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 [1-5]. In 1993, Rochester Institute of Technology (RIT) began the design of a curriculum leading to the Bachelor of Science in Software Engineering [6, 7]. We held the strong belief that the software engineering body of knowledge had matured and grown sufficiently distinct from other computing disciplines that a new curriculum was indeed needed. We developed our curriculum from the ground up rather than adding a small set of software engineering courses to an established curriculum in computer science or computer engineering. The curriculum was designed to meet the software engineering program criteria specified by the Engineering Accreditation Commission of ABET [8], the private governing agency responsible for accreditation of engineering, technology and computing programs in the US. In 2001, our first class graduated from the program with the first degrees granted by an accredited software engineering baccalaureate program in the United States.

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. These different types of models serve differing purposes. Consider an architect's mockup of a building compared to a structural engineer's finite element model of the structural support system. The former serves an artistic purpose primarily while the later is serving an engineering purpose. A cornerstone of modern engineering practice is performing a rigorous analysis on the engineering models. 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 is partially due to the relative immaturity of software engineering, but it also reflects a key distinction between software and traditional engineering: whereas the latter builds on numerical computation and continuous functions, software is more appropriately modeled using logic, set theory, and other aspects of discrete mathematics. Most of the models stress relationships between software components, and numerical computation is the exception rather than the norm.

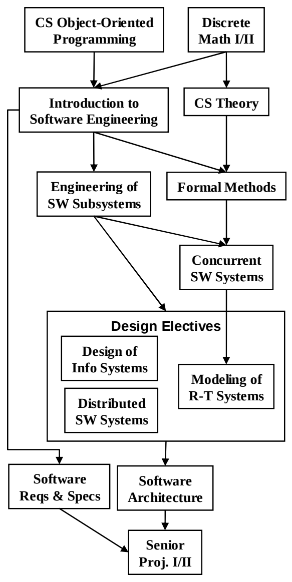
3. Modeling throughout the Curriculum

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 [9] to those expressed in mathematically rigorous languages such as Alloy [10] and FSP [11]. 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. A flowchart for our curriculum is at [12]. Figure 1 shows the sequencing of courses this paper discusses. Each course runs for ten weeks, meeting four hours per week. Except for the three courses within the box labeled "Design Electives" these are all required courses in our program. They are taught by faculty in Software Engineering except for the CS and Math courses shown. These software engineering courses are from the "design side" of our program. There are also required and elective courses on a "process side." Those courses also place an emphasis on modeling though in most cases they are not working with UML models. The right side of this sequence is where our students are exposed to formal modeling techniques. These courses have design course prerequisites to ensure that the formal methods are studied from the perspective of their use in designing software systems.

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



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.

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. The courses are offered by the Department of Computer Science and are common to the Computer Science, Computer Engineering and Software Engineering programs. Students are exposed to class diagrams in UML notation beginning in the middle of the first course. From this sequence of three 10-week courses, we expect our students to develop solid programming skills. 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. Beyond this there is little discussion of overriding principles motivating the design activity. This is a programming-first approach with delayed introduction of objects which has worked better with our students than an objects-first or design-first approach.

In a traditional pedagogy, which uses a lecture and lab format, students are typically passive learners. Active engagement of the student is often missing. There is ample evidence in the engineering education literature[13-15] that actively engaging the student results in the long-lasting learning that goes beyond what is needed for the next exam. Over the last several years we have reworked our curriculum to use active learning techniques with an emphasis on problem-based learning. All of our courses are taught in studio lab format where we have no distinction between lecture and lab. The typical classroom session seamlessly weaves lecture, class exercises, computer-based demonstrations, group work, and student use of computers. This presents the course material to students in ways that accommodate a variety of learning styles.

Students in the three computing disciplines take an introduction to software engineering 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[16] has 20 students per section with one faculty member. Due to program growth, we are experimenting with 40 students per section covered by one faculty member and one or two upper-level 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. Class sessions typically are composed of a short lecture component, class exercises and team meetings. Several class exercises are modeling exercises. 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. During class reviews teams present their designs which are then critiqued by the instructor and other teams. In grading, instructors emphasize the importance of clarity in the design description. As they develop a larger scale system in this course, our students first begin to appreciate the importance of design modeling. In reflective comments at the end of the course, students identify that their modeling skills are not adequate to handle the design of these systems with a larger number of classes interacting in new ways. The students are eager to learn more about modeling larger software systems.

3.2 Modeling in a Course on Design Patterns

The next course, Engineering of Software Subsystems, is nicknamed the "Patterns Course" since it is based on [17]. The course covers most of the patterns in [17] using a problem-based learning (PBL) pedagogy. The course is broken into 4 units with each unit having individual and team learning elements. The emphasis is on modeling software systems using the patterns selected for the unit. 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 unit team meetings to discuss the unit project work. Each unit project involves 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 grading of these assignments places more weight on the sound analysis of the design than on a complete implementation of it. Depending on the size of the system, teams are often asked to implement only a subsystem of the overall design.

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. We ask students to model problems, such as, a general framework for a two-player board game, a medical picture archive and communications system (PACS) and a system to allow interoperation with several chat servers. The students are encouraged to discuss the problem with one or two other students during the remaining class time in the first course session. At the beginning of the second class each student will individually submit a first cut at a UML class model for the problem. Students are assured that they will receive most credit for the assignment if they exhibit due diligence in completing it. A full and complete model is not sought. 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, some teams present their models to the entire class. In the last part of this second class, the instructor leads a discussion of ways in which groups of classes in these models relate to each other and to the solution of the problem. Especially where there are commonalities, the instructor will point out where established patterns were used in one or more models and motivate the advantage of discussing the design at this subsystem level rather than an individual class level.

The last assignment challenges the students' modeling abilities in new ways that are quite relevant to co-op work experiences they will see shortly. Each unit team is given the final code and documentation for a student project submitted for our introductory software engineering course. This is the team-based project course that most students in Engineering of Software Subsystems completed within the last term or two. These projects are typically under 2 kSLOC in size. The project is from one or two years back so that few students had this as their project and the instructor absolutely ensures that for those students who had done this project their submission is not the one selected. All teams work with the same code base. 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. Teams individually choose whether to do this manually or with tool support. The team will also need to assess the quality of the documentation that the student team provided to determine if it provides an accurate guide for recovering the class model. The team also 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 a refactoring of the code base by following the principles that the course stresses and applying their newly gained knowledge of design patterns. Each team is required to implement a portion of their refactored model with design pattern usage the team would like to explore. This emphasizes an incremental approach that always maintains a fully working system.

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. There are metric models of object-oriented software[18] that quantify the design principles we stress but, unfortunately, the tools we used so far measure from a code base and not a model of the system. Access to a code base is too late to have an effect on early design tradeoffs. In our assignments, we require students to manually compute some simple metrics, such as, class size and average class coupling from their design models. The refactoring project does, however, begin with a code base. As part of the initial reverse engineering, the teams use the Eclipse Metrics plug-in[19] to compute program metrics on this Java project. Usually, there are a number of areas in which the original project exceeds some metric targets. Teams use this information to help guide their refactoring efforts and work to improve on the project's metrics with their refactored implementation. Teams have been quite proud of their efforts that completely eliminated all red flags from the metrics tool suite.

The students are challenged by the modeling work that they do in the Engineering of Software Subsystems course. The course closes with a fun modeling exercise. After breaking the class into small groups, five design pattern cards[20] are dealt to each group. Each card has on its face one design pattern. The task is for the team to think of a project in which their hand of design patterns could be applied. Teams are allowed one draw of replacement cards for their hand. At the end of the exercise each team has two minutes to present to the class their project and its design. The class votes for the best design patterns use and the winning team is awarded a small prize.

We have evidence that this approach to building modeling skills works[21]. A quantitative comparison with a non-PBL version of the course matches the research on problem-based learning[13]. 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. This paper has already noted issues with formal mathematical models in software engineering. Compared to models used by traditional engineering disciplines, software modeling is difficult to grasp because it does not have an inherent connection to a physical entity. 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 have something to offer to the engineering of quality software.

4.1 "Theoretical" vs. "Practical" Modeling

Our approach begins with our students taking two courses in discrete mathematics. The initial curriculum then followed with a single course, Formal Methods for Specification and Design. Several years into the program, we realized that this course was too broad, covering what might be termed both the science and engineering of formal modeling. We felt that the science was in the domain of the computer scientists and that the engineering application of formal modeling was in the software engineering domain. We wanted the emphasis within software engineering to be on what we sometimes refer to as "practical" formal models. To reflect this viewpoint, we added a requirement for students to take a standard introductory course in computer science theory, which includes the topics of languages, finite state machines, pushdown automata, Turing machines, and basic computability theory. The Formal Methods course builds on this improved knowledge base by focusing 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[22] 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

When we did an assessment after adding the theory of computing course we felt that our students were still not making a strong enough connection between their formal models and the resulting implementations. To highlight this connection, and the potential of model-driven development, we made an existing elective course[23], Principles of Concurrent Software Systems, a program requirement.

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[11] satisfies both of these criteria. Based on finite state machines, the basics of FSP modeling has been seen by the students before and is easy to grasp. Individual sequential FSP models use standard finite state machine semantics (mutually exclusive states, instantaneous execution of actions causing transitions) that are defined using a text notation. Students do not have difficulty modeling non-concurrent FSP's. Modeling of concurrent systems is accomplished by composing multiple sequential FSP's 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 compared to "industrial-strength" tools is that students can quickly learn LTSA; with just a few hours of studio classroom time, students know how to work within the LTSA environment. Features provided by LTSA allow the student to experiment with their models. A trace tool lets the student manually execute the model by triggering actions and watching how the system reacts. By running a simulation and studying the actions available at each state the student can determine if the synchronization within the model is working correctly. The model checking features allow for analysis of deadlock conditions, safety violations and progress failures. For safety issues, the student defines a correct sequence for actions to be executed by the composite process. The model checker determines if there is any trace of execution within the composite FSP that would violate this sequence. Progress checks determine if there are regions of the composite FSP in which actions that must be executed can never be triggered. By analyzing the model the student gains a better understanding of the concurrency and synchronization requirements for the system.

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 provide that feature and students will manually do the implementation. Students think about mappings from model elements to implementations during in-class discussions and while answering unit questions. They must consider trade-offs between an exact implementation of the FSP formal semantics and implementation efficiencies. This can yield a somewhat 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 have since changed the syllabus for the course to explicitly discuss the formal semantics for each FSP feature when it is presented. Students now recognize that while they may not be "doing discrete math" they are applying it in the design and analysis of these 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 composite FSP has exponential growth for the number of states in the system. 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 actually 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 at the same time fits within the capacity of the LTSA.

We teach Principles of Concurrent Software Systems using a problem-based learning pedagogy. There is a minimal amount of traditional lecturing to cover the semantics of FSP features. Class time is spent doing instructor or student-led modeling exercises. The instructor will also discuss the current unit questions and project with each team. The day before each class, any student can make a request on a course bulletin board for the instructor to discuss a particular topic during the next class session. Students like PBL because they are given some control over class content. Individual students or teams can request what they think will best aid their learning: lectures, instructor-led exercises, or instructor discussion with the team.

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[24]. The course follows the treatment of UML modeling of real-time systems given in [25]. Course projects are completed in pairs—a software engineering student teamed with a computer engineering student.

The requirements and architectural design project has the team create a requirements specification for a consumer device, such as, a pedometer step counter or a home blood pressure monitor, based on the user manual for the 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 clear statement of the system requirements and requires the team to model the behavioral requirements in a UML statechart, create a class-level design and set of sequence diagrams, and implement the complete system. The third project is a complete model-driven development emphasizing statecharts as a mechanism for behavioral modeling of real-time and embedded systems. In this project 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. Typically, the team will be able to create a fully-functioning application entirely from within the statechart model. For this project we have used a four-function calculator and garage door opener controller as systems to implement. 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.

5.1 Modeling of Distributed Systems

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. Subsystems for distributed applications often rely on message passing protocols where the communication diagram models prove particularly valuable.

5.2 Modeling in Information Systems Design

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.

5.3 Requirements Modeling

All students in our program are required to work in a team on a two-quarter senior capstone project. Two upper-division courses serve as required prerequisites for taking the senior project courses. 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.

5.4 Modeling of Software Architectures

Software Architecture is also a prerequisite for senior project. In this 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[26]. 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. The written documentation and class presentations must include multiple architectural views. Similar to early choices made on development projects in industry, the short timeframe for the assignment requires teams mostly rely on what they can gather from documentation rather than in-depth work with the framework. Their analyses often make comments on the quality of the documentation itself.

Students have struggled with the level of abstraction required in the Software Architecture course. We originally had this course positioned as the third course in our program typically taken early in the third year. Students did not possess sufficiently developed modeling and analysis skills to handle the high level of abstraction. We have since moved software architecture to late in our program. With its new position in the curriculum, the software architecture course appropriately acts as the culminating required course for the design-oriented aspect of our curriculum.

6. Problems Still to Solve

This paper has discussed our approach to infuse software systems modeling throughout an undergraduate software engineering curriculum. It has shown some of the places where we learned from what we did and made corrections. 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. Since it is not uncommon for practicing software engineers to have differing interpretations of these relationships these students unfortunately will be in good company. Fortunately, our faculty have agreed on semantics for the common UML elements we use in coursework, particularly the multi-section introductory courses.

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. To adequately cover use case analysis would take too much time in that particular course. 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 Appropriate Tool Support for Modeling

It is important to provide tools to assist students with developing their models and performing analyses where that is possible. We have available both Rational Rose and the popular UML editing tools that work within Eclipse. We have chosen not to spend class time on helping students become proficient with these tools. Students are made aware of the tool availability and they often choose based on the learning curve.

Recently, we have learned of tools that provide model-based metric analysis in addition to code-based analysis[27]. Incorporating these into our design courses, particularly Engineering of Software Subsystems, will allow spreadsheet-like what-if analyses at modeling time and eliminate current manual calculations.

6.3 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 created a model but did not use it to directly drive the implementation. Students do not trust that the FSP model represents a correctly functioning system. They do not trust 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. The students who take the elective Modeling of Real-Time Systems course are very excited about model-driven development when they can get an autocoded implementation of their statechart model simply by selecting a menu option. We believe that we will need to wait for further development of model-driven development methodologies and tools before we incorporate it into required courses.

7. Success of Modeling throughout the Curriculum

We designed the Software Engineering program with RIT's traditional focus on career-oriented education in mind. Since almost all of our students enter the workforce upon graduation, rather than continuing with graduate studies, 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. We also have anecdotal reports of our students' preference for a modeling-first approach to their work.

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. Their skill in software system modeling improved in subsequent courses with practice at higher levels of model abstraction.

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 but could not pinpoint the problem. At this point, he could have built a skeleton implementation and observe its operation. It might have been many hours of testing and debugging to uncover what may be a subtle problem in the protocol. 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 student discussed this problem with the hardware designers and they acknowledged that he had uncovered a problem with the protocol. The model highlighted the exact problem that was latent in the system and eliminated typical finger pointing between the hardware and software engineers.

7.3 Top Compensation

To the extent that salary is an expression of value to an employer then our program is quite successful within the RIT community in our career-oriented mission. RIT's Office of Cooperative Education and Career Services tracks the hourly cooperative rate, and full-time annual salaries that students receive upon graduation[28]. The software engineering undergraduate program has a higher median starting full-time salary than Computer Science, Computer Engineering, Information Technology, and all other undergraduate engineering programs except for Microelectronics Engineering. The SE program's hourly rate for cooperative employment ranks third in that group of 18 undergraduate programs, behind only Microelectronics and Computer Engineering.

8. Conclusions

The RIT undergraduate program in software engineering aims to instill an engineering mindset in students as they progress through the program. Formal mathematically-based modeling is a key characteristic of contemporary engineering practice. 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 the traditional curriculum models for computer science or computer engineering programs, we were able to design a curriculum in which modeling applied to software development has a prominent place in the curriculum. We believe that this emphasis on modeling throughout the curriculum is a distinguishing characteristic between the science and engineering of software development. As research in model-driven development and model-driven architecture 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] S. McConnell and L. Tripp, "Professional Software Engineering: Fact or Fiction?" *IEEE Software*, Nov/Dec 1999, pp 13-18
[3] D. L. Parnas, "Software Engineering Programs Are Not Computer Science Programs." *IEEE Software*, Nov/Dec 1999, pp 19-30
[4] P. J. Denning, "Who Are We? The Profession of IT" *C. ACM*, v44, n2, pp 15-19 (2001)

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

[6] 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,.

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

[8] Accreditation Board for Engineering and Technology, Criteria for Accrediting Engineering Programs. http://www.abet.org/Linked%20Documents-UPDATE/Criteria%20and%20PP/E001%2006-07%20EAC%20Criteria%202-9-06.pdf

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

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

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

[12] RIT Software Engineering, Program Flowchart http://www.se.rit.edu/docs/documents/se_flowchart.pdf

[13] M. Prince ""Does Active Learning Work? A Review of the Research," *Journal of Engineering Education*, v93 n3, July 2004.

[14] M. Prince and R. M Felder "Inductive Teaching and Learning Methods: Definitions, Comparisons, and Research Bases," *Journal of Engineering Education*, v95n2, April 2006.

[15] D. Jonassen "Everyday Problem Solving in Engineering: Lessons for Engineering Educations," *Journal of Engineering Education*, v95n2, April 2006.

[16] 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.

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

[18] S. R. Chidamber and C. F. Kemerer "A Metric Suite for Object Oriented Design." *IEEE Transactions on Software Engineering*, v20n6, June 1994.

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

[20] Industrial Logic, Design Patterns Playing Cards, http://www.industriallogic.com/games/dppc.html.

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

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

[23] 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.

[24] 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.

[25] B. P. Douglass, *Doing Hard Time – Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison Wesley, Reading, 1999.

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

[27] SDMetrics.com, *Software Design Metrics Tool for UML*, http://www.sdmetrics.com.

[28] RIT Office of Cooperative Education and Career Services "All Programs - Salary Data and Program Information." http://www.rit.edu/~964www/salary_program/AllPrograms.htm.