If You’re Not Modeling, You’re Just Programming: Modeling throughout an Undergraduate Software Engineering Program

James Vallino

Department of Software Engineering, Rochester Institute of Technology
Rochester, NY 14623-5608, USA
J.Vallino@se.rit.edu

Modeling is a hallmark of the practice of engineering. Through centuries, engineers have used models ranging from informal “back of the envelope” scribbles to formal, verifiable mathematical models. Whether circuit models in electrical engineering, heat-transfer models in mechanical engineering, or queuing theory models in industrial engineering, modeling makes it possible to perform rigorous analysis that is the cornerstone of modern engineering. By considering software development as fundamentally an engineering endeavor, RIT’s software engineering program strives to instill a culture of engineering practice by exposing our students to both formal and informal modeling of software systems throughout the entire curriculum. This paper describes how we have placed modeling in most aspects of our curriculum. The paper also details the specific pedagogy that we use in several courses to teach our students how to create, analyze and implement models of software systems.

1. Introduction

There has been much discussion of software development as an engineering profession and what changes are necessary in the undergraduate education of software professionals for the profession to move forward [1-3]. In 1993, Rochester Institute of Technology (RIT) began the design of a curriculum leading to the Bachelor of Science in Software Engineering [4, 5]. We developed our curriculum from the ground up rather than by adding a small set of software engineering courses to an established curriculum in computer science or computer engineering. We considered software development to be primarily an engineering endeavor with modeling as a core component as it is in the traditional engineering disciplines.

2. The Difficulty of Modeling Software Systems

A hallmark of engineering design is the use of models to explore the consequences of design decisions. Sometimes these models are physical prototypes or informal drawings, but the *sine qua non* of contemporary engineering practice is the use of formal, mathematical models of system structure and behavior. Unfortunately, the current practice in software engineering is such that rigorous models from which one could derive significant properties are either too rudimentary or so tedious to use that it is difficult to justify the incremental benefit in other than the most critical of systems. This reflects a key distinction between software and traditional engineering: whereas the latter builds on numerical computation, software is more appropriately modeled using aspects of discrete mathematics. The models stress relationships between software components, and numerical computation is the exception.

3. Modeling throughout the Curriculum

We designed our curriculum to provide a focus on the principles and practices for the engineering of software systems through their entire life cycle. Our answer to the topical question, "How does modeling integrate into the software engineering curriculum?" is "It should be emphasized throughout the entire curriculum." Despite the difficulties described in the previous section, our curriculum stresses modeling throughout from more informal models expressed in the UML [6] to those expressed in mathematically rigorous languages such as Alloy [7] and FSP [8]. This emphasis on modeling is reflected in two of our ten program outcomes:

1. Model and analyze proposed and existing software systems, especially through the use of discrete mathematics and statistics.
2. Analyze and design complex software systems using contemporary analysis and design principles such as cohesion and coupling, abstraction and encapsulation, design patterns, frameworks and architectural styles.

Students develop their modeling skills starting with basic object-oriented design and progress through the remainder of the curriculum to higher levels of modeling abstractions in all areas of software engineering including architecture, requirements, verification and validation, and formal models. This paper describes how we incorporated modeling into most of the courses in our curriculum. Figure 1 shows the sequencing of courses this paper discusses. Except for the three courses within the box labeled "Design Electives" these are all required courses in our program. These software engineering courses are from the "design side" of our program. There are also required and elective courses on a "process side."

This paper first describes how we introduce our students to abstraction through modeling and move them from a programming view of software development to an engineering view. Next is a description of our use of mathematically formal models where our overall goals are three-fold: to acquaint our students with modern modeling tools, to connect the courses they take in discrete mathematics to real applications, and to persuade them that mathematics has much to offer to the engineering of quality software. In the context of these formal models we introduce our students to model-driven development. The paper concludes with a description of problems still to be solved and indications of success of modeling in a software engineering curriculum.

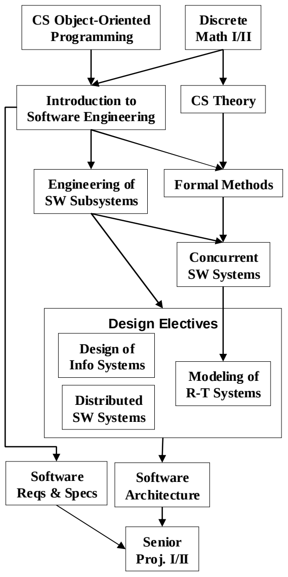

Fig. 1. Modeling in RIT's Software Engineering Design Courses

3.1 Basic Object-Oriented Modeling

The students in our program spend their first year studying the fundamentals of object-oriented programming. Three courses cover topics in basic programming, object-oriented technology, data structures, and algorithms with simple complexity analysis. Students are exposed to class diagrams in UML notation beginning in the middle of the first course. Modeling discussions stay at rather low levels, considering questions, such as, which nouns might represent objects in the system or state within the objects and which verbs are behaviors in an object. The design activity is mostly concerned with the design of single classes and interactions between pairs of classes.

Students in the three computing disciplines take an introduction to software engineering[9] during their second year. This is the first course taught by the Department of Software Engineering faculty. The main component of this course is a term-long team-based project using teams of 4 or 5 students. This course covers topics such as roles on a software development team, software development lifecycles, requirements specification, design principles, and user interface design. Each team develops a product from requirements through product iteration deliveries. For the first time, students are confronted with subtleties in the UML such as the distinctions between associations, aggregations and compositions. Teams must document their designs using UML class diagrams, sequence diagrams and statecharts. As they develop a larger system in this course, our students first begin to appreciate the importance of design modeling.

3.2 Modeling in a Course on Design Patterns

The next course, Engineering of Software Subsystems, covers most of the patterns in [10] using a problem-based learning (PBL) pedagogy. Instructors lecture for no more than 6 hours throughout the entire course. The traditional lecture time is replaced with active learning by the students doing class exercises and holding team meetings to discuss the project work which emphasizes modeling software systems using the patterns. The two team projects involve the design of a software system in the 2 to 3 kSLOC range. The team models its solution in the UML using class diagrams, sequence diagrams and statecharts. Discussions with the instructor center on the tradeoffs in various design approaches and the appropriateness of design pattern usage.

The first and the last assignments in the Engineering of Software Subsystems course particularly highlight the emphasis placed on modeling of designs. Students are confronted with a modeling challenge in the first class when they are given a design problem to solve using the modeling skills that they have developed through three quarters of computer science programming courses and one software engineering course. At the beginning of the second class each student will individually submit a first cut at a UML class model for the problem. The second class is divided into three parts. First, groups of three or four students will work together to create a consensus model incorporating the best aspects of the individual models. Next, teams present their models to the entire class. Finally, the instructor leads a discussion on ways in which groups of classes in these models relate to each other and to the solution of the problem pointing out where established patterns were used and the advantage of discussing the design at this subsystem level.

The last assignment challenges the students' modeling abilities in new ways. Each unit team is given the final code and documentation for a student project submitted for our introductory software engineering course. The first task is to reverse engineer the code to obtain a UML class model for the system and identify any, most likely inadvertent, design pattern usage. The team must capture dynamic models for the program by creating sequence diagrams for two significant program features. After gaining an understanding of the as-built system model each team will propose and implement a refactoring of the code base by following the principles that the course stresses and applying their newly gained knowledge of design patterns.

In design presentations throughout the course, teams must discuss how their modeling activities have considered design principles, such as, encapsulation, coupling, cohesion, and separation of concerns. As mentioned earlier, a cornerstone of modern engineering practice is the use of quantitative models to do early design analysis. In our assignments, we require students to manually compute some simple metrics, such as, class size and average class coupling from their design models. As part of the initial reverse engineering in the refactoring assignment, the teams use the Eclipse Metrics plug-in[11] to compute program metrics. Teams use this information to guide their refactoring efforts and work to improve on the project's metrics with their refactored implementation.

We have evidence that this approach to building modeling skills works. A quantitative comparison with a non-PBL version of the course matches the research on problem-based learning[12]. There is a statistically significant improvement in student satisfaction with and perceived learning from the course. The students also have a greater appreciation for the course textbook which they now must actually read because of the minimal lecture pedagogy used.

4. Formal Modeling

While the models discussed to this point have semantic definitions there is often disagreement between practitioners in their understanding of those semantics particularly when dealing with UML constructs. Disagreements, such as these, rarely exist when the models have a formal mathematical definition. The modeling is capturing logical interconnections and relationships between components using discrete mathematics rather than numerical attributes using continuous mathematics. The software engineering design models are difficult to analyze because of the complexity of the systems being designed and built. Despite these shortcomings, we believe it is important for our students to see that mathematical formalisms indeed undergird software design and provide benefits for engineering quality software.

4.1 "Theoretical" vs. "Practical" Modeling

We believe that the science of formal modeling is in the domain of the computer science and the engineering application of formal modeling is in the software engineering domain. Our approach begins with our students taking two courses in discrete mathematics followed by a computer science theory course, which includes the topics of languages, finite state machines, pushdown automata, Turing machines, and basic computability theory. We want the emphasis within software engineering to be on what we sometimes refer to as "practical" formal models. Our Formal Methods for Specification and Design course focuses on the development of mathematical models of software systems, and applying those models to the analysis of system properties, and to verifying design and implementation decisions. This course has used formalisms such as Z, VDM, and, most recently, Alloy[13] to capture system behavioral requirements, and uses simulation, and proof to analyze system properties. The assignments and projects are almost exclusively modeling and model checking exercises.

4.2 Finite State Process Modeling of Concurrent Systems

For a modeling methodology to be useful for the design of concurrent systems it should meet two criteria. First, the formalisms should be at a level that reduces the scale and complexity of the system sufficiently to allow the software engineer to analyze its important concurrent properties such as deadlock and progress checks. Second, there should be tool support available so that the analysis is done mechanically rather than by hand. The Finite State Process (FSP) modeling technique described by Magee and Kramer[8] satisfies both of these criteria and is the methodology emphasized in our Principles of Concurrent Software Systems[14]. Individual sequential FSP models use standard finite state machine semantics (mutually exclusive states, instantaneous execution of actions causing transitions) that our students easily grasp. Students do not have difficulty modeling non-concurrent FSPs. Modeling of concurrent systems is accomplished by composing multiple sequential FSPs into a single parallel composition. This is where students often struggle getting synchronization aspects of the model correct.

A tool called the Labeled Transition System Analyzer (LTSA) allows students to edit and analyze their FSP models. A major advantage of the LTSA is that with just a few hours of studio classroom time, students can do productive work within the LTSA environment. Model checking features provide analysis of deadlock, safety violations and progress failures.

Having mathematically proven that the model does not contain any anomalous behaviors, the intention is to keep the implementation as closely tied to the model as possible. To complete this model-driven development, it would be optimal to generate an implementation of the model via autocoding. The LTSA does not have an autocoding feature requiring students to do manual implementations. Students think about mappings from model elements to implementations. This yields a mechanical conversion to generate the code for the concurrency framework captured in the model.

When we initially taught this course, we did not explicitly cover the formal FSP semantics. We assumed that the students would recognize the application of discrete mathematics in the finite state machines that are the basis of the FSP semantics. We were quite surprised, then, when over 75% of the students answered “Not applicable” to the question, “How much did this course require you to demonstrate an ability to model and analyze proposed and existing software systems, especially through the use of discrete mathematics and statistics?” We added discussion of the formal semantics for each FSP feature. Students now recognize that while they may not be “doing discrete math” they are applying it in the design and analysis of concurrent systems.

Each of the projects we assign requires the team to use a model-driven development approach. One problem with FSP modeling is state-space explosion. The larger projects that we assign in this course will commonly have millions of states in the composite. While LTSA can handle systems of this size, a naïve approach to modeling will exceed the capacity of the tool. This aids student learning, in that it forces them to model the system at a level of abstraction that captures all the essential concurrency issues while fitting within the capacity of the LTSA.

4.3 Model-Driven Development

One course in our curriculum has model-driven development at its core. This elective course, Modeling of Real-Time Systems, is in our multi-disciplinary real-time and embedded systems course sequence[15]. The requirements and architectural design project has the team create a requirements specification for a small consumer device. The team does a UML use case analysis of the product followed by an architectural design and high-level class structural design. In the second project, the instructor provides a statement of requirements and the team models the behavioral requirements in a UML statechart, creates a class-level design and set of sequence diagrams, and implements the complete system. The third project is a complete model-driven development using statecharts for behavioral modeling of real-time and embedded systems. The students explore the code generation features of the Ilogix Rhapsody modeling tool they have been using throughout the course. The teams create a statechart-based definition of the system behavior and automatically generate C++ code for the application. A final individual project requires students to model a system, such as an auto power window controller, and reverse vending machine, with an identification of actors, a UML use case analysis, class structural design, and system dynamic modeling using sequence diagrams and statecharts.

5. Modeling in Other Design Areas

The previous sections described how our Engineering of Software Subsystems course sets the foundation for our students' use of design modeling and abstraction, and the way we present formal modeling to our students. This section describes how design-oriented courses throughout the rest of our program reinforce the software engineer's reliance on modeling and abstraction.

In the Principles of Distributed Software Systems course students work with the Concurrent Object Modeling and Architectural Design Method. This method follows the traditional UML approach, with a heavier emphasis placed on interaction models and communication diagrams.

Entity-Relationship-Diagrams, considered by some to have been a precursor to object-oriented class models, are the models that students develop and analyze in Principles of Information Systems Design. The course also requires teams to use J2EE Blueprints and enterprise-level patterns as abstractions in their information system designs.

In the Software Requirements and Specifications course our students see modeling techniques for expressing software requirements. Students model system requirements using UML activity diagrams and by applying analysis-level patterns. The course also exposes the students to Data Flow Diagrams and Nasi-Scheiderman diagrams as legacy modeling techniques that they may need to understand if they are required to work on older systems that had originally used those two methodologies.

In the Software Architecture course, students are challenged with understanding and developing models of software systems at the highest levels of abstraction. They must model the system from multiple architectural perspectives[16]. Views include, for example, structural, process, deployment, and concurrency. Systems are also assessed based on quality attributes in the areas of availability, modifiability, performance, security, testability, and usability. We also teach this course in a problem-based format. Assignments include preparing one-page executive summary memos that describe the effect a new technology will have on a product, and to advocate for a product line approach for a new development project. Case studies provide prominent examples of architectural analyses in the course. Teams select an open-source or well publicized architectural framework and perform their own architectural analysis of it.

6. Problems Still to Solve

This paper has discussed our approach to infuse software systems modeling throughout an undergraduate software engineering curriculum. This section will describe some of the problem areas that still remain.

6.1 Using a Consistent Subset of UML

We are not satisfied that we have chosen the right aspects of the UML to cover in each of our courses. We need additional emphasize on the semantics for basic UML class relationships. In several of our design-oriented courses we give a short UML quiz early in the term. There are many students who continue to have difficulty distinguishing the semantic differences between association, aggregation and composition.

We originally used use case analysis of requirements in our introductory course. The analyses that teams submitted were so poor that we questioned whether there was any educational benefit. In this case, we opted for an agile approach and switched to user stories to specify requirements. We felt that this was adequate for this introduction to software engineering, which is taken by students in computer science, computer engineering and software engineering, as long as the SE students saw full use case analysis in our Software Requirements and Specifications course.

6.2 Getting Students to Trust Their Models

Our students are comfortable with model-driven development when the models are class-based models. They still grapple with other abstraction models such as the concurrency models seen in Principles of Concurrent Software Systems. Students do not trust their FSP model and their ability to use the model to create a working implementation. We have observed, however, that the emphasis on modeling gives students an improved understanding of the system requirements and the thread synchronization points, which is a benefit even if they abandon the model during implementation.

7. Success of Modeling throughout the Curriculum

RIT's traditional focus on career-oriented education means that almost all of our students enter the workforce upon graduation and their employers are a major stakeholder in the outcomes of the program. Discussions with campus recruiters and members of our Industrial Advisory Board have indicated an existing emphasis on or a strong move toward modeling using the UML. While we would not attribute the success of our students in their employment only to our program's emphasis on modeling, we do believe, however, that it is a prime factor that attracts employers to our students.

7.1 Preference for a Modeling-first Approach

A review of co-op employment evaluations provides anecdotal evidence of the value of our students' training to their employers. An engineering manager in an aerospace company, which has hired many of our students on co-op and in full-time positions, commented that the students have a strong focus on capturing requirements and system modeling. An engineering vice-president, who has hired several of our students and sponsored senior projects, commented that our graduates match up favorable against some software engineers who have been working for him for five years. A non-SE RIT faculty member, who manages interns for a health insurance provider, noted a significant difference in how software engineering students learn about a system. The SE students ask questions about components, architecture, and interactions between the components, preferring a higher-level and more abstract model-driven discussion. The computer science and information technology students tend to quickly ask for examples of working code and begin understanding the system from the bottom up. The SE students overwhelmingly believe they formed the base for this methodology in Engineering of Software Subsystems when they were forced to think abstractly about their projects using design patterns rather than code implementations.

7.2 Analysis of Formal Models

Even though the LTSA tool used in our concurrent systems course is not "industrial strength", one student used it on a co-op assignment. The student sensed that there was a problem in a protocol that he was asked to implement. The student remembered the features provided by LTSA; with an afternoon of effort he modeled the protocol, executed traces, and uncovered a progress failure that prevented the protocol from continuing to completion under certain circumstances. The model highlighted the exact problem that was latent in the system thus eliminating many hours of debugging and finger pointing between the hardware and software engineers.

8. Conclusions

The RIT undergraduate program in software engineering instills an engineering mindset in students. Our program exposes our students to both the informal modeling, which is more prevalent in software engineering practice, and formal modeling, which has benefits derived from its underlying mathematical rigor. Without the constraints of traditional computer science or computer engineering programs, we designed a curriculum in which modeling applied to software development is prominent throughout the curriculum. We believe that this emphasis on modeling is a distinguishing characteristic between the science and engineering of software development. As research in model-driven development progresses, we will adapt our curriculum to ensure that our students graduate with an ability to model complex software systems using state-of-the-art practices and abstractions.

9. References

[1] M. Shaw, "Prospects for an Engineering Discipline of Software." *IEEE Software*, v7, n6, Nov/Dec 1990, pp.15-24

[2] D. L. Parnas, "Software Engineering Programs Are Not Computer Science Programs." *IEEE Software*, Nov/Dec 1999, pp 19-30

[3] T. B. Hilburn, "Software engineering education: a modest proposal." *IEEE Software*, v14, n6, Nov/Dec. 1997, pp 44 – 48

[4] J. F. Naveda and M. J. Lutz, "Crafting a baccalaureate program in software engineering." *Proceedings of the Conference on Software Engineering Education & Training*, April 1997,.

[5] Department of Software Engineering, Rochester Institute of Technology, http://www.se.rit.edu.

[6]M. Blaha and J. Rumbaugh *Object-Oriented Modeling and Design with UML (Second Edition)*. Prentice-Hall, 2005.

[7] D. Jackson "Alloy: A Lightweight Object Modeling Notation." ACM Transactions on Software Engineering and Methodology (TOSEM) v11, n2, April 2002, pp. 256-290

[8] J. Magee and J. Kramer *Concurrency: State Models and Java Programs*. John Wiley & Sons, 1999.

[9] Ludi, S., Reichlmayr, T., and Natarajan, S. "An Introductory Software Engineering Course That Facilitates Active Learning," *Proceedings of ACM SIGCSE Conference*, St.Louis, MO. February, 2005.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*. Reading: Addison-Wesley, 1995.

[11] Eclipse Metrics Plug-in, http://metrics.sourceforge.net/.

[12] J. Vallino "Design Patterns: Evolving from Passive to Active Learning." *Proceedings of the Frontiers in Education Conference*. Boulder, CO. November 2003.

[13] M. Lutz "Exploratory Mathematics: Experiences With Alloy In Undergraduate Formal Methods," *Proceedings of 2006 American Society of Engineering Education Conference*, Chicago, IL. June 2006.

[14] M. Lutz and J. Vallino "Concurrent System Design: Applied Mathematics & Modeling in Software Engineering Education," *Proceedings of 2005 American Society of Engineering Education Conference*, Portland, OR. June 2005.

[15] J. Vallino and R. Czernikowski "Thinking Inside the Box: A Multi-Disciplinary Real-Time and Embedded Systems Course Sequence," *Proceedings of Frontiers in Education Conference*. Indianapolis, IN. October 2005.

[16] L. Bass, P. Clements, and R. Kazman *Software Architecture In Practice*. Addison-Wesley, 2003.