Proceedings of the 2nd

Educators’ Symposium

Co-located with

MoDELS 2006

the ACM/IEEE 9th International Conference on Model
Driven Engineering Languages and Systems

Editor
Ludwik Kuzniarz

Research report

Genova, Italy
October 02, 2006
Symposium Chair
Ludwik Kuzniarz, Blekinge Institute of Technology, Sweden

Program committee
Thomas Baar, EPFL, Switzerland
Jutta Eckstein, IT-Communication, Germany
Gregor Engels, University of Paderborn, Germany
Robert France, Colorado State University, USA
Holger Giese, University of Paderborn, Germany
Mieczyslaw Kokar, Northeastern University, USA
Kai Koskimies, Technical University of Tampere, Finland
Ludwik Kuzniarz, Blekinge Institute of Technology, Sweden
Gianna Reggio, University of Genova, Italy
Pascal Roques, Valtech Training, France
Sotiris Skevoulis, Pace University, USA
Michal Smialek, Warsaw University of Technology, Poland
Jean Louis Sourrouille, INSA Lyon, France
Miroslaw Staron, IT University of Göteborg, Sweden
Perdita Stevens, University of Edinburgh, UK
Tarja Systa, Technical University of Tampere, Finland
Juha-Pekka Tolvalen, University of Jyvaskila, Finland
Inkeri Verkamo, University of Helsinki, Finland
Paula Filho Wilson de Pdua, UFMG, Brasil
Preface

Putting the model-driven development (MDD) approaches and technologies for software-based systems vision, in which development is centered round the manipulation of models, into practice requires not only sophisticated modeling approaches and tools, but also considerable training and education efforts. To make people ready for MDD, its principles and applications need to be taught to practitioners in industry, incorporated in university curricula, and probably even introduced in schools.

Industry is striving to improve their practice of software development by adopting MDD. The adoption, however, is determined by the availability of skilled software engineers who are educated and trained in modeling and model-driven development. On the other hand, teaching model-driven development skills slowly influences the practices in industry with an increasing number of graduates capable for realizing the vision of MDD

The educator's symposium at the MoDELS conference, the premier conference devoted to the topic of model-driven engineering of software-based systems, is intended as a forum in which educators and trainers can meet to discuss pedagogy, use of technology in the classroom, and share their experience relevant to teaching modeling techniques and model-driven development.

The first educators symposium was organized at MoDELS 2005. The leading topic of this symposium is the synergy between industrial needs, influences on education and vice versa. A special emphasis will be put on the synergy between industrial needs and university education.

The papers accepted for presentation address the issues of industrial relevance of the education, which was one of the main topics of the symposium, such as:
- experiences with teaching modeling throughout the software engineering curriculum
- using project-based learning as a vehicle for teaching modeling
- teaching modeling through student projects where parts of tools are implemented
- teaching modeling in the context of J2EE applications
- using an artificially created software development laboratory as a means of enhancing the motivation for learning modeling

All papers determine model-driven software development as the necessary skills for the future software developers.

The diversity of authors from various countries from 2 continents provides an opportunity to compare the industrial views on modeling – from modeling being a desired skill in industry to modeling being only a surplus (while the foreseen competence was in the tools and technologies).

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

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.

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. 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
James Vallino

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

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

Nov/Dec 1990, pp.15-24
If You’re Not Modeling, You’re Just Programming: Modeling throughout an Undergraduate Software Engineering Program

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns Elements of Reusable Object-Oriented Software. Reading: Addison-Wesley, 1995.
Teaching software modeling in a simulated project environment

Robert Szmurlo, Michał Śmialek
Warsaw University of Technology, Warsaw, Poland
szmurlo, smialek@iem.pw.edu.pl

Abstract. Teaching software engineering in the academia always faces the problem of inability to show problems of real life development projects. The courses seem to be unable to properly show the need of using software modeling as important means of coping with complexity and handling communication within the project. The paper presents format of a course that tries to overcome this. It focuses on application of modeling tools in a realistic software engineering environment. The objective is to teach best practices of software design and implementation with the use of UML. The students can practice design and communication techniques based around CASE tools in teams of 12 to 14 people. The paper summarizes 5 years of experience in teaching modeling with CASE tools. Authors present a concept of how to simulate the roles of architects, designers and programmers as close to reality as possible. This is accomplished unlike in standard courses where the students implement their own designs. In the presented approach students have to implement the design of their fellow team members at the same time cooperating to achieve a common goal. Educational benefits and drawbacks of such an approach are presented. The paper also discusses the problems of organizing laboratory work for a large group of students. Authors present the tasks and their arrangement during the course.

1 Introduction

When teaching software engineering in the academia (and perhaps also in the industry) we face a very important problem of inability to show the reality of an actual software development project. Two elements seems to be important here: scale and communication. A typical project in the software industry produces tens or hundreds of thousands of lines of code which means hundreds or thousands of classes (in Java, C# or other OO language). Moreover, such a project involves many developers that play different roles (architects, designers, etc.) and need to communicate efficiently. On the other hand, typical group projects in the academia involve two to four students that together produce several hundred or perhaps several thousand of lines of code. With such a scale it is relatively easy to manage the code without any special models or modeling tools. With only a couple of students involved, the communication between them is relatively simple and can be accomplished through frequent “code analysis” meetings.
Problems in teaching scale and communication result in that the graduates coming to the industry are not prepared to participate in real scale projects. This seems to be one of important causes for poor usage of modeling tools and applying in many projects an approach of “code first, then document”. Many real projects seem to duplicate the process of creating code gained in courses where a project produces a couple of hundred of lines of code.

Considering the above, there seem to be a very serious need to design a course where the students would learn how to apply modeling techniques in a realistic development environment [1, 2]. This environment would need to be based on an assignment to create a significantly sized system in a team of more than 10 student developers. Such an environment would allow the students to apply the knowledge they gained about modeling and software lifecycle process during traditional lecture-oriented classes. It would also allow to apply modern CASE tools in an environment where such tools are really needed.

When trying to design a format for such a course, several problems arise. The most important of them is how to organize students in larger groups. Large group contradicts “fairness” of marking as it allows poor students to “hide” behind those that really do the work. Another problem is communication in a large group of students, as they normally don’t meet regularly “at work”, as people in real projects do. Yet another issue is how to divide responsibilities and distinguish between analysts, architects, developers, testers (and so on) [1].

In this paper we describe a course format designed by us which tries to take into account the above issues. The main goal was to teach the students how to use software modeling tools to organize their work in a realistically sized project. The main idea here is that students should be organized in larger “project teams” (of more than 10 people). At the same time they need to be marked in smaller groups (of up to 3) and also individually. An important constraint for the course is the time assigned to it (30 hours for common lab work plus work at home). This necessitates certain simplifications of the development lifecycle. Finally, the goal of the course was to teach proper organization of group work around good quality architectural models that are consistent with precise requirements models. This makes it necessary to organize work around components with well defined interfaces assigned to small groups. These components should be independent enough to prevent from a situation where a poorly performing group endangers other groups in the same project causing the whole project to fail and lower the marks significantly. This all should lead to a situation where the students would not hesitate to apply best modeling practices gained during the course in their real working environments after graduation.

In the following sections we will present the designed format in detail. We will also analyze the teaching results as based on the observations of several simulated projects throughout five years (almost 500 students in around 32 projects). We will also present survey results that show the level of student’s understanding of software modeling and its role in the development process. We will also present certain limitations of the course and possibilities for its further improvement.
2 Course format

The course entitled “Object Oriented CASE Tools” is taught as part of an MSc degree in Computer Engineering at the Warsaw University of Technology. The students are assumed to have knowledge of software modeling in general and UML [3] in particular. Most of the students should have already taken a course in object oriented analysis and design or equivalent.

![Diagram](image)

**Fig. 1.** Project team divided into sub-teams with assigned models and code

The presented course is taught for five years already. It consists of 30 hours of lab sessions plus student’s work at home (individually and in groups). The students are suggested a wide range of textbooks for the course [3-7], where [3] describes the software development process (including analysis and design) being the basis for the course. Throughout the five years the course format had evolved, however the main concept remains unchanged. The assignments are based on larger groups of students (around 12-15 people). During one semester, several such groups (usually 5-6) are formed to constitute “project teams”. These teams divide themselves into smaller sub-teams (architects + designers/programmers). This division is illustrated on Fig. 1. Usually, a sub-team of architects consists of three students. Designer sub-teams involve basically two students.

The project teams receive an assignment to write a small but complete software system according to a specific requirements specification. The teams have to design their systems by dividing them into components and then into classes. This is done under constant supervision of the teachers. Components forming the architectural model are managed by the architectural sub-team (see Fig. 1). Individual components are assigned to design sub-teams. These sub-teams have to design and then implement classes realizing interfaces of assigned components.
The development efforts of design sub-teams are controlled by the architectural sub-team. Architects integrate the whole system and assure proper communication between other sub-teams to make this integration feasible.

Fig. 2. Mappings between models and code produced during the course

All the development efforts in a project team are centered on the usage of a CASE tool. The teams have to produce three UML models: component model, design class model and interaction model (sequence diagrams). These design models are based on a use case model supplied by the teachers as part of the assignment. Moreover, the students have to write scenarios for the assigned use cases. Teachers assure that the models are consistent and strongly suggest the use of layered architecture. An important part of the work is to generate code from models and assure their synchronization. Detailed design models (class and interaction) are checked for compliance with the architectural model (component and interaction). Code is checked for compliance with detailed design.

The relationship of models produced during the course by project teams is illustrated on Fig. 2. The architects (in cooperation with designers) produce component diagrams and also design all the interfaces to these components. Use case scenarios are designed with architectural interaction diagrams compliant with the component model. The design sub-teams have their components as-
signed and then produce appropriate class diagrams that define the structure of each of the components. Architectural interaction diagrams are transformed into detailed design interaction diagrams. These describe the details of each of the interface operations, as to be implemented inside the considered component.

The project teams constitute themselves during the first meeting of the course which takes 2 hours. This time is also used by the teachers to explain all the details of the course format to the students. It can be also noted that at this moment the students can choose not to attend the course (for some students the course is not obligatory and they can exchange courses during the first two weeks of semester). The disturbance caused by some of the students resigning is minimized by making the original teams a bit larger than expected average. In the end, for each student group counting originally around 26-28 students, two project teams of 11-13 get formed. After the initial meeting, the remaining time is divided into three phases. In the first phase (6 meetings), the students have to design the system. During the second phase (5 weeks) they implement it. Finally, the last phase (3 weeks) is devoted to testing.

The limited time the teams have to develop the system puts certain constraints on the size of the assignment. The system should be simple enough to be implemented by inexperienced developers in around 60-70 hours (800-900 man-hours). Having this limitation, the system can be quite small but non-trivial which we will discuss later. Additional constraint about the system is the ability to divide the project teams into sub-teams responsible for their parts (components). This constraint is necessary for two reasons. The students work individually at home and thus they must be able to concentrate on specific parts of the system. The second reason is the necessity to mark the students’ progress on a more individual basis.

In order to cope with the above issues an appropriate choice for the assignment domain area is very important. In our five years experience with the course we have chosen three different domains for the applications to be built by the students. In the first year, the students had been creating a management system for an academic departmental library. The system was to be able to handle the book inventory and manage subscriptions. The requirements for the project assumed around 20 use cases concentrated on several actors: library staff members, library managers, system administrators and subscribers. In the second year the assignment was to design and implement the simulator of an intelligent building. In this assignment, the students were to design a home appliance control system (HACS). The system should had simulated such devices as automatic curtains, indoor/outdoor lights, gates, air conditioners, clocks, bells, security sensors, etc. Here, the main focus was put on designing a distributed system of devices and a communication protocol between them. The central role in the architectural design was played by a message dispatcher module, finally implemented by the architects. This assignment seemed to be perfectly suited for separating clearly the individual parts (components) of the system. The sub-teams were implementing individual devices which they could choose from the list given above. The architects sub-team was responsible for design of the communication protocol
and message dispatcher module. This domain area proved to be well suited for teaching modeling, at the same time allowing for clear division of work between sub-groups (it was also used by others, see [1]).

![Diagram](image)

**Fig. 3.** Typical component diagram of a dean’s office system which shows complexity being too high for the limited course schedule.

In the third and fourth edition of the course, students were to design and implement a management system for the dean’s office. In this project the system should support registering student candidates, promoting candidates into students, employing teachers, storing the students’ final grades, etc. The system was divided into several functional packages. The division was performed on the basis of actors interacting with the system. The assignment given to the students distinguished the roles of a candidate, student, lecturer, dean and dean’s office staff member. Figure 3 shows a component diagram suggested as a static architectural model of the system. The requirements for the system included more than 20 use cases and proved to be too complex for the course. In the second
year, the assignment was modified so that the students were explicitly asked to limit the number of implemented use cases.

In the most recent edition of the course, we have simplified the project requirements and reduced the number of use cases in the assignment. In this project, the students have to design and implement an electronic ticketing system of a theme park. The requirements for the system contain just eight major use cases (with two extra for larger teams). Appropriate use case diagram is presented in Figure 4a. Having such a functional requirements model, the system can be easily divided into four (or five) distributed application modules based on the actor involved. The presented use case model was communicated to the students through a model template file (see Fig. 4b - “Use cases” package). This file divides the model to be delivered by the students into Requirements, Architecture and Detailed Design (Components). The structure of the model and division into major packages is consistent with the project’s lifecycle and facilitates communication based around requirements and architectural design.

During the first meetings in the design phase, the teachers present and discuss the functional requirements. They also enforce some design practices. The main design concept used in classes is the application of an extended three layer architectural framework. We introduce three traditional architectural layers: presentation, business logic and application logic and add an additional data access component layer. The introduction of data access layer plays a special role. It allows to integrate the vertically divided system. In a classical approach the three
layered model is divided horizontally. This means that usually, separate groups of designers and programmers create each layer of the system. In our course we divided the tasks vertically. Each sub-team designs and later implements the three layers of an assigned “vertical” module. We have chosen this solution because this allowed sub-teams to work independently and later facilitated fair assessment of their work. Moreover, the students had the opportunity to learn the roles of all the layers in a layered architecture.

The overall project timeline is divided into three phases: design, implementation and tests. Throughout the phases, the students can choose to play two different roles. Some of them constitute the architectural sub-team and the rest form 4 to 5 sub-teams of designers/programmers. In Figure 5 we present in detail the tasks performed by both types of sub-groups. In the first phase each designer/programmer sub-team creates the designs for the assigned use cases (two chosen from Fig. 4). The architects create an overall component diagram containing all the components of the system (including data access layer components) and design the database structure. During the second phase, the designed components, database and interfaces are implemented.
The design and implementation phases of the project are associated with a concept of exchanging designs between sub-teams. In the second part of the course every sub-team has to implement scenarios and components designed by another sub-team. The two sub-teams simply exchange their design models. Students have to explain their designs to members of the other sub-team. This approach allows to gain knowledge on the kind of information that should be included in the design models, and on proper ways to present this information. This also is a good way to illustrate various kinds of obvious mistakes and mis-designs that had been made. In the last phase of the course, the students have to design and run acceptance tests of the system. Within the described division of the course into design, implementation and testing, the course has several milestones, set to allow for marking of the students' progress and systematic work. In Figure 5 the milestones with suggested marking system are presented. The marks sum to 100 points. Generally, half of the points are assigned for the quality of design and the rest - for the implementation. The majority of the points are assigned to sub-teams. However, some of the points (15%) are assigned for the performance of the whole project team.

In the first editions of the course we have tried to simulate the iterative project lifecycle. However, the limited time did not allow us to simulate it properly. In the recent course, we have limited the lifecycle to just one “normal” iteration with a second “corrective” iteration. This allows us not to make the process shift into a pure waterfall (see eg. [8, 9]). The second, short iteration allows us to simulate the iterative lifecycle [10] to some extent through a second deadline for acceptance tests. The first deadline simulates tests after an initial iteration in a project. Normally, the tests fail demonstrating to the students the problems with waterfall approach, where it is normal for the system not to pass the tests (for various reasons) endangering seriously the project. In the second “iteration” the students do not produce new functionality but correct (or enhance) these parts of the system that caused the tests to fail. With this approach we can show to the students the need to organize their projects with a “risk buffer” of an iterative process.

During the course students play two major roles: architect and designer / programmer. Both roles have to accomplish different tasks. In Figure 5 we present a scheme for their tasks which illustrates the individual and shared tasks with different colors. Tasks common to the whole project team are marked with dark gray. The division into individual and shared tasks is motivated by the need to grade individual sub-teams and and at the same time to give importance to group work character of the project.

Important part of the course is devoted to the administrative aspects of a software development project. The students have to use a CASE tool available to them in the University’s labs. For the purpose of this course we have used Enterprise Architect by Sparx Systems [11]. The tool allowed for integrating models made by individual sub-teams into a common model. This common model could be automatically published on a project team’s web page. The web pages
were used by the teams to communicate (including communication of changes in the architecture and interface implementation) and verify progress.

Apart from the CASE tool, the teams had to organize their implementation and testing environments. For this purpose, each team was given their own VMware virtual machine [12]. The architects (in cooperation with other team members) could choose their own operating system and development environment in which they wanted to implement the system. The teams normally choose .NET (3 teams last year) or J2EE (4 teams) as their development frameworks. The architects are responsible for maintenance of the overall structure of the system and the design and implementation of the data access component layer. They design and synchronize the component model of the system with specific diagrams of individual components. One of architects in each team is also responsible for maintenance and administration of the VMware machine. The other architects were personally responsible for database management, web page maintenance and keeping up-to-date the UML documentation of the whole system. The teams create their own websites which integrate the whole documentation of the project, most recent information and announcements. The website plays a crucial role as the second communication medium between students in the team and complements the meetings during the classes. The website also constitutes a report deliverable for the whole project.

3 Teaching results

The teaching results can be divided into three aspects: quality of produced systems and team work, quality of produced designs and student’s comprehension of modeling. The first two aspects can be judged by the quality of deliverables and generally by measuring how successful individual project teams were. To measure the third factor we have prepared a post-course questionnaire.

Five years of experience in teaching the course leads to a general conclusion that the proposed format proved to be quite successful. This success can be measured by the fact that out of 32 project teams (12-15 students each) only 1 (throughout the five years) did not succeed in delivering the assigned system. In all other cases the systems delivered were functional enough to pass most of the prepared functional tests. It has to be noted though that in the first three editions of the course, most of the teams did not deliver all of the required functionality. Nevertheless, the students managed to deliver fully operational systems with good user interface design.

The most important problem during classes was the organization of team work. This was caused by the fact that the students were really not prepared to work in a larger group. This was a totally new experience to them. For this reason, the course gained an opinion of a “hard one” - one necessitating more work from the students than an average course. Despite this, the students participating in the course have shown high commitment and motivation. They were additionally motivated to work by fellow students - project team members. The teachers could notice that the project teams became real teams with good inter-
personal relationships. Only in some cases (2-3 teams out of 32) the team work did not succeed. This was especially the case when the architectural group did not perform well. It is thus crucial to have the architectural team being carefully chosen by the project team with the help of the teacher.

Another issue is associated with poorly performing design teams. The course format proved to be well suited to accommodate for such situations. None of the projects so far suffered because of one of the design teams did not do their job or did it too late. Vertical division of work made applications prepared by the students quite independent. Even without one of the modules, the applications could work simulating easily the lacking functionality.

Yet another issue associated with team work is the organization of the project's lifecycle. We have noticed that the iterative approach to system development cannot be properly simulated because the number of classes during the course (15 × 2h) is too limited. The course could actually simulate just around one-and-a-half weeks of a real project. In the first editions of the course we tried to simulate two or even three iterations. However, this only caused too much chaos when trying to perform testing sessions three times during the course. It also proved to be infeasible for the students to produce a reasonable part of the system to test in 4-6 weeks (course time allocated for one iteration). In the last edition of the course we have modified the format to simulate just one iteration with the possibility for another “half-iteration”. This was done by fixing two deadlines for tests. The first deadline was for preliminary tests, where a limited functionality of the system was tested. After this, the teams had two weeks for fixing the problems found and including the remaining functionality into the system. This proved to be a better solution which did not cause any chaos but at the same time allowed the students to appreciate certain benefits of the iterative approach.

Quality of team work is closely related to the quality of produced deliverables (models). In the initial editions of the course the students were to deliver systems with around 20 or more use cases. In the last two editions we have significantly limited the number of required use cases (to even less than 10). This was caused by noting that the students have sacrificed modeling when pressed by the deadlines, and concentrated mostly on coding. Lack of modeling caused poor quality of the systems and later problems with testing. In the last editions, the teachers have concentrated on enforcing proper modeling practices. This caused that the teams produced better designs leading to easier integration of systems and significantly less problems during testing.

A common problem in teaching modeling is how to motivate students to produce good quality models. Usually, they produce models that they later implement in code. The models are quite simple and the students feel that they could develop the system faster if they could get rid of the “unnecessary pictures” (which they have in their heads). To prevent this, in our course we have simulated the process of exchanging design information between separate sub-teams. In the second part of the course, the students were obliged to implement components designed by another sub-team. This forced them to present and explain
their project to others. During this process both the presenting and the listening team (under the teacher’s supervision) were gaining experience on the kind of information that has to be included in the design. Additionally, the students became aware of many obvious errors, mis-designs and shortcomings of their projects.

Another problem is associated with the usage of modeling (UML) tools. Very often, the students are taught to code the system first and then prepare the documentation. Such habits could be noticed during the described course. Some of the teams tried to reverse engineer their code in order to produce the required design models. The teachers tried to prevent such practices by frequent inspections of code and models. On the other hand, many of the teams did not need such inspections as they have noticed all the benefits of automatic code generation and kept code synchronized with the models without teacher’s intervention. The marking system developed for the course promotes the teams that keep their designs up-to-date. However, we have noticed that even with this system in place, the students needed additional motivation. This motivation was gained through a systematic use of the CASE tool. The students realized that the tool allows them to communicate design decisions and at the same time relieves from the burden of writing the “uninteresting” parts of code (the code structure).

<table>
<thead>
<tr>
<th>No.</th>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>It would like to attend such courses during the studies more often.</td>
<td>4.0</td>
</tr>
<tr>
<td>2</td>
<td>Good architecture design in UML helps to communicate during the process of system development.</td>
<td>4.2</td>
</tr>
<tr>
<td>3</td>
<td>The course have helped me to understand the process of system implemen-</td>
<td>3.7</td>
</tr>
<tr>
<td></td>
<td>tation with UML.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>The course will allow me to integrate faster with a team building a real</td>
<td>3.6</td>
</tr>
<tr>
<td></td>
<td>system in the future.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>It is not possible to create a good large scale and complex software</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td>system without modeling (eg. in UML).</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Application of CASE tools is necessary in the process of software</td>
<td>4.2</td>
</tr>
<tr>
<td></td>
<td>development.</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Results of a voluntary questionnaire conducted on a group of 75 students.

To measure some of the teaching results of the course, associated with the student’s comprehension of modeling practices, we have prepared a questionnaire to be filled by the students. In this voluntary, anonymous questionnaire, 75 students out of total 90, have participated. Students were to assign 1 to 5 points for each statement, where 1 point meant: ‘I definitely disagree’ and 5 points ‘I definitely agree’. This means that the more points were assigned the
stronger agreement was expressed. The results of this questionnaire are presented in Table 1.

Most students taking the course accepted the innovative form of the laboratory (4.0). The students had the opportunity of playing different roles in a realistic environment. There were architects who handled the system globally and controlled the compatibility of all components. Other students could concentrate on details of smaller parts of the system. There were also some purely technical roles like keeping the webpage of the project and administration of the computer system where the software was running (VMware). The students could thus find a role that suits them best. On the other hand, some of the students were quite sceptic about whether they would like to take more of such courses due to amount of work the course necessitates.

Answers to questions 2, 5 and 6 have shown that the students really appreciate modeling and the role of CASE tools as an important element of software development. On the other hand, they seem to be more sceptic about applying the knowledge and experience they gained in real life (question 4). This might be caused by their poor opinions on the quality of software development process in an average software house. Answers to question 3 show that the students were not totally satisfied with the outcomes of the course. Their additional opinions expressed when discussing with the teachers show that they feel the course did not fully simulate a real project. They also felt that too little stress was put on transforming UML models into the technologies of their choice (.NET or J2EE).

4 Conclusions

Current curricula for software engineering courses seem to ignore the need to prepare students for work in a real-scale software project. It can be argued that this is caused by applying traditional approaches which concentrate on “fair” assessment of individual students. This however is in contrast to the needs of the industry. In the presented course we have tried to accommodate both the needs of academia and the needs of industry. We try to simulate a real industry project, at the same time introducing a system which allows for quite fair marking. This is accomplished by proposing an architectural framework with vertical division into separate applications in a three layered architecture. Additional data access component layer allows for relative independence between the sub-teams.

The course simulates just one (and a half) iteration of a real project. Experience shows that realistically, only up to 10 average use cases can be implemented. This is due to the fact that the course can simulate just below two weeks of full time work in a real project. Despite this limitation, the teams could produce a reasonable system. Even in this limited lifecycle, the students gain the capability to communicate with other members of their team through models. They also gain experience in conducting project meetings, organizing team work. Most importantly, the students really start appreciating modeling and the use of CASE tools - not only as means of documenting the system but also as facilitators of actual development process.
The authors still see certain elements that need improvement in the course’s format. One important element here is the simulation of an iterative lifecycle. This however, seems to be feasible only in a course spanning for a longer period (a two semester max-course). Another problem is associated with the use of distributed component technologies. This knowledge is not even among student taking the course. This means that some of them have to study .NET or J2EE individually. An idea here would be to introduce an initial “hello world” application to be prepared by sub-teams in order for them to gain some initial knowledge of the chosen technology. This seems to be crucial as it is practically impossible to teach design modeling abstracting from the technology used.

5 Acknowledgement
We wish to thank Sparx Systems for supplying us with their CASE tool system (Enterprise Architect) for the purpose of this course.

References
Teaching modeling of reactive systems

Øystein Haugen
University of Oslo
oysteinh@ifi.uio.no

Abstract. We present one approach to the teaching of modeling at University of Oslo. We aim to combine semi-real projects with home-grown tool pieces integrated with commercial tool support. In this paper we discuss our approach, the tools and our projects as well as our motivation for doing it this way.

1 Introduction

The students of Department of Informatics of University of Oslo are exposed to modeling in several different ways. The beginner students are introduced to programming via Java in their first course. Then modeling is given a chance. The approach that we present in this paper is given to students of the second or third year of the Bachelor studies, and is a project course. Some Bachelor courses need to be project courses implying students working in groups of 3-5 and receiving only pass/fail rather than the grades A-F.

The purpose of the course which is the base for this paper (INF2120 – Project in Modeling1) is multifaceted as most university courses are:

1. The students should learn how to apply what they should have learned already.
2. The task of the course should be so big that it would be impossible to perform it alone.
3. The students shall experience project work.

Fearing that only applying what you have learned would be too dull, we added a few more learning objectives.

4. to learn how concurrent, reactive systems are built,
5. to learn how to combine technical, reactive systems with more administrative database systems,
6. to connect to (semi-)professional software that provides very cool and useful functionality, and finally,
7. to experience a variety of interconnected tools.

1 http://www.ifi.uio.no/inf2120/ Administrative information in Norwegian, but course material is mainly in English.
The paper is organized as follows. First we present the format of the course in more detail, and then we present the topics for the projects the two years the course has been taught. We present the engineering method before we go into the tools that we apply in the course. To give the reader more intuition, concrete examples of student solutions are presented before the experiences and student reactions are summed up.

I would like to thank all my students and assistant teachers over the two years for their hard work and enthusiasm, and especially I would like to thank the three participants of Group 23 this year whose work I have been allowed to use for illustration, Øyvind Dahl, Tommy Gudmundsen and Halvord Vinger.

2 The format of the course

The course is a so-called project course and the grade is formally only Pass/Fail. Individual students of project teams may fail due to individual examination.

2.1 Organization

The course is supposed to cover 33% of one semester workload spread evenly (almost) over the spring semester lasting from early January to early June. The course has run twice and we have had 40 students each year. The students are divided into two exercise groups, each exercise group assigned to one group teacher. There are four hours group time in the beginning of the semester and only two hour group time during the last half. In the beginning two hours group time is spent in a room with computer terminals connected to the Linux cluster of the department. This is called “lab-time”.

There are two hours lecturing each week. I give most of the lectures, but I have sometimes let others do the lecturing on some subjects such as the database modeling.

Within each exercise group the students are divided in project groups ranging from 4 to 6 members. The groups are composed randomly to break up super-teams and let the students experience having to cooperate with total strangers. This approach has proven both constructive and destructive for the obvious reasons. Some students dislike to be separated from their buddies. In most project groups one or two people become the drivers and it is not always the case that they manage to get all the others working properly. On the other hand all the teams manage to reach an adequate level of achievement, but a few students drop out – more or less voluntarily.

The group teacher act as supervisor for the project teams and may have extra supervision with them on demand.

All the lectures and supplementary material are available on the web and we build up a FAQ for difficulties experienced throughout the course.

The students are encouraged to cooperate not only within the group, but also between the groups. Furthermore we welcome any solutions that the students can find by using the Internet.
2.2 The project

Each time the course runs we make a new project, but the two projects that we have had, have clear similarities. All the project groups get the same task. The project description defines three drops with very hard deadlines.

The first drop simulates the requirements phase and the students provides a report containing a UML description with use cases and sequence diagrams as the main ingredients combined with class diagrams and composite structures. The UML descriptions must also be accompanied with explanatory text.

Following the delivery of drop 1, all deliverables are made available on the web and another designated group will criticize the deliverable. A few days after the deadline we have presentation and criticism. The group teachers sum up their evaluation and reference the opinion of the course lecturer who also indicates informally a grade in the range A-F. This procedure of evaluation and criticism is repeated for every drop, reversing who criticizes whom.

The main goal of drop 1 is that the students appreciate the extra precision offered by UML 2 sequence diagrams, and experience the problems and benefits of model consistency.

Drop 2 is the result of the design phase where the result is a UML model with state machines in addition to the class diagrams, composite structures and sequence diagrams. The main goal of drop 2 is that the students realize the benefits of behavioral consistency between the sequence diagrams and the system of state machines.

Drop 3 results in an executable implementation. The requirement is that the implementation must be transformed directly from the UML model. There are two different ways to present drop 3, either the project team gives a flashy demonstration, or they show how they validate the system through testing and/or walkthroughs. The main goal of drop 3 is to execute the system in a semi-real environment.

3 The topics

We are cooperating with the telecom operator Telenor so that we can send and receive SMSes and position GSM phones. This gives us good opportunities to make projects that aim at making interesting and semi-real applications.

In 2005 our aim was to find the nearest bus stop to the user based on positioning his/her mobile phone, and then find the first bus for a given destination. We connected real-time to the Trafikanten traffic system of Oslo and could get from there actual real positions of all buses based on GPS navigation terminals within each bus. We had also the accurate positions of the bus stops (limited to one bus line). The students were encouraged to find modifications to the suggested services. Typically the user would send an SMS like "goto Blindernveien" and receive an SMS answer with the next few buses from the stop "Blindernveien". There were also modifications such as to give different starting points than your own destination or to give a delay.

In 2006 we modeled the private surveillance system "ICU" where positioning is used to place your buddies on a GoogleEarth map. Modifications include user
interface with SMS where you may ask "position Trine" and get the return "Trine is 300 meters from Blindern" provided that Trine is in your buddylist and Blindern is a registered hotspot. Furthermore you may look Trine up on GoogleEarth provided that she is inside Norway. (Outside Norway we only get in which country she is.)

4 The engineering method

This course is not mainly a course in software methodology. It is a course for gaining experience. The students are encouraged to use the approaches they have been taught so far [1]. We see that the students are reasonably good in making conceptual models with class diagrams and that they are able to produce simple textual use cases to supplement their use case diagrams. They are not so familiar with more formal modeling where also the behavior must be defined in detail by the modeler.

Our supplementary literature includes two chapters authored by Birger Møller-Pedersen, Thomas Weigert and myself [2, 3]. The important theme here is to show that UML has good means to ensure consistency between the requirements (defined by sequence diagrams) and the imperative design (defined by the set of state machines). We also demonstrate to the students how asynchronous messaging is practical in situations where a possibly large number of actors use the system simultaneously.

Since the students are expected go all the way from the requirements specification to an executing application, we have found that it would be helpful for them to have some examples to look at and copy from. Therefore I created a system that demonstrated SMS communication and positioning as well as producing data for GoogleEarth. It had no database and no buddy lists. This “basic functionality service” was released when we had had the review of drop 1 because I did not want them to be influenced by this example model in their requirements specification.

After drop 2 we released another model that demonstrated the use of the database. This example model showed the recommended architecture where one state machine encapsulates the Oracle database. Furthermore it gives the java-magic that is needed to apply SQL clauses to the database. The reason for providing these example models is the recognition of the fact that modern modeling and programming can be seen as “design by example”. Furthermore it is much easier to give a concrete example of certain low-level details than to describe it in a lengthy document. Actually we try and provide both since there are help documents for all the pre-made units of the system.

The way we have chosen to conduct the project work is by no means obvious, and we are not sure that we will follow exactly the same path the next time. We have chosen not to put a lot of emphasis on the project organization itself. We have let the students (with supervision from group teacher) find their own organization. The main reason is that the course is about experiencing what you have learned and there are other courses that focus on project organization. We may choose in the future to give more detailed guidance on project roles and responsibilities as a countermeasure against the evolving of a “dictator” in the project group.

The second issue is whether we should try and apply a more iterative approach even for such a small project. One could think about reaching a very small minimal
system early and then evolve that system on later drops. One problem with an iterative approach is that we need time to teach advanced sequence diagrams and state machines. Furthermore there is quite a multitude of tool details to be handled to be able to generate an executing system. This is the reason why we did not want the students necessarily to reach an executable model for drop 2. We did not want to focus on programming and tool details before the design principles were mastered.

5 Tool Ingredients

Tools are important in software engineering. But we often find that no tool is perfect and no single tool can cover all we want. In our course we use a number of tools for modeling and programming as well as a set of libraries and frameworks. In this section we shall go through them and give our reasons for why we include them.

![Diagram of toolset](image)

**Fig. 1.** The Ifi UML Toolset for spring 2006 in the modeling project course INF2120

In Fig. 1 we give a graphical illustration of the different vehicles that we apply in our course.

5.1 Eclipse

Using Eclipse has become very common in universities, and our reasons are not very different from others. Since we have based our in-house computers very much on Linux it is a pleasure to find something that can bridge over the operating system gap over to the students’ own laptops. Eclipse also provides the students with an adequate environment for Java programming and debugging.

The experience is overall very good, but some initial problems are encountered regarding the initial setup and the differences between versions 3.0 and 3.1.
5.2 IBM Rational Software Modeler

IBM Rational Software Modeler – RSM – is a commercial UML tool that we have acquired through the IBM Scholar Initiative. The students also have the tool on their laptops.

It is large, and running it requires a fairly powerful computer and many students find that they need more RAM before their laptop again behaves adequately. The tool is comprehensive and covers most corners of UML 2. Their tool for Sequence Diagrams, however, is not at all adequate and we have found that we needed to find something better. The main problems are that RSM lacks support for gates and for decomposition, both features that are absolutely necessary since consistency between sequence diagrams and state machines is one of our key engineering invariants.

Furthermore it does not seem to handle asynchronous signaling at all which must be an unintended bug.

Otherwise the tool has the Eclipse look-and-feel and supports most of the state machine mechanisms.

5.3 SeDi – our Sequence Diagram Editor

SeDi is the name of the in-house sequence diagram tool made by a Master student (Andreas Limyr) as part of his Master work and refined by Master student Frank Davidsen. The software is built on Eclipse modules such as GEF and EMF.

SeDi can create sequence diagrams with gates and decomposition and it can produce, but not read, the UML2 format. Most symbols are included, but not all. Some symbols are included in the editor, but they do not properly snap and glue to the lifelines. Thus SeDi is more than a simple drawing program, but less than what we want in the future.

That it does not integrate seamlessly with the UML2 repository makes it in fact very separate from the RSM, but still the students find it simple to use. SeDi can be imported as a plugin within the Eclipse environment of RSM such that it on the surface looks well integrated with RSM.

5.4 The UML-to-JavaFrame Transformer and the corresponding UML profile

A very crucial piece of software is our UML compiler. We transform the UML model represented by the UML2 repository to a set of java files. These java files are based on a framework called JavaFrame which was made by Ericsson Research in Norway a few years ago and made available to us on a non-commercial basis.

The transformer is made by Asbjørn Willersrud, a Master student, as a summer job in 2005.

The UML compiler cannot compile the whole UML2. It takes for granted that the system consists of active objects described by state machines. The system may be hierarchical, but the leaves are all state machines. Every node in the system aggregation graph that is not a state machine must be described by a composite structure. All connectors connect via ports. Thus the transformer transforms a subset
of UML, and since we also apply a few stereotypes, we may say that we transform a model based on a JavaFrame UML profile. This is similar to what most others do when they generate code from UML.

The transformer has been very stable and the students have found close to no errors in it, but we have ourselves improved it through a couple of versions during this semester.

5.5 JavaFrame

As mentioned above JavaFrame is a framework in Java. It contains a UML runtime system (provided your UML model is as mentioned above). You may deploy your system in several threads or in only one, on several machines or on only one. That the systems are thread-safe is one of the benefits that we emphasize for our students. Typically their projects are such that an unknown number of concurrent services are to be served. How can this be done without undue stress? Our answer is sessions modeled as state machine instantiations.

The JavaFrame is also a library of useful concepts for asynchronous signaling which may be seen as a small vocabulary to be used in the transitions. The transitions are written in pure Java, but they are always fairly simple. Complicated algorithms can of course be put in methods (again written directly in java).

Before we got the UML compiler (2005) we coded JavaFrame systems manually and that was quite effective, too. In fact one could say that we instructed the students to be the compiler since we gave them the coding templates. The problem with the manual coding is of course that in the end the student are tempted beyond resistance to finish the design on the Java-level and the UML model becomes obsolete.

5.6 Oracle

The course is intended not only to give the students experience in modeling reactive systems such as telecom or surveillance systems. We also want to give the students experience in modeling and applying databases. We must admit, however, that the databases that the students are asked to model are quite simple, but they do experience that modeling the database is different from modeling the reactive system.

It is also a point that we are able to combine the reactive world and the database world. Our answer to the interface is simple, but effective. The database is encapsulated by a state machine. The students may themselves choose the protocol between this database-keeper and the rest of their system. This architecture has the big advantage that the state machine takes care of the synchronization of the requests, and furthermore represents an interface that can easily be substituted.

We have chosen to apply an Oracle database because this has already been used in one of the beginner courses that they must have had, but we could easily switch to other implementations of SQL-bases or even simple sequential files and transient tables if we thought that would be appropriate.
5.7 PATS and Trafikanten

Since we believe it is motivating for the students that we make semi-real systems, we are happy that we have been able to attach to some third party vendors that have provided us with real-world functionality. By the term “semi-real” we mean that the students are making programs that in principle do something useful or funny. There may be restrictions such that you will not actually afterwards use the made systems, but their functionality is close to what you would have used.

PATS (Program for Advanced Telecom Services) is a research agreement between the Norwegian University of Science and Technology (NTNU) Department of Telematics, Ericsson, Telenor and Hewlett-Packard, but University of Oslo and others have been able to piggy-back on the initiative. The services that we have been using are the ability to get and send GSM SMS messages and to position GSM mobile phones.

We have no doubts that being able to make services that actually work on your mobile phone is considered “sexy” by the students.

Trafikanten is a service organization that provides schedules etc. for the public transportation system of greater Oslo. They track in real time all buses and trams such that their positions are known. From this you may get a projection of when a given vehicle will arrive at a designated stop. We have been allowed to access this information through a simple http protocol. We have also been given information about the whereabouts of some bus stops (not the whole base).

In both the PATS and the Trafikanten case we encapsulate the communication with them in special kinds of ports and special signal classes. In this way the students access these external services just like they were internal state machines.

6 Examples

In the following we shall give examples of what was produced by one of the student project groups during spring 2006 where the project was to make a personalized surveillance system “ICU”. The text of the diagrams are fairly small and in Norwegian, but this is not very significant since we explain in the text what the examples show.

6.1 Drop 1: Sequence Diagrams and composite structures

We would like to show from the first delivery of one group how they very early tried to model hierarchically and how the sequence diagrams were reasonably consistent with the composite structures. This group also applied textual use cases to motivate their sequence diagram, but I have omitted them due to space limitations.
We see in Fig. 2 that the students have not yet applied ports. This top level context collaboration will never be transformed anyway. It is only the System:JSD23 that will be their application, and that is broken down in Fig. 3.

The reader should not see these illustrations as examples of endorsed architectures, but rather as the result of very early work in a Bachelor course.

We show also two sequence diagrams in Fig. 4 and Fig. 5 where there is reasonable correspondence between the lifelines and the composite structure parts and where decomposition is used in a proper, consistent way.
The reader should ignore that the syntax of guards are not graphically correct.

Fig. 5. System level “Register User”

We see in Fig. 5 that the students know that the combined interaction should be extra global which means it should go beyond the frame graphically, but that our ScDi tool at that time did not have the feature. We have since then made a version that does handle guards and extra global combined fragments, but we did not require that the students went back and changed their diagrams.

6.2 Drop 2: State Machines and sequence diagrams

In the second deliverable the focus was on design of the executable behavior i.e. the state machines. Naturally the designs of the composite structures and the sequence diagrams had to be slightly updated when the students got feedback and their experience level increased.
We will follow the same group and see that their system level composite structure has changed as shown in Fig. 6.

![Fig. 6. Drop 2 composite structure of system level](image)

We see that they have now introduced the ports and got rid of the “scheduler”. Since they plan to use different kinds of ports they have more connectors between the different parts. The students have also introduced a third level and detailed the upper right part JSD23Kontroller.

Following their structure modifications they do the necessary modifications of the sequence diagrams and decompose further into Fig. 7.

![Fig. 7. Controller level “Register User”](image)

The main improvement here is that they introduce a session state machine which they model as shown in Fig. 8. The state machine is close to trivial, but that is actually a good thing since it does the job even when there are many users registering...
at the very same time. It shows that the state machines motivated by the lifelines of the sequence diagrams effectively represent a separation of concerns.

In the second deliverable the students were asked not to detail the insides of the transitions, but possibly to indicate the eventual sending of signals. They were asked to give the transition a meaningful name that would intuitively tell what goes on in the transition. For the English speaking readers the Norwegian in the diagram is not at all intuitive, but this cannot be attributed to UML.

There were other state machines that were more involved than our single example here, but mostly the state machines are themselves very simple. Once the architecture is reasonably correct, the creation of the state machines can be distributed among the project members. This was practiced to a certain extent, but we realize that many of the groups had one (or two) champions that did most of the modeling, while the others participated in discussions and did trivial work delegated to them by the champion.

![State machine diagram](image)

**Fig. 8.** “Register User” session state machine

### 6.3 Drop 3: Executing and validating the model

In principle when the students came to the third and final deliverable, they should be able to execute their model more or less by pushing the compile-and-run button. Of course this was not true for two reasons. First, making an executable model requires a rigor that was not necessarily obtained in the second deliverable. Second, they needed to fill in the insides of the transitions and to implement the database management state machine.

Our example project group had a very good design in the second deliverable and did very few modifications for Drop 3. They filled in the transitions easily and executed according to specification. To validate that the system actually did execute according to the specification, the students were asked to describe a (small) number of tests with the UML Testing Profile [4, 5]. In **Fig. 9** we see the test context of the application of our example student project group.
Not all the projects groups were as good as our example in deliverable 2 and therefore had to do more modifications to get the system running. Still all groups showed steady progress during the course and all of them reached an acceptable and executable result. All demonstrations were successful.

7 Experiences

We summarize here some of the experiences seen from the lecturer.

1. Going from informal UML to executable UML is non-trivial
2. Decomposition of lifelines in sequence diagrams can be learned and brought to productive use.
3. Manually ensuring consistency between sequence diagrams and composite structures is difficult. Tool support is much wanted.
4. To model with state machines and asynchronous signaling requires some maturing, but once grasped, most students become very enthusiastic.
5. In the end the students see the essence of consistency between the sequence diagrams and the set of state machines (as well as the intermediate composites).
6. That the system in the end actually runs with “cool” things like mobile phones is an appreciated added value.

The students were given an electronic questionnaire that they filled in anonymously as one means by the University to assess quality. 22 out of 31 eligible students returned the questionnaire. We summarize below the sentiments of the students.
1. State machines were just ingenious! Great for overview and amazingly simple.
2. Nice to work with (semi-) real technology such as the PATS laboratory interface.
3. Eclipse is great. Nice to work on a cross-system platform.
4. Sequence diagrams were difficult at first, but after a while they felt natural and we understood how to use them and benefit from them.
5. The use of an Oracle database had little significance. The database was very simple and could as well have been replaced by a simple file.
6. Most students were reasonably positive towards Rational Software Modeler (RSM), but almost everyone had some negative experiences to balance the positive remarks.
7. SeDi – the sequence diagram editor plugin – was considered all right, but almost all of the students thought that better integration with RSM (and the UML2 repository) would be very desirable.
8. The UML to JavaFrame transformer was considered “surprisingly good”. Code generation became a favorite for many students, but they also pointed out some usability problems that would not have happened if they had read the manual.
9. The project work caused the most traumas. Many students experienced that the groups were unevenly composed, which in fact was intentional. This caused conflicts and uneven distribution of the workload. Especially the hardworking students found this frustrating. Some students also reported problems of the project due to difficulties with configuration management.
10. The students were reasonably satisfied with randomly composed groups. They had no desire for stratification of the students prior to group formation.
11. The students would have liked more hands-on lab time with supervision and less blackboard-oriented group teaching.
12. Everybody liked hard deadlines.
13. Most students had benefit from peer-criticism, both being criticized and doing critics. The evaluations from lecturer and group teachers were also considered very valuable.
14. Most students also accepted the informal A-F grading on drops. A few students found the grading definitely positive, while one person was clearly against such informal grading.
15. In the final evaluation, the students thought that the course had been a good one. 68% said that the course should receive a B. They were more satisfied with the teachers and less satisfied with the tools.

8 Conclusions

The challenge of teaching modeling is to give the students a smooth ride from a fairly simple and informal world to a more complex and formalized world. We aim to make projects that are bigger than one person can handle easily, but still manageable by a group within the timeframe of one semester. We emphasize consistency between different, but related descriptions. While consistency is a much appreciated property of programming languages, students seem to expect otherwise from modeling
languages. We found that executing a final semi-real system with mobile phones and GoogleEarth is a very motivating approach.

All students were motivated by getting the final system to work, and many students experienced that once the threshold of getting the system to execute had been passed, to improve and expand the services were simple.

All groups were able to have something running that did useful things, and they learned how to control their ambitions according to the time available. The central idea of consistency became clear to the students during the course and the third deliverable showed fairly good consistency. Still it is obvious that the tool support that we lack the most, is that of consistency between sequence diagrams and state machines. Can the sequences of the interaction diagrams be fulfilled by the set of state machines? We plan to make such a tool as an Eclipse plug-in.

We applied a multitude of tools and the students had few problems with that. The students were able to experience that commercial tools are not obviously better than home-grown ones.

The students appreciated modeling with state machines when they got the hang of it, but for some students it took a considerable time to reach such a level. That the models could be made executable was a big advantage to support the understanding of state machines – a concept that you do not find as a first class concept in programming languages such as Java.

All together the reactions and results of the course are such that we believe that we will try and continue approximately the same approach also next year, but as always we shall have to find a new project idea. Here are some means that could improve the course further:

- Let Drop 2 demand a very simple, but executable system from which the final system can evolve. This could induce a more iterative development.
- Put more emphasis on project work and project participant roles.
- Increase hands-on lab time and replace blackboard time by project group appointments.

9 References

UML Based Modeling and Development of J2EE Applications: Course Experiences

Dániel Varró

Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Budapest, Magyar tudósok körútja 2.
varro@mit.bme.hu

Abstract. This paper reports on teaching a graduate-level course on UML based modeling and development for multi-tier web applications with J2EE technologies at the Budapest University of Technology and Economics. The main educational challenge addressed by the course is to find a synergetic balance for educating UML modeling concepts, J2EE technologies and advanced commercial modeling tools. As such, students do not only acquire modeling expertise but they also gain experience in industrial technologies and tools, which is a prerequisite for many job offers in Hungary.

Keywords: UML modeling, J2EE, modeling and development tools.

1 Introduction

The Faculty of Electric Engineering and Technical Informatics at the Budapest University of Technology and Economics is the largest faculty of Hungary in the field with educating approximately 500 new students each year in computer engineering. For this reason, various courses are offered on both the undergraduate and graduate level for educating software engineering. Students learn the core UML notation, object-oriented software development and Java programming on the undergraduate level in various software engineering related courses, exercises and labs. Graduate-level courses extend elementary (undergraduate-level) concepts with advanced modeling techniques and technologies.

This paper reports on a four year experience in teaching a graduate-level course in the 8th semester on UML based modeling and development for multi-tier web applications. This course is jointly offered as a compulsory lecture for approximately 80 students each year of the two most important major curriculums (i.e. specialization fields) having intensive focus on software engineering.

---

* This work was partially supported by an IBM Faculty Award, the Sensoria European IP (IST-3-016004), and the János Bolyai Scholarship.

1 The official separation between undergraduate and graduate levels in the Hungarian educational system is still an ongoing activity. Anyhow, the term “undergraduate-level” will refer to the first three years of studies and “graduate-level” means the final two years to comply with the terminology for European educational systems.
One of the curriculums has intensive focus on the modeling and development of dependable services and systems while the other put its focus to advanced technologies and tools in software engineering. Thus, it is an educational challenge how to satisfy this dual focus of the two curriculums within a single course offered for students having different interests.

As a consequence, the purpose of the course is to teach the model-based development of multi-tier web applications by using advanced J2EE technologies and commercial-off-the-shelf (COTS) modeling and development tools. Our experience so far has demonstrated that it is possible to find a synergetic balance that enable students to acquire skills, concepts and best practices of UML modeling, the Java 2 Enterprise Edition (J2EE) technology and advanced COTS tools within a single course.

Modern web applications typically have a multi-tier architecture where the data layer, the business layer and the web (client) layer are clearly separated, in fact, these components are frequently resided on different nodes. The J2EE technology was selected for the course as a demonstration platform of multi-tier web applications as it is based on open standards including Enterprise Java Beans (EJB) for the business layer, Java Server Pages (JSP) and Java Server Faces (Faces) for the thin web-client layer.

Additional motivation behind the course takes several less pragmatic but more realistic (sometimes even contradicting) issues into consideration.

- **UML modeling.** Since elementary UML and software engineering concepts are already educated on the undergraduate-level, a graduate-level course should address more advanced topics including modeling tricks and best practices, etc. Despite this fact, a sad educational experience is that as basic UML is taught in the 3rd semester, while this course is in the 8th semester, many of the students forget what they have already learned, therefore, briefly revisiting the UML language cannot be avoided.

- **J2EE technologies.** The Hungarian labor market of software engineering is still rather looking for appropriate employees having in-depth knowledge in a certain technology (or even a certain tool). Since pure UML and modeling skills are not yet such appealing in a CV, it should be complemented with leading industrial technologies like J2EE or .Net.

- **Advanced COTS tools.** As pointed out in [8], the use of complex COTS tools has the risk of distracting the attention of students from modeling, and putting some additional burden on them. However, our experience shows that students at a graduate-level rather see this as an advantage, which creates a realistic look-and-feel for the entire course. Frequently, they will be using these tools after their graduation (i.e. within a year) in real projects.

The rest of the paper is structured as follows. In Section 2, the available educational resources and constraints are discussed. Then in Sec. 3, a detailed overview is provided on each major phases of the course. Later, Sec. 4 addresses various educational aspects such as the development process, the used modeling and development tools, and evaluation and feedback from students. Finally, Section 5 concludes the paper.
2 Educational Resources

Since this is a university course, the available educational resources are highly restricted and for bureaucratic reasons, these restrictions are quite static. In order to attain a meaningful course, unofficial solutions are provided.

Lectures and labs. The course is officially given in the form of two 90 minute lectures (per week) for 14 weeks. Obviously, such a “theoretical UML” and “theoretical J2EE” are of little practical relevance, especially in software engineering.

Therefore, facultative labs and consultations are offered in addition once a week where (i) the basics of tool usage are shown and (ii) questions concerning the homework assignments are discussed with the tutors.

The contents of the lectures and facultative labs are highly synchronized with the tasks the students need to carry out for their homework assignments. This way, conceptual materials discussed during the lectures are applied by the students as soon as possible.

Homework assignments. As a main part of the course, the students are required to run a multi-phase project as their homework assignment in order to design and implement a relatively complex web application using UML for modeling and J2EE as the underlying technology (which shows certain resemblance to the Object-Oriented Analysis and Design course in [6]).

Homework assignments facilitate team work: a group of four or five students forms a team, which has to report regularly to their tutors (acting as project leaders). In order to stress the importance of the homework, fifty percent of the final grade is derived from the evaluation of their project work.

During the project, there are six deadlines with different tasks ranging from UML modeling and analysis to model-based testing to J2EE implementation. In case of high mid-term workload of the students, one deadline is allowed to be postponed with a week after negotiating it with the tutor. Obviously, the next deadline is not shifted to have a realistic scenario and a continuous pressure for the deadlines.

Based on the feedbacks from students, the preparation of the homework usually takes around 90-150 hours for each member of the team. Approximately, 40 percent of this time is spent on modeling, 45 percent on implementation related issues, and 15 percent on testing. The large distribution depends on two main factors: (i) the enthusiasm of the group, and (ii) pre-course background knowledge of the team in J2EE technologies. Within the given time-frame, an enthusiastic group prepares such a J2EE application, which is frequently used later as a reference work in job interviews.

Teaching staff. According to the regulations of the university, one PhD student is assigned as an official tutor for the course to help the lecturer. Given a course with 80 students, this is highly inadequate, since our experience shows that one tutor should be assigned to each 10 student in an ideal case.
This is resolved by recruiting former students of the course as tutors. In most of the cases, these former students are (i) either working already in the industry in the area, and they help on a voluntary basis (ii) or they are writing their Master’s Thesis on a related topic. This way, they can also share their own experience when guiding the teams.

3 Course Overview

As discussed in Sec. 1, the main goal of the course is to find a synergetic balance between teaching concepts and best practices of UML modeling, the J2EE technology and advanced COTS tools. While the core structure of the course follows a typical software development process, the following final agenda of the course is a result of several years of evolution. This agenda is overview in Fig. 1 in a schematic way in the form of a UML activity diagram by depicting the main outputs of each phase of the project as object flows (possible iterations are not illustrated).
1. Basics of the J2EE technology (Sec. 3.1)
2. Requirements analysis (Sec. 3.2)
3. Platform-independent modeling (Sec. 3.3)
4. Platform-specific models + implementation of business logic (Sec. 3.4)
5. Model-based testing (Sec. 3.5)
6. Advanced modeling concepts + Implementation of user interfaces (Sec. 3.6)

Obviously, the outputs produced by subsequent phases are typically used in the subsequent phase in a rather regular way, thus such inputs are not depicted explicitly to improve clarity. In Fig. 1, we only highlighted how analysis-level UML diagrams are used later in the testing phase.

In earlier years, the order of model-based testing and user interface implementations was swapped multiple times, furthermore, the course started previously in the traditional way with requirements analysis. This year, we experimented with an early user interface prototype, but this experience is not enough to draw major conclusions from it.

3.1 Part I: Basics of J2EE technology

Unfortunately, different graduate students have very different background knowledge on J2EE technology since most of the courses on that technology can be taken only on a voluntary basis. Therefore, the course provides a complete introduction for J2EE in the very beginning, prior to UML related issues.

In the previous years, J2EE was introduced “on request”, i.e. in the middle of the course after constructing platform-independent UML models. However, student feedback showed that those who are unfamiliar with the J2EE technology had problems with understanding and modeling architectural considerations (by e.g. analysis classes).

*Enterprise Java Beans (EJB).* Enterprise Java Beans (EJB) are one of the most fundamental parts — the main business data layer and business functionality — of the J2EE platform, which defines a layered architecture for scalable, distributed application development including several Java standards and APIs.

While various types of beans are available, the two most important, and most commonly used ones are discussed in details, namely, entity beans and stateless session beans. *Entity beans* are application-level, persistent business data objects which are kept synchronized with an underlying relational database. *Session beans* implement the business logic of the application. After deployment, EJB instances are executed by an application server, which is responsible for efficiently providing many high-level services (e.g. transactions, security).

Since these lectures primarily aim at providing an introduction to the EJB technology (mainly following [13]), the detailed discussion of different high-level services are postponed to later phases of the course. On the other hand, a special emphasis is put on the architecture of J2EE applications, which is indispensable for the discussion of analysis classes.
Java technologies for user interfaces. The core technology for creating user interfaces for J2EE applications is called Java Server Pages (JSP). JSP provides an XML based language for designing dynamic web pages as graphical user interfaces. When a form is submitted, the corresponding business functionalities provided by EJB session beans are called by servlets. Java Server Faces improve the JSP technology by providing more complex handlers for user interface events and validators for form data in order to provide rich web user interfaces.

Homework. The first homework assignment requires the implementation of a pair of interconnected entity beans and a session bean with one query, create and delete method as the business logic (similarly to the class diagram extract of Fig. 5). As for the user interface, a creation/modification, and a query form are required.

The first homework assignment is different from all the upcoming assignments in two ways. First, it should be carried out individually, moreover, its evaluation has a “go/no go” nature, i.e. its completion is required for every student in order to qualify for the team work part.

3.2 Part II: Requirements analysis

After an introduction to J2EE technology, the course goes through the typical phases of a traditional model-driven software engineering project, thus it starts with requirements analysis.

In this phase, students learn what techniques are frequently used for capturing requirements, with special emphasis on how to capture workflows, use cases, and how to detail use cases afterwards. On the UML level, activity diagrams, use case diagrams, and the basics of sequence diagrams are overviewed.

Furthermore, in addition to a notation guide, modeling examples are also presented (e.g. from previous student projects), including not only good modeling practices but bad model smells as well.

Best practices and guidelines as discussed in [4] for properly constructing diagrams are also discussed. The reason for this is that inexperienced students tend to draw everything on a single diagram, which often results in scattered and hard to read diagrams. Moreover, guidelines such “actors should be named according to their role and not according to their job titles” or “start a use case diagram by placing the most important actor on the top left corner” are not frequently considered by first-time modelers.

Homework. The team work project with groups of 4-5 students starts from the second homework assignment with requirements analysis. First each group selects a group leader who is responsible for the balanced distribution of work within the group.

There is an exception for those students who already successfully passed a previous course on J2EE technologies, which is not a prerequisite for the current course.
As emphasized in [8], it is difficult to provide a realistic exercise for requirements analysis in a university course as problems are frequently overspecified by the teacher. This is tried to be resolved by selecting the scope of the project where the teacher can act as the domain expert. For this reason, previous projects included the construction of a publication portal, a football championship management and a conference management system.

Initially, a high-level initial text is provided for the groups as a starting point, which is refined later by an interview session where the different groups may ask questions from the domain expert (i.e. the teacher) in order to construct the main workflows.

After constructing the workflows, the students need to specify use cases for the functional requirements of the system. They need to assign at least a priority and a responsible person for each use case as depicted in a sample use case diagram of Fig. 2. The reason for this is that the designated system is deliberately specified to be too complex, therefore, it cannot be designed and implemented as a whole within the given time constraints of the semester. Thus, it is the role of the students to prioritize use cases so that a meaningful fraction of the system will be designed, and this way, different groups will design different systems for the same topic.

As a general guideline, each student should be responsible for 4 or 5 business-level use cases. They do not need to design and implement more, but they need to carry out what they “signed up for”. Again, this is for simulating an environment which is sufficiently close to that of a real project.

The homeworks are presented in a review meeting for the tutor of the group with a project presentation. There, the tutor may require to reengineer certain parts of the model / implementation if major problems have arisen, which makes the development process at least slightly iterative.

3.3 Part III: Platform-independent modeling

By the term “platform-independent modeling”, we refer to the construction of various analysis and design models still independently from the underlying J2EE technology, which forms the third part of the course. As before, lectures discuss the UML notation and semantics, modeling style, best practices and guidelines, and also present some bad smells for models.

---

Fig. 2. A sample use case diagram in a conference management system
Based on the gathered requirements, students learn how to construct analysis classes (of the Unified Process [10]) in order to separate business data (Entity classes), business functionality (Control classes) and user interfaces (Boundary classes). An extract from a student diagram is depicted in Fig. 3.

Students’ experience shows that control classes are very useful for designing web applications where business functionality usually involves the manipulation of several entity classes, thus traditional object-oriented localization of behavior is not always applicable.

On the other hand, students frequently find the construction of Boundary classes as tiresome and somewhat artificial. Therefore, this year we experimented with constructing prototypical user interfaces already in the very early phase of the project.

The detailed description of a business functionality is detailed by UML 2.0 sequence diagrams, which also shows the architecture-level interaction between analysis classes. If an Entity class has an obvious stateful behavior (like articles in a conference management system), the corresponding statecharts are also constructed (see Fig. 4). It is worth pointing out that the trigger events in such statecharts (which belong to an Entity class) are typically business methods of the Control classes.
Additional restrictions and details of Entity classes and business methods of Control classes are specified by using OCL constraints. In fact, it is worth pointing out that the use of OCL constraints proved to be one of the most problematic tasks for students. Hopefully, advances of modeling will soon provide rich textual editors for OCL (as it is supported by Borland Together 2006).

**Homework.** Unsurprisingly, platform-independent modeling using UML is a key homework assignment, therefore, additional consultations are offered to the students by the tutors.

In addition, during the last lecture before the deadline, a voluntary group presents its models to the audience. This is reviewed by the lecturer and also by the students in the audience. Since inexperienced students frequently commit similar flaws in modeling, such a session normally reduces the number of typical design errors in the homework.

A typical source of design flaw of students is to mix up roles with persistent business entities in analysis class diagrams (e.g. to use only a general *User* role already in Fig. 3 instead of domain specific roles like *Author*). Furthermore, technology-oriented students tend to directly create class diagrams in the style of the target implementation technology. For instance, since EJB entity beans typically do not support generalization, no generalization is used between analysis classes. Another common misunderstanding of students is that while a platform-independent model is independent from the target technology, it already reflects architectural decisions. For this purpose, the pros and cons of various data validation approaches are discussed in details (e.g. data validation in Boundary classes vs. data validation in Control classes).

In this phase, tutors may reject the modeling homework assignments of their group when major modeling flaws are identified. Rejected homeworks should be corrected and resubmitted within a week, and it does not allow additional delay with upcoming homework assignments.

### 3.4 Part IV: Platform-specific modeling and EJB implementation

The next part of the project focuses on platform-specific modeling and EJB implementation as a joint homework assignment. For this purpose, EJB proves to be a nice technology as the differences between platform-independent (PIM) and platform-specific (PSM) modeling can be easily identified already on the class diagram level (for instance, compare the Fig. 3 and 5).

On the other hand, many platform-independent to EJB-specific mappings are non-trivial, therefore, we can show that there are typically various PIM-to-PSM mappings for a given PSM technology, and it is a design decision to select the most suitable one for a project. For instance, there are various ways how to get rid of inheritance in an analysis model when deriving an EJB class diagram (e.g. keep the superclass as in case of *Author* vs. *User* in Fig. 3 and 5 or keep the subclasses). As a conclusion, there is typically no single silver bullet PIM-to-PSM mapping for a platform.
The initial lectures on EJB technology (Sec. 3.1) are now extended with a detailed discussion on the object-relational mapping (i.e. how to persist entity beans in a relational database), and platform-specific design patterns such as session facade or data transfer objects. Normally, it is worth pointing out for the students that the concepts of design patterns are more general than the original object-oriented design patterns in [9]. In fact, the J2EE technology has its own collection of design patterns [3].

Further advanced EJB topics (like security, transactions, etc.) are also presented in this stage. Moreover, main features of the new EJB 3.0 standard [14] are discussed as well. Our experience shows that the presentation of new standards is of utmost importance as students learn that being always up-to-date with standards is a crucial knowledge.

Homework. As their next homework assignment, students need to construct main platform-specific diagrams (mainly class diagrams, see Fig. 5 and sequence diagrams), and they need to implement the business logic of their application by using the EJB technology.

While they already know the basics of the EJB platform from Sec. 3.1 and they have a precise platform-independent model (Sec. 3.3), they are now capable of concentrating on the adequateness (elegance) of their implementation by following EJB best practices.

3.5 Part V: Model-based testing

While initial unit tests are carried out for the EJB implementation of the business logic in the previous phase in an ad hoc way, the next phase of the project discusses systematic model-based approaches for requirements-driven testing.

Various object-oriented testing strategies with different scope (based on [5]) are discussed with minor adaptation.

- Since a control flow graph can be constructed from the source code or from scenarios captured by sequence diagrams (see Fig. 6), a collection of entry/exit paths provides a set of test cases with high branch coverage.
- Flattened statecharts can be extended with a response matrix to test the result of potentially erroneous transitions, thus making the statechart specification to be deterministic. Then the N+ test strategy incorporates all

\[^4\] A fully specified and deterministic statechart is frequently a minimum design requirement in many embedded or safety-critical system [11]
significant execution paths of the flattened state model by executing loops only once. Note that in many cases, the statecharts of business objects are typically much more simple than that of embedded systems.

- **Domain analysis** is a frequent test strategy for deciding whether a business method respects all its invariants. These invariants are usually defined by OCL constraints which were identified during modeling.

**Homework.** The corresponding homework assignment is to test certain business methods by using the test strategies presented during the lectures (while user interface tests are out of scope for the course). Since these test strategies typically yield a large number of test cases, the students are not required to test all their business methods. However, the selected methods should be tested in depth in order to attain a 100% statement coverage and a branch coverage if possible.

### 3.6 Part VI: Advanced modeling concepts + Implementation of user interfaces

In the very final part of the course, lectures are decoupled from the actual student projects. Advanced modeling concepts are discussed in lectures while the students are implementing the graphical user interfaces to finalize the implementation of their system (by using the JSP or Java Server Faces technology discussed in Sec. 3.1).

One lecture addresses refactoring and reverse engineering. These two core modeling concepts are already applied implicitly by the students. For instance, the Eclipse based modeling tool used throughout the course supports various refactoring and reverse engineering operations (e.g. from EJB to UML class diagrams), thus students already know what to expect from these concepts.

Another topic which is discussed in detail is the creation and use of UML Profiles. These topics are selected as stereotypes are very frequently used in everyday modeling practice to denote “home standards” within a corporation. While the theoretical foundations of stereotyping are too complex for the students, selected standard UML Profiles which focus on a certain domain (like the UML Profile for Performance, Schedulability and Time [12]) turn out to be less abstract.

Finally, the concepts of metamodeling, domain-specific languages and model transformations are also mentioned, since the code generation facilities of various EJB tools highly rely on these concepts. An unofficial objective of these lessons is to advertise related courses of our research group for the fall semester on (1) Eclipse-based development and (2) model transformation-based systems engineering. On the one hand, since an Eclipse-based user interface is being used by the students during the UML course, Course (1) is a straightforward continuation to learn how to write Eclipse plugins. On the other hand, metamodeling and model transformation techniques serve as the core technology of UML-based model-driven development, thus Course (2) is also a conceptual follow-up of the current course.
Homework. The final part of the homework assignment aims at the implementation of the graphical user interfaces using the JSP and/or Faces technologies.

For the final project presentation, both the teacher and all tutors are present (at least in an ideal case), thus the final impression on the homework achievements of different groups are guaranteed to be synchronized. This also enhances a more justified evaluation for each group.

By default, each member of the group receives the same score for the homework, unless the tutors or the group members themselves indicate that certain members were more active. Since only positive discrimination is applied, the evaluation of the tutors and the opinion of the group members are correlating in most of the cases. In previous years, we experimented with a more individual grading system, but our experience was that students could be grouped into these two categories, which is not completely fair but no complaints have arrived from students about that. Since the same number of functionalities with similar complexity is implemented by a group each year, the homework grading compares the achievements of a group within the semester as well as previous editions of the course.

Obviously, there are always students who disappear during the semester and thus, they automatically fail their homework assignments, and thus the course. In case of losing a team member, their team mates only need to compensate for the missing functionality that their own tasks highly depend on (e.g. a login functionality). Despite the very intensive workload during the semester, only 5-6 students fall out this way each year, which is a very good number (at least at our university). As a consequence, real group work can be carried out.

4 Further Educational Aspects

4.1 The Development Process

The software development process followed by the students during the course is a combination of the Rational Unified Process (RUP) and eXtreme Programming (XP), although iterativeness (which is a main commonality of the two approaches) is quite restricted due to the constraints of an academic course.

The development process highly facilitates team work, which is common to both approaches. Regular project meetings are held within a group for discussions and main design decisions. We recommend for the groups to agree on a fixed agenda (and especially, time constraints) for a meeting. Knowing that one has e.g. 15 minutes to discuss an issue keeps a discussion much more focused.

Group work was also facilitated by the use of the CVS version management system. Fortunately, the modeling tool (see Sec. 4.2) offered a model-level (and not only file-level) interface for version control by calculating model differences and conflicts, etc. before committing changes to the CVS server. However, despite this support, the resolution of all conflicting changes during a commit procedure still required approximately 15 minutes, thus additional “locking” emails were used by several groups to minimize their effort for uploading their models.
On the one hand, the course teaches modeling and model-based testing, which is typical for RUP in a J2EE context [7]. On the other hand, pair (even group) programming is quite frequently used either during the implementation phases or in the initial phase of the course when experimenting with new technologies.

Review meetings held for closing a phase of the project are again a common project management approach in the industry. Finally, classroom training in EJB technology is provided by the tutors in (voluntary) labs.

4.2 Tools

During the course, students use Rational Software Architect (RSA) – a commercial tool from IBM – for preparing their projects. This tool supports modeling in UML 2.0 and J2EE development by integrating Websphere Application Developer (WSAD) and the Websphere Application Server. Alternatively, the students could choose the JBoss application server for J2EE deployment. During the testing phase, JUnit test cases are used which are also deployed on the application server (thus they are executed as server-side and not simple client-side applications). Since the RSA tool is built on the Eclipse platform, the students will also gain experience with this modern open development platform.

Of course, there are several pros and contras for using a COTS tool. As pointed out in [8], COTS tools may have (1) licencing problems, (2) complex interfaces which distract the attention of students from modeling, and (3) some own dialect of UML.

In order to address Problem (1), we used free educational licenses provided by the IBM Academic Initiative [1]. While reinstalling the application was required here and there due to software crashes or specialties of the underlying OS (e.g. Java 1.5 had to be uninstalled prior to installing RSA under Windows, and reinstalled afterwards), RSA was executable both under Windows and Linux. Thus the different configurations of the students’ machines were not really a problem.

We organized facultative labs to overcome problems with tool usage (Problem (2)). Our experience shows that one four-hour lab for EJBs, one for Java Faces /JSPs, and one for the UML modeling tool and testing are quite sufficient for the students to start working.

Obviously, Problem (3) still remained, since RSA does not fully support the UML 2.0 standard, but this disadvantage is balanced by the fact that students have real experiments with using a commercial tool. Naturally, these tool experiments are not always positive. For instance, there is a definite learning curve for RSA in order to avoid “risky” operations (such as removing a deployed project from the application server without undeployment). Moreover, RSA contains various bugs which corrupt the XML-based deployment descriptors (e.g. when a project is not undeployed from the application server before closing the project), which need to be fixed manually, thus hindering the smooth development process. However, the students also realize that this is the typical case in the industry: they will need to work in a production line with modern tools and technologies which are too complex to be free of bugs.
Fig. 6. Sample exercise from exams: Design a minimal test set with 100 % branch coverage for the scenario defined by the above sequence diagram.

Our experience shows that students at a graduate-level rather see this as an advantage, which creates a realistic look-and-feel for the entire course. Frequently, they start using these tools after their graduation (i.e. within a year) in real projects.

On the other hand, it must be pointed out that RSA does not yet provide an in-depth support for all phases of model-driven software engineering. Most of the identified weaknesses were related, especially, to certain model transformation tasks, which results in a limited automation for reusing analysis models either for testing purposes or for deriving EJB specific models.

4.3 Final student assessment

Since most advantages of investing into modeling are observable only from second or third project, during the exam, students are solving small modeling and EJB programming exercises which are similar to those they did during their homework assignment. The exercises try to combine different skills acquired during the course. For instance, students need to design a minimal test set which provides 100 % branch coverage for the scenario defined by the sequence diagram of Fig. 6.

In the final grading, both the homework assignment and result of the exam counts with 50 percent.
4.4 Course feedback

Naturally, a university course should continuously improve in response to students’ feedback. Students’ feedback is collected from three main sources.

1. First, there is a university-wide questionnaire which needs to be filled by the students at the end of the course. However, the evaluation criteria on this questionnaire is too general to provide in-depth feedback for the course. The results of this questionnaire are published at the university web site, but sadly, the results have no impact at all.

2. As usual, the quality of the homework assignments and the exams are also frequently show not only the knowledge and efforts of an individual student but also the quality of a course. Since there is already four years of experience, the lecturer and the tutors now have some experience to identify the most problematic parts of the course for the students.

3. For the most in-depth evaluation is obtained by organizing a student forum right after the completion of the final homework assignment when the joys and nuisances of their J2EE projects are still fresh. This is an open discussion forum when the teams can tell everything what they did not like in the course (ranging from inappropriate tool support to high workload). They frequently propose some solutions to these problems already at the forum.

The positive industrial feedback of the course is reported by several students who successfully applied for a job typically in J2EE development with knowledge acquired during the course. Motivated groups also frequently participate in a 24 hour Hungarian programming contest [2] organized by IBM where the students need to provide solutions for a complex development problem using various IBM products including Rational Software Architect.

5 Conclusions

This paper reported on a graduate-level course on UML based modeling and development for J2EE applications where the main emphasis was to find a synergistic balance for educating UML modeling concepts, J2EE technologies and advanced COTS modeling tools. As a consequence, students acquire more skills in using modern tools, which is probably a main difference compared to other UML courses.

Taking an industrial aspect for evaluation, we are proud that (i) several students taking the course managed to get a job using the acquired knowledge (ii) excellent homeworks were used as a reference work for job interviews, and (iii) numerous former students are still actively participating as tutors to share industrial experience with students.

Last year, two new courses have been announced: one on Eclipse-based development, and one model driven systems engineering. In the future, we plan to provide experience report also on these courses, however, at least one or two years of experience is definitely required prior to that.
References

9. E. Gamma, R. Helm, and R. J. ans John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
Transitioning model-driven development from academia to real life

Clarindo Pádua, Bruno Pimentel, Wilson P. Paula Filho and Fabiana T. Machado

Synergia Systems and Software Engineering Laboratory
Computer Science Dept. - Federal University of Minas Gerais
Av. Antônio Carlos, 6627 - Belo Horizonte - MG – Brazil
{clarindo, brunoisp}@dcc.ufmg.br; wppf@ieee.org; fabi@dcc.ufmg.br

Abstract. This paper presents a case study on the transition of a model-driven software development process, from an educational environment, to a laboratory that develops real life information systems. Although this laboratory, named Synergia, is internally organized as a traditional software development company, it also retains important academic characteristics. Synergia uses the Praxis software development process, which was tailored from its educational version, in order to consider the technology, schedule and cost constraints of real life projects, with specific emphasis on the Usability discipline. Such tailoring has also provided important feedback to the original, educational process.

Keywords: Model-driven development, university-industry cooperation, process tailoring, usability.

1 Introduction

This paper presents a case study on the transition of model-driven software development process, from an educational environment to a laboratory that develops real-life information systems. The target environment is Synergia, a laboratory for software and system engineering, housed in the Computer Science Department, in the Federal University of Minas Gerais, Brazil. It provides software, systems, training and consulting solutions for Brazilian public agencies and private companies.

Synergia also provides an interesting model of cooperation between university and industry – although it is internally organized as a traditional software development company, the laboratory also retains important academic characteristics. In such setting, the transition of significant academic knowledge – model-driven software development, in the case described here – to real life takes place in a seamless manner. In this cooperation model, the university provides high-quality IT services, while it gains professional experience and financial resources.

Synergia uses a tailored version of the Praxis model-driven software development process in its software and systems engineering projects. Although the Praxis process has been designed and applied primarily for education and training in software
engineering, it provides tailoring guidelines, which must be interpreted according to the current situation of the development organization.

The transition effort provides benefits to both educational and industrial versions of the Praxis process. Strict, well-established development and management models for software engineering defined by the educational process are seamlessly inherited by the tailored version. Also, the experience obtained with the application of the industrial process provides important feedback for the continuous improvement of the educational process version.

This paper builds on previously published material on the Synergia laboratory model [1] and the Praxis software development process ([2], [3], [4], [5]). We summarize the relevant features of Synergia and Praxis in the second and third sections, respectively. The central contribution of this paper is in section four, which discusses how Praxis was transitioned to use in real-life projects at Synergia. The last section summarizes conclusions and lessons learned.

2 A university-based software development laboratory

Synergia roots come from a 1993 cooperation contract, between its parent computer science department and one of the largest Brazilian telecommunications companies, for developing engineering applications. In this section, some relevant aspects of its current profile are discussed.

Synergia has currently 55 people in its staff – 15 undergraduate students, 13 graduate students and 27 non-student graduates (10 with M.Sc. degree), most with a computer science background. The laboratory is organized in multi-functional project teams, with specific profiles for requirements and analysis, design and implementation, testing, quality management, usability, process engineering and project management.

Professors of the Computer Science Department staff the senior management board, which defines organization strategies and supervise the activities of intermediate level managers. These manage day-to-day activities and submit strategic issues to the board. This organization relieves the professors from operational tasks, which might hamper their academic activities.

Synergia takes part in education, research and development activities, seeking to bridge academia and the local information technology community. This cooperation model provides interesting gains to all parties involved: the university earns important financial resources to support and develop its infrastructure, while providing high-quality practical software engineering education for undergraduate and graduate students; the government clients acquire high-quality IT solutions through a transparent, mature development process, and become acquainted with solid, transparent software acquisition procedures; and the Brazilian software community gains highly capable and educated professionals, trained in software engineering best practices and technologies, and familiar with quality assurance and process improvement cultures.

The following sections summarize important features of Synergia’s operation. For further detail, see [1].
2.1 Portfolio

Synergia provides software and systems engineering, training and consulting services, mainly to Brazilian public agencies. Recent significant cooperation projects had as customers a state financial auditing agency, a state planning and management agency, and a municipal legislative body. Typical projects deliver information processing and retrieval systems, ranging from 800 to 3000 function points in size, with usual project duration between 7 and 14 months. System development projects form the bulk of the laboratory workload, but there have been smaller contracts for business modeling, system specification, training, consulting, and research.

2.2 Human resources development

Synergia is thoroughly integrated in the academic environment, offering an environment for practical software engineering education, where undergraduate and graduate students work with up-to-date software processes and technologies, driven by current market demands.

Synergia receives no government subsidies and has to be self-sustained, operating in the market like a real company. This competitive environment requires achieving competitive productivity, driving the development of staff proficiencies, and of technology and process solutions.

Undergraduate students that work in Synergia receive scholarships, and are allowed to tailor their work schedules to meet course commitments. Undergraduate trainee work is planned and supervised, to balance its contribution to their education as students with the laboratory needs.

Graduate students receive a scholarship supplement, very attractive when compared to those granted by public research agencies. The graduate students can also acquire non-research competencies, such as customer relations or project management; they are encouraged to align their academic research interests with Synergia’s technology and project demands. Some undergraduate and graduate students become fully employed in Synergia, after they complete their studies.

Considering this environment and the pool of capable students provided by the parent university, Synergia has no difficulties in staffing capable and experienced teams. Even when a project team includes inexperienced people, these often have strong learning potential. These teams often contribute to the improvement of the processes and technologies applied on the software projects.

A high-potential staff may also present drawbacks, such as a high turnover rate. Since Synergia’s staff proficiency is well known in the local technology market, some private companies – and sometimes, even some Synergia’s clients – often try to recruit Synergia’s professionals, causing losses of key people during critical phases of a project. Another drawback is that, since this kind of person usually welcomes technical challenges, they tend to dislike routine project tasks and medium-complexity projects. A Synergia current weakness is the lack of mature human resources policies; hence the staff does not always perceive a long-term career prospect, and may be seduced by more traditional corporate or public service environments.
2.3 Strategy and internal organization

*Synergia* has a relatively good internal structure, but still lacks some strategic business and academic definitions. It may be difficult to align those strategies in a public university, where there is some objection to seeking funding from project customers or other non-traditional sources. Acceptable strategies must consider both business and academic goals; moreover, they are constrained by university regulations and policies. Often, possible private clients or partners do not see *Synergia* as a real-life professional-level organization, but rather as an academic, research-minded group, supposedly having “higher costs” as a consequence of its processes. Ironically, many former *Synergia*’s collaborators were hired by the same organizations to set up quality and process cultures.

*Synergia*’s top clients, who usually understand that *Synergia* provides a much higher quality level, and minimal schedule and cost overruns, compared to local market practices, however, do not share that view. For instance, current products have shown around 0.018 field defects per function point. Also these clients usually appreciate the fact that *Synergia*’s projects usually cover the whole development spectrum, from requirements to deployment, and even to maintenance and evolution, if desired; lower cost suppliers generally do not offer all of those services.

Although the academic objectives of education and research must be accomplished, *Synergia*’s self-imposed rules of the game require financial self-sufficiency. Its long-term survival depends on constantly obtaining new development contracts, but also on keeping a core of senior engineers and managers, committed to long-term goals. Therefore, the marketing strategy and the human resources policies must be aligned.

3 The Praxis software process

Praxis is a software development process, designed to support education and training in software engineering. It is currently in version 2.1[6]. It has been applied in undergraduate and graduate courses and industrial training programs; some results of those applications are discussed in [2] and [3].

3.1 Use case development states

Praxis models the development of each product use case as a sequence of development states. The name of each state evokes how far the use case has advanced towards complete implementation and acceptance. Each state is associated with a set of resulting work products, which become inputs to the following steps. Table 1 summarizes the Praxis standard set of use case development states. A detailed discussion of those states is in [4].

Reaching a new state requires passing a quality assurance gate, that is, an objective verification procedure, performed either by independent reviewers or by automated scripts. Depending on the state, they may include several kinds of inspections, tests,
and audits, as detailed in [5]. This provides for more objective state transitions, which are not solely dependent on the developer’s personal appraisal.

Table 1. Praxis use case development states, with respective outputs and quality gates.

<table>
<thead>
<tr>
<th>State</th>
<th>Output</th>
<th>Quality gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified</td>
<td>Use-case diagrams (requirements view).</td>
<td>Management review</td>
</tr>
<tr>
<td>Detailed</td>
<td>Use-case descriptions; low-fi prototype; interface and non-functional requirements (requirements view).</td>
<td>Requirements inspection.</td>
</tr>
<tr>
<td>Analyzed</td>
<td>Use-case realizations; detailed class diagrams (analysis view).</td>
<td>Analysis inspection.</td>
</tr>
<tr>
<td>Designed</td>
<td>Design use-case descriptions; concrete user interfaces diagrams and specifications; hi-fi prototype (external design view).</td>
<td>External design inspection.</td>
</tr>
<tr>
<td>Specified</td>
<td>Test specifications (test design view).</td>
<td>Test design inspection.</td>
</tr>
<tr>
<td>Realized</td>
<td>Design use case realizations (internal design view); source code and unit test scripts for lower architecture layers.</td>
<td>Unit tests; implementation inspection.</td>
</tr>
<tr>
<td>Implemented</td>
<td>Source code and test scripts for upper architecture layers; executable code.</td>
<td>Integration tests; implementation and internal design inspection.</td>
</tr>
<tr>
<td>Verified</td>
<td>Verified executable code; user documentation.</td>
<td>System tests.</td>
</tr>
<tr>
<td>Validated</td>
<td>Validated executable.</td>
<td>User appraisal.</td>
</tr>
<tr>
<td>Complete</td>
<td>Complete code and artifact baseline.</td>
<td>Quality audit.</td>
</tr>
</tbody>
</table>

3.2 Model transformations

For the first seven states, every development state transition centers on a model transformation; the last three states are concerned with verifying and validating those models and associated code. Every model uses UML, augmented with textual and tabular attachments. An overview of these transformations is shown in Figure 1.

In the first three use case development states, developers work on the analysis model. This model describes the problem to be solved, in two views. The requirements view shows the product functional requirements, expressed as analysis use cases, that is, essential use cases, whose behavior is specified by low-formality text. The analysis view represents problem domain concepts, user and system interfaces, and algorithms and control flows as analysis classes. The most important items in the analysis view are the analysis use case realizations: conceptual, technology-free collaborations of the analysis classes, which must satisfy the behavior specified by the analysis use cases.

In the following four states, the central artifact is the solution-oriented design model. Its external design view contains the design use cases, that is, detailed descriptions of user-product interactions, which refine the analysis use cases. Design use cases represent the contract between users and product, mediated by concrete user interfaces and interaction mechanisms, designed to conform to the requirements. From this contract, test designers develop test specifications, contained
in the test design view. Finally, in the internal design view, the design use cases are realized by collaborations of internal design classes. These reorganize and detail the analysis classes, to meet non-functional requirements, while using the chosen implementation technologies.

Code is generated from the design model. It is divided in product code, which actually implements product functions, and test code, which ensures their conformity with requirements. Both the design model and the code are built upon a process framework, which provides for reusing product code, but also user interaction patterns, internal realization patterns, test specifications and test scripts.

![Praxis model transformations](image)

**Figure 1.** Praxis model transformations.

### 4 The Praxis-Synergia Software Process

Praxis-Synergia is the tailored industrial version of Praxis, adopted by Synergia, which applied its practical experience to tailoring and enhancing its own version of Praxis – the **Praxis-Synergia software process**. This process is almost fully aligned with standard Praxis, but there some important differences in the techniques, procedures and patterns used by Synergia project teams.
Both business and technology aspects of Synergia’s software development projects have driven the tailoring effort. Firstly, the demand for higher productivity led to use more process automation tools, which in turn required more stable artifacts and procedures. Secondly, recent development projects of web-based software products, for instance, have demanded fine-tuning of design, implementation, testing, quality management, and deployment procedures. The enhancements were based on previous experience, literature and also on available support tools. These procedures are currently better fit to current project technology constraints, but there are yet several improvements to be done.

It is important to point out that, although Synergia is located in an academic environment, there are still many challenges when trying to adapt an educational software process to the needs of a real-life organization. Staff resistance on changing processes, patterns and procedures can still be a concern, but people are usually more receptive to process improvement. In fact, the engineers themselves propose many enhancements.

The remainder of this section discusses the most relevant aspects of Synergia’s tailoring of the Praxis process.

4.1 Technical aspects

Analysis model abstraction level. Ideally, the analysis model should be completely technology-free, containing conceptual problem descriptions only. In practice, its abstraction level may depend on factors such as product complexity, team experience, and process and technology maturity.

Better results are achieved when experienced designers participate actively in requirements and analysis. Complex interface behaviors and non-functional requirements are not mirrored in functional size estimation, so that appropriate forecasts of development complexity must consider solution features. Experienced developers may contribute to better understanding the technological implications of such complex requirements, providing important inputs for effort and cost estimation, and for choice of architecture and technologies. As a by-product, those developers become proficient in the requirements of each product.

Analysis model detail level. To accomplish the model transformations as described previously, it is necessary to define an adequate level of detail for the analysis use cases descriptions. It is important to provide, at that moment, detailed information to allow thorough validation by the customers, but also a comprehensive evaluation of technology and architectural feasibility by the designers. The sooner this analysis takes place, the better for minimizing technology-related risks.

Our experience has shown that some analysts tend to overload the use case descriptions with business logic, or even interface requirements. This is not much of a problem for small educational projects, but, for larger business applications, such bloated use cases are difficult to understand, validate and maintain. Thus, Praxis-Synergia came to include a set of guidelines that help the analysts to balance
functional requirement descriptions with object-oriented representations of interface requirements and business rules.

**Design model formalism level.** Synergia’s experience confirms the importance of matching the level of model formalism to the needs of the model consumers, that is, downstream developers. Low-formality, textual use case event flows ease user participation and appraisals, during requirements elicitation. More formal use case flow descriptions, used in the Praxis design model, convey more precise information to developers and testers, avoiding duplication and ambiguity. However, consuming and maintaining these diagrams is harder; increased standardization and training become necessary.

High-fidelity prototypes have shown to be far more effective than UML models, in helping developers to grasp design use case behavior and interface design, especially in the case of less experienced professionals. However, serious problems arise when these developers rely exclusively on the prototypes, neglecting the information in the models. In this case, their implementation generally turns out to be incomplete and non-compliant with the design standards. Although tests usually spot these mistakes, the resulting rework is clearly undesirable. Training and management guidance are necessary to enforce compliance of the implementers with the external design view.

**Design model quality gates.** The external design view of the design model has also been in the focus of attention, by process engineers and quality analysts involved in Praxis-Synergia. The most relevant issues at this stage are related to standardization, change control and alignment between external design and implementation component frameworks. Although much work has been done in this direction, there is yet some ground to be covered. One important aspect, however, is the Synergia's tailoring of the quality gates related to the external design view of the design model. Since external design is the most upstream design activity, the stability level of the external design view has a major impact over dependent tasks, such as test design, internal design and implementation.

In standard Praxis, the quality gate for external design is a simple formal inspection. In Praxis-Synergia, this has been unfolded to include the following:

- a requirements-mapping review, where the external design view is verified against the elicited system requirements;
- a testability review, where the external design view is evaluated according to the information needs of the testing team (that is, whether there is enough information to enable test design);
- a specific external design review, where each design use case is validated according to the organization’s standards and the needs of the internal designers and implementers.

This unfolding provides a better control of the model transformation procedures embedded in the Praxis-Synergia process, providing project management with a rigid, process-oriented task precedence control. This has shown to be critical for quality assurance and rework avoidance, in the ensuing activities.
Design model evolution. Concerning the design model evolution throughout the project, certain practical aspects must be observed. Firstly, all systems are subject to requirements changes, either requested by the users, or resulting from requirements defects, found by the development team. Such changes must cascade from external design view to test specifications and internal design, before reaching code. Keeping the design model up-to-date requires an experienced development team and a highly disciplined project management.

Secondly, keeping complete model-code synchronization in real-life projects is a complex task, which can be seriously aggravated by technological limitations of the modeling tools. In the Synergia practice, the internal design view usually keeps updated architectural features and lower (entity) layer classes. Classes and collaborations in the upper (control and boundary) layers are much harder to keep synchronized, and continuous synchronization is not required for them.

Design model reuse and technology restrictions. Synergia’s experience has shown that the reusable design framework provides better results when the interface model elements and the interface code components are aligned. This alignment may constrain design options, restricting interface design to elements that match the component framework; on the other hand, it yields higher standardization and productivity levels during design, coding and testing. The end of the Elaboration phase is a good point to find a compromise among standardization, productivity and usability goals, by tailoring design and code frameworks to the project, in order to accommodate product-specific features.

Standard Praxis promotes the reuse of functional components, represented by abstract design use cases, which embody common user interaction patterns. Synergia’s experience shows that, unless the developers are very experienced, this kind of reuse should be restricted to simpler, more common patterns, which have easier consumption and maintenance.

Testing. System testing is also a discipline which had to be adapted in Praxis-Synergia. The main reason for this is related to technology restrictions in the Web-based projects commonly executed in Synergia; Web technologies impose some additional constraints, compared with the Swing-based Java boundary components, supplied by the standard Praxis framework. These constraints cause higher complexity involved in automating testing of web-based user interfaces.

This is specially the case of the standard Praxis testing scheme, where integration tests are specified by the test team, using spreadsheets for test procedure and test case specifications; these are implemented by the developers as automated JUnit scripts, using so-called inspector classes, which bridge the gap between the external design view and the actual interface code components.

In Praxis-Synergia, on the other hand, test cases are specified with a support tool, allowing the set-up of test data with more complex relationships. Also, the test team specifies a subset of tests that must be used for integration testing, which is manually performed by the developers.

The Praxis-Synergia test team then doubly checks the integration tests, in order to ensure an acceptable confidence level. In addition, the test team, after carrying out a
careful cost-benefit analysis, in order to decide which tests should be automated, executes acceptance tests. This analysis considers use case complexity, priority and stability, confronting these against the cost of creating and maintaining a regression test suite.

Praxis-Synergia stresses the importance of the quality gates associated to unit testing. Our experience has shown that quite significant quality improvements come just from the application of test-driven implementation. When unit tests are inspected for test coverage, as done in Synergia, greater levels of quality and project control are achieved.

**Usability.** The Praxis-Synergia process has given a strong emphasis on the Usability discipline. This is not explicitly included the standard process version; its inclusion is probably the largest single difference between the tailored process and its standard parent. For this reason, details of the Usability discipline in Praxis-Synergia are described separately, in Section 4.3.

### 4.2 Managerial aspects

After having established relatively stable technical processes, another major improvement achieved by Praxis-Synergia focused on the project management discipline. Although standard Praxis defines project management procedures and artifacts, Synergia’s projects often present a more challenging managerial scenario, where larger teams, within tighter cost and schedule constraints, develop complex software products with larger scopes. Also important, the Synergia environment requires the coordination of various projects running at the same time and sharing resources. Recent enhancements were based on the PMBoK methodology, focusing primarily on scope, time, cost and risk management techniques. Integrated tools for planning, controlling and reviewing were made available, and project managers have applied Earned Value Analysis, when tracking project progress. All scope evolution is strongly attached to the model-based quality gates described above, to ensure objective control of both scope and quality.

Praxis-Synergia added enhanced support to the project management discipline, helping the elaboration of detailed plans, reports and checklists. The use of project management tools, integrated to product life-cycle features and measurement activities, is much more extensive than in the standard process, where rather simple spreadsheets are considered as sufficient for educational purposes.

One of the aspects yet to be improved in Praxis-Synergia involves the quality audits. In the standard process, these audits focus on the process artifacts, using them to evaluate the performed processes. A more proactive procedure should employ direct observation, measuring and assessment practices, to achieve a more realistic view of the actual performed processes.
4.3 Approach to Usability

Usability courses are taught in our Department, in the same undergraduate and graduate programs whose software engineering courses offered a test bed for the evolution of the Praxis software development process. A Usability process was developed by one of us (C. Pádua) and used in research and educational activities. This process models usability concepts using UML and object orientation. In Praxis-Synergia, this Usability process was the basis of a whole new Usability discipline, not existent in the standard Praxis version.

The design of the Usability discipline considered a set of requirements [7], derived from the objective of making it adequate for both industrial and educational usage:

- **Ease of teaching and learning**: Similar usability concepts applied to software products also apply to a software development process. The process should be adequate to its context of use, which includes its users, tasks (development activities) and work environments (software development projects and education environments).

- **Process organization suitable for teaching**: The process should be structured in modular and hierarchical units (sub-processes and artifacts) to facilitate teaching in sequential steps and at different levels of abstraction. The process organization should allow teaching using independent module units. Also, it should be adequate to the performance of software development projects by small teams of students.

- **Seamless integration to other process disciplines**: Usability is related to many other disciplines, such as Business Process Modeling, Product Engineering, Ergonomics, Design, and Human Cognition. In the development of the Usability process, however, the focus was directed to software engineering and business process modeling, which are more closely related to software development.

- **Base on widely recognized standard and up-to-date techniques**: This requirement is important for both educational and industrial purposes.

**Discipline design.** The above requirements have influenced the design of the Usability discipline in Praxis-Synergia. Many recognized works ([10], [11], [12]) approach usability in software development as a specific process, centered on user interaction issues. In the IBM-Rational Unified Process [9], usability is approached in the form of a roadmap with prescriptions on how to integrate usability related activities and results to process elements (activities, artifacts, etc) of the other disciplines. Many other works available in the literature of the area present concepts, techniques and methodologies related to usability, but these aspects are not organized as structured development process. *Synergia* preferred to approach usability as a separate discipline, within the Praxis-Synergia process, considering both educational and strategic objectives.

**Teaching considerations.** A separate discipline provides an organized framework, both for teaching and for use in an industrial setting. The Usability workflow includes activities (both atomic activities and sub-processes), input and output artifacts, and
quality gates. Figure 2 shows an UML activity diagram describing the workflow of the Usability discipline.

Figure 2. Usability workflow in Praxis-Synergia.

As it happens with the standard Praxis disciplines, the Usability discipline contains guidelines for distributing its activities along Praxis-Synergia phases and iterations, easing their application, from a managerial perspective. Phases and iterations may be mapped to course units or course project stages, guiding the distribution of tasks along those units or stages. Also, the Usability discipline adds discipline-specific
examples, templates and other instructional materials, to those provided by standard Praxis.

Usability process elements, like activities and artifacts, have many intersections with other process elements in the disciplines of software engineering and business modeling. For instance, the usability activity of user and task analysis is closely related to the activities of business processes modeling and requirements elicitation of business modeling and software engineering, respectively. These overlapping activities should be carried out in an integrated form, so as to avoid waste of time and effort, and inconsistencies in their results.

The Usability discipline was designed to expose the student or developer to widely recognized standardized and up-to-date techniques. The processes proposed by Earthy [13] were taken into consideration in the design of the Usability activities. Other recognized modeling techniques were incorporated to the discipline. For instance, the Usability activity of Context of use analysis produces a model that is composed by UML diagrams and textual descriptions. UML static diagrams are prescribed to model user roles (or actors) hierarchy [10]. UML diagrams are also used to produce a conceptual model of objects and their relationship in the problem domain [15]. Textual description, UML models (using state, activity, or interaction diagrams, or collaborations), and scenario description [8] are prescribed for task modeling [11]. The user mental models [16] can be described textually, complemented by UML models.

The Usability discipline also prescribes the Interaction design activity, making use of UML models in different levels of abstractions [10].

Strategic considerations. Synergia’s approach to usability as a separate discipline has also the strategic objective of promoting process evolution and maturity in the area of usability. This approach facilitated forming a usability team in the software development project environment, easing the training of the technical staff, and contributing to the evolution of the usability process. This evolution has been clearly observed in the staff behavior and attitudes towards usability, according to the maturity scale as proposed by Earthy [14]. Also, quantitative usability results are obtained from delivered products and customer appraisals.

In the Praxis-Synergia process, the Usability team has the important task of assisting user interface evaluation, in several moments of the development life cycle. The first task, however, is related to the development (or adjustment) of the external view framework to the specific characteristics of the product to be developed. Furthermore, in web-based projects, it is the responsibility of the Usability team to implement the static part of the user interfaces. These can be employed to allow users to validate system requirements using a high-fidelity prototype, which also provides a better stability level in the overall project life cycle. After the executable code is ready, the Usability team evaluates the final usability perspective, validating the external view design. In the final stages of the development life cycle, the Usability team is employed again, but this time to perform usability experiments involving the final product and the users. These activities allow Synergia to improve its Usability concepts and achieve higher acceptance levels of the developed products.
5 Process feedback

The Praxis standard educational process has been evolving since its official launching, in 2001, and even before that, in a predecessor process. Most of this evolution consisted of fine-tuning changes, reflecting the findings of its application in educational projects. However, there have been also some strategic changes, resulting from feedback from its application in Synergia.

For instance, the first Praxis version was document-centric, where the main artifacts were documents patterned after their corresponding IEEE standards; models were seen as work artifacts, mere stepping-stones to the documents, whose production was required. The Synergia experience showed that customers in Brazilian market almost never demanded those documents, except for the SRS (Software Requirements Specification); therefore, those documents were seldom written. When written at all, keeping the documents synchronized with the models proved to be a major hassle. Therefore, in the current educational version, the models themselves contain all the information required by the IEEE standards, so that automated scripts, if necessary, may extract the documents. This is done, for instance, for the SRS in Praxis-Synergia.

Also, the first Praxis version followed a dominantly waterfall life cycle, since this is much easier to map into a sequence of course modules than a spiral life cycle. The Synergia practice showed the advantages of a mostly spiral, use-case driven life cycle, for risk mitigation and project control. This style was adopted in the educational process, as far as the mapping to course modules remained feasible.

6 Conclusions

This paper has briefly described the application of a model-driven academic process in a professional controlled environment, in which both sides can gain experience and improvements. This is achieved by the special structure found in Synergia – a software and system engineering laboratory housed in a Brazilian public university. Such proximity to human resources development, research activities and public agencies demand provide an environment to adequately transition technology – in this case, model-driven development – from academia to real life.

The Praxis-Synergia process, a tailored version of the standard educational Praxis process, was developed from theoretical and empirical considerations, based on Synergia’s approximately 10-year experience. The current use-case driven, model-based style of the Praxis educational standard process is a direct consequence of the feedback from its tailored Synergia version.

Acknowledgments. The authors would like to thank IBM Rational for supporting this work, within the IBM Academic Initiative.
References


