private String name;
  private boolean canSetName;
  public Student() {
    canNameBeSet = true;
  }
  public boolean setName(String aName) {
    boolean wasSet = false;
    if (!canSetName) { return false; }
    canSetName = false;
    name = aName;
    wasSet = true;
    return wasSet;
  }
  public String getName() {
    return name;
  }
}
```

The implementation above includes an additional check *canSetName* to ensure that the variable is only set once. Note that the *wasSet* variable is available to be used by before and after code injections, so they can base their behaviour on the success (or failure) of the set or get operation.

4.5.3 Defaulted Attributes

A defaulted attribute provides an object with a default configuration that can be overwritten:

```
class Student {
  defaulted name = "Unknown";
}
```

The underlying Java implementation is shown below.

```
public class Student {

  private String name;
  public Student() {
    resetName();
  }
  public boolean setName(String aName) {
    name = aName;
    return true;
  }
  public boolean resetName() {
    name = getDefaultName();
    return true;
  }
  public String getName() {
    return name;
  }
  public String getDefaultName() {
    String aName = "Unknown";
    return aName;
  }
}
```

The subtle differences between an initialized and a defaulted attribute include the following: First, a defaulted attribute is specified in the constructor, whereas an initialized attribute is not. Second, a defaulted value is guaranteed to return its default if it has not been initialized (or if it has been reset), whereas an initialized attribute guarantees only the particular value after construction (which can be set to null afterwards).

4.5.4 Unique Attribute

The unique attribute guarantees its uniqueness within a particular class.

```
class Student {
  unique String name;
}
```

The implementation in Java is shown below.

```
public class Student {
  private static List<String> allNames = new ArrayList<String>();
  private String name;

  public Student(String aName)
  {
    if (!setName(aName))
    {
      throw new RuntimeException("Unable to create student due to name");
    }
  }

  public boolean setName(String aName)
  {
    boolean wasSet = false;
    if (indexOfAllName(aName) != -1) { return wasSet; }
    String oldName = name;
    name = aName;
    wasSet = true;
    if (wasSet) { addAllName(aName); removeAllName(oldName); }
    return wasSet;
  }

  public String getName()
  {
    return name;
  }

  public static boolean addAllName(String aAllName)
  {
    boolean wasAdded = false;
    wasAdded = allNames.add(aAllName);
    return wasAdded;
  }

  public static boolean removeAllName(String aAllName)
  {
    boolean wasRemoved = false;
    wasRemoved = allNames.remove(aAllName);
    return wasRemoved;
  }

  public static String getAllName(int index)
  {
    String aAllName = allNames.get(index);
    return aAllName;
  }

  public static String[] getAllNames()
  {
    String[] newAllNames = allNames.toArray(new String[allNames.size()]);
    return newAllNames;
  }

  public static int numberOfAllNames()
  {
    int number = allNames.size();
    return number;
  }

  public static boolean hasAllNames()
  {
    boolean has = allNames.size() > 0;
    return has;
  }

  public static int indexOfAllName(String aAllName)
  {
    int index = allNames.indexOf(aAllName);
    return index;
  }
}
```

Uniqueness is provided by filtering incoming values on the unique attribute's set method. Prior to setting the value we check for uniqueness, following the successful setting of the attribute the new value is added to the list of current unique values, and the old value is removed.

The implementation of uniqueness uses a static class level API for managing the list of existing attribute values, which is based entirely on the API for list attributes. This mechanism allows for uniqueness to be guaranteed without requiring that all instances of the class be instantiated. As such, the methods are purposely public such that developers can provide their own optimized approach to managing uniqueness. If such a mechanism were not in place, then our approach would not scale well.

The *lazy* keyword can be used to remove the attribute from the constructor, similar to immutable attributes presented in Section 4.5.2 .

Uniqueness can also be qualified along an association end with a multiplicity of one, an example of which is shown below.

```
unique Integer flightNumber on airline;
```

The implementation differs in the following respects. First, the static list tracking all unique values becomes an attribute of the qualified association. Second, the *set* method references attribute of the qualified association end not a static list. Third, a check is added to the *set* method that links the qualified association end, to ensure uniqueness if the attribute is set prior to linking the association end. The first two changes reflect small changes to the code above. The additional code required to implement the third change is shown below.

```
public boolean setAirline(Airline newAirline)
{
  boolean wasSet = false;
  if (newAirline != null && newAirline.indexOfAllFlightId(flightId) != -1)
  { return wasSet; }
  Airline oldAirline = airline;
  // existing association implementation code
  if (wasSet && oldAirline != null){ oldAirline.removeAllFlightId(flightId);}
  if (wasSet && newAirline != null){ newAirline.addAllFlightId(flightId); }
  return wasSet;
}
```

4.5.5 Autounique Attributes

The Umple language also supports autounique attributes as shown below.

```
class Student {
  autounique Integer id;
}
```

The implementation of autounique builds on the implementation of unique presented in the previous section. The difference is that the autounique attribute is automatically set in the constructor to the next available value. The implementation in Java is shown below (we filtered out the aspects that are similar to the unique implementation).

```
public class Student {
  private static int nextId = 1;

  private int id;
  public Student() {
    if (!setId(getNextId()))
    {
      throw new RuntimeException("Unable to create student due to id");
    }
  }
  public int getId() {
    return id;
  }

  public static boolean setNextId(int aNextId)
  {
    boolean wasSet = false;
    nextId = aNextId;
    wasSet = true;
    return wasSet;
  }

  public static int getNextId()
  {
    return nextId++;
  }

  // API implementation as available for unique attribtues

}
```

The autounique process can be manually configured (i.e. setNextId() and setId()) to support loading autounique objects from a persistence mechanism (i.e loading an object from a database – requires that the same persisted autounique value is assigned to that object instance).

4.5.6 Constant Class Attributes

A constant class level attribute is identified using the *const* keyword. The UML modeling standard is to underline; a convention that is difficult to achieve in a development environment as most developer code is written in *plain* text.

```
class Student {
  const Integer MAX_PER_GROUP = 10;
}
```

The underlying implementation in Java is shown below.

```
public class Student {
  public static final int MAX_PER_GROUP = 10;
}
```

4.6 Attributes with Multiplicity of Upper Bound Greater Than 1

Umple includes support for attributes that might contain multiple values (i.e. cardinality > 1). The Umple notation uses a square brackets [], which typically refers to arrays in programming languages like Java, PHP or Ruby. An example list attribute is shown below:

```
class Student {
  String[] nickname;
}
```

The underlying implementation is Java is shown below:

```
public class Student {
  private List<String> nicknames;

  public Student() {
    nicknames = new ArrayList<String>();
  }

  public boolean addNickname(String aNickname) {
    boolean didAdd = nicknames.add(aNickname);
    return didAdd;
  }

  public boolean removeNickname(String aNickname) {
    boolean didRemove = nicknames.remove(aNickname);
    return didRemove;
  }

  public String getNickname(int index) {
    String aNickname = nicknames.get(index);
    return aNickname;
  }

  public String[] getNicknames() {
    String[] newNicknames = nicknames.toArray(new String[nicknames.size()]);
    return newNicknames;
  }

  public int numberOfNicknames() {
    int number = nicknames.size();
    return number;
  }

  public int indexOfNickname(String aNickname) {
    int index = nicknames.indexOf(aNickname);
    return index;
  }

}
```

More complex cardinalities (i.e. 1 -- 3..5) can be achieved by using the association notation presented in the following chapter. To be consistent with the treatment of primitive types like an *int* or *double*, the generated code provides access to a copy of the entire list using primitive arrays as opposed to the unmodifiable lists that, we will see, are used by associations.

4.7 Summary

This chapter analyzed the syntax, semantics and pragmatics of attributes. We studied how attributes are used in practice, and discovered the difficulty in extracting modeling abstractions from analyzing source code. Our approach used manual inspection, which, although subject to human error, is comparable to analysis by automated tools since there are so many special cases to be considered.

We also demonstrated how attributes are represented in Umple and showed the code-generation patterns used when Umple is translated into Java. When compared to the code generated for attributes by existing tools, we believe our patterns have a great deal to offer. In the following chapter we analyze the impacts that supporting associations has on code generation.

Chapter 5 Syntax and Semantics of Textual Associations

UML classes involve three key elements: attributes, associations, and methods. But current implementations of object-oriented languages, like Java, do not provide a distinction between attributes and associations. The focus of this chapter will be to investigate the implications of providing separate syntax and corresponding semantics for associations, as distinct from attributes that were discussed in the last chapter.

Tools that generate code from associations currently provide little support for the rich semantics available to modellers such as enforcing multiplicity constraints or maintaining referential integrity. In this chapter, we show source code from existing code-generation tools and highlight how the issues above are not adequately addressed. We then outline code generation patterns currently available in Umple that resolve these difficulties and address the issues of multiplicity constraints and referential integrity.

Umple syntax allows the declaration of an association at the same *level* of abstraction as the declaration of a class. A developer no longer needs to declare instance variables in classes to implement associations. Except for reflexive associations, associations intrinsically involve more than one class. Unless we make judicious design choices, creating an explicit association abstraction may have consequences that negatively impact the object-oriented programming paradigm. Firstly, doing this would potentially increase coupling. The association would be coupled to the two associated classes, and the two classes might be coupled to the association if their methods needed to access the association in order to perform needed operations. Also, it has been a key tenet of OOP that a class looks after its own data; the association abstraction makes this potentially no longer the case. The challenge is to design an association abstraction at the programming level that adheres to the original spirit of OOP, and provides increased abstraction and better engineering capabilities.

Figure 26 shows how associations are represented textually and diagrammatically in Umple. On the right is a UML class diagram with three classes and two one-to-many associations. The code on the left in the equivalent in Umple. The '--' means that the association is bidirectional navigable (more on this later). It is also possible to use '->' or '<-' to indicate that navigation is possible in only one direction. The full set of UML multiplicity symbols may be used.

```
class Student
{}

class CourseSection
{}

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

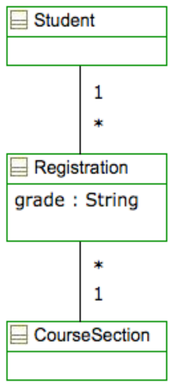


Figure 26: Umple class diagram for part of the student registration system

In addition to showing an association embedded in one of the two associated classes, it is also possible to show an association 'on its own', thus:

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

Other information such as role names can also be provided:

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

Besides simply providing improved abstraction, explicitly coding associations might have the following advantages: 1) Reduction in bugs, since the compiler can enforce various design constraints and less code would need to be written. The current implicit nature of associations in standard object-oriented languages results in bug-prone code since there is no general mechanism to enforce things like referential integrity. 2) Faster development due to the need to write less code and reduced numbers of bugs to fix.

In this thesis we demonstrate the effectiveness of the Umple approach to associations by showing the reduction in the amount of code needed by elimination of the 'boilerplate' code that would otherwise have to be coded. We also demonstrate that Umple associations can be used effectively to code applications. However, we leave the empirical study of bug reduction and development speedup to future work.

5.1 Related Work

Several studies [130-134] propose approaches to formalizing the semantics of associations. They generally agree on the interpretation of associations, but do not address uniqueness and ordering of associations.

Other studies in the literature refer to two types of associations, static and dynamic [135, 136]. Static associations, a view we adopt, represent structural relationships between classes, where the association is enforced throughout the lifetime of links between instances of those classes. *Dynamic* (or *contextual*) associations are only enforced during the interactions of the two objects. Miliev [9] proposes yet another view of associations: *intentional* associations that encapsulate the intention of association of each participating object. Instead of restricting associations from containing duplicate links, Miliev's approach modifies the way in which an association end maps objects from the other association end depending on the characteristics of the association ends. An example where assuming uniqueness fails to provide the necessary abstraction is shown below in Figure 27.

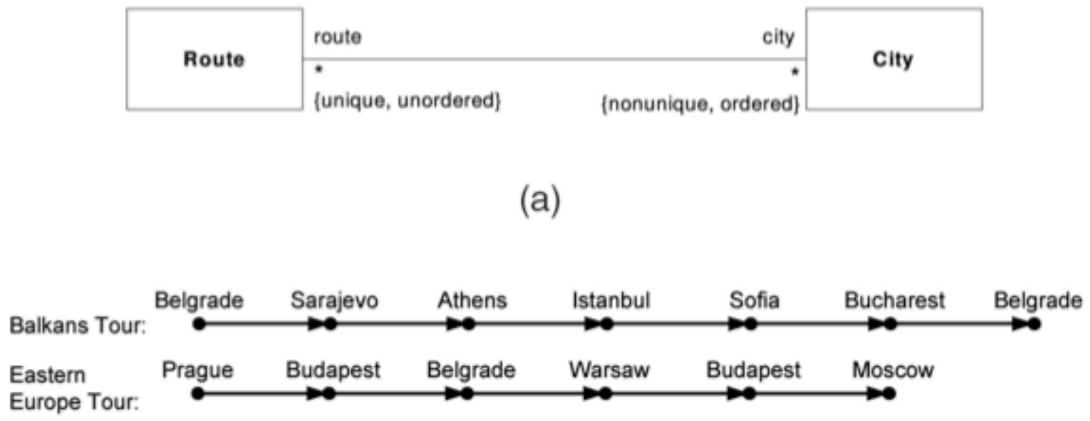


Figure 27: An example of nonunique association ends from [9]

In the example above, we see that a route can contain multiple links to the same object. Miliev uses the Z language to provide different association implementation depending on the properties of the association (i.e. ordered, unique, etc). Umple currently only supports ordered, unique set semantics to define association ends. To support additional qualities such as being unordered and / or nonunique, Umple would first need to introduce the appropriate syntax into the language in a usable way, and only then would we look at the implementation specifics. To determine the potential usefulness of qualities such as unordered, one could first look to expand on our study of attributes and associations presented in Section 4.3 and Section 5.2 to investigate the types of List structures used in practice.

Acknowledging deficiencies in automated code generation of UML associations and multiplicities, Wang and Shen [137] propose a run-time verification approach for UML association constraints. Østerbye [138] proposes supporting association referential integrity with a reusable class library that ensures the consistency of the relationship is maintained.

Executable UML (xUML) [91] was introduced by Steve Mellor; its aim is to provide a (yet to be approved) specification of an unambiguous subset of UML that can be executed using model compilers. The Umple compiler also behaves as a model compiler and provides a concrete implementation of a subset of UML. However unlike Executable UML, Umple integrates with standard object oriented languages, and supports a wider range of multiplicity, as well as a variety of other features not present in executable UML.

The Executable Foundational UML [59] is a computationally complete and compact subset of UML. Lazar et al. [139] provide an action language based on fUML with a concrete syntax inspired by OCL. This fUML language does not explicitly define associations as does Umple, but rather uses variables to define association ends as shown below for our Student/Mentor example.

```
def students : Student [0..*];
def mentor : Mentor [0..1];
```

The Alf language [61] (an RFP submission for OMG's upcoming UAL specification) presents another computationally complete language aimed at providing a textual modeling notation. Alf defines associations using a similar external notation to Umple. An example association in Alf representing our Student/Mentor example is shown below.

```
assoc LifeCoach {
  public student:  Student[0..*];
  public mentor:   Mentor[0..1];
}
```

Alf supports higher order associations, the implementations of which would be considerably different than those available in Umple; which only supports binary relationships.

The Umple approach to implementing a UML action language is distinct from the official OMG approach in three aspects. First, Umple makes a textual representation for other UML modeling elements available and integrates the textual action language with the textual diagram representation. Modelers can create and edit models diagrammatically or textually, and can embed the action language textually.

Second, Umple's bottom-up approach attempts to raise the abstraction level of widely-adopted programming languages to include modeling abstractions and action semantics, effectively overcoming limitations associated with using a programming language as a modeling action language. Such an approach allows us to continuously use UML and the action language in building real systems of considerable complexity.

We raise the abstraction level of base programming languages by iteratively executing the following language refinements (LRs):

- LR-1. Make available additional, and more abstract, language constructs based on necessity, experience, and experimentation
- LR-2. Restrict and modify statements to decouple the action language from the underlying target system
- LR-3. Discover new language constructs for inclusion in our action language.

A third upcoming distinction between Umple and Alf is Umple's native support not only for class diagrams but also of state machine abstractions; the current Alf approach only addresses the former.

5.2 Associations In Practice

In the previous chapter we discussed an empirical study of existing open source software to analyze how attributes are used in practice. This study was extended to also include the analysis of associations. Recall that we identified candidate attributes based on certain criteria (e.g. attributes are simple types that also have a get method, or public accessibility). The process applies almost identically for associations, except for the final step (see Table 15 for a review of the filters used to distinguish simple from complex variables).

The process of seeking associations in our subject systems resulted in 350 candidate association ends. During our manual review of candidate variables, we further filtered the set down to 235 candidate association ends by removing any internal variables that were neither set in the constructor, nor available via a set/get method. These variables appeared to represent dependencies and not association ends (e.g. Readers, Streams, and Maps).

Below, in Table 22, we highlight some distribution statistics of the 235 candidate association ends. Please note that the categories are not mutually exclusive so the column sum will not be 100%.

In addition to tracking the distribution of set and get methods, the following observations were made:

- For associations with an upper bound greater than one, some implementations provided direct access to the list structure and others provided list accessor methods like add and remove.
- Of the 235 association ends, 42 (17.9%) were defined using collections (Map, Set, Hash, List) and hence most likely represented associations with an upper bound greater than one

Table 22. Distribution of set/get methods and availability in constructor.

Category	Frequency	%	Description (if required)
Set/Get Methods	67	29%	All variables that had both a set and get method.
Set Method	89	38%	All objects that at least had a set method.
Get Method	120	51%	All variables that at least had a get method.
No Set Method	54	23%	All immutable variables links as the variables have no set method.
Only Get Method	39	17%	Internally managed variable links (no set method and not available in the constructor).

Note that public instance variables are considered to have set and get methods as discussed in the previous chapter.

When analyzing the open source systems it was difficult to match association-end variables to one another. This was because many associations linked to external resources (and most likely represented one-way associations). There was little evidence of referential integrity between association-ends, implying that the application developer *using* the object model would have to maintain the correct multiplicities himself or herself. This difficulty in analyzing how associations are used in practice provides some motivation for our work, as greater traceability can be achieved using a model-oriented language like Umple. This is because developers can explicitly define associations in one location and the association can be accessed and modified in a consistent manner.

5.3 Associations in UML Diagrams

To get a better understanding of the types of associations used in practice, we set out to analyze a significant set of publically available UML diagrams. By analyzing such diagrams we should be able to get a better understanding of how associations are used in practice. We analyzed 1536 associations (not just association ends as in our analysis above) based on two UML specifications (v1.5 and v2.1.2) and seven UML profiles (MARTE, Flow Composition, ECA, Java, Patterns, rCOS). A summary of the multiplicities in use from these sources is summarized in Table 26.

For comparison, we also analyzed example UML models found in "Object-Oriented Software Engineering: Practical Software Development using UML and Java" by Lethbridge and Laganière [2], as well as those in our own repository of UML modeled systems available at [3]. This repository represents systems modeled and built using Umple. These systems were intended to demonstrate real designs that could be the basis for actual systems.

Table 23: Usage of Association Multiplicities in UML

Occurrences in the UML specs			Rank in the three sets		
Multiplicity	**Frequency**	**Percent**	**In UML Specs**	**Examples in Book**	**In Umple Repository**
0..1--*	274	19.1%	1	4	2
1--*	273	19.0%	2	1	1
->	190	13.2%	3	9	5
--	158	11.0%	4	2	3
0..1--1	129	9.0%	5	N/A	7
1<-*	86	6.0%	6	N/A	6
0..1<-*	75	5.2%	7	N/A	4
0..1--0..1	56	3.9%	8	6	N/A
1..*--*	55	3.8%	9	N/A	N/A
Other	142	9.9%	N/A	N/A	N/A
Total	1438	100.0%			

Table 24: Example Usage of Association Multiplicities in the Book by Lethbridge [2]

Occurrences in the book			Rank in the three sets (for comparison)		
Multiplicity	**Frequency**	**Percent**	**In UML Specs**	**Examples in Book**	**In Umple Repository**
1--*	39	39.8%	2	1	1
--	15	15.3%	4	2	3
1--1	13	13.3%	N/A	3	N/A
0..1--*	11	11.2%	1	4	2
1 <- *	4	4.1%	N/A	5	6
0..1--0..1	4	4.1%	8	6	N/A
0..n--1	3	3.1%	N/A	7	N/A
n--*	2	2.0%	N/A	8	N/A
->	2	2.0%	3	9	5
Other	5	5.1%	N/A	N/A	N/A
Total	98	100.0%			

The top nine types of associations, where the categories distinguish associations both by multiplicities at each end and by navigability, are shown in Table 23. The rank of actual usage (where rank 1 is the most frequently-encountered multiplicity category) is given for the three sets of class diagrams we analyzed, as described above. Note that in this and the subsequent two tables the order of ends is irrelevant; so for example 0..1--* is the same as *--0..1. Table 24, organizes the data based on example usage from [2].

Table 25, organizes the data based on our repository of UML models. These models can be viewed online at [3]. The repository includes 28 unique models including elevators, airlines, police procedures, warehouse inventory, and even the Umple meta model itself.

Table 25: Usage of Association Multiplicities from Model Repository [3]

Occurrences in the Umple repository			**Rank in the three sets (for comparison)**		
Multiplicity	**Frequency**	**Percent**	**In UML Specs**	**Examples in Book**	**In Umple Repository**
1--*	108	43.4%	2	1	1
0..1--*	34	13.7%	1	4	2
--	27	10.8%	4	2	3
0..1<-*	23	9.3%	7	N/A	4
->	22	8.8%	3	9	5
1<-*	12	4.8%	6	5	6
0..1--1	4	1.6%	5	N/A	7
1--1..*	3	1.2%	N/A	N/A	8
--1..	3	1.2%	9	N/A	9
Other	15	6.0%	N/A	N/A	N/A
Total	249	100.0%			

Table 26: Summary of Usage of Association Multiplicities

Occurrences in all three sets, with directionality considered			**Occurrences in all three sets, ignoring directionality**		
Multiplicity	**Frequency**	**Percent**	**Multiplicity**	**Frequency**	**Percent**
1--*	420	23.5%	1--*	522	29.2%
0..1--*	319	17.9%	0..1--*	418	23.4%
->	214	12.0%	*--*	414	23.2%
--	200	11.2%	0..1--1	134	7.5%
0..1--1	134	7.5%	1..*--*	93	5.2%
1->*	102	5.7%	0..1--0..1	62	3.5%
0..1->*	99	5.5%	1--1	42	2.4%
0..1--0..1	62	3.5%	1--1..*	27	1.5%
1..*--*	59	3.3%	0..1--1..*	23	1.3%
Other	176	9.9%	Other	50	2.8%
Total	1785	100.0%	Total	1785	100.0%

Overall the UML specification models and the models in the book have five of the top nine multiplicity categories in common. A summary of multiplicity usage for all three sources (UML specs, book examples and our Umple repository) is shown in Table 26. In the summary on the left, we consider the directionality of the association; on the right, we ignore the directionality.

The most frequent multiplicity categories if you consider directionality are one-to-many, optional-one-to-many, many-directed-to-many, many-to-many and optional-one-to-many. If you combine directed and bi-directional associations then the top categories include: one-to-many, optional-one-to-many, many-to-many, optional-one-to-one, and mandatory-many-to-many.

After analyzing over 1500 different modeled associations, it is clear that approaches like xUML [91] (whereby only a subset of the UML multiplicities can be modeled) provide reasonable coverage for most applications. The same is true of the code generators to be analyzed in Section 5.6 ; these provide little support for association multiplicities beyond differentiating *one-ends* from *many-ends*.

But, as shown above, 11% of the UML specifications fall outside of the simple cases currently supported (3% if you ignore directionality) and despite the additional complexity of supporting all multiplicity types; it should be of both academic and practical relevance to explore all types of association relationships.

As an aside, only unary and binary associations were observed in practice; our work on modeling associations in Umple will be based on that same constraint.

In the next section we analyze association multiplicities from first principles in order to identify all possible multiplicity combinations, and to understand how to model and generate working systems from them.

5.4 Analyzing All Possible Multiplicity Combinations

Let us consider an association between a Mentor and a Student.

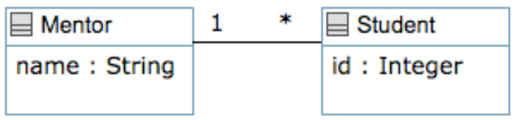


Figure 28: An example binary association

An association has a multiplicity at each end that describes how many instances of one class can be linked with the other class. In Figure 28, a Mentor can link to any number of Students, but a Student must be assigned to one and only one Mentor. Multiplicities play an important role in the implementation of an association, since implementation will vary depending on the combinations of multiplicity ends.

To understand the implications of explicitly describing associations in a programming language, we must first know the types of associations, and in particular the type of multiplicities that can be bound to an association. The nine multiplicity types that can be bound to each end of an association are described below in Table 27. In this table, n and m stand for arbitrary integers greater than one.

Table 27. Multiplicity Possibilities for Associations (Shorthand in Parentheses).

Multiplicity Notation	Lower Bound	Upper Bound	Description
0..1	0	1	Optional-One – Item is either present or not.
0..n	0	n > 1	At Most n – At most n items, or none at all.
0..* (*)	0	undefined > 1	Many – Any number of items can be present, or none at all.
1..1 (1)	1	1	One – The item is mandatory.
1..n	1	n > 1	Mandatory At Most n – At least one item is mandatory up to a maximum of n items.
1..*	1	undefined > 1	Mandatory Many – At least one item is mandatory with no maximum.
n..n (n)	n > 1	n	Exactly n – Exactly n items are mandatory.
m..n	m > 1	n > m	From m to n – At least m items are mandatory up to a maximum of n items.
m..*	m > 1	undefined > m	At least m – At least m items mandatory with no maximum.

Each multiplicity end requires a much different treatment in the code, except that the 1..n case can be coded as a special case of m..n (both are mandatory with a fixed upper bound) and the 1..* case can be coded as special case of m..* (both are mandatory without a fixed upper bound).

5.4.1 Bidirectional Associations Between Two Different Classes

We will first analyze associations between two distinct classes that are navigable in both directions; in other words where both linked objects are conceptually ‘aware’ of the relationship. In total there are 28 different patterns of such binary associations possible, as listed in Table 28. Each of these requires slightly different code for implementation.

Note that x -- y is equivalent to y -- x, so for example, 0..1 -- n is equivalent to n -- 0..1; for this reason, the upper diagonal in Table 2 is left blank. Also note that the variables ***m*** and ***n*** are not assumed to be the same on both ends of the association. For example, 0..6 -- 3..4 association would fall under the 0..n -- m..n category.

Table 28. The 28 Possible Bi-Directional Non-Reflexive Associations

0..1	0..n	*	1	n	m..n	m…*
0..1 -- 0..1						
0..1 -- 0..n	0..n – 0..n					
0..1 -- *	0..n -- *	* -- *				
0..1 -- 1	0..n -- 1	* -- 1	1 -- 1			
0..1 -- n	0..n -- n	* -- n	1 -- n	n -- n		
0..1 -- m..n	0..n -- m..n	* -- m..n	1 -- m..n	n -- m..n	m..n -- m..n	
0..1 -- m..*	0..n -- m..*	* -- m..*	1 -- m..*	n -- m..*	m..n -- m..*	m..* -- m..*

Shaded cells show cases where both sides are ‘mandatory’. Bold associations with thick borders indicate the common cases as observed in the previous section. Note that 1..* is a subset of the more generic m..* and 1..n is a subset of m..n.

Several of our examples below will use an association between a Mentor and a Student (e.g. Figure 28 above). We use this example as it is logical to vary the multiplicity and still have a system that makes sense. Figure 28 shows the case where a Student always has one Mentor, but a Mentor can have zero or more Students.

In subsequent sections, our implementation will be presented in Java, and we will assume Mentor has a member variable ***student*** when the upper bound of the multiplicity is 1, and ***students*** if the upper bound is greater than 1. The same applies for Student, with a ***mentor*** member variable for a multiplicity of one, and ***mentors*** for greater than one. In Umple the notation for Figure 28 is:

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

5.4.2 Unidirectional (Directed) Associations

A directed association is navigable in one direction only. Only one object of the pair is aware of and can manage the relationship. For example, one could write in Umple:

```
association {* Mentor -> * Student;}
```

In the model above, Mentor is aware of the associated Students, but a Student is unaware of any Mentor’s to which he/she might be associated.

Because one end is unaware that it is part of a link, the unaware object can be unknowingly linked to multiple objects (i.e. a * relationship is generally implied); resulting in seven possible combinations for code generation, `* -> 0..1`, `* -> 1`, `* -> *`, `* -> m..n`, `* -> n`, `* -> m..*` and `* -> 0..n`.

Without injecting additional complex code, the system will not be able to manage the association when changes occur to the unaware side, such as Student deleting him/herself. If such situations must be managed, then a bi-directional association should be used. In practice we find that the vast majority of associations would benefit from being bidirectional. Doing so enables functionality that tends to be required anyway, and where the functionality is not immediately required, the code is better suited to meet unanticipated future needs.

It should be pointed out that an advantage of using a tool like Umple is that a change from directed to undirected associations could be accomplished almost instantly at a later stage of development if necessary. When associations are written manually, the work involved in this change can be time consuming and error prone.

5.4.3 Reflexivity and Symmetry

Where both ends of an association are the same class, we must consider several special cases. A truly reflexive association allows an object to be linked to other objects of the same class *including itself*. An association called lives-at-same-address would be reflexive.

It is also common to have *irreflexive* associations with both ends being the same class. The added constraint is that a given object cannot be linked to itself. For example, a *mother* relationship is irreflexive as you cannot be your own mother. It should be noted that some literature call all associations from a class to the same class 'reflexive'.

A *symmetric* association describes a mapping that reads the same as its inverse, for example, a *spouse* association. An asymmetric association is not reversible, for example, a child relationship. Finally, an anti-symmetric association is asymmetric except that is allows a relationship to self. For example the relationship *being-present-at-birth* is anti-symmetric.

One could encode the mentor-student example using a single class, where all objects are Persons, and some persons can mentor others. This would be an irreflexive asymmetric association, since one cannot mentor oneself and the meaning of the association would be different in each direction. The Umple notation would be: `association {* Person mentor -- 0..1 Person student;}`.

The Umple language natively supports symmetric and asymmetric associations. Anti-symmetric associations are currently not supported as we find them to be quite rare (they may be supported in the future). The distinction between reflexivity and irreflexivity is currently not explicitly managed, but applications can relatively easily be coded to prevent (or allow) an object to be linked to itself.

Asymmetric associations require code that is effectively identical to associations between two different classes - as shown in Table 28 - except that the lower bound of both ends must be zero. The top seven associations from Table 28 are therefore possible as asymmetric associations from one class to itself. The reason the lower bound must be zero is to prevent infinite regress. For example, in the case of class Person in the asymmetric association above, if every mentor Person *had* to have a student Person, and since every student is also a mentor (by virtue of being a Person), there would be an infinite chain of persons requiring a mentor.

A symmetric association specifies links between different instances of the same class, and must have the same multiplicity on each end. The diagonal of Table 28 gives the cases to consider.

As a result of all the above analysis, a total of 42 different possible association types have been identified (28 for ordinary binary associations, 7 for unidirectional associations, and 7 for symmetric associations). In the following section we highlight certain implications that these association types will have on code generation. This overview will serve as a guide when comparing existing code generation tools and also as a template for building our own code generation for systems programmed in the Umple modeling language.

5.5 Implications for Code Generation

The implementation of associations in a language like Java impacts the following aspects of a class.

First, the class will have an additional member variable to reflect the other end of an association. One and optional-one multiplicities can be declared as instance members of the other type, while many multiplicities can be declared as lists (or some other type of collection class) containing objects of the other type.

Second, the constructor may need an additional parameter to ensure mandatory association ends like 1 or 1..*.

Finally, the class requires methods to set, get, add and remove links between objects. To be consistent with the model of the associations, the implementation of those methods should ideally maintain the referential integrity of all linked pairs of objects, as well as ensure that multiplicity constraints are upheld.

In the following section, we analyze how existing code generators deal with the various combinations of multiplicities and to what extent they behave according to the structure outlined above. We then discuss the code generation available from the Umple language.

5.6 Existing Code Generators

In the previous section we provided an overview of all the possible binary associations. We briefly outlined the concerns that code generators should keep in mind when generating code for associations. We will now look at existing tools to see how well they translate the semantics of associations into a programming language.

The UML modeling tools considered are the same as those identified in the previous chapter in Table 20. Each tool was configured to generate Java code for a simple 1 -- * relationship shown in Figure 28.

This simple relationship was used to illustrate the tools' source code transformation templates for a *one* versus a *many* multiplicity end. The generated code provided in the following sections has been modified only to provide a consistent layout/format, and comments have also been removed to save space.

5.6.1 Code Generation Patterns

In general, all tools analyzed provide two basic code generation templates, one for 0..1 and 1 (referred to as *one*) multiplicities and a second for m..n multiplicities (referred to as *many* where $m \geq 0$, $n > m$, $n > 1$).

The template pattern for *one* multiplicities would generate a member variable to refer to the other association end, as shown below.

```
private <ClassName> <associationEndName>;
```

The template pattern for *many* would generate a reference to a List or Set structure that could contain multiple references to the other association end, as shown below.

```
private <ListStructure> <associationEndName>;
```

Some tools provide explicit code generation patterns for *n* relationships (where $n > 1$), as well as m..* relationships (where $m \geq 0$). Some tools provided explicit get and set methods in addition to creating the necessary member variables. A discussion of each code generation pattern will be provided based on the tools analyzed.

5.6.2 ArgoUML

ArgoUML is an open source modeling platform that provides code generation for Java, C++, C#, PHP4 and PHP5 (see for Table 20 for the URL). Below is the generated code for Mentor and Student as shown in Figure 28.

```
import java.util.Vector;
```

```
public class Mentor {
  public Integer id;
  public Vector myStudent;
}

public class Student {
  public String name;
  public Mentor myMentor;
}
```

The generated code provides a mechanism to access each end of the relationship. The generator provides little validation or constraint checking to ensure the relationship is maintained, and the attributes are made directly available without the inclusion of accessor (get and set) methods.

In general, all 0..1 and 1 multiplicity ends generate similar structures as seen in the Student class above, and all m..n multiplicity ends (where $m \geq 0$ and $n > m$ and $n > 1$) generate similar structures to the Mentor class.

5.6.3 StarUML

StarUML is an open-source modeling tool. Below is the generated code for the Mentor and Student example:

```
public class Mentor {
    public String name;
    public Student student;
}

public class Student {
    public Integer id;
    public Mentor mentor;
}
```

StarUML's generated code does not account for the *many* multiplicity; the code is either un-compliable or unusable without modification.

5.6.4 Bouml

Bouml is a free tool based on UML 2 that provides source code generation for C++, Java, Idl, PHP and Python. Below is the generated code of Figure 28.

```
class Mentor {
  private List<Student> student;
  private String name;
}

class Student {
  private Mentor mentor;
  private int id;
}
```

The source code generated above is very similar to that of ArgoUML. The Bouml source code does not provide any mechanism to test, or ensure the constraints outlined in the model; this code must be written by hand after code generation. In addition, the source code is incomplete as no reference to the *java.util.List* class is provided; this means that the generated code must be maintained by hand to ensure proper compilation into byte code.

5.6.5 Green Code Generator

Green UML is another UML editor that can generate source code from a class diagram. Below is the generated code for the Mentor and Student example.

```
import java.util.List;
public class Mentor {
  private List<Student> student;
  java.lang.String name;
  public Mentor(List<Student> student2) {
    student = student2;
  }
}

public class Student {
  private Mentor mentor;
  int id;
  public Student(Mentor mentor2) {
    mentor = mentor2;
  }
}
```

Green does provide some additional code generation support by creating custom constructors based on the association. Green supports the following types of multiplicities: 1, n, m..*, and * (where n > 1 and m >= 0).

Green provides some enforcement of constraints, although the implementation is awkward and not scalable. Below is the implementation of the mandatory relationship where a Mentor must have *n* Students (e.g. n = 3).

```
public class Mentor {
  private Student student3;
  private Student student2;
  private Student student;
  java.lang.String name;

  public Mentor(Student student4,
              Student student5, Student student6) {
    student3 = student4;
    student2 = student5;
    student = student6;
  }
}
```

This implementation provides little opportunity to access or manage the collection of students; instead each must be accessed explicitly by name. It also does a poor job of maintaining the constraint; as the variables could be set to *null*, violating the model's intention.

Finally, Green provides some additional code to manage m..* relationships. Below is an example implementation of a 2..* relationship.

```
import java.util.List;

public class Mentor {
  private List<Student> student;
  java.lang.String name;

  public Mentor(List<Student> student2) {
    student = student2;
    student.add(new Student());
    student.add(new Student());
  }
}
```

The implementation above presents two issues. First, the potentially unwanted side effect of creating and inserting additional entities into the list argument (i.e. students). Second, the code generator assumes that a default (and empty) constructor exists for the Student object, an assumption that might not always be valid and could result in a generated system that does not compile.

Although Green UML does attempt to provide some additional source code generation to manage the various types of association multiplicities available; the results provide little, if any, added benefit in representing the model's intentions.

5.6.6 Rational Software Architect (RSA) and RSA Real-Time

IBM's Rational Software Architect (RSA) and RSA Real-Time are full-fledged development environments that support model-driven development including source code generation from UML diagrams. The following is the code generated by these tools:

```
import java.util.Set;
public class Mentor {
  // code to declare+set+get attribute id cut to save space

  public Set<Student> students;
  public Set<Student> getStudents() {
    return students;
  }

  public void setStudents(Set<Student> students) {
    this.students = students;
  }

}
```

```
public class Student {
  // code to declare+set+get attribute id cut to save space

  public Mentor mentor;
  public Mentor getMentor() {
    return mentor;
  }

  public void setMentor(Mentor mentor) {
    this.mentor = mentor;
  }
}
```

RSA's model transformation into Java code provided some flexibility regarding the template patterns including (a) which Java collection to use, and (b) whether or not to include get and set methods for the attributes and association ends. As with all other source code generators, no distinction between the various possible one or many relationships are presented in the generated code; leaving the implementation of the modeling constraints up to manually-written code. In addition to providing simple set and get methods, RSA's member variables representing the association ends are also public, presenting an encapsulation issue. This is an important oversight considering the code already provides set and get methods.

5.6.7 Eclipse Modeling Framework (EMF)

EMF is described in detail in Section 3.8.1 . As shown below, classes are generated as interfaces.

```
import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.ecore.EObject;

public interface Mentor extends EObject
{
  EList getStudents();
}

public interface Student extends EObject
{
  Mentor getMentor();
  void setMentor(Mentor value);
}
```

The implementation of each class above is shown below. For clarify, we do not include the generic accessor methods (eSet, eGet, eIsSet, eUnset) as described in Section 4.4.6 .

```
public class MentorImpl extends EObjectImpl implements Mentor
{
  protected EList<Student> students;
  protected MentorImpl() { super(); }

  public EList<Student> getStudents() {
    if (students == null) {
      students = new EObjectResolvingEList<Student>(Student.class, this,
                        AssociationPackage.MENTOR__STUDENTS);
    }
    return students;
  }
}

public class StudentImpl extends EObjectImpl implements Student
{
  protected Mentor mentor;
  protected StudentImpl() { super(); }

  public Mentor getMentor() {
    if (mentor != null && mentor.eIsProxy()) {
      InternalEObject oldMentor = (InternalEObject)mentor;
      mentor = (Mentor)eResolveProxy(oldMentor);
      if (mentor != oldMentor) {
        if (eNotificationRequired())
          eNotify(new ENotificationImpl(this, Notification.RESOLVE,
                   AssociationPackage.STUDENT__MENTOR, oldMentor, mentor));
      }
    }
    return mentor;
  }

  public Mentor basicGetMentor() { return mentor; }

  public void setMentor(Mentor newMentor) {
    Mentor oldMentor = mentor;
    mentor = newMentor;
    if (eNotificationRequired())
      eNotify(new ENotificationImpl(this, Notification.SET,
                   AssociationPackage.STUDENT__MENTOR, oldMentor, mentor));
  }
}
```

The EMF implementation of the Mentor and Student class with respect to associations has a similar implementation as for attributes discussed in Section 4.4.6 , with the following differences:

- A more complex *get* method that supports a proxy mechanism for retrieving objects
- The *get* method also includes the Observer pattern to notify others of a change to the object when accessed via the proxy.
- A second *get* method is available (basicGetMentor) that circumvents the proxy check.

Similar to attributes, EMF consolidates multiplicities into two categories 'one', or 'many'. No additional constraint checking or validation is available to ensure the modeled multiplicities are enforced.

There is no doubt that the code generated from EMF is more complex than that of the other code generators analyzed. Upon closer analysis, the primary benefit of the EMF approach is the availability of the Observer pattern to watch for changes to associations. The use of an interface / implementation approach for code generation could also be seen as a benefit (although one might argue that it adds unnecessary complexity – and if classes are implemented as interfaces, what are interfaces implemented as?). Finally, EMF, much like all other code generators analyzed above, provides no referential integrity or multiplicity constraint checking on modeled attributes or associations.

5.7 Generating Association Code using Umple

The existing UML code generation tools analyzed in the previous section fall short of providing robust code to implement associations. The generated code provides little implementation support either to manage referential integrity or to ensure multiplicity constraints (beyond the ’one’ vs. ‘many’ distinction).

Tilevich investigated several approaches to implementing associations as classes in the Standard Template Libary (STL) [140] (these should not be confused with UML association classes). Part of his analysis considers the problem of representing associations, and in particular the use of non-pair-associative containers. An example is shown below.

```
class Student {
  Mentor mentor;
}

class Mentor {
  List<Student> students;
}
```

For simplicity, the associations are implemented as public members. In the example above, the containers (i.e. variables) share no relationship between one another (i.e. non-pair). This non-pairing relationship implies that the relationship between both classes must be explicitly maintained. In concrete terms, a system must set both ends of the link; setting the mentor to the student and then adding the student to the mentor. The same is true when removing the link between the objects.

The drawbacks of using a conventional non-pair-associative container within the class are (a) it is hard to provide a consistent API for adding, deleting and searching the container, and (b) understanding the associations within the code is difficult as there is little way to determine one-to-many versus many-to-many relationships without analyzing both association ends as well as their interactions between one another. Next, [140] prescribes the use of special-purpose classes for each association, so that associations are externalized and kept together. An example from [140] is shown below.

```
OneMany<Department*, Project*> ProjsByDept;
```

Instead of using a class to externalize associations, Umple considers associations as first-class entities with their own distinct syntax as shown below:

```
1 Department -- * Project.
```

The syntax above can be defined directly within a class for convenience, or externally using the association construct (please refer to Section 3.5.2).

In this section, we provide our approach to code generation and identify implementation patterns that go beyond the capabilities of current tools and add support for referential integrity and managing multiplicity constraints. The code generated based on the association above does in fact use non-pair-associative containers; but because this code is generated and not manually maintained the limitations as identified by Tilevich do not apply.

5.7.1 Defining Association Variables

The first pattern to emerge is the distinction between having a multiplicity upper bound equal to one versus greater than one. For convenience, we will use UB for upper bound and LB for lower bound.

Table 29: Member Variable Patterns

Multiplicity Constraint	**Pattern**	**Example**
UB = 1	ObjectType associationEnd;	Student student;
UB > 1	List<ObjectType> associationEnd;	List<Student> students;

5.7.2 Association Ends as Constructor Parameters

The next patterns relate to a class's constructor signature. The constructor defines how objects should be created and indirectly affects the order in which objects can be instantiated. Three signatures emerge from the various multiplicities:

a. The association end is not required (LB=0) and is not part of the constructor

b. Exactly one, the upper and lower bounds are exactly one

c. Mandatory Many, (LB > 0 and UB > LB)

The patterns in Table 30 work well when the multiplicity of at least one association end is zero, allowing the creation of one object before the other.

Table 30: Constructor Argument Patterns

Multiplicity Constraint	Pattern	Example
LB = 0	Empty	(no argument)
LB=UB=1	ObjectType anAssociationEnd	Student aStudent;
LB > 0 && UB > 1	List<Student> someAssociationEnds	List<Student> allStudnets;

Below is an example constructor where a Mentor has exactly one Student.

```
public Mentor(Student aStudent)
{
  if (!setStudent(aStudent))
  {
    throw new RuntimeException("***");
  }
}
```

By using the *setStudent* method (which we will discuss in the interface patterns section), we are able to encapsulate *how* students are set, including the verification that the set operation is indeed valid (i.e. association multiplicity constraints are not violated). If we are unable to assign the student, then an exception is thrown. The exact error message is not shown for simplicity.

When the upper bound is greater than one (and the lower bound is not zero), we must initialize a list of associated members. We can use a similar mechanism as shown for the lower and upper bound of one, using the *setStudents* (instead of setStudent) method to delegate the action and verification of assigning the objects.

```
public Mentor(List<Student> allStudents)
{
  students = new ArrayList<Student>();
  boolean didAddStudents = setStudents(allStudents);
  if (!didAddStudents)
  {
    throw new RuntimeException("***");
  }
}
```

A chicken-and-egg issue is manifested when neither end of an association has a lower bound of zero; meaning that each end requires the other in the constructor, resulting in deadlock as neither constructor can be called before the other.

We will consider the one-to-one mandatory relationship as first shown in Figure 18. Here each Mentor must have exactly one Student, and vice versa.

```
public Mentor(Student aStudent)
{
  __initialized = false;
  __deleted = false;
  student = aStudent;
  if (student != null)
  {
    if (!student.setMentor(this))
    {
      throw new RuntimeException("***");
    }
  }
  __initialized = true;
}
```

The constructor for Student would be similar. Due to the difficulty in creating *two* instances simultaneously, the system temporarily allows the one-to-one constraint to be violated. But, as shown below, the object instances are unusable until the relationship is satisfied due to constraint checking. This is achieved by injecting code into all methods to verify the relationship (as shown below).

```
public Student getStudent()
{
  if (!isMandatoryStudentSatisfied()) { throw new RuntimeException("***") }
  return student;
}

public boolean isMandatoryStudentSatisfied()
{
  return __initalized && !__deleted && student == null;
}
```

Here is how a mandatory relationship must be created using the API generated by Umple.

```
Mentor m = new Mentor(null);
Student s = new Student(m);
```

Our original code generation approach attempted to create both objects simultaneously with the use of a contructor that includes the parameters from both ends of the relationship (i.e. all parameters for Mentor and for Student). Although the original approach worked well for simple cases, it broke down when we considered more complex examples such as chains of mandatory relationships, as shown below.

```
class Student {  }
class Mentor { 1 -- 1 Student; }
class Supervisor { 1 -- 1 Mentor; }
```

Our current approach supports the relationships defined above.

```
Student student = new Student(null);
Supervisor supervisor = new Supervisor(null);
Mentor mentor = new Mentor(student,supervisor);
```

The approach also works for mandatory many relationships (LB > 0 and UB > LB). The only difference is that the constructors now take a list of objects instead of a single object, and the constraint verification considers not only the presence of a object, but also the number of objects. As modelers, we are highly suspect of such *chained* mandatory relationships in general, we feel that our support for such seemingly obscure cases does not detract from the quality of the probably more common 1 -- 1 relationship as originally discussed.

5.7.3 Get Method Code Generation Patterns

Next, we will consider the interface to access an association end. Table 31 is an outline of the interface available to a Mentor based on the multiplicity at the Student end. The pattern in the middle column is based on a generic association end name (name), and the association end's type (type).

Table 31: Method Signature Patterns for Accessor Get Methods

Multiplicity Constraint	**Pattern**	**Example**
UB = 1	getName() : Type	getStudent() : Student
UB > 1	getName(int index) : Type	getStudent(int index) : Student
	getNames() : List<Type>	getStudents() : List<Student>
	indexOfName(Type aName) : int	indexOfStudent(Student aStudent) : int
	numberOfNames() : int	numberOfStudents() : int
	hasNames() : boolean	hasStudents() : boolean

The *getStudent* implementation is shown below.

```
public Student getStudent() {
  return student;
}
```

The difference between mandatory one (1) and optional one (0..1) is that the student member might be null in the optional case; whereas the 1 multiplicity will never allow null.

When the upper bound multiplicity is greater than 1, there are four common accessor methods as shown below.

```
public Student getStudent(int index) {
  return students.get(index);
}
public List<Student> getStudents() {
  return Collections.unmodifiableList(students);
}
public int numberOfStudents() {
  return students.size();
}
public boolean hasStudents() {
  return students.size() > 0;
}
public int indexOfStudent(Student aStudent) {
  return students.indexOf(aStudent);
}
```

Although you have access to all associated students, you are not able to alter the association by manipulating a list retrieved using the *get* methods shown above. To change the number of elements you must use the available *add* methods as shown below. The reason for this is to prevent the caller of API methods from being able to violate the multiplicity constraints or corrupt the referential integrity. Many implementations of the 'many' ends of associations simply pass the collection of objects around; however, we explicitly ensure that this never happens. For example, in the Java generated code shown above we use an unmodifiable list, whereas in the PHP generated code we send a copy of the list structure.

5.7.4 Set Method Code Generation Patterns

We now consider the manipulation interface to add, remove and set links of an association end. Consistent with our previous examples, we will be adding Student instances to a Mentor object based on various multiplicity constraints. Table 32 describes the generated interface.

Table 32: Method Signature Patterns for Set Methods

Multiplicity Constraint	Pattern	Example
UB = 1	setName(Type aName)	setStudent(Student aStudent)

UB > 1	addName(Type aName)
	removeName(type aName)
		addStudent(Student aStudent)
		removeStudent(Student aStudent)

The implementation of set methods is considerably more complex than get methods. First, set methods must undo any existing links between objects and establish the new links. Second, the methods must ensure referential integrity; when creating one end of a binary association they must create the other end as well. In addition, the API itself could be expanded to include inserting and removing instances at a particular offset as is available with getMethod semantics. The need for such enhancements might become apparent based on additional analysis of systems written and refactored into Umple; but to date such a facility has been necessary.

Let us begin with the case where the upper bound is one. When the relationship is optional, the following scenarios must be considered.

If adding a new link, be certain to set the inverse link as well. Conversely, if the inverse link has already been set, then do not set it again. For example, if adding a Student to a Mentor, be sure to add a Mentor to the Student (but ensure this call does not result in infinite mutual recursion).

If replacing or removing an existing link, be careful to remove both directions of the link. For example, if a Mentor can only have one Student, then when assigning a Mentor to a new Student, be sure to unassign that Mentor from the existing Student.

Whenever creating a new link, mandate that the multiplicity constraints on both ends are satisfied. If a Mentor can only have four Students, then do not allow a Student to add a Mentor such that the Mentor would now be linked to five Students.

Finally, whenever removing an existing link, ensure that the multiplicity constraints on the existing objects are satisfied. If a Mentor must have at least two Students, then do not allow a Student to set itself to a new Mentor if the existing mentor is at its two-Student minimum.

We will outline two examples (one where UB = 1, and the other where UB > 1).

Below is the implementation of *setMentor* as defined in the Student class as part of the 0..1 Mentor -- 0..1 Student association.

```
  public void setMentor(Mentor newMentor) {
    if (newMentor == null) {
      Mentor existingMentor = mentor;
      mentor = null;

      if (existingMentor != null &&
          existingMentor.getStudent() != null) {
        existingMentor.setStudent(null);
      }
      return;
    }

    Mentor currentMentor = getMentor();
    if (currentMentor != null && !currentMentor.equals(newMentor)) {
      currentMentor.setStudent(null);
    }

    mentor = newMentor;
    Student existingStudent = newMentor.getStudent();

    if (!equals(existingStudent)) {
      newMentor.setStudent(this);
    }
  }
```

Next are the implementations of addStudent, and removeStudent for the Mentor class as part of the 0..1 Mentor -- * Student association.

```
public boolean addStudent(Student aStudent)
{
  if (students.contains(aStudent)) {
    return false;
  }
  Mentor existingMentor = aStudent.getMentor();
  if (existingMentor == null) {
    students.add(aStudent);
    aStudent.setMentor(this);
  } else if (!existingMentor.equals(this)) {
    existingMentor.removeStudent(aStudent);
    addStudent(aStudent);
  } else {
    students.add(aStudent);
  }
  return true;
}

public boolean removeStudent(Student aStudent)
{
  if (!students.contains(aStudent)) {
    return false;
  } else {
    students.remove(aStudent);
    aStudent.setMentor(null);
    return true;
  }
}
```

As mentioned earlier, in total there are 42 different combinations (28 for associations between different classes, 7 for directed associations and an additional 7 for symmetric associations where both ends are the same class). The spirit of each implementation follows the general guidelines as shown in this section. The specifics of all other combinations can be explored online at [3].

5.7.5 Code Generation Patterns for Support Methods

In addition to establishing relationships between objects, we also include methods to query the minimum and maximum bounds of a relationship. Due to space constraints, we omit the full details of the support methods, but Table 33 highlights the interface.

Table 33: Interface for support methods

Multiplicity	Interface
n, m..n, m..*, 0..n, 0..*	minimumNumberOfStudents() : int
n, m..n, 0..n	maximumNumberOfStudents() : int
n	requiredNumberOfStudents() : int

5.8 Summary

This chapter analyzed the extent to which associations are used in practice. First, we considered seven open source projects and used a systematic approach to uncover candidate associations extracted from the source code of those projects. Next, we looked at a series of UML models, analyzing over 1800 associations from real systems, example systems, and from our repository of UML models. We determined that in practice a small subset of the most-often-used association multiplicity combinations (1--*, 0..1--*, *--*,0..1 -- 1, and 0..1--0..1) accounted for about 85% of all uses of binary associations; leaving about 15% for the remaining types of association multiplicities.

Next, we discussed the syntax of binary associations in the Umple model-oriented language. We identified the 42 needed combinations of multiplicities for association ends to allow for full binary support of associations in a modeling language. We analyzed the impact on code generation based on supporting such a wide range of multiplicity ends. We reviewed the code generation of five modeling tools, and found that none of them dealt with the complexities or concerns identified during our analysis. This may be one reason why code generation is not as widely used in practice as might be expected. Two important qualities captured at the modeling level were not being managed by the generated code; dealing properly with multiplicity constraints, and managing referential integrity. These missing qualities of the generated code imply that the application developer will have to manage the constraints and integrity by hand; or, that the generated code must be manually edited (both situations are far from ideal). We then provided an overview of how associations can be used in Umple. In particular, we showed how code generated from Umple addresses both the multiplicity constraints imposed by the model and the referential integrity when managing association ends.

Chapter 6 Modeling Software Patterns

In the previous chapters we outlined a textual syntax to capture the semantics of UML attributes and associations. This chapter will describe how several well-understood modeling and programming concepts can be described in Umple. The syntax and semantics presented in this chapter help to further reduce the amount of boilerplate code required. We will call the concepts 'patterns'; some are classic patterns from the Gang of Four book [114], whereas others are programming idioms that have been part of programming since long before the notion of patterns became well known.

6.1 Singleton

Umple supports a singleton keyword as shown below:

```
class Application
{
  singleton;
}
```

This declares that the class should follow the Singleton pattern as presented in the Gang of Four book [114]. The Java code generated from the *singleton* keyword is shown below:

```
public class Application
{
  private static Application theInstance = null;
  private Application() {}

  public static Application getInstance() {
    if(theInstance == null) { theInstance = new Application(); }
    return theInstance;
  }
}
```

Singleton classes are generated with empty constructors meaning that attributes and association ends must not be required at construction. For Umple, this implies that attributes must each have a default value and that association ends must have a lower bound of zero. Once a singleton is declared in Umple, other objects can safely access the singleton via the *getInstance* static method; which in turn initializes (if required) and returns the globally accessible instance.

6.2 Equality, Immutability and Keys

Support for the equality predicate is an important feature of a class to allow support for sets, maps, vectors and hashes of instances of the class. Languages like C# and Java specify a contract for their equals and hash code methods, which in turn enable the efficient implementation of collection objects. The most basic notion of equality is *reference equality* where each object is uniquely identified. This notion of equality is inherited by default in Java. Reference equality presents issues for systems where different object instances can be considered the same (e.g. when data is sent and then later received from an external source). For testing purposes, the other extreme can also be useful where all attribute and association values of an object are compared; this is known as *full equality*. By default, Umple classes implement reference equality. If a class contains a unique *key*, then equality is evaluated based on the elements that form part of the key (as will be presented shortly).

The equals contract is required to maintain an equivalence relationship between two non-null variables, supporting properties of the relationship between them such as reflexivity, symmetry, transitivity, and non-nullability.

The difficulty in object equality stems back to an *object identity crisis* as discussed by Baker in [141]. Baker's approach to object equality is to consider the greatest number of properties as possible. Baker includes all immutable components when analyzing immutable objects, and refers to referential equality for mutable objects (i.e. ones that have mutable components). Enforcing *what should be used to define equality* should be a design decision and not necessarily one algorithmically enforced by the implementation language (a view shared with Rupakheti and Hou ([10] – page 109). Our approach will introduce an equality syntax, providing a useful tool to define the equality pattern that best suits the developers needs' without dictating or enforcing its use.

A review of the current implementation issues with equality is highlighted by Rupakheti and Hou [10]. They cite problems including: flawed examples in textbooks, flawed code generation in Eclipse and Netbeans, as well as buggy code within the Java JDK. In their paper they review three types of equality including: type-compatible, type-incompatible and hybrid-equality. The diagram from [10] (and reproduced below in Figure 29) visualizes the difference between type-compatible and type-incompatible equality.

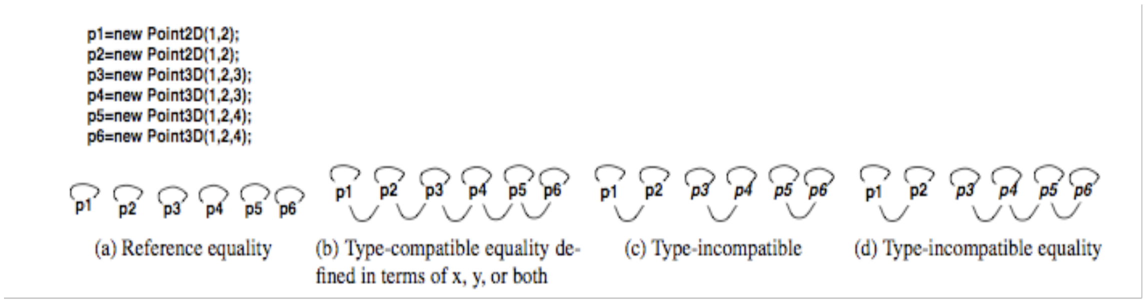

```
p1=new Point2D(1,2);
p2=new Point2D(1,2);
p3=new Point3D(1,2,3);
p4=new Point3D(1,2,3);
p5=new Point3D(1,2,4);
p6=new Point3D(1,2,4);
```
p1 p2 p3 p4 p5 p6
(a) Reference equality
(b) Type-compatible equality defined in terms of x, y, or both
(c) Type-incompatible
(d) Type-incompatible equality

Figure 29: Visualizing Different Types of Equality (from [10])

To summarize, type equality is based on a certain number of properties of an object and compatibility means that objects within an inheritance hierarchy *can* be equal (and therefore reducing the set of equality types to the super class); whereas incompatible equality checks will also be false for different objects within a hierarchy. Please note that incompatible equality checking breaks the Liskov Substitution Principle (LSP) [87] and therefore in these cases the equals method cannot be used where subtypes can be substituted for supertypes. The hybrid-equality mechanism allows mixing type-compatible and type-incompatible equals between different subclasses of a hierarchy by introducing a *delegate* method to help ensure symmetry. The authors also suggest that equals implementation be optimized for inclusion in collections of objects, whereas other types of equality (e.g. full-equality for testing purposes) be relegated to different method names like *similar* or *identical*.

Rupakheti and Hou then studied four Java projects to see how *equals* was implemented. Common problems encountered included suspected flaws in null checks, type casting, and implementing equals for other purposes.

Liskov and Guttag suggest using reference equality for mutable objects and value equality for immutable objects [142].

Cohen [143] provided a similar overview of equality. Cohen promotes the use of composition over inheritance to deal with the issue of maintaining the symmetry property of equality, and he also promotes the use of a helper methods (called *blindlyEquals* instead of *equalsDelegate*). Stevenson [144] provided a more technical analysis applying the Template Method Design pattern to deal with equality. In [145], commentators to Cohen's article suggest including an equivalence class method to allow the developer to determine how equality should behave when checking subclasses against superclasses.

Our approach to immutability, equality and keys has incorporated much of the lessons learned above, and is also based on the pragmatics of current development environments. In Umple, we provide the developer with the ability to define a key that comprises both attributes and association ends from which the equality (and hash code) can be properly implemented.

Inheritance equality is implemented using a template method pattern to maintain the LSP. By providing developers the flexibility to incorporate the use of an object *key*, Umple allows developers to concentrate on what constitutes the unique identity of an object as opposed to the deep technical implications of its implementation. Conversely, maximum flexibility is permitted as the functionality of an object key can be completely omitted and the task of uniquely defining equality can be left to the developer.

6.2.1 The Umple ‘key’ Keyword to Support Equals and Hashing

To facilitate equality checking in Umple, we introduce the *key* keyword. As seen in the previous literature, once the subset of attributes (and associations) of a class have been identified as relevant to determining equality, then the mechanics of implementing the equals (and potentially also the hashCode) methods results in mostly boilerplate code.

By abstracting the elements of equals into a defined *key*, we allow developers to determine what constitutes relevant data of an object for determining uniqueness and the generated code can take care of the mundane (and error prone) implementation. Below is a simple example of a key for a Airline object.

```
class Airline
{
  immutable name;
  address;
  Date createdOn;

  key { name }
}
```

Above, we see that an Airline is uniquely identified by its name. By defining the key to be *name* the generated Java code for the equals method would be as follows.

```
public boolean equals(Object obj)
{
  if (obj == null) { return false; }
  if (!getClass().equals(obj.getClass())) { return false; }

  Airline compareTo = (Airline)obj;

  if (name == null && compareTo.name != null)
  {
    return false;
  }
  else if (name != null && !name.equals(compareTo.name))
  {
    return false;
  }

  return true;
}
```

Here are a few key highlights regarding the implementation of equals above. First, we are using the runtime *getClass* method to ensure the two objects are of the same class, as opposed to simply checking if they are part of the same hierarchy using instanceOf. The issue of allowing superclasses to be equal to subclasses causes symmetric equivalence validation problems as the superclass might be equal to a subclass, but not vice versa. This is discussed at length in [143] and [145]. Once the objects are determined to be the same class, we perform an equality comparison on each relevant key property; ensuring that comparisons of *null* properties are correctly handled. The method delegates to equality checking of each property to ensure consistency when using the same object type within a key.

The implementation of equals also satisfies the equals contract of the JDK (1.4 and above): it is reflexive, symmetric (due to our strict enforcement that only objects of the exact same type can be compared), transitive, consistent, and not equal to null.

In addition to implementing equals, the *key* property also creates the corresponding hashCode() method; allowing for the efficient use of such objects as an index in any class that implements a hash table. The implementation is shown below.

```
public int hashCode()
{
  if (cachedHashCode != -1)
  {
    return cachedHashCode;
  }
  cachedHashCode = 17;
  if (name != null)
  {
    cachedHashCode = cachedHashCode * 23 + name.hashCode();
  }
  else
  {
    cachedHashCode = cachedHashCode * 23;
  }

  canSetName = false;
  return cachedHashCode;
}
```

Our hashing approach is based on Josh Bloch's Effective Java[146] where the use of prime numbers 17 and 23 help ensure unique hash code for different objects.

As the hashCode of an object should remain constant, two additional code segments were added. First, we cache the results for improved performance. Second, we include a mechanism to ensure that an object does not change after having called hashCode; using the same technique as described in Section 4.5.2 for allowing the lazy-instantiation of immutable objects.

The implementation of equals and hashCode abides by the prerequisites of the equals contract of the JDK since v1.4. It provides a high level pattern to describe the meaning of uniqueness without having to be burdened with the implementation details (and complications) based on that meaning (e.g. the improper and incorrect implementation of equals). The key allows developers to define keys across all properties; Umple enforces immutability on all key properties in the same manner as if the ‘immutable’ keyword had been used.. And, finally declaring a key is optional; should the implementation needs of the developer be different from what the above implementation supports, the key can simply be left undeclared.

6.2.2 Keys on Attributes and Associations

An Umple key can be comprised of attributes, as well as associations, as illustrated by the following example.

```
class Flight
{
  id;
  * -- 1 Airline;
  key { id, airline }
}
```

In this example, a flight is uniquely identified but not only by the id, but also by the Airline that it belongs to. This design allows flights on different airlines to have the same flight numbers without breaching equality. Keys can be used on attributes and association with multiplicities of one (UB=1) or many (UB>1).

6.3 Pre/Post Conditions and Operations

The default implementation for access methods (such as set and get) of attributes and associations has been discussed in detail in previous chapters.

Embley et al. [147] observed that modeling language constructs are not sufficient to express many types of constraints so, for completeness, a general-purpose sublanguage is required to augment the modeling language. OCL performs this role for UML, as we will discuss in the coming paragraphs. Our approach in Umple is instead to use the base language (Java, PHP, Ruby, etc.) as the general purpose sublanguage, which should reduce the mental burden of attempting to introduce yet another *new* language (Umple is not a new language, but instead can be viewed as an enhancement or preprocessor to existing languages).

Object Constraint Language (OCL) is an expression-based language that enables modellers to enhance a UML model with constraints and queries. In particular, OCL is used to specify invariants of classes, as well as pre and post-conditions that apply to operations and state transitions.

OCL provides both metamodel and a formal mathematical approach to provide semantic descriptions of the language. These approaches are examined by Flake in [148], where he analyzes the elements presented in the OCL standard, but not in the formalism. The formal semantics lack descriptions for ordered sets, global OCL variable definitions, UML Statechart states and OCL messages. Flake presents an extension to the formalism to deal with these problems.

OCL is a context-free language that is designed to be only an *observer* and not to alter the underlying state of the model entity being constrained or queried. The major limitation with such an approach is that OCL cannot be used to take actions based on the constraints; thus requiring yet another mechanism to manage the behaviour of the system following a constraint violation.

In summary, a limitation with current mechanisms to describe modeling languages is that their definitions reference a specification document and not a technical artefact like an XMI document of the metamodel. In general, existing modeling languages require intermediate models to add additional structure to the semantic descriptions that are not available in the definition of the language itself. In addition, the semantics of these languages are defined (either directly or indirectly) by natural language statements.

Skene [57] critiques visual languages, stating that a diagram can be only a projection of a model, as there is typically insufficient information within a diagram to unambiguously determine the intent of the modeller. He concludes that diagrams should be used to reference aspects of a model with a concrete textual representation.

OCL at its core is quite similar to existing programming languages like C or Java. But, Bauerdick's [149] analysis found several syntactic and type errors in the UML specification.

Umple has been developed as a programming language first with model language features added to enhance that language. Umple has eliminated the intermediary sub-languages like OCL and instead provides a more direct link to the target platform language. By using the underlying target language we allow for very rudimentary control over the system, but we also provide the possibility of extending the modeling concepts to incorporate higher-level constraint definitions, allowing for a syntax like OCL.

In the next section, we will describe a mechanism for *injecting* either action semantics or object constraints within the generated Umple API code. This code injection technique allows for complete customization of system functionality and lays the foundation for adding Umple syntax to support higher level languages like Alf and OCL.

6.3.1 Injecting Custom Behaviour using *before* and *after* Keywords

As introduced in Section 3.8.12 , Umple provides a basic aspect-oriented approach for weaving advice into systems using the before and after keywords. Umple's implementation is currently limited to support only before and after advice; but support for *around* is being considered. Also, Umple's approach to pointcuts is built directly into the advice syntax (i.e. first define a pointcut based on a certain predicate and then define advice for a particular pointcut). Extending Umple to support explicit pointcuts could be added to the language, but further analysis outside the scope of this work is required to ensure the appropriate trade-off between added complexity and added freedom.

To support pre and post-condition advice in a generic manner, we introduce the *before* and *after* keywords to the Umple language. Let us begin with a simple example.

```
class Person {
  name;

  before setName {
    if (aName != null && aName.length() > 20) { return false; }
  }

  after setName {
    System.out.println("Successfully set name to : " + aName);
  }
}
```

In the code above, we have a constraint that names cannot be more then 20 characters using the before injection. We also have a debugging injection that logs each time the attribute is set, using the after injection.

The code provided in the *before* block will be run prior to desired operation (i.e. setName) and the code block provided in the *after* block runs after (or just before returning) from the desired operation.

The Java code generated from the Umple code for the *setName* method is shown below.

```
public boolean setName(String aName)
  {
    boolean wasSet = false;
    if (aName != null && aName.length() > 20) { return false; }
    name = aName;
    wasSet = true;
    System.out.println("Successfully set name to : "   aName);
    return wasSet;
  }
```

For pragmatic purposes, to avoid relying on third party AOP tools, the implementation of before and after is injected into the join point (i.e. method call). Much like the decision to integrate more closely with EMF is left to the developer so too is the decision to integrate with 3rd party AOP tools.

The before and after mechanisms can be used with any Attribute as discussed in Chapter 4 or Association as discussed in Chapter 5. A summary of the operations is described below.

Table 34: Applying before and after operations to Attributes and Associations

Operation	Applies To (UB = Upper Bound)
setX	Attributes, Associations (UB <= 1)
getX	Attributes, Associations
addX	List Attributes, Associations (UB > 1)
removeX	List Attributes, Associations (UB > 1)
getXs	List Attributes, Associations (UB > 1)
numberOfXs	List Attributes, Associations (UB > 1)
indexOfX	List Attributes, Associations (UB > 1)

Before and after can also be applied to an object's other generated methods such as the constructor. This mechanism can be used, for example, to provide additional constraints to a class, or to initialize several internal variables. Example usage is shown below.

```
class Operation {
  query;

  before constructor {
    if (aQuery == null)
      { throw new RuntimeException("Please provide a valid query"); }
  }

  after constructor {
    if (DEBUG) { System.out.println("Created " + query); }
  }
}
```

The code above provides a precondition on the initialization of the query attribute (ensuring it is not null). There is also a DEBUG trace statement that is executed following the completion of the constructor. The Umple code above is translated into the following Java.

```
public Operation(String aQuery) {
  if (query == null)
    { throw new RuntimeException("Please provide a valid query"); }
  query = aQuery;
  if (DEBUG) { System.out.println("Created " + query); }
}
```

An operation can have several before and after invocations. This chaining effect allows before and after statements to focus on a particular aspect of the system being described such as a precondition check of inputs, or a post-condition verification of the state of the system.

The examples shown above provide a single location / join-point emphasis where Umple allows for specific code injections on a particular API call. Umple does, however, support some crosscutting capabilities including wildcard inclusions.

```
class Student
{
  firstName;
  lastName;
  cityOfBirth;

  before get*Name { System.out.println("accessing the name"); }
}
```

The example above would inject the additional code prior to accessing the firstName and secondName attributes, but not the cityOfBirth attribute. Umple also supports exclusions using the not (!) operator. The same semantic example from above is shown below using the ! filter.

```
class Student
{
  firstName;
  lastName;
  cityOfBirth;

  before get*,!getCityOfBirth { System.out.println("accessing the name"); }
}
```

It should be noted that the syntax of Umple's before and after mechanism is purposely generic with a relatively fine-grained level of control. Currently, Umple only supports injecting code into the constructor method, attribute APIs, and association APIs. Future work is being considered to extend this capability for user-defined methods. The intent of this mechanism is to act as a building block to include additional constraint-like syntaxes for common conditions such as non-nullable, boundary constraints and access restrictions. By including before and after code injections at the model level, additional code injection facilities are possible at the model level, without having to modify the underlying code generators. For example, the immutable property discussed in Section 4.5.2 is implemented internally using before conditions on the set methods.

6.3.2 Injecting Custom Behaviour using Mix-ins

In addition to supporting before and after advice from aspect-oriented programming, Umple also includes a convenient mix-in mechanism to weave additional features into existing models. Umple's approach to mix-ins allows developers to directly inject structure and behaviour into a class without the need for using either inheritance or composition.

For example, our Student/Mentor example from Section 3.8 can be extended with additional attributes, associations, advice and custom action semantics as shown below.

```
// Original example from Section 3.8
class Student {}
class Mentor { 0..1 -- * Student; }

// Mixing in additional behaviour in the Student class
class Student {
  Integer id;
  * tutor -- * tutee;
  before getMentor { System.out.println("Accessing Mentor");}
  public void toString() {
    System.out.println("Id: " + getId() + ", Mentor: " + getMentor());
  }
}
```

The result of applying the mix-in above to the Umple code from Section 3.8 would be equivalent to the following Umple code (and the resulting Java / PHP / Ruby code would be the same as well).

```
class Student {
  Integer id;
  * tutor -- * tutee;
  before getMentor { System.out.println("Accessing Mentor");}
  public void toString() {
    System.out.println("Id: " + getId() + ", Mentor: " + getMentor());
  }
}

class Mentor {
  0..1 -- * Student;
}
```

The benefit of allowing mix-ins is that they provide developers with the ability to effectively deal with cross cutting concerns such as bridging two components with additional attributes and associations), bringing features of aspect-oriented programming into an object-oriented syntax.

Currently our mix-in behaviour only supports *augmenting* a class with additional structure and behaviour, but with our upcoming support of state machines, we are looking to also support the *removal* and *replacement* of structures and behaviour.

6.4 Summary

Umple is more than simply a textual notation for UML. Umple provides a full-fledged programming language enhanced with model-oriented concepts. In additional to our textual notation for attributes and associations; this chapter provided an overview of a few common patterns available natively in Umple. The primary patterns include supporting singleton objects, managing keys, immutability, and equality. An additional low-level feature of Umple is the ability to enhance methods operating on model entities such as attributes and associations with pre- and post- operations. These generic operations provide the foundation for adopting a yet-to-adopted UAL standard for modeling actions. For now, this mechanism has been used internally to ease code generation for immutable attributes.

The foundations that motivated the development and guided the use of Umple have been discussed at length in the preceding chapters. The next chapter considers the impacts that Umple can have as a development and modeling language.

Chapter 7 Quality and Validity of the Research

In order to ensure that the results presented in this thesis are of high engineering quality and are as valid as possible from a scientific perspective, several approaches need to be followed. These include:

a) Ensuring that the decisions made are grounded in empirical data and sound analysis.

b) Demonstrating that the engineering process used follows best practices.

c) Showing certain desirable properties of the results have been achieved.

d) Demonstrating that the final results can be used by software engineers for real tasks by showing that they have been used in practice to solve problems arising from approach a.

e) Conducting usability evaluation and other forms of qualitative evaluation, such as grounded theory studies, to discover weaknesses in the result, feeding back the results into further development.

f) Conducting experiments in which the results are compared formally and quantitatively with the best-available alternative approaches.

In this thesis we have performed approaches a through d. We leave approaches e and f for other researchers and future work, largely because Umple has so many features that performing e and f if done well would likely require a second PhD thesis.

In general all six of the above approaches should be conducted in an iterative manner. The results of approaches c, d, e and f should generate additional empirical data that can be used as input for additional sound engineering work, b, and lead to more and more adoption and practical use of the tool, d. In addition, the sound engineering work, b, is needed in order to ensure that the tool has sufficient quality to be put into actual use, d. Many university research tools have failed to be adopted because they are simply prototypes tested on ‘toy’ problems.

Regarding approach a, as discussed in Chapter 2, our first step in dealing with the model-code divide, was to understand from software practitioners what they like and disliked about model-centric and code-centric development approaches. The results indicated that many developers revert back to coding despite strong agreement that modeling approaches tend to be the better fit for most software development tasks. As we discussed, this observation aligns with our professional experiences in software development.

With this evidence in hand, we developed Umple, in which the model is the code and the code is the model. Throughout this thesis we have shown how our decisions have been justified by what we believe is sound analysis, another key part of approach a.

To facilitate adoption and other types of evaluation (approaches d, e and f above), we put into place an engineering process centered around test-driven development, approach b. This is discussed in the next section and is designed to mitigate the risk that the participants are distracted by weaknesses in the tools as opposed to the underlying concepts being evaluated. The process allows our team to react to feedback and analysis from our studies and make changes to Umple for follow-up studies.

This chapter is organized as follows: in the next section we present our test-driven approach (approach b) used by Umple developers to balance the constant change with the stability necessary to build industrial examples and allow for formal evaluation. Our approach to constructing and testing the Umple language is generalized and could be extended to other research languages as a process model to mitigate the risks of constant change. Next, we evaluate the comprehension qualities of systems built in Umple relative to their base language such as Java (approach c). This evaluation provides some evidence that systems written in Umple can exhibit more readable qualities and be less complex. Next, we evaluate Umple's code generation process, also an aspect of approach c. Finally, we provide four case studies of actual systems that were built and deployed using Umple; one of which is Umple itself (approach d).

7.1 Building Umple using a Test Driven Design (TDD) Strategy

The tools and language to implement Umple are built using a test-driven approach. This approach enabled our team to quickly develop a functional version of the language, without hindering future development or features. Test-driven development (TDD) [150] enables the software to evolve based on feedback received from early adopters, and enabled our team to use early versions of the Umple language to develop and enhance future versions (i.e. the Umple toolset and language that was originally written in Java is now 100% implemented in Umple). For more details on testing and software quality, the reader is referred to [151].

The following project characteristics have been observed during the development of Umple; these align with the cited benefits of following a TDD approach.

- **Maintaining modular system design**. For example, we moved from parsing Umple code using Antlr [80] to an in-house solution that focuses on the niche of building libraries of languages and to deal with limitations of earlier releases of the Antlr library. Conversely, we moved away from a custom templating mechanism to using JET Templates. Finally, we recently incorporated Xtext and Eugenia to allow our models to be shared using XMI and Ecore.

- **Fewer feature regressions**. As a software system evolves, newer features will conflict with existing ones (either as a result of added functionality, inaccuracies in existing implementation, or based on updated requirements).
- **More confident refactoring**. Based on the two points above, our team members are more willing to make internal changes to the system with high confidence about not introducing latent bugs, avoiding or at least detecting feature interaction, and maintaining existing features unchanged. This has helped to keep the code base concise, consistent and flexible in the face of enhancements and changes in technologies.

The qualitative benefits observed above are consistent to test-driven approaches, but it should be noted that benefits are subjective by nature and merely align with past test-driven projects of the team members.

The testing and quality approach developed to support Umple provides some valuable insights into patterns and processes to test a *new* programming language; especially in cases where the language is implemented in itself. Taken out of context, simply claiming to have a lot of automated tests does not necessarily imply much about the underlying quality of a software system. But, as you will see in the coming sections, when you combine test-driven development with a structured approach to manage application development, bug fixes and feature requests, one can become much more confident regarding the health of a software system.

The following subsections outline the testing infrastructure in place; as well as the software development processes used to perform the following tasks:

- Adding new features to the system
- Making minor modifications to an existing feature
- Resolving and identifying defects

7.1.1 Testing Infrastructure

The Umple tools include: a parser, a component that encapsulates Umple's metamodel, a synchronization engine (between diagram and text), as well as several code-generators and model-to-model transformation engines. Testing these components helps to provide confidence that Umple code is correctly tokenized and processed into an internal representation consistent with the metamodel. The internal representation is then tested to ensure it is properly translated into other artefacts (either additional models like EMF, Yuml, xUML; or source code such as Java or PHP or Ruby).

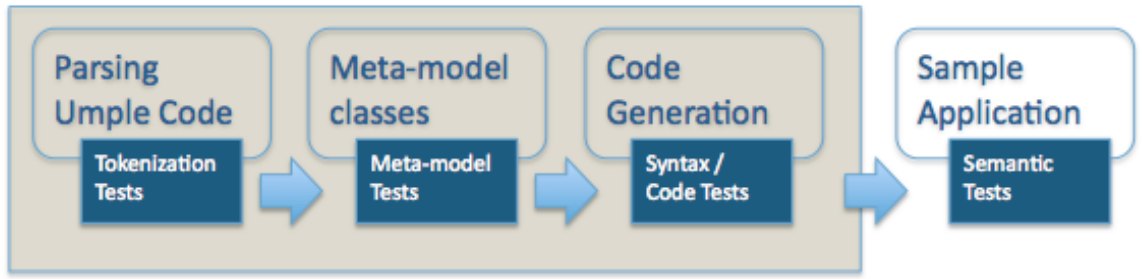
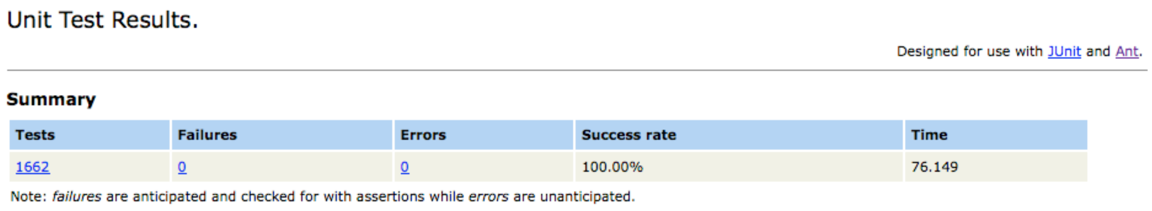


Figure 30: Umple Testing Infrastructure

The Umple testing process in Figure 30 is only capable of testing within the scope of Umple. In other words, you are only testing Umple and not testing the set of possible systems created using Umple. This means that, so far, you can only test that the outputs are syntactically correct. To achieve the additional level of testing whereby you validate the semantics of system built using Umple, one must build and test sample Umple applications and perform the testing against those sample systems. Currently, Umple provides sample application for each base language supported by Umple: Java, PHP, and Ruby.

At present, there are over 1600 tests that span all areas above and are run as part of our automated quality process as shown below in Figure 31.


Unit Test Results.

Designed for use with JUnit and Ant.

Summary

Tests	Failures	Errors	Success rate	Time
1662	0	0	100.00%	76.149

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Figure 31: Umple Automated Testing Report

In the subsequent sections we provide an overview of the each aspect of the Umple testing approach.

7.1.2 Testing the Umple Parser

Testing the Umple parser is centered on the tokenization of Umple code. Our tests ensure that Umple models are parsed and tokenized as we expect.

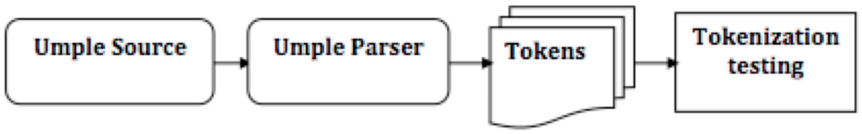


Figure 32: Process to Test the Umple Parser

A simple parser test is shown below that verifies the class *Student* written in Umple is properly parsed, and then properly tokenized by the UmpleModel.

```
@Test
public void emptyClass()
{
  String input = "class Student{}";
  String expectedOutput = "[classDefinition][name:Student]";

  UmpleModel model = new UmpleModel(new UmpleFile("test.ump"));
  UmpleParser parser = new UmpleParser(model);

  boolean answer = parser.parse("program", input).getWasSuccess();
  Assert.assertEquals(true, answer, "Unable to parse Umple code");
  answer = parser.analyze(false).getWasSuccess();
  Assert.assertEquals(true, answer, "Unable to analyze Umple code");
  Assert.assertEquals(expectedOutput, parser.toString());
}
```

The pattern for parser-related tests is as follows.

```
@Test
public void someSyntaxToVerify()
{
  // Step 1: Load external source file
  // Step 2: Parse file (ensure parsing successful)
  // Step 3: Verify tokenization
  // Step 4: Clean up
}
```

To verify the tokenization process, we opted to include a convenience toString method in the UmpleParser. This is a mechanism to facilitate testing as well as debugging; we must of course not forget to test the toString method as well.

At present, there are about 200 parser related tests for Umple.

7.1.3 Testing the Metamodel Classes

Testing the metamodel classes ensures that Umple will be able to maintain valid internal representations of a model, i.e. instances of the metamodel. Umple's metamodel is presented in Section 3.7.2 . This most closely aligns to unit testing as described in most TDD books (such as [150]). These tests follow the standard unit testing pattern:

```
@Test
public void someSpecification()
{
  // Step 1: SetUp
  // Step 2: Execute
  // Step 3: Verify
  // Step 4: TearDown
}
```

It is important to document not only how the system behaves under normal conditions, but also how it behaves in abnormal scenarios where, for example, preconditions are not satisfied.

Here is a sample test case for the Multiplicity metamodel class. Below we see that setting the range on a Multiplicity properly sets both the upper and lower bound.

```
@Test
public void setRange_ExplicitBounds()
{
  Multiplicity m = new Multiplicity();
  m.setRange("1", "2");
  Assert.assertEquals(1,m.getLowerBound());
  Assert.assertEquals(2,m.getUpperBound());
}
```

At present, there are about 700 metamodel-related tests for Umple.

Some may question the value of such *simplistic* tests. It should be noted that the test is merely an example to demonstrate the structure of a meta-model test and is simplistic to demonstrate the overarching structure of a metamodel test. But, more importantly, the spirit of following test-driven design (as well other *driven* approaches) is the concept of evolving design through tests. By following a test-driven approach, the tests (and the ability to run them over and over again in an automated fashion) is a welcome side effect but the true power of the approach is in the initial design whereby you first *exploit* the common uses of your meta-model and only then do you concern yourself with the implementation.

7.1.4 Testing Code Generators

The Umple language provides several model-to-model and model-to-code transformations. The input to a code (or model) generator is a populated metamodel instance and the output is the base language or simply a transformation into another model syntax. In addition to overall setup and tear down, the high-level approach to testing code generation is shown below.

```
@Test
public void verifyGeneratedCode()
{
  // Step 1: Prepare Metamodel
  // Step 2: Run Code Generator
  // Step 3: Verify results
}
```

To prepare the metamodel instance, there are two approaches: populating the model *by direct calls* to the available API of the metamodel (or perhaps using a mock object facility[152]); or, *parsing source code* using the existing infrastructure (as shown above in Section 7.1.2).

The first approach (to populate the metamodel instance directly using the API) has several drawbacks: The setup code can become quite cumbersome and complex, which could make the test code less readable and maintainable. And, it can also be error prone, as the testing developer must properly populate the metamodel. The primary benefit of this approach is the isolation of the code generator's behaviour from that of the code that parsed the source and generated the metamodel instance. Issues related to the parsing phase (translating Umple code into the Umple metamodel instance) of the project would not interfere with testing the code generator.

For example, the following code creates a Student class with three attributes (id, name and program).

```
UmpleModel m = new UmpleModel(null);
UmpleClass student = m.addUmpleClass("Student");
student.addAttribute(new Attribute("id","Integer",null,null,true));
student.addAttribute(new Attribute("name","String",null,null,false));
student.addAttribute(new Attribute("program","String",null,"SEG",false));
```

The second approach where the test begins at the Umple *source code* that is then parsed into the metamodel (which is then used as input to the code generators) has two primary benefits. First, it is easier to express a system in its own syntax as opposed to building it using a metamodel's API. Second, you provide additional testing for the parsing aspect of the system. The obvious drawback is that these tests are no longer pure unit tests, and that failing tests in this component could be resulting from the parser or the code generator.

Here is the same example from above, but written using the Umple syntax directly.

```
class Student
{
  Integer id;
  name;
  program = "SEG";
}
```

Regardless of the approach, the metamodel needs to be populated before the code generation can be tested. Instead of crafting a new means to populate that model, it makes more pragmatic sense to simply reuse the existing (and tested) parsing approach as described in the previous section.

By parsing the model code directly, an added benefit is that you can create a generic TemplateTest to manage the test artefacts (i.e. input model code, expected output system code); leaving the testing mostly boilerplate-code free.

The outline of such a template class is shown below.

```
public class TemplateTest
{
  @Before
  public void setUp()
  {
    //configure paths to Umple data files
    //this can be configured to support multiple languages
  }

  @After
  public void tearDown()
  {
    //clean up any temporary or generated files
  }

  public void assertTemplate (String modelFile, String expectedGeneratedFile)
  {
    // Parse / tokenize modelFile
    // Create an instance of meta model
    // Generate code for the underlying system
    // Compare the actual generated code with the expectedGeneratedFile
  }
}
```

The method signatures will vary slightly depending on the type of code generator that is being created; but the overall structure remains intact. Below is a code snippet of relevant lines in the Umple TemplateTest class.

```
public class TemplateTest
{
  public String pathToInput;
  public String pathToRoot;
  public String language;
  public String languagePath;

  @Before
  public void setUp()
  {
    pathToInput = "test/cruise/umple/implementation";
    pathToRoot = "../cruise.umple";
    language = "Java";
    languagePath = "java";
  }

  @After
  public void tearDown()
  {
    //omitted for brevity
  }

  public void assertUmpleTemplateFor(String umpleFile, String codeFile)
  {
    assertUmpleTemplateFor(umpleFile, codeFile, null);
  }

  public void assertUmpleTemplateFor
                 (String umpleFile, String codeFile, String className)
  {
    UmpleModel model = createUmpleSystem(pathToInput,umpleFile);

    if (model.getUmpleClass(className) == null)
    {
      Assert.fail("Unknown class:" + className);
    }
    String actual = model.getUmpleClass(className).getGeneratedCode();
    File expected = new File(pathToInput,codeFile);
    System.out.println(actual);
    SampleFileWriter.assertFileContent(expected,actual);
  }

  public  UmpleModel createUmpleSystem(String path, String filename)
  {
    UmpleParser parser = new UmpleParser();

    String input = SampleFileWriter.readContent(new File(path,filename));
    ParseResult result = parser.parse("program", input);

    if (!result.getWasSuccess())
    {
      Assert.fail("Syntax Failed at:" + result.getPosition());
    }

    UmpleModel model = parser.getModel();
    model.setUmpleFile(new UmpleFile(new File(path,filename)));

    result = parser.analyze(true);
    if (!result.getWasSuccess())
    {
      Assert.fail("Semantics Failed at:" + result.getPosition());
    }

    return model;
  }
}
```

With the infrastructure shown above in place, adding new code generation tests is straightforward, as the template encapsulates the distracting elements of the test setup. A sample code generator test is shown below.

```
@Test
  public void Association()
  {
    assertUmpleTemplateFor(
      "AttributeTest.ump",
      languagePath + "/ AttributeTest."+ languagePath +".txt","Student");
  }
```

The test above requires a model file (AttributeTest.ump), as well as a source code file based on the selected language. In Umple, we currently support Java, PHP and Ruby. Using the test case above, the same model file can be reused to test against the Java generated code (java/AttributeTest.java.txt), the PHP generated code (php/AttributeTest.php.txt), and the Ruby generated code (ruby/AttributeTest.ruby.txt). This infrastructure can easily be extended to add testing for other generated languages.

At the time of writing, there are about 300 code generation / syntax tests.

7.1.5 Testing Generated Systems

The previous sections described testing the Umple system itself, but not Umple generated systems. The semantics of Umple's modeling components are quite rich so it is also important to provide adequate testing of generated systems to ensure that the semantics of an Umple model is upheld in the underlying base language (i.e. Java, PHP or Ruby). This level of testing ensures the appropriate behaviour of the generated Umple executable artefacts, which is essential to support our industrial case studies.

Let us consider a simple example of testing the semantics of a class attribute.

```
class Student {
  name;
}
```

The specifications for an attribute as defined above includes the following properties and behaviour: the attribute is included as a constructor argument, the attribute can also be modified and retrieved. Based on the above description of an attribute, we could write the following tests (the tests are written using JUnit4 syntax).

```
@Test
public void attributeBehaviour()
{
  Student s = new Student("james");
  Assert.assertEquals("james",s.getName());
  s.setName("henry");
  Assert.assertEquals("henry",s.getName());
}
```

This test can be equally expressed in PHP using PHPUnit as shown below (an xUnit testing framework for PHP applications).

```
public function test_attributeBehaviour()
  {
    $s = new Student("james");
    $this->assertEqual("james",$s->getName());
    $s->setName("henry");
    $this->assertEqual("henry",$s->getName());
  }
```

By capturing the properties and behaviour of systems built with Umple, we are able to build up an extensive library of executable specifications which more concretely demonstrate the realized behaviour of the system, as opposed to its documented behaviour (and as most software practitioners are aware, it is common for documentation to get stale and out of sync quickly [153]).

7.1.6 Managing Defects and Minimizing Regressions

Despite having over 1600 automated tests that span all facets of the toolset from parsing to code generation, our system will inevitably contain defects. In this section, we discuss how our testing infrastructure allows for better defect management by representing bugs as failing tests, effectively diminishing the time and effort required to perform regression testing.

When a defect is uncovered, it might be one of the following:

1. Defects in the way in which the Umple language is tokenized into an abstract syntax tree
2. Incorrect population of the Umple metamodel instance from the tokenized language.
3. Inappropriate behaviour of the metamodel classes
4. Syntax errors in the generated base language code (e.g. Java, PHP or Ruby)
5. Semantic errors (i.e. the incorrect behaviour) in the generated base code

In addition to the defect scenarios above, there is a possibility of usability and training defects where the intended or expected behaviour differs from its actual implementation. Dealing with these types of defects is outside the scope of our work, and instead, we focus our attention on well-defined, repeatable issues.

As defects are uncovered, it is not always apparent which category of defects has been uncovered. In situations where the root cause of a defect is unknown, we resort to bottom-up defect resolution approach. The bottom-up approach resembles our testing process in place. We start by verifying the tokenization, then the metamodel, followed by the generated code, and finally, the application level testing.

Step 1: Identify the problematic Umple code with a failing test

The first step is to identify the Umple component responsible for the problematic outputs. This process may involve a few iterations to isolate the exact symptoms causing the issue, but that is not always necessary.

For illustration purposes, our sample defect is that untyped attributes are not reflected in the generated code. The following code shows the potentially problematic Umple code.

```
class Student { id; }
```

The test case would perform an end-to-end high level test that properly documents the identified issue with a failing test.

Step 2: Verify the tokenization of the problematic Umple code

With our high-level failing test in place, we now analyze each step of the process to identify the root cause of the problem. We start with the Umple parser.

The process for verifying the parser are already available. We simply add an additional test using the problematic Umple code as the input and we verify the output.

To continue with the example above, we first make sure that untyped attributes are properly parsed and tokenized. The expected result "[class][name:Student][attribute][name:id]" represents a *toString* view of the tokenization sequence used to assert equality in a human readable form.

```
@Test
public void untypedAttributes()
{
  assertParse("untyped.ump", "[class][name:Student][attribute][name:id]");
}
```

If this test fails, we resolve it and re-run our test from Step 1. If that test succeeds then it is likely that the problem is now properly resolved and the debugging process is complete. If not, then we move on to the next step.

Step 3: Verify the instance of the Umple metamodel

Once the Umple source has been shown to parse correctly (but that the observed issue persists), we then validate that the instance of the Umple metamodel is consistent with the Umple input. Here, we are ensuring that the metamodel was properly populated following the parser tokenization process.

To test the metamodel, we enhance the test identified in Step 2 as follows.

```
@Test
public void untypedAttributes()
{
  assertParse("untyped.ump", "[class][name:Student][attribute][name:id]");
  UmpleClass aClass = model.getUmpleClass("Student");
  Assert.assertEquals("Student",aClass.getName());
  Attribute attr = aClass.getAttribute("id");
  Assert.assertEquals("id", attr.getName());
  Assert.assertEquals("String", attr.getType());
}
```

Here, we assert that the Student class is created, and that it has an attribute of type *String* with the name *id*. If this test fails, we follow the same steps from the previous step: re-test our high-level test and proceed to the next step only if that test still fails.

Step 4: Validate the proper behaviour of the metamodel

Once the Umple code appears to be parsed correctly, and the metamodel is properly populated, we then investigate if there is any special behaviour that is performed by the metamodel instance that may not be handled properly.

For example, an Attribute has an operation *isPrimitive* which checks for the Umple predefined types, and perhaps this operation is not functioning as expected. Below is a test case demonstrating the expected behaviour.

```
@Test
public void isPrimitive()
{
  Attribute av;

  av = new Attribute("a",null,null,null,false);
  Assert.assertEquals(true,av.isPrimitive());

  av.setType("String");
  Assert.assertEquals(true,av.isPrimitive());

  av.setType("Integer");
  Assert.assertEquals(true,av.isPrimitive());
  av.setType("Double");
  Assert.assertEquals(true,av.isPrimitive());
  av.setType("Boolean");
  Assert.assertEquals(true,av.isPrimitive());
  av.setType("Date");
  Assert.assertEquals(true,av.isPrimitive());
  av.setType("Time");
  Assert.assertEquals(true,av.isPrimitive());

  av.setType("Address");
  Assert.assertEquals(false,av.isPrimitive());
}
```

Dealing with this type of testing is difficult to categorize, and each scenario will need to be analyzed individually. If the behaviour of the metamodel appears to be working correctly (but our high-level test still fails), we continue to the next step.

Step 5: Compare the expected versus actual generated code

Next, we analyze the expected code versus actual generated code. Here, we are testing that the syntactic translation of the Umple metamodel instance into the generated base language is correct.

The example test case would resemble the following code.

```
@Test
public void untypedAttributes()
{
  assertUmpleTemplateFor("attribute.ump","attribute.java.txt”,"Student");
}
```

Where the “attribute.ump” would be the problematic Umple code and the “attribute.java.txt” would contain the desired Java code to be generated from the model.

<u>Step 6: Test the behaviour of the generated code</u>

If all other tests are passing successfully, the final aspect to test in the Umple system is that the generated code conforms to the semantics of the model. It might be the case where we are *producing* what is believed to be the correct code, when in fact the generated code does not behave as the intended by the model.

In our on-going example presented above, we write unit tests against a sample application that contains a *Student* with an *id* attribute.

```
@Test
public void constructor()
{
  Student s = new Student("x");
  Assert.assertEquals("x",s.getId());
}

@Test
public void setAndGetStringAttribute()
{
  Student s = new Student("x");
  s.setId("y");
  Assert.assertEquals("y",s.getId());
}
```

The steps outlined above provide a high-level approach to deal with issues as they arise. The most important first step in the process is to create a *failing* test that exhibits the invalid behaviour of the system. The granularity of this test is not important, as we have developed a systematic approach to verify each step of the Umple compiler to help determine the root cause of the issue.

By adopting a test-driven approach, we can grow our regression test suite to deal with new issues and at the same time mitigate the risk of regressing on existing functionality. In the next section we look at how the current infrastructure supports future potential enhancements.

7.1.7 Enhancing the Umple Language

A missing feature can also be described as a defect; in other words, something is missing that should not be. The approach to adding new features can follow a similar path as described in the previous section, but that might not always be possible; resulting in more of a top-down approach compared to the bottom up approach for *regular* defects.

By starting at the last step (Step 6) from the previous section, our team is able to specify the desired behaviour of a new language feature without that feature being available. And, because

Umple is written in itself, we are able to implement most behaviour using existing Umple constructs.

In the example below, we will be adding an OCL-like constraint syntax placed on attributes. The semantic test below assumes the OCL constraint that *age >= 18* for any Student.

Before implementing the change, we first write a test case in the base language demonstrating the desired functionality. This test would need to be translated into each language. Below is a sample test case written for Java.

```
@Test
public void cannotSetTo17()
{
  Student s = new Student(18);
  Assert.assertEquals(18,m.getAge());
  Assert.assertEquals(false,s.setAge(17));
  Assert.assertEquals(18,m.getAge());
}
```

Next, we use Umple itself to implement the feature using existing constructs.

```
class Student
{
  Integer age;
  before setAge {
    if (aAge < 18) { return false; }
  }
  after getAge {
    if (age < 18) { throw new RuntimeException("Age must be >= 18") }
  }
}
```

Once the behaviour is validated with sufficient (and passing) test cases within our base languages we then enhance the parser and metamodel with the new language constructs.

The potential Umple syntax might look like the following.

```
class Student
{
  Integer x;
  // this is the potential invariant syntax
  inv x >= 18;
}
```

Next, we migrate the custom code written in the behaviour tests into the code generation process to validate the generated syntax. Following that, we deploy a new version of Umple itself and update the original behaviour tests to use the new language constructs (as opposed to having to write the behaviour by hand, as was required *before* the feature was available). These tests themselves remain relatively unchanged; we simply update the tests to use the new language constructs.

In pure TDD methodology, the process is not just about testing; but rather about designing the system in a modular fashion maintaining low coupling and well-defined interfaces. The process is also about capturing the intention of the software (i.e. automated tests) that can be easily verified (i.e. re-running the test suite) so that little effort goes to waste.

For example, the act of manually testing and modifying (i.e. debugging) an application until it works benefits only the developer performing the task. It cannot be replicated easily, as the debugging steps are not documented and are lost once the debugging exercise is complete. Conversely, by capturing the testing process through automation, all developers can benefit as knowledge is gained about the true behaviour of the system that can be easily re-run and re-verified.

In the case of building a new programming language (or in the case of Umple, extending existing base languages), we first need to be concerned with testing the tooling itself. But, because the outputs of such systems are systems too, they can also be tested (i.e. semantic testing of systems generated using the new language).

In addition, because Umple is implemented in itself, we are able to capture the debugging effort of new code generation behaviour in our automated tests, and then modify the underlying Umple language to replicate that behaviour natively, as shown with the OCL constraint example. In summary, we enhance the Umple language so that we can refactor our Umple tooling (which is written in Umple) to make use of the enhanced language elements; eating our own dog food, so-to-speak.

7.1.8 Validating Umple through Testing

Our first step towards validating the merits of Umple as a language was to provide the necessary constructs to enable continuous enhancements and improvements to the language, to the code generation, and to the supporting Umple tools. In the previous sections, we outlined the compartmentalization of Umple into well-defined components including a parser, a metamodel, a code generator (and model transformer) as well as a test-bed for validating the semantics of systems built using Umple. We demonstrated a generic approach to testing programming languages as well as provided some patterns for resolving defects in the tools as they are discovered.

The following section presents our next approach to validate Umple; analyzing it to see whether certain desirable properties are present. In particular, we will look at certain program comprehension qualities of an Umple-built system.

7.2 Improving Program Comprehension with Umple

Umple, we anticipate, should help increase software program comprehension by allowing developers to describe a system at a more abstract level, and also by significantly reducing the amount of code that needs to be written and later understood relative to base languages like Java, PHP and Ruby. A summary of the findings discussed in this section is published in [19].

As discussed earlier in the thesis, Umple's objective is to simplify software systems by incorporating higher-level abstractions in programming languages. One of the motivations for our work is our research reported in Chapter 2 indicating that a large part of the development community remains steadfastly code-centric; hence visual modeling tools are not being adopted as widely as might be desired. Another motivation is that there is much repetitive code in object-oriented programs; and a language like Umple, which incorporates abstractions to remove boilerplate code, should be able to promote understandability and reduce code volume.

This section analyzes properties inherent to an Umple program (versus its equivalent in the generated base language) that allow developers to work at a higher abstraction level, providing the opportunity at reduced complexity. We compare several systems developed using Umple vs. their counterparts in Java, PHP. Finally, we discuss general benefits to comprehension that appear to be achieved using Umple's model-based syntax.

7.2.1 Reducing Complexity and Improving Readability

Software complexity metrics take into account properties such as the overall size of the system, the classes and its methods. Readability refers to how easily a text can be understood and tends to be based on local factors such as number of identifiers per line [154].

Aggarwal claims that source code readability is critical to the maintainability of a software project [155]. Buse and Weimer [154] conducted empirical work to isolate the quality of readability; trying to eliminate context and complexity from their study. Using a tool called SnippetSniper, they asked participants to rate readability of snippets of code, each limited to about 3 statements. They found a strong statistical correlation between having fewer identifiers and characters per line and readability where *less-is-more*.

Buse and Weimer also suggested that new programming languages should encourage readability by providing language constructs that take advantage of their findings. Umple follows their recommendation, as we illustrate the following sub-sections.

In addition to encouraging readability, Umple also allows developers to write less-complex code. Mohan et al.'s [156] investigation into three spatial complexity metrics showed that they all shared a strong correlation to variation of a significantly simpler metric; lines of code (LOC). As the authors suggest, more complex metrics do not seem to offer any advantages over LOC to measure complexity. We therefore adopt LOC as our complexity metric, and compare lines of Umple code to both lines of code in Java systems that were converted to Umple, as well as lines of generated code that the Umple compiler emits.

There are arguments for and against considering white-space and comments in lines-of-code calculations, but as long as we maintain a consistent choice, this issue is moot. Using lines of code is criticized if used to measure human programmer *productivity*, but that is not the concern here.

Sajaniemi [157] states that variables are not used in arbitrary ways, and that there are patterns that re-occur in programs. Umple has enhanced the declaration of a *variable* by distinguishing between attributes and associations. Umple further enhances the semantic meaning of an attribute by providing additional notations that better highlight the variable's intention such as immutable, unique and defaulted. A discussion of Umple attributes is presented in Chapter 4.

7.2.1.1Reducing Complexity with Umple Associations

The following Umple code shows two classes linked by an association.

```
class Mentor {
  1 -- * Student;
}
class Student{}
```

The declaration of an association consists of a single line of Umple. If the association had instead been implemented in Java, an extra 90 lines of code would typically have been required. This includes declarations of member variables in both classes, methods to add, delete, query and iterate through links, as well as code in the constructors. Some of the needed code serves to enforce the multiplicities. From our study of associations presented in Section 5.2 , referential integrity was not observed and often the many-end was presented as a list structure; with code to access the association being calls to generic methods that were defined on that list structure. As such, no additional code is required in the associated classes. It should be noted that such an approach has its drawbacks (i.e. difficulty maintaining referential integrity). Regardless, due to the lack of *extra* code observed in practice we considered only the declaration of the variables needed for the one-to-many association in our analysis.

```
//In the Mentor class
private List<Student> students;

//In the Student class
private Mentor mentor;
```

To assess readability, in Table 35 we see a comparison of the declaration of the Mentor-Student association in Umple versus Java.

Table 35: # Characters of Umple versus Java

	1 line Umple	2 lines Java	% Improvement
# Characters	12	49	75%
# Identifiers	3	5	40%

Note that we do not count ‘private’ as an identifier, since it is a keyword. Also, the Java code follows best practices of making variables private and using genericity (templates), but even without this style, the improvement on lines of code and characters per line would have still been 54% and 25% respectively.

First, we see that Umple can declare an association in just one line (where Java requires two). Second, we see that Java can take up to four times as many characters compared to Umple. Finally, for associations, the number of identifiers is fewer in Umple.

In addition to declaring the variables for associations, the Java language must also provide the necessary methods to manage them. To appreciate the complexity of properly managing associations, below is the generated implementation of just one of the required methods, whose job is to add a Student to a Mentor as defined by the association above.

```
public boolean addStudent(Student aStudent)
{
  if (students.contains(aStudent))
  {
    return false;
  }

  Mentor existingMentor = aStudent.getMentor();
  boolean isNewMentor = existingMentor != null
    && !existingMentor.equals(this);
  if (isNewMentor)
  {
    aStudent.setMentor(this);
  }
  else
  {
    students.add(aStudent);
  }
  return true;
}
```

With Umple, large amounts of boilerplate code are thus avoided. One simply declares the associations and lets the compiler take care of the rest. This benefits both the developer and the subsequent maintainer. Please refer to Chapter 5 for more information on generating source code for Umple associations.

The boilerplate code above might appear overly complex, so we will demonstrate by example its necessity. The examples are written in Java, and would be unchanged in an Umple program. They illustrate the usage and traversal of instances of Student and Mentor. All examples assume the following objects have been instantiated to the following types: Mentor mrJones, Mentor missHenderson, Student andrew, and Student jamie.

First, the addStudent method should disallow duplicate links.

```
mrJones.addStudent(andrew);

//This call will return false;
mrJones.addStudent(andrew);
```

Second, a Student and Mentor can be linked from either end of the bi-directional relationship.

```
// Student sets the Mentor
jamie.setMentor(mrJones);

// Or, Mentor adds the Student
mrJones.addStudent(andrew);
```

And finally, a Student can only be linked to one Mentor (for a one-to-many relationship), so re-assigning a Student to a new Mentor must remove the link to the existing Mentor.

```
// Add Jamie as a mentor to Mr. Jones
jamie.setMentor(mrJones);

// Mr. Jones will no longer
// be Jamie’s mentor
missHenderson.addStudent(jamie);
```

The above analysis shows that by introducing the association abstraction, Umple substantially improves readability, measured in the number of identifiers or characters to make basic declarations, and also substantially reduces complexity, measured by the number of lines of code the developer or maintainer must deal with.

7.2.1.2Reducing Complexity with Umple Attributes

Umple attributes provide several opportunities to reduce code complexity. First, attributes can remain untyped (defaulted to a String implementation) and generate both set/get accessor methods, which further reduces the code volume and code density.

The *grade* attribute in the Registration class could be declared as follows in Umple;

```
grade;
```

In contrast, the same attribute as declared in Java would require code similar to that shown below; assuming best practices are followed.

```
private String grade;

public void setGrade(String aGrade)
```

```
{
  grade = aGrade;
}

public String getGrade()
{
  return grade;
}
```

In Umple, the API for set methods returns a Boolean to reflect whether or not the attribute (or association) was properly set or not – with the addition of attribute contraints it is not guaranteed that *setting* an attribute will update its value (i.e constraint fails and method return false).

From the perspective of readability, the Java code declaration contains an extra identifier and 12 extra characters. If the 'String' data-type had been made explicit in Umple, there would have been smaller, but still noticeable savings.

Regarding complexity, the Java version has an additional 6 or 8 lines to capture the get/set API (the number would vary depending on formatting preference).

Let us now consider an attribute that can contain multiple values. The Umple code to manage a list of contact addresses is shown below.

```
Address[] contacts;
```

UML also allows multi-valued attributes (although it uses a slightly different syntax). An alternative to the Umple syntax above would be to use of a unidirectional one-to-many association. However, attributes typically represent properties of a class and hence have a simpler implementation, lower coupling and greater reusability of the attribute's data-type. Please note there are also subtle differences in code generation between attributes and unidirectional associations. For example, by default attributes are included in the constructor of an object (unless explicitly qualified with the *lazy* keyword) whereas only associations with lower bounds > 0 are included in the constructor's signature.

The code needed to declare a multi-valued attribute in Java is shown below. Readability is improved because there is a one fewer identifier needed and 39% fewer characters.

```
private List<Address> contacts;
```

Complexity is also significantly reduced since the mechanism to manage a list attribute does not have to be coded. This API includes methods like adding and removing entities, retrieving one or all entities as well as asking how many entities belonging to the instance.

By analyzing Umple systems from first principles by analyzing lines of code and code density there is seems to be quantifiable support that Umple systems could be more readable and exhibit qualities of high program comprehension. In the next section, we analyze actual systems built using Umple and compare them to similar systems written in a base language.

7.2.2 Analyzing Systems Written in Umple

Umple was designed to bring additional abstractions to OO languages like Java. As such, using Umple you can "say more" by "writing less". In the previous section, we illustrated the potential for Umple code to be more concise and written in fewer lines of code than existing OO languages. We used Code Analyzer [158], an open source application to track software source metrics, to count lines of code of Java, PHP and Umple files.

In the subsequent section we provide a detailed analysis of program size of two systems, one written in Java and the other in Umple, to compare the relative sizes of the two systems.

We then investigate two systems written in Umple to assess the amount of generated code required (in Java, PHP) compared to the source Umple code.

7.2.2.1Refactoring Umple into Itself

Before we compare systems written in Umple to the code the Umple compiler generates, it is important to first gauge how well the source code generated from Umple compares to the source code written by a developer *without* the use of Umple.

As it turns out, we can use Umple itself to make this comparison, since Umple version 1.0 was written in Java, and version 1.3 (the next stable version) was a re-write largely into Umple itself. In this section we discuss the process of refactoring Umple into itself, and in the next section we analyze the source code of both systems (Java-Based versus Umple-Based).

The effort to refactor Umple into itself provides two main benefits. First, we provide a practical test-bed for identifying language and code generation features required to build real systems. As we encountered code blocks in the Java code that were not supported in Umple; we enhanced Umple, adding the necessary support for the missing feature so that we could continue with our migration from Java to Umple. Second, it provided a realistic migration scenario demonstrating that Umple can be used in existing systems and more importantly existing systems can be migrated towards Umple. As most software development effort is maintenance, it is important that Umple integrates nicely not only into the base language, but also the existing software process and toolset [159]. Requiring a big-bang effort to migrate towards Umple is not practical and would severely limit its adoption.

As we refactored our Java based Umple system into an Umple based system, several less-than-ideal modeling decisions were uncovered in the Java code; these were then reflected in Umple's visualization as a less-than-ideal model. This can somewhat be explained by the fact that the design of the first model for Umple was separate from its implementation, with few facilities beyond reverse engineering tools to visually inspect the dependencies and relationships added to the system that were not part of the original design.

The original Umple metamodel, developed in UML (not Umple) is shown below in Figure 33 and the current Umple metamodel (as developed in Umple itself) is shown in Figure 11 (Section 3.7.2).

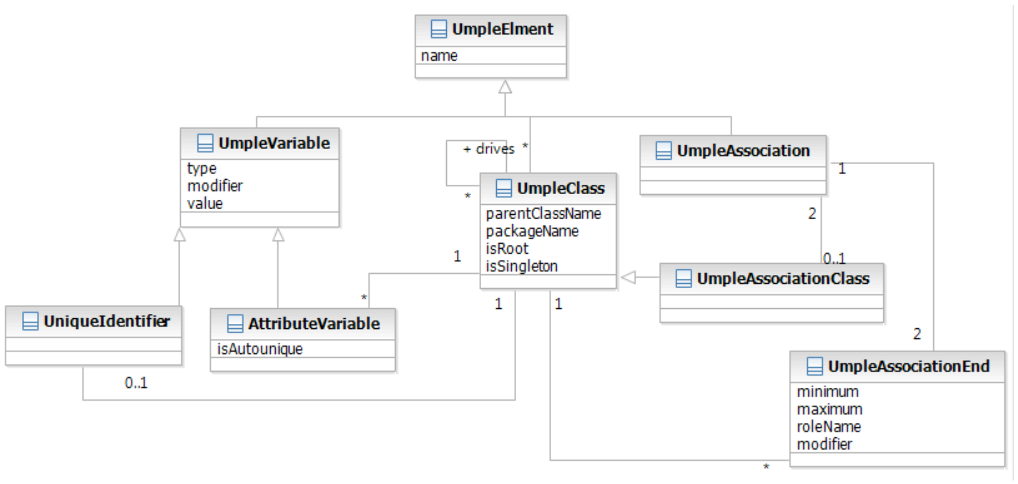


Figure 33: Original Umple Metamodel

These less-than-ideal modeling decisions that were made were not uncovered until we attempted to model the Umple system itself in Umple. For example, an *AssociationVariable* (in addition to an Association itself) was introduced to capture the *variable* being used in the generated code to represent one end of an association, instead of using the *UmpleAssociationEnd* directly; resulting in redundant code. This resulted in the association between UmpleClass and UmpleAssociationEnd being omitted from the implemented system resulting in no direct link between the two classes. A third example is the addition of an unmodelled class *GeneratedClass*, which, if given the opportunity, would have been modeled quite differently. The current implementation is tightly coupled to the UmpleClass, when in reality code generation is not always compartmentalized by class, which is particularly true for model-to-model transformations (where code generation is at the entire model level, not at a class-by-class level).

These issues and omissions were only discovered during our migration of Umple towards Umple, and some issues remain. Had a language like Umple been available for our first version of the system, these issues could have been avoided due to the following properties of the Umple language:

- Additional (and yet to be modelled) elements such as extra classes, or associations added while working with the model would appear in the UML diagram, providing a visual cue for discussion during code reviews.
- Absent elements such as classes and associations missing during the translation of model to code would not occur at all because the model itself is the code and missing entities would have been discovered during the *development* of the model in Figure 33.

Further improvements to the underlying metamodel can be achieved, including those less-than-ideal examples identified above. Nonetheless, Umple provided an excellent environment for *model-level* code inspections and as enhancements to the model are made, the results will instantaneously be achieved in the code itself; as they are one-in-the-same.

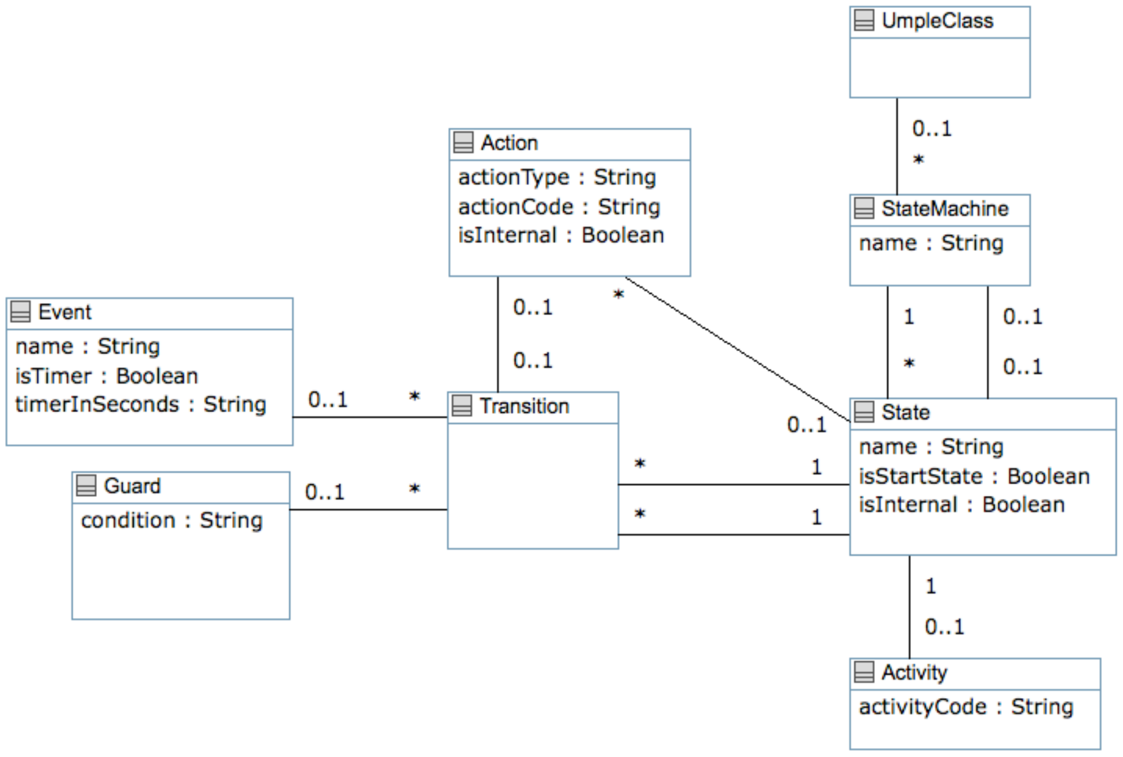


Figure 34: Umple State Machine Meta Model

If we now compare the quality of the code relative to the model of upcoming features in Umple such as state machines (modeled entirely in Umple and is shown above in Figure 34), the models are much more concise, and by virtue of being designed in a model-oriented language are of higher quality with no possibility for translation errors between model and code (as they are one and the same).

We are not stating that a model developed in Umple is without issues. Rather, we are stating that the implementation of an Umple model represents the exact intentions of the model itself with no translation errors, omissions or additions. This is because the mode and code are one in the same without the need for a process to transcribe the model into code; a process that was required in the case the first version of Umple.

Code generated from Umple has similar qualities as code written by hand. We based our code generation templates on the Java code that was originally written by hand when developing the first version of Umple. In the next section we will evaluate Umple's source code generator by analyzing how well the generated code follows the practices of high quality code generators.

7.2.2.2Evaluating the Umple Source Code Generators

Model-based approaches allow for more automation, but the complete and cost-efficient implementation is very difficult to achieve [160]. Some manual coding and manual verification is still required because MDA technologies currently do not generally provide an efficient approach to generating the *entire* system and because most software re-use will be from non MDA systems not written within the context of (i.e. not compatible with) MDA [160].

The code generation approach in "Real-time Java from an automated code generation perspective" [160] compensates for the issues identified above; but a major limitation in our view is that the extension mechanisms to support arbitrary code impose unnecessary complexity into the system. In essence, a *class* in a class diagram would result in two implementation classes; a super class that contains all of the generated code and a sub class that can be augmented with developer code (outside of the system).

Umple provides a simple yet effective approach to both issues above; first Umple provides several mechanism to *write all code in the model*, and second Umple explicitly supports the integration of external artefacts; providing an in-model bridge to support code re-use.

In addition to describing the code generation techniques, [160] also defines the framework under which source code generators can be evaluated. Below we provide an overview of how well Umple compares to the code generation requirements.

7.2.2.3(R1) The Code Generation Should be Easy to Validate

This requirement is broken into three sub-categories

- R1.1: The generated code should be compact
- R1.2: Model-to-code traceability should be direct

- R1.3: Code generator development tools should allow easy model navigation

Generated Umple code is of similar complexity to code written by hand (R1.1). The approach used for code generation mimicked the implementation of the Umple system itself (and was then later replaced by Umple). Umple's translation to code is direct with no additional hierarchical constraints or support classes required. Requirement 1.2 was verified using automated unit tests, which was discussed in Section 7.1 . The suite of Umple tools does provide some visualization to navigate models, but this area has been identified as a candidate for future work (R1.3).

7.2.2.4(R2) Separation of Concerns

This requirement is broken into three subsections.

- R2.1: Separating generated from manually-written code should be easy
- R2.2: The separation between functional and non functional semantics should be maintained
- R2.3: The integration of concurrent and sequential semantics should be easy

To achieve R2.1, the code generator typically uses an inheritance structure where the super class contains the generated code based on the model and a stubbed-out (i.e. un-implemented) sub-class is generated to allow for manually written code. Umple allows *manually-written* code to be provided within the model itself, making R2.1 readily satisfied. This approach also makes the generated code less complex; as the underlying representation of a class-level model is now just a base-language class.

But Umple does provide a separation using source code comments to provide the distinction between code generated due to the model and additional code written by the modeller.

7.2.2.5(R3) The Generated Code Should be High-Quality

Despite the mantra that generated code need not be inspected (just like you would not inspect byte-code or machine code generated from traditional compilers), the quality of the generated code remains important as it must still conform to industry coding standards and it might be verified, maintained or controlled manually.

This requirement is subjective by nature. However, Umple aims to provide generated code that looks like code written by hand. An empirical evaluation of the generated code has been delegated to future work, but as mentioned, the generated version of Umple was based on the written-by-hand first version.

7.2.2.6(R4) Expressiveness of Target Programming Language

This requirement is broken into two sub-requirements.

- R4.1: The concurrency semantics of the target programming language should be expressive and fully encompass the computational model of choice
- R4.2: The target language should support OO semantics

Support for concurrency is being addressed by other researchers with the upcoming implementation of state machines into Umple's syntax.

Umple's expressiveness is explicitly based on the target programming language and is therefore satisfied as Umple is based on both Java, PHP and Ruby; three languages that provide OO semantics and they encompass the computational requirements of the models described in Umple.

7.2.3 Analyzing LOC of Umple (Java Versus Generated Java)

The breakdown of lines of code of Umple v1.0 is shown below in Table 36.

Table 36: Umple v1.0 Java Code

	LOC	% Total LOC
All Java Code	12,938	100%
Java code written by hand	5,481	42%
Java code generated by Umple (which did not yet exist)	0	0%
Java code generated by grammar (from 701 grammar LOC)	7,457	58%

The first version of Umple, written completely in either Java or as a BNF-style grammar, was around 13 KLOCs, excluding template code and test code (see next paragraph). Of that, 58% was generated by the Antlr parser code-generator and 42% was written *by hand*.

We have explicitly excluded our templating code to translate Umple into other base languages for simplicity. This templating code is a form of data, written using JET Templates [79] to describes what the output of the compiler should look like, and the templates are compiled into Java code. The JET package is currently outside of Umple's domain. It was never the intention to *model* our code generation templates in Umple, as JET already provides an efficient mechanism to build a code generator. We also did not consider our test code for this analysis; as the amount of test code written depends on one's software process and in particular one's attitudes towards automated testing.

We believe that comparing v1.0 of Umple to v1.3 is fair, since Umple was not available when v1.0 was written, yet v1.3 has similar requirements and features. Our analysis below should therefore provide reasonably unbiased evidence as to whether an Umple system has more, less, or the same amount of code compared to a Java implementation of the same system.

Our transition from Umple v1.0 to Umple v1.3 included the following changes:

- Almost half of the manually written code was converted into Umple code.
- Our parser and grammar code were also converted to Umple code. This change was brought about to deal with limitations encountered using Antlr [80] (a discussion of which is outside the scope of our work).
- We followed a Test-Driven approach to development, resulting in significantly more test code, but as discussed above, we have ignored that issue, since we could just as well have had the test code in v1.0.

The breakdown of lines of code of Umple v1.3 is shown below in Table 37.

Table 37: Umple v1.3 Java Code

	LOC	% Total LOC
All Java Code	9,449	100%
Java code written by hand	3,103	33%
Java code generated by Umple (from 1998 Umple LOC)	3,280	35%
Java code generated by grammar (from 469 grammar LOC)	3,066	32%

The Umple code written for v1.3 replaced about 43% of the manually written Java. Table 38 compares the amount of manual code written in v1.0 to the amount of manual plus generated code from v1.3.

Table 38: Comparing Umple Generated Code

	v1.0	v1.3	% Change
Java code written by hand	5,481	3,103	-43%
Java code generated by Umple	0	3,280	--
Total LOC of Java	5,481	6,383	+16%

From the above, we see the *generated* system in v1.3 had an additional 16% more lines of code than v1.0, but as with code from any compiler, this should not need to be seen by software developers. Part of the excess code is the generation of methods in the API which might not actually be used anywhere; some of this would have been omitted in the hand-written code. The difference seems also within reason if you consider that development between v1.0 and v1.3 included bug fixes, language enhancements, and overall maintenance.

Most importantly, over half of the Java code in the running system was generated by Umple. Total manually written code in v1.0 (excluding tests, templates and grammar code) was 5481 lines of Java. Total manually written code in V1.3 was 5101 (3103 lines of Java and 1998 lines of Umple). This represents a very conservative savings of 7%. Savings would be greater if the addition of features and improvements to the grammar were taken into account.

One more consideration is that for v1.0 we also maintained a UML model. For v1.3 we no longer needed to maintain this UML model; the Umple code *was* the model, and if we wanted a diagram we could just display it.

In the following sections we provide additional examples comparing the size of Umple based systems and their generated Java, PHP and Ruby equivalents.

7.2.4 Comparing LOC of Systems Written in Umple

In the previous section, we saw that the code generated by Umple was roughly similar to source code that would be generated by hand. We compared two relatively similar version of Umple in terms of feature set, one of which was partially migrated to Umple. In this section, we will look at five systems built using Umple and compare the Umple version versus the base language version.

The five systems being analyzed include:

- Umple (v1.10): The latest version of the Umple tooling written in Umple+Java
- Elevator (Java): An elevator simulation written in Java, details are provided below.
- Elevator (PHP): The Umple+Java simulation re-written in Umple+PHP
- Schedule Management: A management tool for a local restaurant chain written in Umple+PHP. Details of the design of the system will be discussed in Section 7.3.3 .
- Distance Learning Reporting: A reporting tool for a computer based training program written in Umple+PHP. Details of the design of the system will be discussed in 7.3.4 .

The latest version of Umple (v1.10) has been completely refactored into the Umple language and includes many additional features such as code injections, additional software patterns, and state machines.

The simple elevator simulator had two versions: one was written in Java using the Swing user interface toolkit and the second was a version of the same system ported to PHP using HTML. The PHP implementation is available for demonstration at [161].

Table 39 provides an overview of the sizes of the five systems. The second column is the 'pure' Umple; i.e. declarations of classes, attributes, associations, state machines etc. The third column is the code for the methods that are embedded in the Umple programs; these are written in the syntax of the base language, but are considered to be an integral part of the Umple code. The fourth column is code that is manually written, is separate from the Umple code, and is compiled by the base language compiler.

Apart from Umple itself (which is 100% written in Umple), all other systems were about ¼ Umple and ¾ custom code. This discrepancy is due to the nature of Umple. The elevator systems, and management tools all had user interfaces that are better written natively in the UI toolkit of the base language (Swing/SWT for Java and HTML for PHP). Conversely, Umple's UI is a set of command line tools and did not require a separate toolkit. Please note that an Umple system contains modeling constructs like classes, attributes, associations and software patterns (LOC Umple Models) as well as embedded base language code (LOC Umple Embed Code). The % of Umple code represents the sum of both Umple model and embedded code.

Table 39: LOCs of Umple Based Systems

Application	**LOC Umple Model Code (A)**	**LOC Umple Embeded Code (B)**	**LOC Hand Coded Java/PHP (C)**	**Total LOC (D)**	**% Umple (A+B)/D**
Umple v1.10	342	10,246	0	10,588	100%
Elevator (PHP)	44	195	214	453	53%
Elevator (Java)	44	139	678	861	21%
Schedule Management (PHP)	35	231	646	912	29%
Distance Learning Reporting (PHP)	174	2,570	6,628	9,372	29%

The amount of Umple represented in each system above (i.e. an average of about 25%) is not necessarily representative because of a somewhat unfair comparison. As was observed in Section 7.2 systems written in Umple can be more concise than if written in their native base language. To take the potential for efficiency gains by using Umple, Table 40 considers the LOC of the same systems above, but considers the amount of *generated* code to provide a more accurate understanding of the weight the Umple model has on the overall size of the system. The amount of generated code is divided into that generated from the model code (column 2), and generated code from the action semantics; i.e. the base language embedded code that is passed through unedited by the Umple compiler (column 3). Columns 3 and 4 are the same as in Table 39.

Table 40: LOCs of Generated Java/PHP Based Systems

Application	LOC Gen. From Umple Model Code (W)	LOC Gen. From Embeded Code (X)	LOC Hand Coded Java/PHP (Y)	Total LOC (Z)	% Umple (W+X)/Z
Umple v1.10	5,737	10,246	0	15,983	100%
Elevator (PHP)	930	195	214	1,339	84%
Elevator (Java)	902	139	678	1,719	61%
Schedule Management (PHP)	128	231	646	1,005	36%
Distance Learning Reporting (PHP)	3,867	2,570	6,628	13,065	49%

The percent of code represented by the Umple model increases dramatically for the Elevator system (up to about 84% of the code base in PHP and 61% in Java) and more conservatively for the industrial systems (36% and 49% of the code bases). This difference can be explained by the fact that the Elevator system is not a commercial software system so has a much simpler user interface relative to the schedule management and distance learning application.

The 10-to-1 potential code savings from using Umple that was theoretically possible as shown in Section 7.2 is supported by our results below. For the five applications below, if we consider Umple in isolation the savings ranged from 26% to 57% for the *real system* and as high as 82% for our elevator simulator.

Table 41: Comparing Umple Code To Generated Java/PHP Code

Application	% Imp. From Umple Model (W-A)/W	% Imp. Both Umple Model and Embed (W+X)-(A+B)/(W+X)	% Imp. Over Model, Embed and Hand Code (Z-D)/Z
Umple v1.10	94%	34%	34%
Elevator (PHP)	95%	79%	66%
Elevator (Java)	95%	82%	50%
Schedule Management (PHP)	73%	26%	9%
Distance Learning Reporting (PHP)	96%	57%	28%

Please note that the calculations are based on the columns from Table 39 (columns A,B,C,D), and Table 40 (columns W,X,Y,Z).

We analyzed the amount of code that is *saved* by using Umple. We performed three comparisons. First, we considered the amount of Umple model code compared to the resulting generated code. Second, we looked at all Umple code (model + embedded action semantics) versus all Umple generated code. Finally, we looked at the overall system (model + embedded action semantics + hand-coded base language) compared to the overall generated system. The % improvements are summarized in Table 41.

The amount of actual savings in the context of the overall system varies depending on the amount of application logic required (i.e. the amount of action semantic application code). Because Umple is geared towards describing models, additional code must be written to manage persistence (i.e. database tier) and display the user interface (i.e. UI tier). When taken in the context of the entire system, the savings drop to between 9% and 28% of our industrial systems and 50% to 66% for the elevator example.

For the elevator systems, we see substantially fewer lines of Umple code relative to the amount of generated Java code. This dramatic improvement can be attributed to the following explanations. First, the system provides a fairly simple decision algorithm to control the simulation; therefore little code is required to manage the state of the elevator system. Second, the system is mostly comprised of code for describing and navigating the domain model of an elevator; something which Umple does really well. In Umple you can easily describe Floors, Elevators, and Passengers as well as the relationships between them.

The code improvements for the Schedule Management system were not as dramatic as the Elevator system, but that is to be expected as the schedule management application represents a real application with much more application and UI logic relative to our simple elevator simulator. This application also had a relatively simple model, also resulting in less code savings.

As demonstrated by the applications above, it is likely that systems built in Umple will generally contain substantially fewer lines of code than equivalent systems written in PHP or Java. As discussed earlier, the LOC metric has been shown to be a fair indicator of complexity (despite being a simple metric).

7.3 *Industrial Examples of Umple*

A practical approach to validating our work is to build real software system using the Umple language. We took three approaches to building real and practical examples of Umple. First, we used Umple to build realistic models of several systems. Second, we used our first versions of

Umple (written in Java) to explore refactoring a Java system into Umple. Finally, we worked with several businesses to solve their real-world problems and built production-quality systems using Umple.

7.3.1 Umple as a Modeling Language

To date, we have built the models for the following types of systems:

1. 2D Shapes
2. Access Control
3. Accidents
4. Accommodations
5. Afghan Rain Design
6. Airline
7. Banking System (2 versions)
8. Election
9. Elevator (2 versions)
10. Genealogy (3 versions)
11. Geographical Information Systems
12. Hotel
13. Insurance
14. Inventory Management
15. Library
16. Mail Order
17. Manufacturing Plant Controller
18. Police
19. Political Entities
20. Private Lending System
21. Real Estate
22. Telephone
23. Traffic Lights
24. University
25. Warehouse

The most up-to-date list of examples is available at [3]. The models above were built based on the following resources [2]. The model created for the *Private Lending System* was later used as a design for an application built using Ruby on Rails and is described below.

7.3.2 Private Lender Business Domain

A private lender acted as our domain expert and is in the business of managing higher risk secondary mortgages. The system being built is designed to manage the full life-cycle of a deal, as well as provide timely reporting for both investors and borrowers from the private lender. In this example, Umple was used as a data model and Ruby on Rails was used as the application platform. Below is the data model of a private lending system; displayed as a class diagram (as it appears in the UmpleOnline tool).

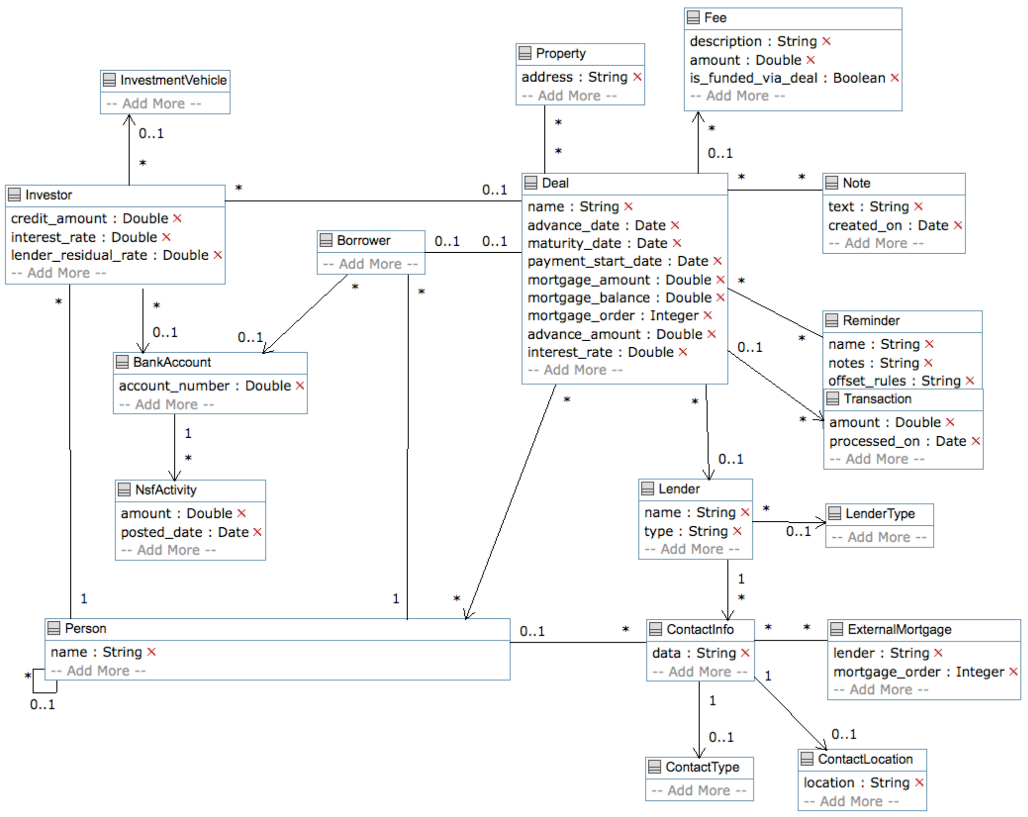


Figure 35: Private Lending Data Model

The diagram view of the model was used as a communication aid as the model was being built and validated with the domain experts. The Umple code to represent the diagram above is shown below.

```
class Deal {
  name; // defaulted to street address of mortgaged property?
  status { Pending, Funded, Open, Default, Closed }

  Date advance_date;
  Date maturity_date;
  Date payment_start_date;
  Double mortgage_amount;
  Double mortgage_balance;
  Integer mortgage_order; //e.g. first mortgage, second mortgage, etc
  Double advance_amount;
  Double interest_rate;

  0..1 -- * Investor;
  0..1 -- 0..1 Borrower;
  * -> 0..1 Lender;
  0..1 -> * Fee;
  0..1 -> * Transaction;
  * -> * Person; // e.g. laywer, accountant, etc
  * -- * Reminder;
  * -- * Note;
}
```

```
class Note {
  text;
  Date created_on;
}

// A simple reminder system to add to a deal
class Reminder {
  name;
  notes;
  offset_rules;  // simple DSL like 1 day after advance_date
  Date notify_me_on; //
  Boolean ignore; // i.e. don't send the reminder anymore
}

// Automatic transaction log stating when payments and collections should be
// made
class Transaction
{
  Double amount;
  Date processed_on;
  type { PaymentToInvestor, PaymentToPrivateLender, CollectionFromBorrower,
PrincipalAdjustment }
  status { Unconfirmed, Confirmed, Declined }
}

class ContactInfo {
  data;
  1 -> 0..1 ContactType;
  1 -> 0..1 ContactLocation;
}

class Lender {
  name;
  type;

  1 -> * ContactInfo;
  * -> 0..1 LenderType;
}

class LenderType {
  type { Internal, External }
}

class ContactType {
  type { Email, Address, Phone, Fax, TTY, IM }
}

class ContactLocation { location; }

class ExternalMortgage {
  lender;
  Integer mortgage_order;
  * -> * ContactInfo;
}

class Fee {
  description;
  Double amount;
  Boolean is_funded_via_deal;
}

// I am thinking instead of having a just Client, Lawyer class, to have a
// generic Person, which could include a lawyer, realtor, etc
// Drop downs could then be based on the type of person
class Person {
  name;
  role { Laywer, Accountant, Banker, Client, Realtor }
  * -> 0..1 Person referredBy;
  0..1 -> * ContactInfo;
}

class Investor {
  Double credit_amount;
  Double interest_rate;
  Double lender_residual_rate;

  * -- 1 Person;
  * -> 0..1 BankAccount;
  * -> 0..1 InvestmentVehicle;
}

class Borrower {
  * -- 1 Person;
  * -> 0..1 BankAccount;
}

class BankAccount {
  Double account_number;
  1 -> * NsfActivity;
}


class NsfActivity {
  Double amount;
  Date posted_date;
}

class InvestmentVehicle {
  name { TFSA, RRSP, Trust, None }
}

class Property {
  address;
  * -- * Deal;
}
```

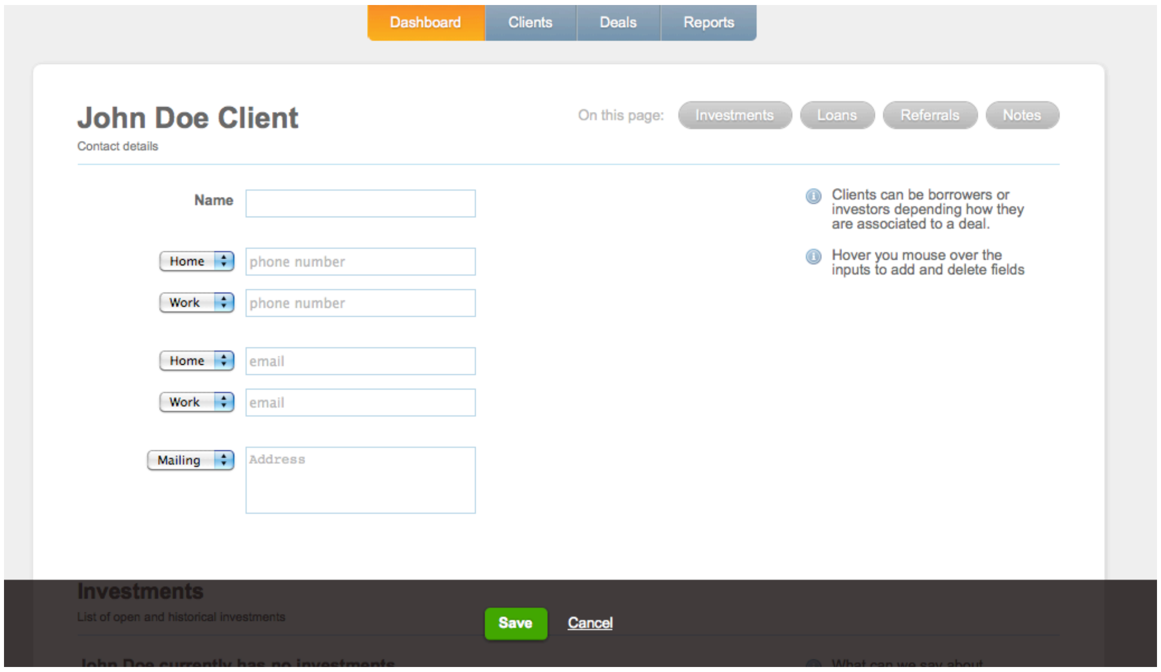
Please note the ease in which comments can be attached to certain aspects of the model without *cluttering* the diagram. These comments acts as developer documentation, as well as to provide some additional information to document the decision making process, as well as provide a means to communicate between developers during development. For example, the comment below was used to justify a change to the model, to which a other developers can respond to or address in future revisions.

```
// I am thinking instead of having a just Client, Lawyer class, to have a
// generic Person, which could include a lawyer, realtor, etc
// Drop downs could then be based on the type of person
```

Other conventions such as TODO, HACK and FIXME can be applied to the textual version of the model and would be integrated into maintenance approaches like Waypoints [162].

It should be noted that the example above uses enumerated types, which are internally modeled as state machines. The display of state machine entities in UmpleOnline is not currently supported in the diagrammatic view of the system and is therefore are not included in Figure 35.

The private lending application was later built using Ruby on Rails, a framework currently not supported by Umple. In this case, Umple was used as a data modeling tool, but our team is looking to provide support for additional languages, such as the Rails DSL, that sits on top of Ruby. A screenshot of the application is shown in Figure 36.



Clients can be borrowers or investors depending how they are associated to a deal.

Hover you mouse over the inputs to add and delete fields

Figure 36: Private Lending Application Screenshot

7.3.3 Schedule Management Web Application

Umple was used to manage schedules for a medium-sized restaurant chain with approximately 250 staff members including servers, bartenders, cooks and managers. This web application incorporates the ability to upload new schedules, allows staff to view those schedules online, as well as provides them with the ability to post to a weekly bulletin board either looking to pick up additional shifts, or drop existing one.

The application was built using Umple, PHP and HTML; it runs on Apache. Below in Figure 37 is the design of the scheduling aspect of the application.

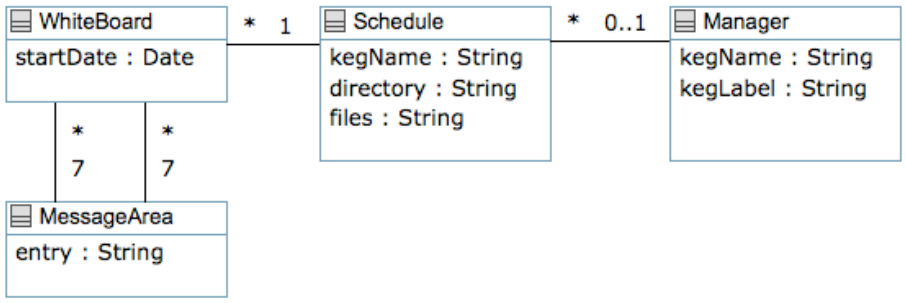


Figure 37: Scheduling Model

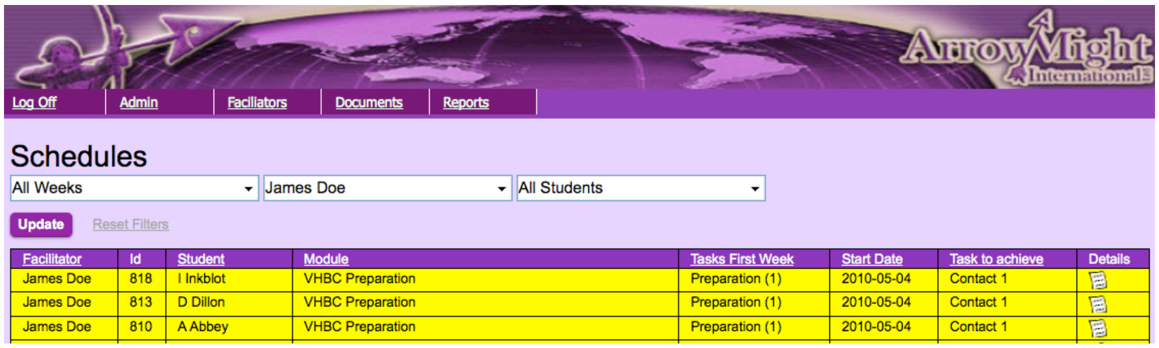
The Umple model for the diagram above is shown below.

```
class Schedule
{
  kegName;
  directory;
  internal files = array();
  1 -- * WhiteBoard;
}

class WhiteBoard
{
  Date startDate;
  * -> 7 MessageArea pickUpShiftMessage;
  * -> 7 MessageArea dropShiftMessage;
}

class MessageArea
{
  String[] entry;
}

class Manager
{
  kegName = null;
  kegLabel = null;

  0..1 -- * Schedule;

}
```

Although the model above is small there was considerable effort saved by using Umple. The model (28 LOC) was combined with an additional 270 LOC of Umple action semantics (i.e. PHP code) resulting in a generated system over 900 LOCs (generated in PHP). Of the generated code, over 600 lines are attributed to the boilerplate code required to implement the model above. Overall, the Umple system represents 67% less code.

7.3.4 Distance Learning Progress Reporting Tool

Umple was used to build a progress reporting tool for several distance learning courses. The application was written using Umple+PHP, with HTML as the user interface and MySQL for storage. A screenshot of the application is shown below in Figure 38.



Log Off | Admin | Facliators | Documents | Reports

Schedules

All Weeks | James Doe | All Students

Update   Reset Filters

Facilitator	Id	Student	Module	Tasks First Week	Start Date	Task to achieve	Details
James Doe	818	I Inkblot	VHBC Preparation	Preparation (1)	2010-05-04	Contact 1	
James Doe	813	D Dillon	VHBC Preparation	Preparation (1)	2010-05-04	Contact 1	
James Doe	810	A Abbey	VHBC Preparation	Preparation (1)	2010-05-04	Contact 1	

Figure 38: Screenshot of Distance Learning Program Reporting Tool

In the business model of this application, a program is made up of several modules, and a module occurs over a set number of weeks. A week has one or more lessons, and a week also has one or more scheduled contacts between a facilitator (similar to a teacher) and a student. During a scheduled contact, the facilitator tracks the answers to several questions like "How is your progress so far?" The UML model is displayed below in Figure 39.

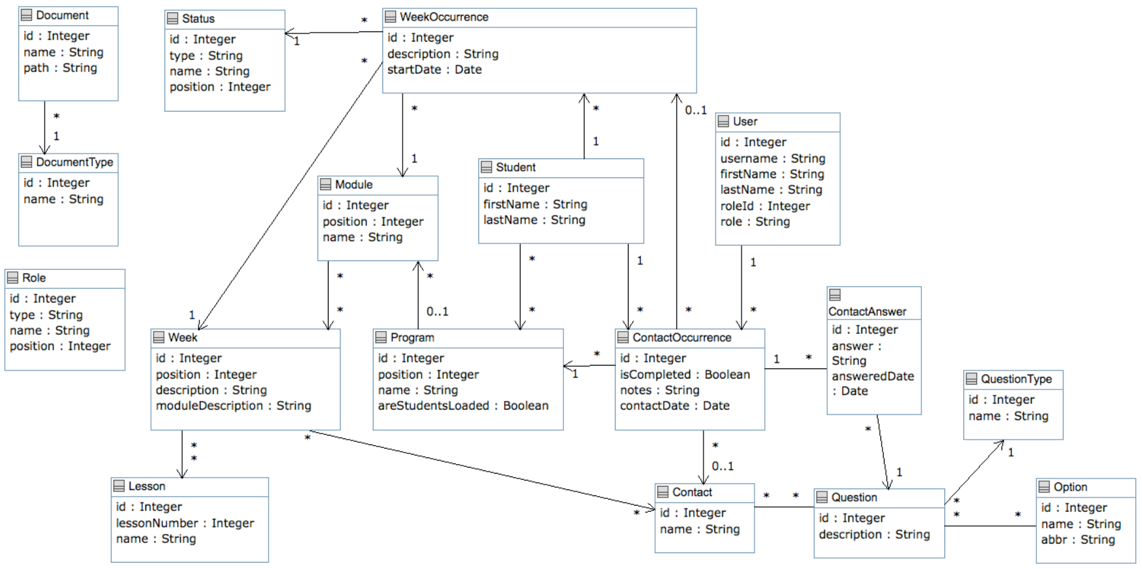


Figure 39: Umple model of a distance learning management application

Because a course can be given at any time, when a student registers, the system uses the week/contact template to create WeekOccurrences and ContactOccurrences which reflect individual contacts between a facilitator and a student.

The Umple model used to generate the diagram in Figure 39 is shown below.

```
class Program
{
  Integer id;
  Integer position;
  name;
  internal Boolean areStudentsLoaded
= false;

  0..1 -- * Module;

  key { id }
}

class Module
{
  Integer id;
  Integer position;
  name;
  * -- * Week;

  key { id }
}

class Week
{
  Integer id;
  Integer position;
  description;
  moduleDescription = null;
  * -- * Lesson;
  * -- * Contact;

  key { id }
}

class Lesson
{
  Integer id;
  Integer lessonNumber;
  name;

  key { id }
}

class Contact
{
  Integer id;
  name;
  * -- * Question;

  key { id }
}

class QuestionType
{
  Integer id;
  name;

  key { id }
}
```

```
class Role
{
  Integer id;
  type;
  name;
  Integer position;

  key { id }
}

class Status
{
  Integer id;
  type;
  name;
  Integer position;

  key { id }
}

class User
{
  Integer id;
  username;
  firstName;
  lastName;
  Integer[] roleId;
  String[] role;

  1 facilitator -- *
ContactOccurrence;

  key { id }
}

class Student
{
  * -- * Program;
  1 -- * WeekOccurrence;
  1 -- * ContactOccurrence;

  Integer id;
  firstName;
  lastName;

  key { id }
}

class WeekOccurrence
{
  Integer id;
  description;
  Date startDate;

  * -> 1 Module;
  * -> 1 Week;
  * -> 1 Status;

  key { id }
}
```

```
class Option
{
  Integer id;
  name;
  abbr;

  key { id }
}

class Question
{
  Integer id;
  description;
  * -- 1 QuestionType;
  * -- * Option;

  key { id }
}

class StudentOverview
{
  * -> 1 Program;
  * -> 0..1 Module;
  * -> 0..1 WeekOccurrence;
  * -> 1 Student;
  * -> 1 User facilitator;
  * -> 1 Status;
}

class ProgramSummary
{
  Integer numberOfStudents = 0;
  Integer numberGraduated = 0;
  ModuleSummary[] moduleSummary;
}

class ModuleSummary
{
  String name;
  Integer numberOfStudents;
}
```

```
class ContactOccurrence
{
  Integer id;
  Boolean isCompleted;
  notes;
  Date contactDate;
  * -> 1 Program;
  * -> 0..1 WeekOccurrence;
  * -> 0..1 Contact;

  key { id }
}

class ContactAnswer
{
  Integer id;
  answer;
  Date answeredDate;

  * answer -- 1 ContactOccurrence;
  * -> 1 Question;

  key { id }
}

class DocumentType {
  Integer id;
  name;

  key { id }
}

class Document {
  Integer id;
  name;
  path;
  * -- 1 DocumentType;

  key { id }
}

class Schedule {
  * -> * ContactOccurrence;
}
```

Please note that the following classes were excluded from the UML diagram: StudentOverview, Schedule, ProgramSummary, and Module Summary. These four classes are secondary to the domain model, but are relevant to the application for summary and reporting features. For example, the Schedule object helps to answer which tasks a facilitator has to accomplish over the upcoming weeks. These objects had several relationships to other classes and would have over-complicated the diagram.

7.3.5 Frequency of Multiplicity Usage in Commercial Umple Systems

In Section 5.2 , we analyzed several open source software systems in an attempt to uncover the types of association used in practice (and in particular the types of association multiplicities in use). We noted the relative difficulty in accurately identifying the multiplicity of association ends in those systems beyond differentiating between *one* and *many*. We then analyzed the multiplicities used in UML diagrams found in textbooks as well as several UML specification documents. In contrast to analyzing source code, it is extremely straightforward to analyze the types of associations in use in UML and in Umple systems because associations are first-class entities of the language and explicitly defined.

In Table 42, we compare the industrial use of associations in Umple compared to earlier analysis of associations found in the UML specifications, in examples from UML books, and finally from our own repository of Umple examples.

We see that the usage of multiplicities reported earlier has some important differences as compared with the usage of multiplicities in the systems built using Umple discussed in this chapter.

Table 42: Frequency of Multiplicity Usage in Umple Based Applications

Industry Use Of Umple			**Rank**		
Multiplicity	**Frequency**	**Ratio (%)**	**In UML Specs**	**Examples in Book**	**In Umple Repository**
1 <- *	10	20.41%	6	N/A	6
* -- *	10	20.41%	4	2	3
1 -- *	8	16.33%	2	1	1
0..1 <- *	8	16.33%	7	N/A	5
0..1 -> *	3	6.12%	N/A	N/A	N/A
* -> *	3	6.12%	3	9	4
0..1 -- *	2	4.08%	1	4	2
0..1 <- 1	2	4.08%	N/A	N/A	N/A
1 -> *	2	4.08%	N/A	N/A	N/A
0..1 -- 0..1	1	2.04%	8	6	N/A
TOTALS	49	100.00%			

The first major difference is that the systems in this chapter made much greater use of directed associations (navigable in one direction only). There are several possible explanations: 1) We may not have correctly identified all associations in the previous work, and may have counted some of them as attributes; Umple just makes it obvious which are associations. 2) The freedom

Umple provides to change the directionality easily may have meant that we were more confident in using a one-way associations, whereas in the systems we examined earlier, bidirectional associations may have been used just in case both directions of navigation were needed.

The second major difference is the prevalence of many-to-many associations. Analagous explanations to those discussed in the last paragraph may apply here too.

7.4 Summary

In this chapter we evaluated the Umple language from various perspective. First, we approached the Umple language and tooling from a systematic approach with regard to quality. We demonstrated a re-usable approach to language design that allows for agile responses to change without overly compromising the need for quality. Next, we analyzed the qualities of an Umple system to demonstrate the potential for improved program comprehension based on the conciseness of the language. Finally, we successfully incorporated Umple into three commercial software projects.

We have demonstrated that Umple has the potential to be a very useful and relevant approach to software engineering.

Chapter 8 Conclusions

Software practitioners and researchers tend to agree that software modeling is considered a good practice in software engineering and a suitable *next step* in applying additional engineering rigor to the practice of software development. However, from our research and experiences, code-centric approaches tend to be the prevalent practice. Our research is motivated by asking *why* this is the case, and *how* to address or improve modeling practices.

Umple is best described as a model-oriented programming language that helps bridge the model-code divide. Below will outline how Umple addresses the issues identified in our first research question (RQ1): "Why do software practitioners resist the current style of software modeling and show a tendency to prefer to design directly in code?"

i. **Habituation:** We argue that Umple is *not* "yet another programming language"; instead, it is an extension to existing base languages including Java, Ruby and PHP. Umple will therefore feel and behave in a way that is familiar to developers since its constructs for modeling elements such as attributes and associations are built on top of their preferred language.

ii. **Efficacy:** Umple maintains strong roots in code-centric thinking with the addition of a model-centric / diagrammatic perspective. By providing an *all-keyboard* solution to software modeling, Umple allows context-assist, short-cuts, and copy-and-paste functionality that code-centric developers are used to. Conversely, Umple dos not abandon the ability to edit diagrams perceived as so important to some. The question of whether text or diagrams are perceived or actually better is now moot; the choice can now be left to developer experience and expertise.

iii. **Politics and Practices:** Another issue that is mitigated by Umple's ability to operate in both code- and model-centric approaches is that an organization's preference towards either is now irrelevant. Umple allows *Java* developers to continue to deliver Java code. It also allows PHP and Ruby developers the same luxury. As more language support is provided for Umple, more software development processes will be able to introduce Umple into their development process without requiring a major shift in underlying processes and tool usage such as application structures, third party frameworks, unit testing, build scripts, continuous integration and bug tracking and software deployment.

iv. **Software Process:** Umple development follows a similar construction path as existing programming languages, but with the advantage that structural prototyping (i.e.

modeling) can be more easily accomplished with fewer lines of code. Umple supports several software development processes such as big-design up-front, iterative/incremental, agile, and evolutionary prototype. Umple can be used to create extensive platform-independent models by relying solely on the modeling constructs of the language and differing in base language implementation. Conversely, a more agile, iterative/incremental approach can be adopted by implementing platform-specific models to address certain features or architectural concerns and then incrementally adding more modeling / coding aspects in tandem. Finally, Umple provides the ability to simulate models [3, 105] to help validate an early prototype.

v. **Tool Weaknesses:** Umple provides different levels of tool integration depending on the type of user. Umple is available online [3], offering a zero-footprint installation. This allows potential Umple developers to *try* the language without having to invest any effort in software installation. Umple includes several command line tools to de-couple the development environment from the compilation process. This allows developers to continue using their preferred text editor such as VI, Emacs, TextMate, etc. Finally, Umple has been deeply integrated with Eclipse with an Umple IDE plugin. Umple has addressed the concern that modeling tools are large and complex. Future work geared towards model level trace and debugging will only further improve support for model-centric development without abandoning Umple's dual code-centric and model-centric roots.

vi. **Intrinsic Utility:** Umple can be developed entirely in the comfort of a text editor without the need to worry about diagrammatic inefficiencies such as layout of the diagram and the amount of mouse 'clicks' required to work in a graphical user interface.

vii. **Software Engineering Education:** UML Class Diagrams can now be taught interactively in any classroom [3]. Teaching UML does not require full-fledged modeling software where features of the tool interfere with concepts of the language (in this case UML). Software educators can use Umple to not only showcase variability in UML models, but they can only show how design decisions affect the underlying implementation code.

viii. **Domain suitability:** Umple has been successfully used in data dominant software as shown in the previous chapter. As more systems are built, it should be clearer in which domains Umple is most suited.

Our second research question (RQ2) asked, “What is the level of modeling adoption in industry, and what are the factors affecting this?” Answers to this question were solicited in the form of a survey and discussed at length in Chapter 2. One prevalent theme from the survey participants was the apparent desire to want (or the feeling of a need) to apply more model-centric approaches to software development. But, most continued development in a code-centric style with very few adopting a model-only approach.

Our proposed solution to help bridge the model-code divide was the development of the Umple model-oriented language. Umple’s approach to modeling is inherently text based with a visual equivalent. We coined the term text-diagram duality where an Umple system can be equally expressed in text as well as in diagrammatic form. By enabling coders to model and modelers to code, as well as enabling the coordination between the two where modelers model and coders code – but in the same language, we have effectively bridged the mode-code divide with a solution that tailors to both software development approaches and both software developer archetypes (model-first versus code-first).

Our final research question (RQ3), “Can programming languages be enhanced with model-oriented constructs and provide benefit to software developers?” helped to frame are research to validate our proposed solution.

To help guide and evaluate our approach to address the model-code divide, we started by creating a taxonomy of software applications and surveyed a number of software practitioners and researchers. This work provided the foundation for our later research and was primarily to answer RQ2 and provide early indications that Umple might succeed at bringing modeling to the forefront of software development.

In Umple, modeling abstractions are incorporated into widely familiar and adopted object-oriented programming languages. At a high level, our approach encapsulates the following concepts:

- Modeling is programming and vice versa. Umple code can be viewed and edited diagrammatically or textually.
- Because models and code are treated uniformly, the need for code generation and reverse engineering is significantly minimized.
- An Umple program can be nothing but UML abstractions. In this case, Umple code is purely a model, with no algorithmic implementations.

To evaluate Umple, we focused on the fact that we were able to produce real commercial products with Umple, as well as on quantitative properties of the language and of the generated system.

Our early focus with Umple was the representation of attributes and associations as first-class entities. We dissected each element to understand how the rich semantics at the modeling level could be (a) represented textually and (b) unambiguously defined for correct execution. We analyzed existing software systems to understand the practice of using attributes and association.

Our next evaluation step was to analyze the Umple program constructs for qualities that promote program comprehension. We uncovered that a system written in Umple can be more concise than similar systems developed in base languages. We found a 10-to-1 code ratio for code saving for platform independent models. For platform specific models (where the action semantics of Umple are the same as the underlying base system) the savings were still considerable at around two-to-one.

Our final evaluation step was to build real systems with the language, to ensure that the foundation of our theory of Umple fit within practical software development. We first migrated the Umple tools to be written in Umple itself, and our experiences led to several techniques for Umple refactoring (known as umplifications) [41]. Later, we worked with three companies where Umple was successfully used to (a) build a new web application for schedule management, (b) enhance an existing online system for tracking distributed learning courses, and (c) model a private lender's business domain which was used later developed in Ruby on Rails.

Future work with Umple is focusing in three domains:

1) Enhancing the Umple language with state machine syntax. By tightly integrating state machines with class diagrams, we hope to further improve the model capabilities within our textual language. Our team's decision to consider state machines as the next model constructs to analyze is based on industry needs and the observation that there seem to be more interesting research opportunities in that area.

2) Empirical studies of software practitioners using Umple. To date we have theoretically demonstrated the benefit of Umple, as well as demonstrated its industrial practicality by building real systems with the language. A next logical step, outside the scope of this thesis, is to evaluate the Umple language and its syntax with a representative set of software practitioners. The output of such studies would help refine the current syntax, identify any

cognitive gaps between the language's syntax, semantics and the pragmatics of the developer's mental model.

3) Improve Umple tools to support better IDE integration including model-level debugging and trace analysis. Umple relies heavily on the compilers and interpreters of supported base languages during Umple *compilation*. This approach effectively forces developers to work at the modeling level but debug at the level of generated code.

Other potential areas of future work include: Investigating code generation optimizations for specific platforms (e.g. deploying an Umple system to a mobile handset versus a large JSEE cloud computing platform). Investigating the integration of automatic refactoring algorithms to deal with consistent refactorings amongst the structural aspects of Umple code, and the algorithmic action semantics (i.e. if I change the name of an attribute, then API calls for that attribute also should be updated).

Despite work with Umple being far from complete, the contributions presented in this thesis lay the necessary foundation for the future of Umple. In summary, the contributions of this thesis include:

- A taxonomy of software applications that can be used in empirical studies to help provide application context when analyzing experiment results [18]
- A survey of software practitioners providing a quantitative perspective on software practitioners attitudes towards [19, 163]
- The implementation and analysis of integrating model-oriented features into existing programming languages; resulting in the Umple language [36, 41, 163]
- An online modeling environment [3] which can be used (a) to introduce people to Umple and popularize it, (b) to create UML diagrams for publications such as this thesis, (c) for software engineering teaching, and (d) as a light-weight development environment.
- The development of four significant systems built using Umple to demonstrate the effectiveness of Umple. The first of these was Umple itself, which was refactored from the first Java-only version; the other three were commercial systems.
- The development of a large online repository of modeling examples available at [3]

Glossary

Action Semantics: Part of the specification of the semantics of a programming or modeling language and is composed of syntax for defining entities, operations and their interactions. In other words, the parts of a programming language responsible for *doing* stuff.

Active Record: [109] An object-relationship pattern and has several concrete implementations. The Active Record pattern is implemented in many languages such as Ruby (on Rails) [110], PHP (symfony), and Python (Elixir).

Advice: [115] The action taken by an aspect at a particular join point and can occur either "before" the join point, "after", or instead of (called "around").

Ant: An XML-based build tool maintained by the Apache that is geared towards building Java projects.

Application Programming Interface (API): A set of capabilities (i.e. methods, attributes, etc) expressed as an interface that enables interaction with other software systems.

Aspect: [115] The modularization of a concern and is composed of pointcuts, advice bodies, inter-type declarations, and possibly classes and methods.

Association: A relationship between one or more classes describing the references or links that will exist at run time between instances of the class or classes. An association can be named, and the ends can be adorned with role names, ownership, multiplicity, and visibility; among other properties.

Attribute: A simple property of an object such as age, name, or date of birth. Attributes should be contrasted with associations, which represent relationships among objects. The data in an attribute does not have any reference back to the object containing the attribute.

Boilerplate Code: Near duplication of sections of code with only slight systematic alterations. Boilerplate code is sometimes referred to as a template implementation, where the code structure is defined with placeholders for each particular case. Examples include set/get methods, add/remove methods, and standardized implementations for software patterns in general (like Singleton). The duplication is sometimes necessary depending on the characteristics of the underlying language, and is not necessarily a sign of poor quality code. Boilerplate code can be reduced using macros, meta-programming, code generation and / or "convention over configuration".

Cardinality: The number of ends participating in a link of an association at run time. In contrast, multiplicity refers to the potential range of cardinalities.

Code-Centric: An approach to software development where practitioners write the code for a system entirely in a (usually textual) programming language; the code is then compiled and linked with libraries to create a working system.

Convention over configuration: A strategy for software design to decrease the variability of design choices and instead relying on an accepted convention, which can lead to simpler systems by exploiting the design assumptions. The process includes conventions relating to naming classes, variables and methods, database design (which also includes naming conventions), project hierarchy (e.g. grouping like entities in a similar location), and many more.

Eclipse Modeling Framework (EMF): A modeling framework and code generation tool based on the Eclipse platform.

EclipseUML (or EclipseUML2): A plugin to Eclipse that allows the creation of UML2 diagrams.

Ecore: The underlying metamodel (written in XMI) that is used for describing EMF models. It is tightly aligned with Meta-Object Facility (MOF).

Emfactic: [85] A text-based editor for managing Ecore models that uses a Java-like programming syntax as opposed to editing the Ecore XMI directly, or building the Ecore model using a visual editor.

Eugenia: [88] A tool that enhances Emfactic / Ecore models with graphical editor annotations to allow to the generation of GMF tools based solely on the annotated Ecore.

Graphical Editing Framework (GEF): An Eclipse-based framework that provides code generation of rich graphical editors for the Eclipse platform including view components, tool palettes and request/commands for editing.

Graphical Modeling Framework (GMF): An Eclipse framework that provides tool generators for building graphical editors that acts as a bridge between tools based on EMF and GEF models. This tool is used to build tools to graphically edit EMF/GEF tools (e.g. to create UML modeling tools based on the UML2 EMF model).

Human-Usable Textual Notation (HUTN): [67] An OMG standard for storing models in a more human understandable format (somewhat as an alternative to XMI). HUTN uses C-like syntax of curly braces instead of XML start and end tags.

Intentional Association: [9] An approach to constructing associations that does not use bag or set semantics to define association ends. Instead, it is defined as an intention of the model which then designates how to derive the collection from the links.

Join point: [115] A point of execution in a program (i.e. method execution or exception handling).

Likert scale: Commonly used in questionnaires where respondents provide their level of agreement to a certain statement or scenario (e.g. highly disagree to highly agree). Similar, but not synonymous to a rating scale.

Mock Objects: An approach to isolated unit testing of an object by simulating the responses of subsidiary objects (i.e. objects that are required due to the design of the system but not currently under test) to limit the affect of these secondary objects on the proper testing of the object under test.

Model-Centric: An approach to software development where the system is instead largely generated from more abstract models created using modeling languages.

Model-Code Divide: The divergence of both practice and attitude towards model-centric and code-centric approaches to software development. The practice of software development appears to be dominated by code-centric approaches whereas a majority opinion is that model-centric approaches are now a 'best practice'.

Model-Driven Development (MDD): A software development methodology based on creating models describing the problem/solution domain, which in turn generate (or support) the creation of source code. Similar terms are model-driven software development (MDSD), model-driven architecture (MDA), and model-driven engineering (MDE) [164].

Meta-Object Facility (MOF): A metamodeling architecture to define UML that is divided into four layers including: M3 (meta-meta model) models to build metamodels; M2 models, which are the result of M3 modeling (i.e. the UML meta model); M1 models, which are the result of using an M2 model (i.e. a particular design using UML); and M0 (data), the real-world objects.

Mix-in: Some code that can be logically incorporated in different places, as though it had been physically copied there, but without the maintenance issues associated with that. It is a similar to an interface with implementation methods and attributes being integrated into the class, as opposed to using inheritance. The concept of the mix-in was first popularized in the Ada language [165]. It allows independently developed code to be injected into a set of classes, and thus supports composition of a system.

Model: See 'software model'.

Multiplicity: A constraint placed on an association end that defines the upper and lower bounds on the number of *links* between objects at run time that can participate in the association relationship. It answers the questions, how many of something, and is it mandatory. It should be contrasted with 'cardinality', which is the *actual* number of links at run time.

Naked Objects: [44] An architectural pattern using software construction whereby the user interface is a direct representation of the domain objects; which in direct completely capture all business logic. It was introduced 2004.

Object Identity Crisis: [141] The crisis is the problem of providing clean, efficient semantics for "object identity" in a programming language.

Papyrus: An open source UML2 modeling and code generation tool based on Eclipse. It is a component of the Model Development Tools (MDT) subproject that provides integrated, user-consumable environment for editing UML (and other related) models.

PHP (Hypertext Pre-processor): A general-purpose scripting language geared towards web development created in 1995.

Platform-Independent Model (PIM): A model of a system that makes no specific reference to the underlying technology platform required to implement the system (i.e. platform agnostic).

Platform-Specific Model (PSM): A model of a system that has references to an underlying technology platform such as programming language, operating system, prototype, or database.

Pointcut: [115] A predicate that matches one or more join points and is associated with certain *advice* to execute. For example, the predicate of a method (the pointcut) with a certain name (join points) is associated with a before execution (advice).

Referential Integrity: A property whereby the bi-directionality of bi-directional references between objects is maintained throughout the life-cycle of a relationship. For example, to ensure referential integrity in a bi-directional association, if an object A points to object B, then when this link is deleted, references stored in both the A and the B end must be deleted.

Rational Software Modeling Tools (RSx): A generic reference to IBM's suite of software modeling tools including Rational Software Architect (RSA) and Rational Software Modeller (RSM). These tools integrate UML modeling with C++/Java development environment supporting round trip engineering between model and code.

Software model: An artefact that represents an abstraction of the software system being built. A model can typically be viewed as a set of diagrams and/or pieces of structured text. It can be recorded on a white board, paper, or using a software tool. A model could use formal syntax and semantics but this is not necessary.

Specification Description Language (SDL): [98] A programming/modeling language aiming to unambiguously describe the behaviour of a system. Originally applied to telecommunication systems, it has been broadened to deal more generally with process control and real-time applications.

State machine: A model defining the behaviour of an entity based on a finite number of states, transitions between those states based on events, and the actions or activities that occur in the system as the entity changes states (e.g. entry and exit actions, or activities while in a state)

Taxonomy: (1) The process of hierarchical classification based on a predefined criteria, scale, or constraint. (2) The model or document resulting from such a process.

Test Driven Development (TDD): A software technique that uses what is known as *Red-Green-Refactor* approach to software construction. The technique dictates that the specification of the software *bit* being built (such as a new class, interface, methods, etc) is written as an automated unit test prior to development the system to meet that specification. In other words, developers write failing tests first, then implement the necessary code to make the tests pass and then work to refine and improve the implementation (i.e. refactor the code) without breaking those automated tests.

Text-Diagram Duality: An observation whereby the underlying representation of an abstraction such as a model can be equally expressed and manipulated both textually and diagrammatically. This observation allows for programming and modeling languages to converge into model-oriented software development where the code is the model (a.k.a. diagram) and the model is the code.

TextUML: [73] A UML2 modeling and diagram generation tool based on Eclipse.

Unified Modeling Language (UML): A standardized general-purpose modeling language that is managed by the Object Management Group.

UML2 Tools: A set of GMF-based editors that allow viewing and editing UML models.

Umlet: An open source UML editor.

Umple: A model-oriented programming language and adds additional modeling elements as first-class entities on top of existing object-oriented languages like Java, PHP and Ruby. Umple supports class diagram entities like associations, attributes and multiplicity, it supports state machine entities like states, events, transitions as well as software patterns like singleton, equality, software mix-ins and aspect-oriented code weaving.

Violet: An open source UML editor.

XML Metadata Interchange XMI: An OMG standard for storing and exchanging metadata and is written in XML. Most commonly used for exchanging UML models.

Xtext: A framework for building programming languages and domain specific language (DSL) editors. Xtext is meant to be the EMF for IDEs.

References

[1] Bourduas, S. "Generation of SDL Specifications from UML and MSC use Cases". *Concordia University. Thesis (Master's),* 2001. Available: http://spectrum.library.concordia.ca/1447/

[2] Lethbridge, T. C. and Laganière, R. *Object-Oriented Software Engineering: Practical Software Development using UML and Java.* New York, NY, USA: McGraw-Hill, 2005.

[3] Forward, A. " UmpleOnline", accessed 2010, http://cruise.site.uottawa.ca/umpleonline/.

[4] Budinsky, F., Brodsky, S. A. and Merks, E. *Eclipse Modeling Framework.* Pearson Education, 2003.

[5] Balz, M., Striewe, M. and Goedicke, M. "Continuous Maintenance of Multiple Abstraction Levels in Program Code," in *Future Trends of Model-Driven Development - FTMDD,* 2010. pp. 68-79.

[6] Vaccare Braga, Rosana T. and Marchesini, Rodrigo H. R. "Implementing Relationships among Classes of Analysis Pattern Languages using Aspects," in *RAOOL '09: Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages,* 2009. pp. 9-16.

[7] Harrison , W., Barton, C. and Raghavachari, M. "Mapping UML Designs to Java". 2000. *ACM SIGPLAN Notices,* vol 35, ACM New York, NY, USA. pp. 178-187.

[8] Jifeng, H., Liu, Z., Li, X. and Qin, S. "A Relational Model for Object-Oriented Designs". 2004. *Lecture notes in computer science,* Springer. pp. 415-436.

[9] Miliev, D. "On the Semantics of Associations and Association Ends in UML". 2007. *IEEE Trans. Software Eng.,* IEEE Computer Society. pp. 231-258.

[10] Rupakheti, C. R. and Hou, D. "An Empirical Study of the Design and Implementation of Object Equality in Java," in *CASCON,* 2008. pp. 9.

[11] Mohagheghi, P. and Dehlen, V. "Where is the Proof? - A Review of Experiences from Applying MDE in Industry," in *ECMDA-FA '08: Proceedings of the 4th European Conference on Model Driven Architecture,* 2008. pp. 432-443.

[12] Forward, A. and Lethbridge, T. C. "Problems and Opportunities for Model-Centric Versus Code-Centric Software Development: A Survey of Software Professionals," in *MiSE '08: Proceedings of the 2008 International Workshop on Models in Software Engineering,* 2008. pp. 27-32.

[13] Levkowitz, H., Holub, R. A., Meyer, G. W. and Robertson, P. K. "Color Vs. Black-and-White in Visualization". *Proceedings of the 2nd Conference on Visualization (VIS),* pp. 336-339, 1991.

[14] Brestovansky, D. "Exploring Textual Modeling using the Umple Language". *University of Ottawa,* 2008. Available: http://www.site.uottawa.ca/~tcl/gradtheses/dbrestovansky/

[15] Böhm, C. and Jacopini, G. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". 1966. *Commun ACM,* vol 9, ACM. pp. 366-371.

[16] Owe, O., Krogdahl, S. and Lyche, T. *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl.* 1st ed., vol. 2635, Germany: Springer, 2004,

[17] Sebeok, T. A. *Current Trends in Linguistics.* 2nd ed., vol. III, Paris: Mouton, The Hague, 1970, pp. 541.

[18] Forward, A. and Lethbridge, T. C. "A Taxonomy of Software Types to Facilitate Search and Evidence-Based Software Engineering," in *CASCON '08: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research,* 2008. pp. 179-191.

[19] 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

[20] Forward, A. " Computer Science PhD Thesis, Appendices, and Supplementary Material", accessed 2008, http://www.site.uottawa.ca/~tcl/gradtheses/aforwardphd/.

[21] France, R. and Rumpe, B. "Model-Driven Development of Complex Software: A Research Roadmap," in *FOSE '07: 2007 Future of Software Engineering,* 2007. pp. 37-54.

[22] Lavaggno, L., Martin, G. and Selic, B. *UML for Real: Design of Embedded Real-Time Systems.* Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[23] Hertel, G., Niedner, S. and Herrmann, S. "Motivation of Software Developers in Open Source Projects: An Internet-Based Survey of Contributors to the Linux Kernel". 2003. *Research Policy,* vol 32, pp. 1159-1177.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] Hannay, J. E., Hansen, O., Kampenes, V. B., et al. "A Survey of Controlled Experiments in Software Engineering". 2005. *IEEE Trans. Software Eng.,* vol 31, pp. 733-753.

[32] 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.

[33] Tzitzikas, Y., Spyratos, N. and Constantopoulos, P. "Mediators Over Taxonomy-Based Information Sources". 2005. *The VLDB Journal,* vol 14, Springer-Verlag New York, Inc. pp. 112-136.

[34] Tukey, J. W., "Data analysis and behavioral science or learning to bear the quantitative's man burden by shunning badmandments," in *The Collected Works of John W. Tukey.* , vol. III, L. W. Jones, Ed. Wadsworth, Monterey CA: 1986, pp. 187-389.

[35] Harris, T. " YUML", accessed 2009, http://yuml.me/.

[36] Forward, A., Badreddin, O. and Lethbridge, T. C. "Umple: Towards Combining Model Driven with Prototype Driven System Development," in *IEEE International Symposium on Rapid System Prototyping (RSP),* 2010.

[37] Medvidovic, N., Egyed, A. and Rosenblum, D. S. "Round-Trip Software Engineering using Uml: From Architecture to Design and Back," in *Proceedings of the Second International Workshop on Object-Oriented Reengineering (WOOR'99), Toulouse, France,* 1999. pp. 1-8.

[38] 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.

[39] Starr, L. *Executable UML: How to Build Class Models.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[40] Hen-Tov, A., Lorenz, D. H., Pinhasi, A. and Schachter, L. "ModelTalk: When Everything is a Domain-Specific Language". 2009. *IEEE Softw.,* vol 26, IEEE Computer Society Press. pp. 39-46.

[41] Lethbridge, T. C., Forward, A. and Badreddin, O. "Umplification: Refactoring to Incrementally Add Abstraction to a Program," in *Working Conference on Reverse Engineering,* 2010. pp. 220-224.

[42] Miller, J. "What UML should be". 2002. *Commun. ACM,* vol 45, pp. 67-69.

[43] Selic, B. "The Pragmatics of Model-Driven Development". 2003. *IEEE Softw.,* vol 20, IEEE Computer Society Press. pp. 19-25.

[44] Pawson, R. and Matthews, R. "Naked Objects: A Technique for Designing More Expressive Systems". 2001. *j-SIGPLAN,* vol 36, pp. 61-67.

[45] Pawson, R. "Naked Objects". 2002. *IEEE Software,* vol 19, pp. 81-83.

[46] Spinellis, D. "On the Declarative Specification of Models". 2003. *IEEE Software,* vol 2, IEEE Computer Society. pp. 95-96.

[47] Lange, C., Chaudron, M. R. V., Muskens, J., Somers, L. J. and Dortmans, H. M. "An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs," in *Workshop on Consistency Problems in UML-Based Software Development, 6th International Conference on Unified Modeling Language, UML,* 2003. pp. 26-34.

[48] Treude, C., Berlik, S., Wenzel, S. and Kelter, U. "Difference Computation of Large Models," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering,* 2007. pp. 295-304.

[49] Alanen, M. and Porres, I. "Difference and Union of Models". 2003. *Lecture Notes in Computer Science,* Springer. pp. 2-17.

[50] Altmanninger, K., Kappel, G., Kusel, A., Retschitzegger, W., Seidl, M., Schwinger, W. and Wimmer, M. "AMOR-Towards Adaptable Model Versioning," in *Proc. of the 1st International Workshop on Model Co-Evolution and Consistency Management,* 2008.

[51] Grönniger, H., Krahn, H., Rumpe, B., Schindler, M. and Völkel, S. "Text-Based Modeling," in *International Workshop on Language Engineering (ateM),* 2007. Available: http://megaplanet.org/atem2007/ATEM2007-22.pdf

[52] Petre, M. "Why Looking Isn't always Seeing: Readership Skills and Graphical Programming". 1995. *Commun ACM,* vol 38, ACM. pp. 33-44.

[53] Gogolla, M., Büttner, F. and Richters, M. " USE - A UML-Based Specification Environment", accessed 2010, http://www.db.informatik.uni-bremen.de/projects/USE.

[54] Jacobson, I. and Cook, S. "The Road Ahead for UML". vol. May, 2010.

[55] Cook, S. " Future Development of UML RFI", accessed 2010, http://www.omg.org/cgi-bin/doc?ad/2008-12-12.

[56] Miller, J. and Mukerji, J., "MDA Guide Version 1.0.1". Object Management Group (OMG), 2003.

[57] Skene, J. and Emmerich, W. "Specifications, Not Meta-Models," in *GaMMa '06: Proceedings of the 2006 International Workshop on Global Integrated Model Management,* 2006. pp. 47-54.

[58] Object Management Group (OMG). " Concrete Syntax for a UML Action Language RFP", accessed 2010, http://www.omg.org/cgi-bin/doc?ad/2008-9-9.

[59] Object Management Group (OMG). " Semantics of a Foundation Subset for Executable UML Models", accessed 2010, http://www.omg.org/spec/FUML/.

[60] 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.

[61] Mentor Graphics Corporation. " Concrete Syntax for a UML Action Language, Action Language for Foundational UML", accessed 2010, http://lib.modeldriven.org/MDLibrary/trunk/Applications/Alf-Reference-Implementation/doc/.

[62] Object Management Group (OMG). " Object Management Group, Unified Modeling Language (UML), Version 2.1.2", accessed 2008, http://www.omg.org/technology/documents/formal/uml.htm.

[63] Oracle. " JavaServer Faces Technology", accessed 2010, http://java.sun.com/javaee/javaserverfaces/.

[64] Zend Technologies Ltd. " Zend Framework", accessed 2009, http://framework.zend.com/.

[65] Anonymous " Hibernate: Relational Persistence for Java and .NET", accessed 2009, http://www.hibernate.org/.

[66] Object Management Group (OMG). " MOF 2.0 / XMI Mapping Specification, v2.1.1", accessed 2009, http://www.omg.org/technology/documents/formal/xmi.htm.

[67] Object Management Group (OMG). " Human-Usable Textual Notation", accessed 2010, http://www.omg.org/technology/documents/formal/hutn.htm.

[68] avishn. " ModSL - Text-to-Diagram UML Sketching Tool", accessed 2010, http://code.google.com/p/modsl/.

[69] Fadila, A. and Said, G. "A New Textual Description for Software Process Modeling". 2006. *Information Technology Journal,* vol 5, pp. 1146-1148.

[70] Fliedl, G., Kop, C. and Mayr, H. C. "From Textual Scenarios to a Conceptual Schema". 2005. *Data Knowl Eng,* vol 55, pp. 20-37.

[71] Bock, C. "UML without Pictures". 2003. *IEEE Software,* vol 20, pp. 33-35.

[72] Steel, J. and Raymond, K. "Generating Human-Usable Textual Notations for Information Models," in *Proceedings of the Fifth International Conference on Enterprise Distributed Object Computing (EDOC 2001), Seattle, Washington, USA,* 2001.

[73] Chaves, R. " TextUML", accessed 2009, http://abstratt.com/.

[74] Demeyer, S., Ducasse, S., Tichelaar, S. and Tichelaar, E. "Why Unified is Not Universal: UML Shortcomings for Coping with Round-Trip Engineering". pp. 630-644, 1999.

[75] Horstmann, C. and Pellegrin, A. d. " Violet UML Editor: Easy to use, Completely Free", accessed 2010, http://alexdp.free.fr/violetumleditor/page.php.

[76] Auer, M., Poelz, J., Fuernweger, A., Meyer, L. and Tschurtschenthaler, T. " UMLet, UML Tool for Fast UML Diagrams", accessed 2010, http://www.umlet.com/.

[77] IBM. " IBM Rational Software Architect Modeling Tool", accessed 2009, http://www-01.ibm.com/software/awdtools/architect/swarchitect/.

[78] Anonymous " The Papyrus UML", accessed 2010, http://www.papyrusuml.org.

[79] The Eclipse Foundation. " Eclipse Modeling - M2T - Home (Jet Project)", accessed 2009, http://www.eclipse.org/modeling/m2t/?project=jet.

[80] Terence, P. " ANTLR Parser Generator", accessed 2010, http://www.antlr.org/.

[81] Forward, A. and Lethbridge, T. C. " Umple Language", accessed 2009, http://cruise.site.uottawa.ca/umple/.

[82] The Eclipse Foundation. " Eclipse Modeling Framework Project (EMF)", accessed 2010, http://www.eclipse.org/modeling/emf/.

[83] The Eclipse Foundation. " Package Org.Eclipse.Emf.Ecore", accessed 2010, http://download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/package-summary.html#details.

[84] Bacvanski, V. and Graff, P. "Mastering Eclipse Modeling Framework," in *EclipseCon,* 2005.

[85] The Eclipse Foundation. " Emfatic", accessed 2008, http://wiki.eclipse.org/Emfatic.

[86] Senac, A. and Sevilla, D. " EMF4CPP", accessed 2010, http://www.catedrasaes.org/trac/wiki/EMF4CPP.

[87] Liskov, B. "Data Abstraction and Hierarchy". 1987. *ACM SIGPLAN Notices,* vol 23, pp. 17-34.

[88] The Eclipse Foundation. " EuGENia: Epsilon Project", accessed 2010, http://epsilonblog.wordpress.com/2008/08/04/eugenia-kick-start-your-gmf-editor-development/.

[89] The Eclipse Foundation. " The Eclipse Graphical Modeling Framework (GMF)", accessed 2010, http://www.eclipse.org/modeling/gmf/.

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

[91] Mellor, S. J. and Balcer, M. *Executable UML: A Foundation for Model-Driven Architectures.* Addison-Wesley, Boston, 2002.

[92] Mentor Graphics Corporation. " BridgePoint", accessed 2010, http://www.mentor.com/products/sm/model_development/bridgepoint/.

[93] Kavanagh Consultancy Limited. " OOA Tool", accessed 2010, http://www.ooatool.com/OOATool.html.

[94] Moten, D. " Xuml-Compiler", accessed 2010, http://code.google.com/p/xuml-compiler/.

[95] Rumbaugh, J., Jacobson, I. and Booch, G. *The Unified Modeling Language Reference Manual.* Boston, MA: Addison-Wesley, 2005.

[96] ITU. " Languages and General Software Aspects for Telecommunication Systems", accessed 2010, http://www.itu.int/rec/T-REC-z.

[97] Piefel, M. and Scheidgen, M. "Modelling SDL, Modelling Languages," in *Cybernetics and Information Technologies, Systems and Applications (CITSA),* 2006. pp. 298.

[98] NTNU. " SDL-2000", accessed 2010, http://www.item.ntnu.no/fag/ttm4115/sdl-2000.htm.

[99] Grammes, R. and Gotzhein, R. "SDL Profiles: Formal Semantics and Tool Support," in *FASE'07: Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering,* 2007. pp. 200-214.

[100] SDL Forum Society. " Towards SDL-2010", accessed 2010, http://www.sdl-forum.org/ftp/pub/SDL-2010/index.htm.

[101] The Eclipse Foundation. " Xtext - a Programming Language Framework", accessed 2010, http://www.eclipse.org/Xtext/.

[102] Microsoft. " Oslo: Making a New Class of Model-Driven Applications Mainstream", accessed 2009, http://www.microsoft.com/soa/products/oslo.aspx.

[103] Purdy, D. " MUrl: A DSL for RESTful Clients", accessed 2009, http://www.douglaspurdy.com/2009/03/20/murl-a-dsl-for-restful-clients/.

[104] Karras, C. " C# Code Generation using MGrammar", accessed 2009, http://www.nootaikok.com/2009/01/c-code-generation-using-mgrammar.html.

[105] Solano, J. "Exploring how Model Oriented Programming can be Extended to the UI Level". *University of Ottawa. Thesis (Master's),* 2010. Available: http://www.site.uottawa.ca/~tcl/gradtheses/jsolano/

[106] van Meegen, M. " Slime UML - the Lean and Mean Modeling Tool", accessed 2010, http://www.slimeuml.de.

[107] Anonymous " PlantUML Open-Source Tool in Java to Draw UML Diagrams", accessed 2010, http://plantuml.sourceforge.net.

[108] Vokáč, M. and Glattetre, J. "Using a Domain-Specific Language and Custom Tools to Model a Multi-Tier Service-Oriented Application - Experiences and Challenges". *Model Driven Engineering Languages and Systems,* vol. 3713, pp. 492-506, 2005.

[109] Anonymous " Active Record Pattern", accessed 2010, http://en.wikipedia.org/wiki/Active_record_pattern.

[110] Tate, B. and Hibbs, C. *Ruby on Rails: Up and Running.* O'Reilly Media, Inc., 2006.

[111] Filman, R. E., Elrad, T. and Clarke, S., Mehmet A. *Aspect-Oriented Software Development.* Boston: Addison-Wesley, 2005.

[112] Rajan, H. and Sullivan, K. J. "Unifying Aspect- and Object-Oriented Design". 2009. *ACM Trans.Softw.Eng.Methodol.,* vol 19, ACM. pp. 1-41.

[113] Pearce, D. J. and Noble, J. "Relationship Aspects," in *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development,* 2006. pp. 75-86.

[114] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* New Jersey: Addison-wesley Reading, MA, 1995.

[115] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J. "Aspect-Oriented Programming," in *ECOOP,* 1997.

[116] The Eclipse Foundation. " The AspectJ Project", accessed 2010, http://www.eclipse.org/aspectj/.

[117] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. "Getting Started with ASPECTJ". 2001. *Commun ACM,* vol 44, ACM. pp. 59-65.

[118] Long, Q., Liu, Z., Li, X. and Jifeng, H. "Consistent Code Generation from Uml Models," in *Software Engineering Conference, 2005. Proceedings. 2005 Australian,* pp. 23-30.

[119] Brisolara, L. B., Oliveira, M. F. S., Redin , R., Lamb, L. C., Carro, L. and Wagner, F. "Using UML as Front-End for Heterogeneous Software Code Generation Strategies". 2008. *Design, Automation and Test in Europe, 2008.DATE'08,* pp. 504-509.

[120] Xi, C., JianHua, L., ZuCheng, Z. and YaoHui, S. "Modeling SystemC Design in UML and Automatic Code Generation," in *Proceedings of the 2005 Conference on Asia South Pacific Design Automation,* 2005. pp. 932-935.

[121] Hoare, C. A. R. *Unifying Theories of Programming.* Prentice Hall, 1998.

[122] Sutton, A. and Maletic, J. I. "Recovering UML Class Models from C++: A Detailed Explanation". 2007. *Inf. and SW Tech,* vol 49, Elsevier. pp. 212-229.

[123] Gueheneuc, Y. "A Reverse Engineering Tool for Precise Class Diagrams," in *Proc. CASCON 2004,* 2004. pp. 28-41.

[124] Kollman, R., Selonen, P., Stroulia, E., Systa, T. and Zundorf, A. "A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering," in *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02),* 2002. pp. 22-30.

[125] Lange, C. F. J. and Chaudron, M. R. V. "An Empirical Assessment of Completeness in UML Designs". 2004. *Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering (EASE '04),* pp. 111–121.

[126] Sutton, A. and Maletic, J. I. "Recovering UML Class Models from C++: A Detailed Explanation". 2007. *Inf. and SW Tech,* vol 49, Elsevier. pp. 212-229.

[127] Norton, D., "Open-Source Modeling Tools Maturing, but Need Time to Reach Full Potential". Gartner, Inc., Tech. Rep. G00146580, 20 April 2007, 2007.

[128] Anonymous " Wikipedia Listing of UML Modeling Tools.", accessed 2009, http://en.wikipedia.org/wiki/List_of_UML_tools