Research Reports in Software Engineering and Management

Proceedings of the 3rd Educators' Symposium at MODELS 2007

Miroslaw Staron (Ed.)

Department of Applied IT

The 3rd Educators' Symposium of the 10th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems

Symposium Proceedings

Edited by:
Miroslaw Staron

Department of Applied Information Technology
IT UNIVERSITY OF GÖTEBORG
GÖTEBORG UNIVERSITY and CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2007
ISSN: 1654-4870

Research reports in Software Engineering and Management
Report number 2007:01

Series editor: Lars Pareto

www.ituniv.se/sem_research

Symposium chair

Miroslaw Staron, IT University of Göteborg, Sweden

Program committee

Magnus Antonsson, Ericsson, Sweden
Thomas Baar, EPFL, Switzerland
Robert France, Colorado State University, USA
Holger Giese, University of Paderborn, Germany
Cesar Gonzalez-Perez, Verdeweek, Spain
Rogardt Heldal, Chalmers University of Technology, Sweden
Oystein Heugen, University of Oslo, Norway
Kai Koskimies, Technical University of Tampere, Finland
Ludwik Kuzniarz, Blekinge Institute of Technology, Sweden
Lars Pareto, IT University of Göteborg, Sweden
Pascal Roques, Valtech Training, France
Michal Smialek, Warsaw University of Technology, Poland
Jean Louis Sourrouille, INSA Lyon, France
Perdita Stevens, University of Edinburgh, UK
Tarja Systa, Technical University of Tampere, Finland
Daniel Varro, Budapest University of Technology, Hungary
Frank Weil, Motorola, USA
Paula Filho Wilson de Padua, UFMG, Brazil

Table of Contents

Preface

Model-driven development approaches and technologies for software-based systems, in which development is centered round the manipulation of models, raise the level of abstraction and thus improve our abilities to develop complex systems. A number of approaches and tools have been proposed for the model-driven development (MDD) of software-based systems, for example UML, model-driven architecture (MDA), and model-integrated computing (MIC). Using models as the primary artifacts in software engineering shifts the focus of the existing software engineering methods from code to models. As the code is the secondary artifact, techniques for estimations, verification and validation techniques, etc. need to be adjusted to take models as inputs. In parallel to transitioning from code centric to model driven development, a transition can be observed from programming oriented, computer science education, to model based software engineering education. Together, these transitions pose new requirements on knowledge goals for students, namely placing more focus on the learning abstract thinking, designing, and creating modeling languages rather than algorithms.

The educators' 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 where educators, researchers, practitioners, and trainers can meet to discuss model-driven development education from three perspectives:

- modeling-related content of courses and curricula: describing what should be taught to students
- pedagogical approaches, theories, and practices: describing how the material should be taught to increase students' learning process
- use of course materials and technology in the classroom: describing how textbooks, modeling tools, and other technology can be used to increase the students' learning process

The symposium contains perspectives from industry, academic faculty, and students.

The leading topic for the symposium in 2007 is transitioning from the traditional, programming oriented, curricula/courses to modern, model based, software engineering curricula/courses. An important aspect is how modeling courses integrate with students' career paths (e.g. how useful are modeling skills for the students' careers).

The students' role is in the focus as it is the students who should benefit from the symposium in a short run. In the long run, it is the industry which has the opportunity to employ skilled professionals. In the heart of this we sterted a joint panel with the doctoral symposium where the researchers, doctoral students, teachers, and industry professionals could discuss the issues how modeling should be taught.

The best paper from the symposium will be published in the Journal of Information and Software Technology published by Elsevier as a promotion of the topics of modeling and education on a broader forum.

Miroslaw Staron
Symposium chair

Fighting the "Formal is Futile" Fallacy

Thomas Kühne
Darmstadt University of Technology
Darmstadt, Germany
kuehne@informatik.tu-darmstadt.de

Abstract

Many students have difficulties regarding formality as a tool that provides value in practice. The typical experience in their studies is that that formal techniques stop being applicable when they would be most helpful. In this talk, I argue that it is important to counteract the undesired "impracticable" image of formal techniques, and then point out how formal specification in modeling can help to reshape how students think about requirements engineering and system specifications.

1. What Do You Mean, "Formal"?

In general, a language or method is considered to be formal if it allows the application of rigorous analyses and proofs, as known from mathematics. This is in contrast to empirical approaches which—paraphrasing Edsger W. Dijkstra—can show the presence of errors, but never their absence. In other words, formality replaces probability by certainty.

Indeed, many early lessons on formal approaches feature small and clean examples, promising powerful ways to deal with the presented problems. When tackled with mainstream technologies, the same examples would lead to much less elegant descriptions whose interpretations allow much less results.

2. What Do You Mean, "Futile"?

> *In theory, there is no difference between theory and practice. But, in practice, there is.* – Jan L. A. van de Snepscheut

All too soon, unfortunately, the above alluded to beauty and utility of formality turns out to be a bait which does not come without a hook. Later lessons on formal approaches typically have to acknowledge that their practicability is rather limited, or ways of dealing with practical issues are introduced that destroy many nice properties, heavily questioning the effort involved in learning and applying formal approaches.

Sadly, students are often left with the conclusion that formality in practice is either infeasible or does not offer any greater value than more mundane alternatives which appear to be easier to grapple with.

3. What Do You Mean, "Fallacy"?

In this talk, I argue that there are a number of recent advances in formal techniques—in particular, regarding their tool support—that make it possible to provide students with a hands-on demonstration of the practical utility of formal techniques.

In particular, I point out how using Alloy [1] and the right examples, students can be made aware of how fuzzy one's thinking can be unless it is challenged by solid validation. Such experiences are destined to reshape how students think about correctly engineering requirements and thoroughly designing systems.

4. What Do You Mean, "Fighting"?

Educators have the responsibility for preventing students from falling for the dark (informal) side. They need to demonstrate unequivocally the advantages of putting in the effort of learning and applying formal techniques. As much as possible, students should be exposed to tools that present them with results which they could not have obtained with mainstream technologies. This way students will see that the use of formal techniques has a value other than passing a course. In turn, this may help students to more easily overcome the challenges involved in learning and appreciating formal approaches.

References

[1] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., Apr. 2006.

ReMODD in Education

Robert France
Colorado State University
france@cs.colostate.edu

The problems that MDD researchers tackle are multi-faceted and inherently complex. This has led to calls for a community-wide research infrastructure that not only enables acccumulation and sharing of MDD research experience and results, but also provides modeling artifacts and resources that can be used to accelerate MDD research.

The Repository for Model Driven Development (ReMoDD) will contain artifacts whose use can significantly improve MDD research productivity, improve industrial MDD productivity, and enhance the learning experience of MDD students.

Artifacts will include detailed MDD case studies, examples of models reflecting good and poor modeling practices, reference and benchmark models that can be used as the basis for comparing and evaluating MDD techniques, patterns reflecting reusable modeling experience, transformations that automate significant software development tasks, descriptions of modeling practices and experience, and modeling exercises and problems that can be used to develop classroom assignments and projects. In this talk I will give an overview of ReMoDD goal and present the current status of the project.

Teaching Domain Specific Modeling

Lars Pareto
IT University of Göteborg
lars.pareto@ituniv.se

This talk describes experiences from teaching domain specific modeling to third year, undergraduate, software engineering students, by the use of problem based learning. Primary learning outcomes were an understanding of the differences between model driven development and model driven architecture, the ability to define graphical domain specific languages, and the ability to define translations from graphical languages to embedded target platforms. The students (which had recently taken courses in embedded systems programming with C, and model driven development with Rational Rose Realtime) were given two existing domain specific languages (Labview / Robolab and Microsoft VPS) and an existing robotics platform (Parallax Stamp), and were asked to implement one of these languages for this platform using the Microsoft DSL toolkit. Two project groups (of 5-6 students each) took on and completed this task, with graphically programmable robots as results. A third group realized the knowledge goals by a graphical agent-oriented programming language, with an accompanying translation onto an agent platform (from a past course).

The course was a five week intensive course; no other courses running in parallel. The organization consisted of the following scheduled activities: a 1h introduction to the problem, a 1h tutorial on the Microsoft DSL toolkit, a weekly supervision, and a weekly workshop with group presentations and demos. The course was examined on the basis of a written group report and by oral examination; grades were individual. The organization assumed that students were familiar with UML, MDD, C and Java, and already accustomed to problem based learning. Notably, however, there were no knowledge prerequisites in traditional programming language related subjects such as compiler design, programming languages, and semantics.

All groups used iterative software development, with iteration length set to one week (except for the first which was two week). Three of the groups presented working language prototypes after the first iteration, and demonstrable translations after the second or third iterations. All groups eventually produced demonstrable language implementations, and all participating students passed the course. The two participating teachers, neither of which had prior experience with teaching domain specific modeling, were positively surprised about the ease of with which domain specific modeling and the underlying technologies were learnt, and about how far the project groups reached within the given time. In particular, three weeks for a demonstrable language implementations was less time than expected.

Our conclusions are that, given that basic knowledge in programming, UML, and model driven development is in place, learning how to design and implement a domain specific language with the DSL toolkit is relatively simple, and well within reach of undergraduate students in software engineering.

A Phased Highly-Interactive Approach to Teaching UML-based Software Development

Egidio Astesiano, Maura Cerioli and Gianna Reggio
DISI, Università di Genova, Italy
astes|cerioli|reggio@disi.unige.it

Filippo Ricca
Unità CINI at DISI*,
16146 Genova, Italy
filippo.ricca@disi.unige.it

Abstract

In a decade of Software Engineering teaching at undergraduate level, we have always attributed great importance to software development course projects and since many years we adopt UML-based development methods. In the attempt at allowing the students to experience the main software development activities in a way as realistic as possible, we have experimented different organizational choices. The paper presents the current organization, based on a rather sophisticated phased development process, with a very high interaction between teachers and students. Our approach is illustrated by the last academic year project and its detailed assessment by means of a questionnaire.

Keywords: Undergraduate Software Engineering course, multi-phase course project, MDA, UML-based development method, Project community forum, Questionnaire.

1 Introduction

The undergraduate course in Software Engineering at the Faculty of Sciences of the University of Genoa, that has just passed its first decade, has faced since the beginning the well-known problem of balancing theory and practice. Since the second year UML has been taught and soon has assumed a central role, very much in the sense, advocated in [7], of an essential conceptual and factual tool for software development. However we have not found easy to achieve together the two goals that we have always considered of paramount importance: to allow the students to experience the main activities of software development within a rigorous framework and to make that experience as near to real life development as possible, a problem also emphasized and discussed in [19], that proposes an interesting simulated project environment. Among the problems not easy to overcome, we single out the effort constraint, both on the students and the teachers side, and the nature and size of the project. But there are other, more specific, technical and organizational difficulties, as we will discuss in the paper.

In the years we have experimented different organizational choices, from restricting the project to deal with a particular step in the development process - say a part of the design or of the implementation - to requiring the students to complete some steps within an overall development mainly provided and illustrated by the teachers. After some unsatisfactory attempts, we have found an organization of the project activities that, together with being more satisfactory, presents some distinctive features that may be of interest, we think, to the Software Engineering educational community.

In essence the relevant features are the following. A UML-based development method is followed throughout the project, which is split into distinct phases corresponding to development steps. But, most distinctively, for each phase the students are asked to perform the relative step on a small subset, the results are evaluated, and the input to the next phase is a common complete solution of the previous one provided by the teachers. Understandably, that process requires a high degree of interaction between students and teachers and the interaction is aptly supported via a Project Community Forum. The proposed organization is still at an experimental stage and thus, to improve the course, we have found useful to make a rather detailed assessment, on the basis of a questionnaire related to the last project and to the well-structured UML-based method proposed.

In the following section, after a brief outline of the context and evolution of our Software Engineer-

*Laboratorio Iniziativa Software FINMECCANICA/ELSAG spa - CINI

ing course, the current organization is motivated and described in detail. In the third section the subject and some relevant data of the last project are given, both to make the presentation concrete and the questionnaire understandable. Finally, the fourth section is devoted to the mentioned assessment via a questionnaire.

2 Teaching Theory and Practice of Software Engineering

In this section, we discuss the experience we gained teaching a Software Engineering course (3rd year B.Sc.) at the University of Genova (Italy).

2.1 Overview of the course

This undergraduate course is planned both to give a general view on Software Engineering and to provide an in-depth knowledge of UML. The course consists of three parts: an overview on general topics in Software Engineering (in the following briefly SE), a detailed introduction to UML, and a project on the development of a software system based on the general concepts on SE and using UML. The prerequisites are programming (Java and C), database knowledge and GUIs implementation using Java. Therefore, the students are assumed to have a good knowledge of OO concepts, in particular of the Java language.

We mainly base our lectures about SE on [14, 18], whereas for UML we use directly the official specification [12] and we suggest (but not require) our students to read [9]. Both for the SE and UML parts, the only material distributed is a copy of the slides used for the lectures, providing a summary and a guide to the individual study of the suggested text-books.

In order to pass the exam, the students have to independently pass two written examinations, in any order, one concerning SE and the other on UML. Moreover they have to develop a multi-phase course project in teams (team size $= 3 \pm 1$). The final grade is computed from the results of the individual parts, accordingly to the formula 45% SE + 20% UML + 35% project.

The expected working load for the average student is of 225 hours (see Table 1). Similarly to [6, 8] we consider the project to be a prominent part, and indeed more than 1/3 of the student time is devoted to it. We think the project to be extremely instructive because for the first time the students experience the development of a realistic system, starting from the requirements, and putting in practice the principles of Software Engineering. Moreover, thanks to the project, students are expected to learn to work in teams and to prepare a project plan including estimates of size and effort, a schedule, resource allocation, time management, configuration control and project risks. Several graduate students, already working in ICT companies, reported that they were able to build on this experience when they had to face real industrial software projects and realized retrospectively how useful it had been.

	hours
Lectures (6 hours per week × 12 weeks)	72
Study day by day	50
Project	82
Final preparation for the exam	17
Written examinations (2 hours each)	4
Total	**225**

Table 1. Working load.

The SE part of the course consists of an introduction to Software Engineering fundamentals, covering both traditional and object-oriented techniques. Topics include requirements engineering, design engineering, software architectures, testing, maintenance, process models (plan-driven and agile), reuse and design patterns.

We teach UML 2.0 starting from its basics and introducing the most important diagrams (Use Case diagrams, Class diagrams, Sequence diagrams, State machine diagrams, Activity diagrams and Composite Structure diagrams). Moreover, considerable importance is given to OCL [11]. The students gain familiarity with each topic by small toy examples at first. Then, during the course, UML is applied to various modeling problems across a variety of application domains.

2.2 Evolution of the project modalities

In our opinion, it is of paramount importance that a Software Engineering course allows the students to experience the various activities performed during the development of a software system, and that the setting for these experiences be as realistic as possible. That requires to give careful consideration to the choice of the subject for the project, balancing the realism of the selected case study against the strict time constraints, which make the time span and the effort required by even the easiest industrial case unfeasible for a course project. Indeed, in our program on Computer Science, courses have up to 12 credits[1], but most of them have

[1] In Italy each course has a number of *credits*, each of them corresponding to 25 hours of the "average" student, evaluating the efforts required to take it. Thus, the most time consuming course occupies a student for up to 300 hours, that is less than two man months.

just 6 credits (less than one man month of student effort) and that was the case with the course on Software Engineering.

In order to reduce the effort required without adopting a toy case study for the project, we initially chose to detail only some of the activities of the development process. Thus, the first year the course was held we decided to restrict the project scope to the design activity. The students were required to produce a design specification, using UML, starting from the requirement specification developed by the teachers for a software system supporting committee meetings on line. That organization was unsatisfactory, since the students were unable to fully grasp the real effect of their design choices, as they were not implementing their own design.

Therefore, the following year, we decided to include the implementation of the produced design using Java[2]. In order to get the extra time for extending the project to include the implementation phase, we asked for more credits assigned to our course (hence more student time) and we got 3 extra credits, going from 6 (150 hours of student time) to the current 9 (225 hours). However, the result was even less satisfactory, because to adhere to the time limits on the student work[3] we had to abruptly stop the project before its completion, as we had underestimated the effort required (we were apparently overoptimistic on the programming capabilities of our average students).

Thus, to be able to include all the phases from requirements to code on a realistic project, and at the same time complying with the constraint on the student effort, we decided to completely restructure the project. The organization discussed in the sequel is still adopted, and we plan to keep it for the near future at least, with small adjustments made each year on the basis of the previous experiences.

We split the development into distinct phases. At the end of each phase the students produce their artifacts restricted to a small part of the system. We collect them and give our *complete*[4] realization back to the students as a common starting point for the next phase. For instance, students are required to design using UML only a few significant classes; the overall class diagram to be implemented is produced by the teachers.

That organization has the following benefits:

- for each task the students learn how to perform it on a small subset of the whole system, with an effort comparable to that required by toy examples, and so they save time, though they are working on a realistic example, and can see the result of the task on the overall system even if they are not producing it themselves;
- the errors possibly introduced in a phase do not propagate to the next one, because the students are starting each stage from the teacher release of the product for the previous one; thus, errors in one phase do not prevent to successfully conclude the project;
- it is easier to compare the products (and hence fairly grade the projects) of different students in each phase, because they are all starting from the same official release.

On the other hand, to guarantee a "realistic" setting for the project, we need real case studies not already developed, convincing "clients", and a non trivial application domain, to experience the difficulties in understanding it. Therefore, we cannot reuse a case study already developed in some book, nor go for the n-th version of the "bank account" or of the "library" case study.[5] In all the projects, one of the authors (G. Reggio) played the role of the client because she had a real interest in using the software to be developed, thus providing a most convincing client.

2.3 Teaching an OO UML-based Software Development Method

Before devising the Software Engineering course described so far, we had earlier experiences, prior to 2002, in teaching OO UML-based software development methods, like COMET [10] and RUP [15], and also in tutoring students in their application. The difficulties encountered during such activities, led two of the authors to propose a new Model-based Adaptively

[2]For the academic year 2003/04 the project required to develop a software system for managing the different categories of questions for a quite complex quiz show (see http://www.slowtrav.com/italy/general/sc_eredita.html), where each question is selected with a complex algorithm depending on the state of the game of the different players.

[3]Being the first time we were using this modality, we were monitoring the time spent by a pilot group; thus, we were able to detect our error and modify the rules accordingly.

[4]The drawback is that this requires a lot of effort on the teacher side.

[5]Most recently, our choices were:

- Academic year 2004/05: software for managing a condominium; in Italy this is a real complex problem where the flat owners are deeply involved.
- Academic year 2005/06: ReqGuru, software for writing use case specifications, including a scenario editor (it has now reached the level of alpha-test, as an Eclipse plug in. *http://www.disi.unige.it/person/ReggioG/reqguru.html*).

Rigorous Software development method [1, 2, 3] (in the following shortly MARS). MARS, which is model-driven and adopts UML, enforces a tighter and more precise structuring of the artifacts for the different phases of the software development process, than required by most MDA compliant methods. That characteristic helps inexperienced developers to speed-up the process and at the same time facilitates the consistency checks among the various artifacts, and hence their final quality. Moreover, MARS strives to balance formalism and easiness of use: the formal background provides the foundational rigor but is kept hidden from the developer.

From the very beginning, we adopted MARS in our Software Engineering course and found it useful for our sample of rather inexperienced users. Indeed, we plan to apply it to real-size case studies in some projects with the industry to see if our promising results carry over to a population of well seasoned developers.

In order to make the paper self-contained we briefly sketch MARS; further explanations on the method can be found in [1, 2, 3]. MARS is a multiview, use-case driven and UML-based software development process. The activities of MARS and the artifacts to be produced are draw in Figure 1. The Concrete Design is based on the Abstract Design that realizes the Requirement Specification, which is built on the Problem Domain Model.

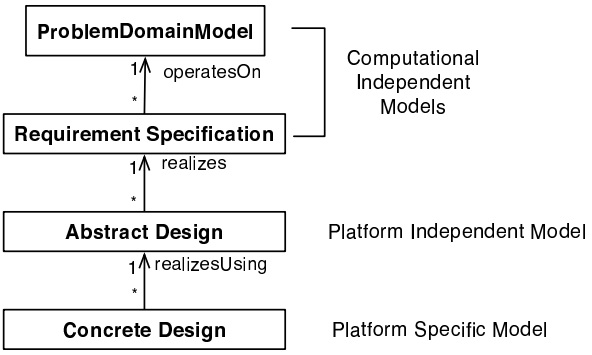


Figure 1. Phases and artifacts.

In MARS the Requirement Specification artifacts consist of different views of the System, including Data view, a description of the data types used to provide a rigorous description of such views. The Use Case view shows the main ways to use the System (Use Cases), making clear which actors take parts in them. Each Use case in the Use Case diagram is accompanied by a textual description following the format in [17].

The (abstract) Design, the structure of which is illustrated in Figure 2, consists of several views of the System:

The Data View defines all data types used by the entities composing the System.

The Static View defines the types (classes) of the entities composing the System.

The Behavior View describes the behavior of a part of the System.

The Configuration View describes the run-time structure/architecture of the System at some given point/situation during its life.

The Additional View describes how some entities of the System behave to get some particular task done.

The Interface View describes the GUIs associated to the entities realizing the interactions with the users.

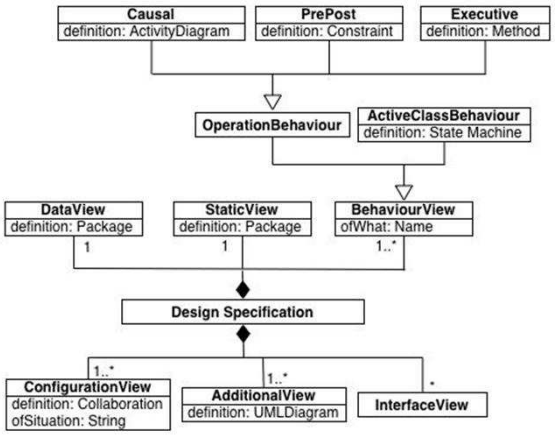


Figure 2. (Abstract) Design specification structure.

All the UML models produced in the various phases of the development process following MARS use only a specific subset of UML (see [2]), that has a well defined semantics.

Moreover, the expressions, the conditions and the constraints in any diagram are expressed by using OCL, and the actions using the UML action language.

MARS is totally compliant with the MDA philosophy. One of the strength of this method is that implementation is reduced to an almost mechanic (i.e.,

not creative) phase. The code can be derived, almost entirely automatically (except for the GUIs) starting from the Concrete design. That phase is currently performed by the students by applying a set of fixed transformation rules.

2.4 The Project: the Current Format in Detail

Currently, the project of our Software Engineering course consists in the development of a software system in Java, focusing on the design and implementation parts. The students start from a given requirement specification, and produce a working system, following the MARS method, and using Visual Paradigm[6] as a tool for writing the required UML models.

The project is split in four phases and for each phase the students have to produce some deliverables by a fixed deadline. After the deadline, we collect, correct and mark the students' artifacts and then we propose a common solution to be used in the subsequent phases. For the whole project lifetime, each common solution is modified and reviewed, to accommodate the feedback from the successive phases (requirements and design iterations).

The four *phases* are the following.

Phase 1 The input is a document containing some requirements of the system. This document includes Use Cases, Use Case diagrams and a UML class diagram representing the domain model. Students inspect it, highlighting possible ambiguities, incompleteness and any kind of requirements problems. The deliverable consists of the inspection report and of two Use Cases not already included in the input document.

Phase 2 The input is our new version of the requirements, revised taking into account the student inspections, and completed. The students develop parts of the Data view and of the Static View and moreover some Additional Views (i.e., sequence diagrams) following the MARS prescriptions. The deliverable is a UML model prepared using Visual Paradigm.

Phase 3 The input is our UML model with the complete Data, Static and (enough) Additional Views. The students define the behavior of a few classes included in the Data and Static Views, using state machine diagrams for active classes (as for example boundaries or executors [2]). For the operations of the passive classes (e.g., stores or calculators [2]), instead, they can choose among UML method definitions in the action language, pre/post conditions in OCL or activity diagrams. The deliverable is a new UML model.

Phase 4 The input is our complete model of the design. The students implement it in Java and the deliverable is a running system.

In the implementation phase students have several levels of freedom. They may choose their favorite IDE (between Eclipse and Netbeans 5.5) and their preferred Java GUI technology[7] (among Swing, AWT, and SWT). The use of GUI builders, such as, for example, Visual editor[8] and Matisse[9], is suggested but not imposed. We suggest to use Junit[10] to test the most complex classes of the System. Before starting the implementation phase we suggest some libraries that we consider useful, but the students can freely decide whether to use them or others of their choice. We do not give the students material or lectures about these technological issues, but let them find by themselves, mainly looking up documentation, tutorial and comparisons on the Internet to make their own independent choices. We think that this freedom is important so that the students can gain a better understanding not only on the existing technology, but also on the process of self-learning which will be of essence in their working days. Moreover, discussing with their colleagues who made different choices, and comparing their different results, they learn the impact of the choice of tools and libraries on the building of a quality product.

It is easy to understand that with such an organization it is crucial to keep in continuous contact with the students. Besides the lectures and the question&answer time, where we can meet face to face with the students attending the course[11], a big help to create an efficient learning community is given to us by Moodle. Moodle[12], a free open source software, is a course management system designed to help educators create on-line learning communities. It is not only useful as repository, but also very useful to manage complex projects. Indeed, the students, during the project development, can solve their problems simply by posting on the Forum, where educators and others students are often available 24h/7d to answer. Moreover, Moodle is

[6] http://www.visual-paradigm.com/

[7] Students are familiar with the underlying GUI technologies because they have attended a course on GUI development.

[8] http://www.eclipse.org/vep/WebContent/main.php

[9] http://www.netbeans.org/kb/articles/matisse.html

[10] http://www.junit.org/index.htm

[11] We have a majority of students working part-time, who are able to participate in person only in a limited way. That required us to adjust the modalities of interaction.

[12] http://moodle.org/

also useful for full time worker-students, as it allows for them to complete the project and pass the exam without attending the course.

3 Academic Year 2006/07 project: Easycoin

In the year 2006/2007 the students developed a simple program for cataloging collections of coins. This program is named *EasyCoin*[13]. The two main characteristics of *EasyCoin* are:

- it does not use a unique catalogue to classify the coins, i.e., it is not based on a specific catalogue;
- the visualization of the information is flexible: the collector can choose which information visualize and in which way (e.g., cards/sections and complete/reduced).

The aim of *EasyCoin* is to help the collector manage both catalogues and collections of coins (the user may switch from a modality to the other). In the modality "manage catalogues" *EasyCoin* provides the following features:

- insert, modify and delete the various information about catalogue entries (entities issuing coins, coin types, and coin issues);
- search in the catalogue;
- sort accordingly to different criteria;
- export in pdf, html and rtf. All reports produced can be viewed on the screen, printed and saved. They respect the visualization format selected (i.e., cards/sections or complete/reduced).
- export and import of the catalogue in a proprietary format (based on XML).

In the modality "collection of coins" *EasyCoin* handles different info about coins (e.g., grade, price, location, collections it belongs to). In that modality the main functionalities of *EasyCoin* are:

- create /delete a collection;
- insert, modify and delete a coin in a collection;
- search coins in the collections;
- compute some statistics on collections;
- export and import data about sets of coins in a proprietary format based on XML.

. The students had to implement the corresponding given design, and we suggested them to divide the work among the members of the group in equal parts (following a decomposition based on the pattern model-view-controller), and to use Junit to test some classes. The average size of the project is about 18.000 LOCs (this value differ from group to group because it depends mainly on the number/type of libraries used) per approximately 150/170 Java classes[14].

As SQL Database we elected to use the freeware H2 Database Engine[15], that permits the embedded connection mode (local connections using JDBC). Before starting the implementation phase we suggested (but not required) to use the iText[16] library for implementing the pdf exporter. One of the best implementations realized by the students is downloadable, togheter with the proposed design, from *http://www.disi.unige.it/person/ReggioG/easycoin.html* (see a snapshot in Figure 3).

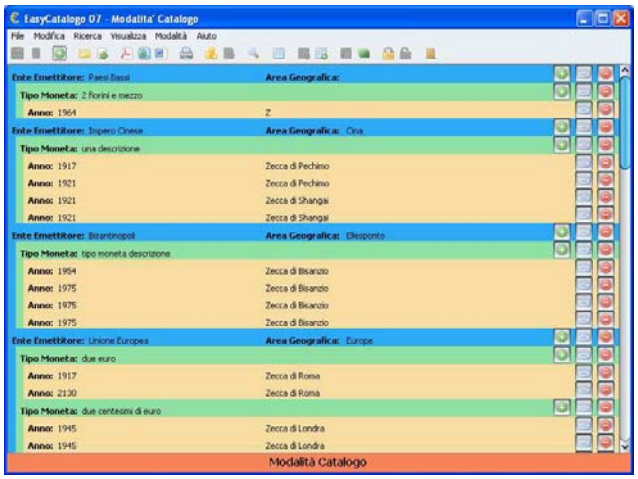


Figure 3. EasyCoin.

[13] Again, one of the authors (G.R.) played the role of client and user, since she is a coin collector.

[14] It could appear unrealistic that undergraduate students at their first serious programming experience were able to produce 73 LOC per hour (i.e., 18.000/246) and, indeed, the results of the questionnaire show that they worked more hours than expected. However, it is important to note that the 18.000 LOCs included comments, blank lines and a large amount of code automatically produced by a prolix GUIs generator. Furthermore, Java is a verbose language. Therefore, though some adjustment is definitely in order here to meet the time-frame, it is not as large as one could expect from the sheer data.

[15] http://www.h2database.com/html/frame.html

[16] http://www.lowagie.com/iText/

phase	delivered	passed	% passed	% drop-out
1	60	60	100%	0%
2	60	60	100%	0%
3	57	57	100%	5%
4 (1* deadline)	34	28	82,3%	-
4 (2* deadline)	12	12	100%	-
4 (overall)	40	40	100%	33,3%

Table 3. Students that have delivered and passed the project.

4 The questionnaire

After completing project, the students were asked to fill-in a questionnaire, in one hour. The purpose was to analyze the answers to better understand their opinions about the project and the MARS method both to calibrate/improve the next editions of the course and to gain further feedback on MARS.

The questionnaire consists of two parts. First we ask general questions (enrollment year, mean of student marks, experience in the industry if any, etc.) to better characterize the student population. Then we move to specific questions regarding the development project (complexity, adequacy of the time allowed to complete the work, etc.) and the MARS method (usefulness, complexity, applicability, etc.). Table 2 shows the 18 questions from the second part of the questionnaire. Students used a Likert scale [13] from 1 (strongly agree) to 5 (strongly disagree) to answer. Some free answer questions concluded the questionnaire.

4.1 Subjects

The subjects answering the questionnaire are the 34 students that delivered the project, meeting the final deadline. Other 26 students participated in the initial phases of the project, but were not able to complete it by the deadline. They are given the choice between completing the project for a later deadline and getting a lower rank (12 of them make use of it), or develop a new project (not organized in phases) during the summer. Table 3 indicates the students that delivered and passed the project and the drop-out rate.

From the analysis of the general questions we obtain the following results. Only 8.8% of the subjects worked full-time as programmer in the industry, 11.7% worked part-time and 79.5% had no working experience. At the question *"What is the mean of your marks?"* 32.2% answer medium, 47% high and 20.8% very high. No one has a low mean. At the question *"How do you describe the typology of the code that you have written until now?"* 73.5% answer only small programs, 17.6% medium programs and 8.9% large programs. Only 26.5% of the students declare to know a development method different from MARS and 29.4% state to have applied before agile methods.

4.2 Results

This section summarizes the main results obtained from our analysis of the answers in the second part of the questionnaire. Some insights can be obtained by looking at the descriptive statistics in Table 4. In this section, only questions associated with "significant" answers (i.e., having median $\neq 3$) will be considered. They are represented in bold in Table 4.

About the project

Students state that the time to complete EasyCoin was not sufficient (**Q1**: 52.9% considered it insufficient, 20.6 sufficient and 26.5% was not certain). They had difficulties to develop the project (**Q7**: 2.9% strongly disagree, 2.9% disagree, 35.3% not certain, 52.9% agree , 5.9% strongly agree) but at the same time they judge the experience of the project useful and formative (see Figure 4). Participants claim that the pictures of the GUIs of EasyCoin, given contextually to the requirements and refined during the design activity, were useful to make the requirements clearer (**Q3**: 67.6% retained it useful, 11.8% no and 20.6% was not certain).

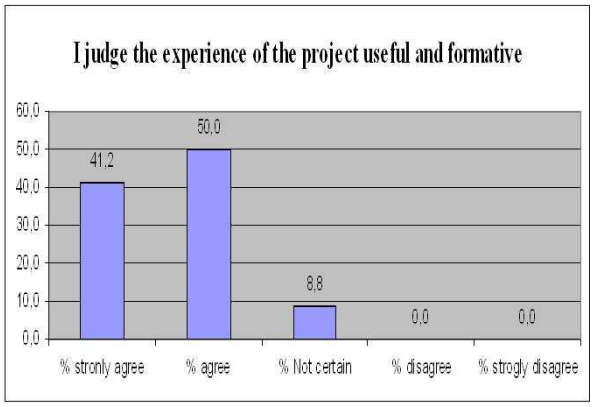


Figure 4. Histogram of Q5.

About MARS

Students find MARS useful (**Q9**:14,7% strongly agree, 64,7% agree, 20,6% not certain) and not difficult to apply (**Q8**: only 5,9% found difficulties to apply it to EasyCoin). They claim that MARS guides "step by step" the developer (**Q13**: 20.6% strongly agree

ID	Question
Q1	I had enough time to complete the EasyCoin project.
Q2	The EasyCoin requirements were perfectly clear to me.
Q3	The pictures of the EasyCoin GUIs helped me to better understand the requirements.
Q4	I experienced no difficulty in developing EasyCoin.
Q5	I consider the experience of the project useful and formative.
Q6	I had enough time to learn MARS.
Q7	I had difficulties in grasping MARS.
Q8	I had difficulties in applying MARS to EasyCoin.
Q9	I found MARS useful.
Q10	In the development activity it is better to use UML without any specific method instead of MARS.
Q11	MARS is too difficult to apply.
Q12	Does MARS balance agility and formalism.
Q13	MARS guides "step by step" the developer in the development activity.
Q14	Using MARS the implementation phase became purely mechanic.
Q15	OCL is useful to understand the diagrams of the design model.
Q16	The application of MARS in real cases takes too much time; hence MARS is not usable in practice.
Q17	I think that MARS could be successfully used in the professional practice.
Q18	In the future projects I will use MARS.

Table 2. Specific Questions of the questionnaire.

	Q1	Q2	**Q3**	**Q4**	**Q5**	Q6	Q7	Q8	**Q9**	Q10	**Q11**	Q12	**Q13**	Q14	**Q15**	Q16	Q17	Q18
Mean	**3,24**	2,76	**2,21**	**3,55**	**1,64**	2,45	3,15	3,27	**2,06**	3,52	**3,7**	2,85	**2,15**	2,79	**2,42**	3,30	2,45	2,73
Median	**4**	3	**2**	**4**	**2**	3	3	3	**2**	3	**4**	3	**2**	3	**2**	3	3	3
$\chi^2 - test$	0,73	-	**0,04**	0,3	**1,6*10^{-6}**	-	-	-	**0,0006**	-	0,17	-	**0,02**	-	0,3	-	-	-

Table 4. Descriptive Statistics and results of the χ^2-test.

and 50.0% agree) and consider OCL useful to understand the diagrams of the design (**Q15**: 5.9% strongly agree, 52.9% agree, 35.3% not certain, 5.9% disagree, 0% strongly disagree).

Chi2-test

χ^2-test [20] was used to gain statistical evidence over the whole set of subjects involved in the experiment. For each question, we have compared the percentage of answers with value > 3 (< 3) against that of answers having value ≤ 3 (≥ 3). We decided to adopt the most commonly used value for the alpha-level, that is, considering statistically significant a test with a p-value lower than 5% [20]. Results of the χ^2-test are shown in Table 4.

4.3 Threats to Validity

This section discusses the threats to validity that can affect our results [20]. The main threat could be due to the fact that the questionnaire was delivered and compiled by the students some days before the written examination and before the grading of the project. Students could have answered not sincerely to the questions hoping to have some benefits (or fearing some threats). To limit that threat we informed the students that the answers to the questionnaire would not influence the final vote of the course nor our opinion of them. Another threat to the validity of results is that the questionnaire was only completed by those who delivered their projects within the first deadline. The opinion of those who did not deliver the project is somehow neglected.

Though the selected subjects represent a population of students specifically trained on Software Engineering, UML and MARS, we doubt that the results obtained can be generalized for industrial senior developers [16]. In any case, only further specific studies, with other students and professionals, could confirm or contradict the obtained results. The threat, always present when experimenting with students, is the lack of experience of the subjects. Some questions require indeed some amount of experience to be answered (for example, "*does MARS balance agility and formalism?*", see [4]) and moreover the majority of the students is facing a realistic system for the first time; they know only one development method learned in this course and they have never applied agile methods. It will be interesting to repeat the questionnaire with professionals.

4.4 Discussion

The experiment was useful for two reasons: to have feedback on the project and to understand what students think about MARS. Even if we have obtained only four statistically significant results (**Q3, Q5, Q9** and **Q13**) other important information can be obtained analyzing the other answers.

We intend to take into account the opinions of the students to improve the next year project. Indeed, we plan to confirm the structure of the course and in particular our idea of having the students develop a realistic project as part of the course, because it has been highly appreciated by the students (see Figure 4). However we have to tune some details. We understand that the time given (1 week for the phase 1, 3 weeks for the phases 2 and 3 and 2 months for the implementation of the system) to complete the project was not sufficient (**Q1**). In particular students complained to have little time for the implementation (this is the phase with higher drop-out rate). Given than the number of credits of the course is fixed (9 credits correspond to 225 hours, of which 82 hours for the project) we can see only two solutions to solve this problem. Reducing the complexity of the project (choosing, for example, a domain better known to students - or reducing the number of requirements to implement) or increasing the team size up to 5-6 students as done in other courses, as reported in [6].

Another lesson that we have learned is about requirements: in the future we will have to pay major attention to their quality. They have been judged by the students as not always clear (**Q2**). As suggested by the students a way to make the requirements clearer is to accompany them with the pictures of the GUIs of the System to realize (**Q3**). As students appreciated the idea of having requirements and pictures, we plan to confirm it for the next years. Improving the clarity of our requirements may seem to lessen the amount of realism, as one referee has observed. However the emphasis of this SE course is on design and implementation. In a second advanced SE course the project is centered around requirement elicitation and analysis.

Even if students are not experienced programmers we think that their opinions/suggestions may be useful to refine/improve MARS. First of all, we have to improve the explanation of MARS (several students had problems to understand the method and needed more time to learn it, others had problems to apply it to EasyCoin; see **Q6, Q7** and **Q8** in Table 4). Surprisingly to us, students consider OCL unequivocally useful to understand the diagrams of the design (**Q15**). This result, though unexpected, goes in the same direction of [5]. Maybe, the most interesting results of the experiment are that students consider the experience of the project useful and formative (**Q5**) and find helpful (**Q9**) and not difficult to apply (**Q8, Q11**) MARS (**Q13**: MARS guide "step by step" the developer).

Some of the results have been confirmed by a set of free answer questions we have included in the questionnaire. In particular the students recognized the need for a method able to guide the programmer in all the phases of the development. Students question the excessive rigidity of MARS, but, at the same time, some of them appreciate the possibility to be precise and rigorous in the development, even if this requires a greater effort. A lot of students ask for a better tool support, able to derive the implementation in automatic way.

5 Conclusion

We have presented and discussed our current approach to organizing a project in software development within an undergraduate Software Engineering course.

Our organization has some distinctive features that are the result of various attempts at allowing the students to seriously experience most of the different activities involved in adopting a well-structured UML-based method. In particular we have tried to overcome some maturity, time and effort constraints on the students side with the main aim of making their experience as realistic as possible. By "as possible" we obviously mean a reasonable compromise that takes into account some constraints, like time and effort, and priorities, such as learning a rigorous UML-based development method and experiencing all the relevant development phases.

Admittedly, our approach is demanding on the teacher side; however we must say that it is also rewarding in terms of understanding the difficulties that the students − and, to a certain extent, the common developer − experience when seriously applying UML-concepts and related methods and tools. In that respect we also found of great help the proposed assessment via a questionnaire carefully analyzed and discussed, an experience that we intend to reiterate.

Acknowledgments

We would like to thanks first of all the students that along these years have taken part in this experience. We also gratefully acknowledge the benefit we gained from some very competent and accurate referee reports.

References

[1] E. Astesiano and G. Reggio. Tight structuring for precise UML-based requirement specifications. In *9th Monterey Software Engineering Workshop*, pages 16–34, September 2002. Lecture Notes in Computer Science n. 2941, Berlin, Springer Verlag, 2004.

[2] E. Astesiano and G. Reggio. Towards a well-founded UML-based development method. In *Conference on Software Engineering and Formal Methods, SEFM 2003*, pages 102–113, 22-27 September 2003.

[3] E. Astesiano, G. Reggio, and M. Cerioli. From formal techniques to well-founded software development methods. In *Formal Methods at the Crossroads: From Panacea to Foundational Support. 10th Anniversary Colloquium of UNU/IIST the International Institute for Software Technology of The United Nations University*, pages 132–150, 18-20 March 2002. Lecture Notes in Computer Science n. 2757, Berlin, Springer Verlag, 2003.

[4] B. Boehm and R.Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison Wesley, 2003.

[5] L. C. Briand, Y. Labiche, M. Di Penta, and H. D. Yan-Bondoc. An experimental investigation of formality in UML-based development. *IEEE Transactions on Software Engineering*, 31(10):833–849, 2005.

[6] K. Cooper, J. Dong, K. Zhang, and L. Chung. Teaching experiences with UML at the university of Texas at Dallas. In *Educators Symposium of the 8th International Conference on Model Driven Engineering Languages and Systems*, pages 1–8, 3 October 2005.

[7] G. Engels, J. Hausmann, M. Lohmann, and S. Sauer. Teaching UML is teaching software engineering is teaching abstraction. In *Educators Symposium of the 8th International Conference on Model Driven Engineering Languages and Systems*, 3 October 2005.

[8] W. Filho. Process issues in course projects. In *In the Proceedings of 27th International Conference on Software Engineering*, pages 629–630. ACM 2005, 2005.

[9] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3th edition.* Addison Wesley, 2003.

[10] H. Gomaa. *Designing Concurrent, Distributed and Real-Time Applications with UML*. Addison-Wesley, 2000.

[11] OMG. *UML 2.0 OCL Specification*, 2003.

[12] OMG. *UML 2.0 Superstructure Specification*, 2003.

[13] A. N. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter, London, 1992.

[14] S. L. Pfleeger. *Software Engineering. Theory and Practice. 2nd edition.* Prentice Hall, 2001.

[15] Rational. Rational Unified Process© for System Engineering SE 1.0. Technical Report Tp 165, 8/01, 2001.

[16] P. Runeson. Using students as experiment subjects - an analysis on graduate and freshmen student data. In *7th International Conference on Empirical Assessment in Software Engineering*, pages 95–102, 2003.

[17] S. Sendall and A.Strohmeier. Requirements Analysis with Use Cases. `http://lgl.epfl.ch/research/use_cases/RE-A2-theory.pdf`, 2001.

[18] I. Sommerville. *Software Engineering. 6th edition.* Addison Wesley, 2000.

[19] R. Szmurlo and M. Smialek. Teaching software modeling in a simulated project environment. In *Educators Symposium of the 9th International Conference on Model Driven Engineering Languages and Systems*, 1-6 October 2006. Lecture Notes in Computer Science n. 4364, T.Kuehne (Ed.), Berlin, Springer Verlag, 2007.

[20] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.

Students can get excited about Formal Methods: a model-driven course on Petri-Nets, Metamodels and Graph Grammars

Pieter Van Gorp Hans Schippers* Serge Demeyer
Dirk Janssens
Department of Mathematics and Computer Science
University of Antwerp
{Pieter.VanGorp,Hans.Schippers,Serge.Demeyer,Dirk.Janssens}@ua.ac.be

Abstract

Formal Methods have always been controversial. In spite of the fact that the disbelief about their usefulness has been corrected by a growing number of applications and even more publications, it remains a challenge to demonstrate the strengths and weaknesses of formal methods within the time constraints of a typical semester course. This paper reports on a new course at the University of Antwerp in which the introduction of a new formalism yields a better understanding of previously taught ones. While the exercises are designed to reveal the limitations of the formalisms used, students remain convinced that their formal models have more value than conventional source code.

1 Introduction

Formal Methods have been praised for facilitating the detection of inconsistencies and/or inaccuracies early in the development process. Still, formality is often falsely associated with the exhaustive specification of proofs in special symbols that are hard to read by most stakeholders. Therefore, they are often believed to be very costly and thus only applicable to very specific applications, such as embedded spacecraft software [10, 3]. To prevent the further spread of such myths in industry, software engineering students at the University of Antwerp are confronted with formal modeling languages in a variety of settings.

On the one hand, providing correctness proofs is part of courses on mathematics, databases, computer arithmetics and computability. On the other hand, several courses illustrate that modeling a system before its implementation assists in the early detection of misunderstandings. For example, already after a minimalistic introduction to the Unified Modeling Language (UML), student teams have lively discussions to reach a consensus on the structure of a domain model before constructing their first distributed network application. Similar discussions are held when student teams are instructed to construct a small compiler. Within this compiler project, students are not allowed to implement a semantical analyzer and code generator using the implicit abstract syntax tree from the parser generation framework directly. Instead, they are required to model a more abstract representation of the source language as a class diagram. The instructor verifies that students do not model implementation-specific concepts and ensures they do not abuse UML compositions and/or association cardinalities. While this exercise does significantly improve the students' knowledge of the UML, the learning curve is rather steep. Therefore, one can hardly expect that students fully appreciate the added value of their modeling task.

Recently, the introduction of a new undergraduate course indicated that this investment in precise modeling did result in more appreciation of further courses in formal methods in general and a model-driven engineering approach specifically. This paper presents that course, called *"Formal Techniques in Software Engineering"*, in more detail, using the following structure: Section 2 presents the role, objectives, structure and examination form of the course; Section 3 presents some of the artifacts to be developed by the students and indicates their relation and educational value; Section 4 summarizes the lessons learned after the first year while the final section concludes this paper.

2 Course Design

This section first describes the role of the course within the curriculum. Secondly, it discusses how the students' background and the complexity of supportive tools affects the objectives. Then, a description of the course structure and examination form clarifies how competences are transferred to the students and how students are evaluated.

*Research Assistant of the Research Foundation, Flanders (FWO)

2.1 Role within the Curriculum

As stated in the introduction, modeling is an integral part of the computer science curriculum at the University of Antwerp: after an absolutely fundamental course on discrete mathematics, several courses rely on formal models with laws to derive properties of the domain or system under study.

For example, instead of focussing an introductory "databases" course on querying concrete databases with SQL, systematic procedures are taught to transfer an Entity/Relationship model into a relational model. Moreover, the course illustrates how the formal nature of the latter model allows one to normalize databases automatically. Similarly, a course on computer arithmetics relies on a model of the standard for floating point arithmetics to reason about the correctness of floating point implementations.

While other courses use languages such as Z, B, SDL and statecharts, they leave the definitions of the involved models and languages implicit. In the third and final year of the bachelor program, the new course aims to teach students how different formal techniques relate to one another instead of leaving them isolated within the other, individual, courses. Primarily, students are taught that in model-driven software engineering, software is developed in different languages, at different levels of abstraction and that there are good reasons to do so. Secondly, the course illustrates how metamodeling and model transformation can be used to maintain the consistency between the models expressed in these languages.

2.2 Course Prerequisites and Objectives

This section describes the prerequisites and objectives that were defined when the course was first planned.

Students can enroll in the course provided they have practical programming experience, practical experience in the use of UML class diagrams and a solid understanding of the formal foundations of computer science (logic, formal languages). In practice, this expertise was provided by courses from the first two years of the bachelor program.

Officially, the expected learning outcomes were initially defined as follows:

> "Based on formal specifications (logical specifications, statecharts, Petri-Nets) the student should be able to build models expressing the intended functionality of a system, to analyse and to verify these models, and to generate a working implementation from them."

These objectives were designed with the AndroMDA code generators in mind [2]. Using this code generator from UML diagrams to Java web applications would bring students in contact with:

- conceptual modeling with UML class diagrams,
- query definition with a subset of the Object Constraint Language (OCL),
- user interface flow modeling with use cases and activity diagrams.

However, supervision of another undergraduate course (in which students have to build a large software system within two semesters) indicated that the use of AndroMDA required a significant learning curve for setting up a correct modeling and build environment. Moreover, despite the significant amount of code generation, students are still required to master the underlying J2EE technologies. Therefore, the use of AndroMDA would put too much weight on the use of a code generator and leave too little more room for teaching how the modeling languages used (class diagrams, activity diagrams, ...), are defined and kept consistent. Since the novelty of model-driven engineering does not consist of using code generation as such, but of adapting code generation environments (by using standard model and code transformation languages), the course objectives were informally adjusted to:

> "The student should be able to express the intended functionality of a system from different viewpoints in different formalisms (Petri-Nets, Graph Grammars) and ensure particular properties (boundedness, consistency, ...) of such models. The student should use state-of-the-art transformation techniques (model animation, model translation and code generation) to integrate distinct models and relate them to a complete implementation. The student should experiment with metamodelling in this context, and acquire an understanding of the benefits and limitations of the 4-layer meta model architecture."

After considering a combination of the DiaMeta [13] and Tiger [9] meta-case tools with the MoTMoT [14] model transformation tool, the AToM^3 tool was selected because of its completeness. AToM^3 offers mechanisms for metamodeling, concrete syntax definition, model transformation and code generation in a self-contained, Python based, environment [6].

2.3 Course Structure

The course takes place in the last semester of the bachelor program. It divides six European Credits (ECTS [4]) across seven theoretical sessions and eleven lab sessions of

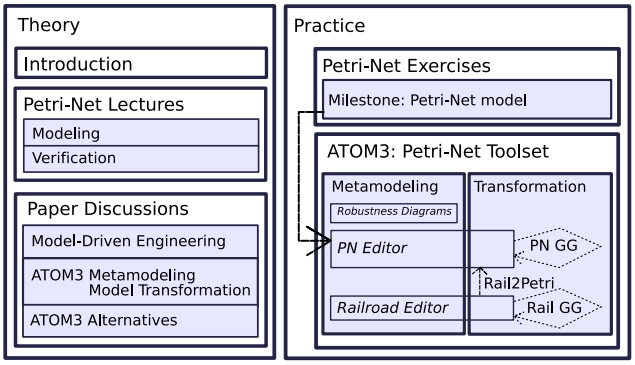


Figure 1. Structure of the Course

two hours per session. The course structure is visualized by Figure 1.

The first part of the theoretical sessions consist of lectures whereas the latter sessions are more interactive: in these sessions, the lecturer leads a discussion on a set of selected papers. The first lecture introduces the students to the Model Driven Engineering paradigm. Based on a concise, yet sufficiently complete textbook [11], the lecturer clarifies the definitions of a model, modeling language, metamodel, transformation definition and a transformation language. The following three sessions introduce the students to the most commonly used variants of the Petri-Net language. After explaining the role of invariants in the analysis of Condition/Event nets, the language is extended to Place/Transition nets with weights and capacities. A number of examples illustrate how the underlying formalism allows one to reason about deadlocks and other properties.

The first three lab sessions make sure that students master Petri-Nets both for modeling and verification. Since students are already heavily loaded with project work from other courses in their final year of the bachelor program, all exercises are made in the controlled context of the lab sessions. In the last Petri-Net session, students are given an assignment that will serve for evaluating their modeling ability. More interestingly, it serves as input for a Petri-Net editor that will be constructed in the second series of practical lab sessions.

The second and final part of the lab sessions distinguish the course from a conventional course on formal methods. In each of these eight lab sessions, students are given a well-defined assignment contributing to the construction of a Petri-Net based toolset using the AToM^3 tool.

Within the first session from this series, the teaching assistant demonstrates the core features of this tool. Moreover, the students are given an idea of what kind of toolset they will construct in the upcoming weeks. In the remainder of the first session, students are already creating their first metamodel, using the Entity/Relationship language. This first metamodeling exercise consists of the definition of the Robustness Modeling language [16]. This language consists of only five modeling constructs: Actors, Boundaries (Interfaces), Controls (Processes), Entities (Domain elements) and Use Cases. The language is selected because it is related to design rules that students have encountered in other software engineering courses. More specifically, the language includes well-formedness rules stating, for example, that user interface elements should not access persistent elements directly. Students realize that it is better to enforce these constraints by means of a specific modeling language than to check them in implementations based on a general purpose programming language. Moreover, students become familiar with metamodeling. At the end of the first session, students hand in their E/R model defining the Robustness Modeling language along with a sample Robustness Diagram that they created with the generated editor.

In the second AToM^3 session, an editor for Condition/Event and Place/Transition Nets with weights and capacities is constructed. The third section is the first hands-on introduction to model transformation: in this session, students are instructed to define a graph grammar realizing the Petri-Net transition semantics. Again, the introduction from the teaching assistant starts from a demonstration of the result that needs to be achieved. More specifically, a correct grammar is executed on a Petri-Net model of the dining philosophers problem. Students see how different rules are combined to detect whether a transition is enabled and whether tokens have already been moved from input places to output places. To illustrate that the presented techniques are not specific to Petri-Nets, a model of a railroad track is presented and a graph grammar is used to move trains across the tracks. These demonstrations are followed by a brief introduction to the core graph grammatical concepts, like the left- and right-hand side of a rule, the node identification mechanism and rule priorities.

At the end of the third session, students hand in their graph grammar for Condition/Event nets. In session four, students generalize this grammar to the Place/Transition variant of Petri-Nets (allowing more than one token per place). The grammar should handle weights on arcs from places to transitions properly. Although this seems like a minor conceptual extension of the Petri-Net variant, the grammar needs to realize a loop over a number of its rules. Since AToM^3 only supports priorities as a control flow structure, the realization of such a loop is not straightforward. In the fifth session, the grammar is generalized one last time by supporting weights on arcs from transitions to places and capacities on places too. At the end of the fifth session, students should be able to animate a Petri-Net model for access control that is part of the theoretical course material on Petri-Nets.

In session six, students return to metamodeling. They are instructed to model the "Railroad" language that was briefly demonstrated in session three. Since this language contains a set of related concepts (trains move across straight tracks in the same way as they drive through a station), it motivates the need for inheritance in the metamodeling language. Therefore, the Railroad language needs to be modeled with class diagrams instead of with Entity/Relationship diagrams. In sessions seven and eight, the AToM^3 lab is concluded with a transformation from Railroad models to Petri-Nets. This assignment challenges students to combine all competences acquired so far. Additionally, they are brought in contact with the additional techniques required to define an exogenous transformation (see Section 3.3). For example, this transformation between different languages introduces students to the need for (and nature of) traceability mechanisms.

The discussions from the four final theoretical sessions are based on three to four papers per session. To prepare for such a paper session, the students should read the papers and answer a few questions. In the lecture itself, the students debate the strong and weak points of a given paper, which results in quite vivid discussions. The goal during these debates is not that one student wins the debate, but rather that all students see the merits and differences in each approach. The final discussion session is based on two papers that generally classify today's model transformation approaches and two papers that each present an AToM^3 alternative in detail. During the debate, students are asked which aspects of their transformations they found difficult to express with AToM^3. Moreover, students need to assess what features of other graph transformation languages address these difficulties.

2.4 Examination Form

The course evaluation consists of two parts: permanent evaluation, and an oral exam with a written preparation.

First of all, the solutions to the lab exercises are taken into account, which test one of the course objectives, namely *"The student should use state-of-the-art transformation techniques (model animation, model translation and code generation) to integrate distinct models and relate them to a complete implementation."* This part of the examination is managed by electronic submissions to the university's electronic learning platform [17]. Since the AToM^3 lab sessions build incrementally upon one another, the teaching assistants have prepared partial solutions corresponding to all AToM^3 related deadlines. This can help students that have failed to meet one deadline in catching up for the next deadline. In practice, students have enthusiastically completed some incomplete exercises at home. The e-learning system has been useful to collect all AToM^3 artifacts within one web-based system. Moreover, it provides a comprehensive overview of the deadlines that have been met by each individual student. However, teaching assistants have selectively relaxed the firm deadlines by allowing e-mail submissions too. This was desirable when students encountered unexpected problems that were due only to unpredictable behavior of the AToM^3 tool.

Secondly, at the end of the semester the students must pass an oral exam with a written preparation. During this exam, the remaining course objectives are tested, namely *"The student should be able to express the intended functionality of a system from different viewpoints in different formalisms (Petri-Nets, Graph Grammars) and ensure particular properties (boundedness, consistency, ...) of such models."* and *"The student should ... acquire an understanding of the benefits and limitations of the 4-layer meta model architecture."* The former is tested with an exercise in Petri-Net modeling and verification while the latter is tested with a discussion on two papers selected from the list read during the paper sessions.

3 Course Artifacts

This section presents some of the models, metamodels and transformations that need to be developed in the new course.

3.1 Petri-Net Editor

Figure 2 displays a simplified version of one of the models used to convince students that AToM^3 can be used to build powerful editors. The Petri-Net models the dining philophers problem with six philosophers. The Petri-Net editor is developed in the second session of the AToM^3 labs. For improving the readability of this paper, the token capacities and edge weights have been omitted. The model can be animated using the graph grammar that is developed in sessions three, four and five.

In Figure 2, each philosopher is represented by a pair of places that are positioned on a diameter of the circle that represents the table. The outer place from such a pair holds a token when the philosopher under consideration is thinking. Therefore, in Figure 2, all six philosophers are thinking. These philosophers are numbered clockwise and with number 1 for the two places on the top of the figure. From the twelve places representing the philosophers, six places hold a token. These six places represent the three spoons and three forks. Since they hold a token, all silverware lies on the table.

After executing some graph transformation rules (that are selected automatically from the grammar), the Petri-Net model from Figure 2 has evolved into that of Figure 3. In the latter (version of the) model, philosophers 1 and 5 are

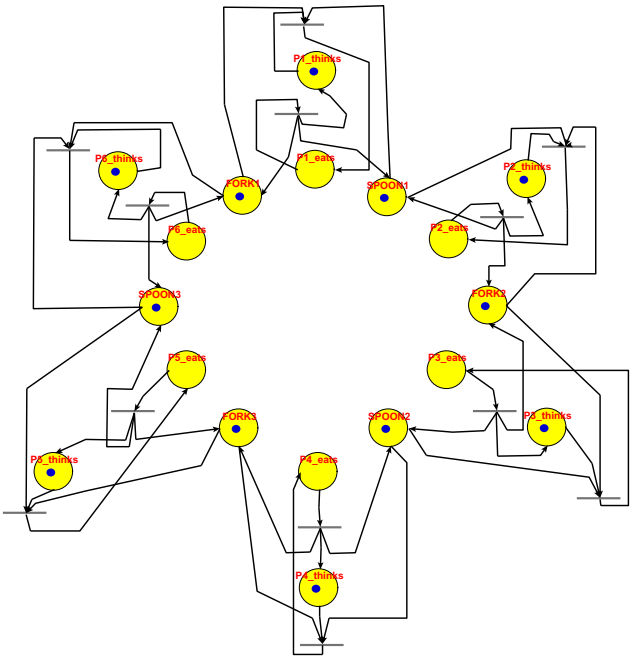


Figure 2. Petri-Net model: Dining Philophers.

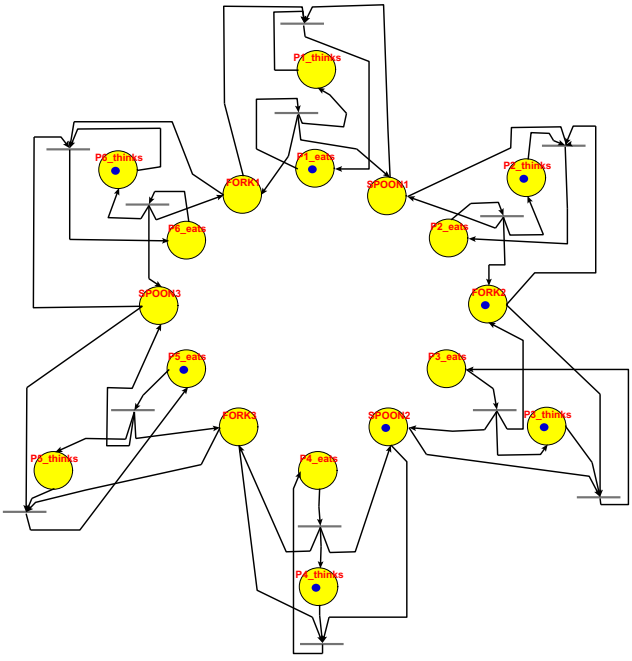


Figure 3. Model during graph grammar execution.

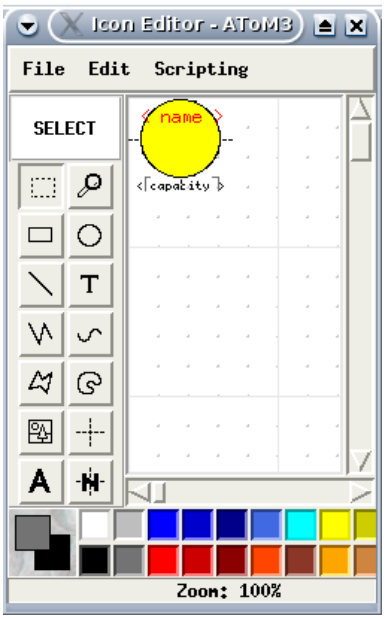


Figure 4. Editor for modeling the concrete syntax of visual language elements.

eating while the others are still thinking. Philosopher 3 can start eating since spoon 2 and fork 2 are still on the table. However, philosophers 2, 4 and 6 have to wait until their neighbours (philosophers 1 and 5) have finished eating and returned their silverware to the table.

The Petri-Nets language can be represented by a straightforward metamodel containing a *Place* class and a *Transition* class, related by an association for input places and an association for output places. Figure 4 shows the AToM^3 editor for modeling the concrete syntax of the *Place* class. This example teaches students how to use the basic constructs for representing individual language elements and for representing relations between these elements. In addition, students are confronted with the limitations of visual modeling. More specifically, the representation of tokens within a place is realized by some pragmatic programming at the level of the Python code that AToM^3 generates from the metamodel and the concrete syntax definition.

The transition semantics for Petri-Nets can be modeled by a variety of graph grammars. However, some essential rules occur in almost any solution. Figure 5 presents a snapshot of the active AToM^3 windows when such a rule is being edited. At the center of the screenshot, the graph transformation rule for incrementing output places is shown. The "order" field at the top of the dialog defines the priority of this rule and is set to 4. Rules with a lower order have a higher precedence. In this example, such rules make sure that the input places of the transition under consideration

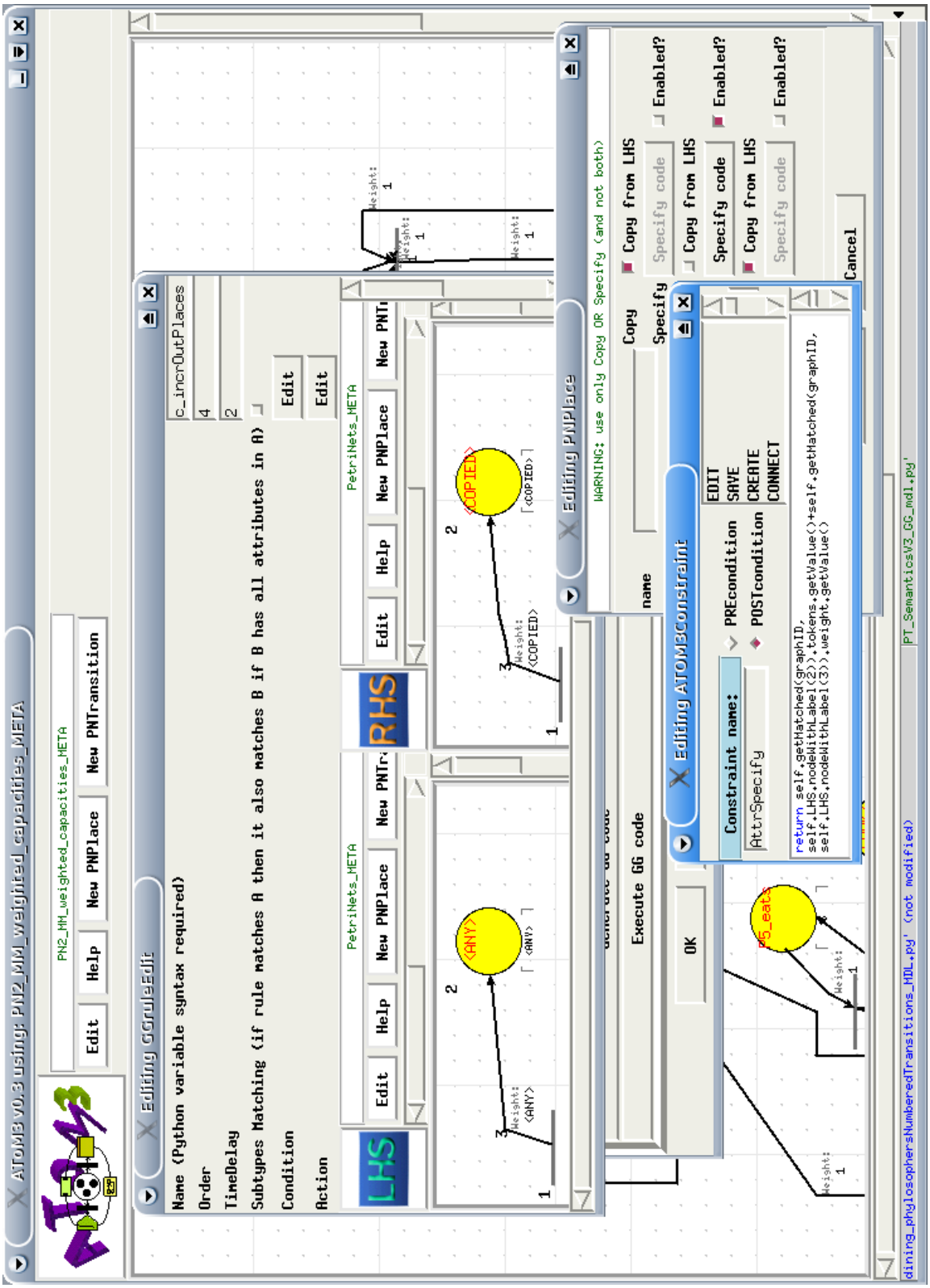


Figure 5. Screenshot of the AToM^3 dialogs for editing graph grammar rules.

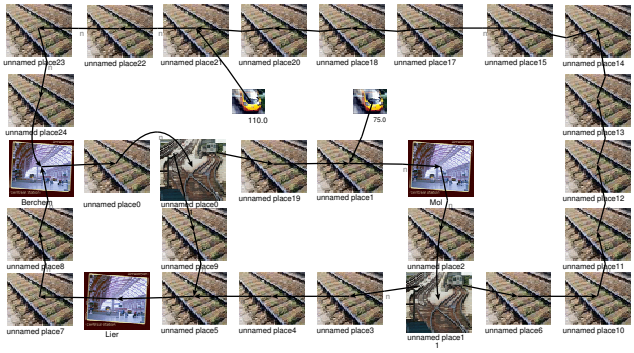


Figure 6. Sample model in the Railroad language.

hold enough tokens to enable the transition. Moreover, it is ensured that not more than one transition is enabled at a time. On the other hand, rules with a lower precedence make sure that the transition under consideration is disabled and that other rules can be enabled again.

Unlike most of today's generic model transformation engines [12, 5], AToM^3 supports the specification of rewrite rules in *concrete* syntax. The left-hand side of the rule in Figure 5 specifies that any transition, called *1*, holding an edge to an output place, called *2*, should be matched. The screenshot does not show that these nodes are constrained further by additional clauses from the "condition" field. The right-hand side of this rule copies the weight of the edge and the name and capacity of the output place. In contrast, the value of the "tokens" attribute from the output place is specified in Python code. The dialog at the bottom center of the screenshot shows that this value is defined by the old value of the "tokens" attribute plus the value of the weight of the incoming link. Within this dialog, the values of the Pre- and Post-condition widgets have no meaning for this example.

3.2 Railroad Editor

The Railroad language requires a somewhat more complex metamodel. In summary, it is desirable to use inheritance between classes such as *Track*, *Station* and *Fork*. Tracks and stations can be connected to exactly one next track while a fork holds a *left* and a *right* outgoing track. A Fork has an explicit property that denotes whether incoming trains will be switched to its left or right track. This property can be altered manually. Figure 6 illustrates what kind of systems can be modeled in the Railroad language. Note that the edges between the tracks, stations and forks define the path along which trains can travel the railroad.

Again, this language is supported by a graph grammar for simulation. As Figure 7 shows, this example brings students in contact with polymorphic matching: the rule for

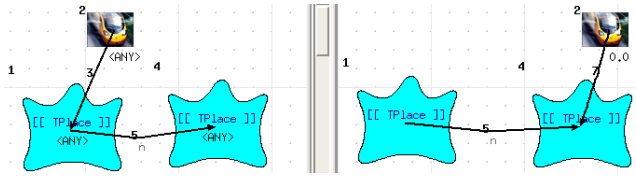


Figure 7. Polymorphic rule for moving trains.

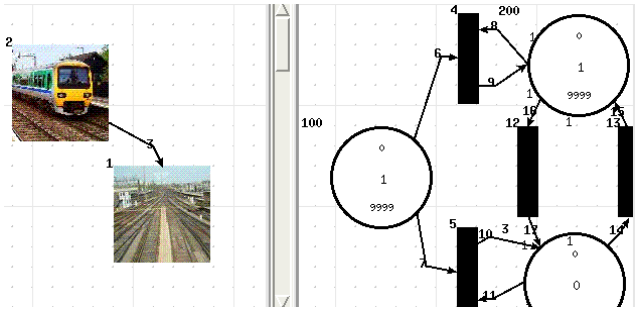


Figure 8. Translation rule for a left-going fork to a Petri-Net pattern.

moving a train from one track to the next one states that as soon as a train is located on a *Place* (a *Track* or a *Station*), it can be moved to the next track. This is realized by representing the connection with label *3* only on the left-hand side of the transformation rule and creating a new connection with label *7* in the right-hand side. All other properties are preserved.

3.3 From Railroad Models to Petri-Nets

The final course artifact is a graph grammar for translating Railroad models to Petri-Net models. Figure 8 displays a fragment from a student's solution. The rule matches *Fork* elements that hold a train and that are configured for moving trains to the left (this property would be visible when opening a property editor for node 1). For such *Fork* elements, the translation rule generates a *Place* element that holds one token (representing the train) and two subpatterns holding a transition, two opposite connections and a place. These patterns are used to encode the behavior of the Train switch: the upper place holds a token when trains should be moved to the left and the lower place holds a token when trains need to be moved to the right. Therefore, this transformation rule generates a token in the upper place and no tokens in the lower place. A transition connects the place from the upper subpattern with the lower place and vice versa. These transitions can be used to switch the *Fork* pattern in the Petri-Net from left to right.

Other rules generate links between the transitions representing the next-track relationship. These rules need to

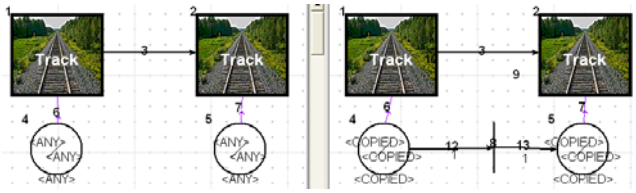


Figure 9. Navigation across traceability links.

employ some kind of traceability information. AToM^3 provides "generic links" for connecting model elements from different languages. However, since the use of this built-in traceability mechanism resulted in obscure error messages, students had to design an ad-hoc solution in Python code. On the one hand, it is unfortunate that not all students have thus not been able to "model" traceability links within the rewrite rules. On the other hand, the exercise illustrated that in the final lab session, most students master the tool "internals" sufficiently to implement an alternative solution creatively.

One student did succeed to apply the built-in traceability feature of AToM^3. Figure 9 for example displays his rule that properly applies generic links to generate a transition between places that correspond to two connected tracks. The left-hand side of the rule contains the two tracks, labeled *1* and *2*, that are connected in the input model. The edges labeled *6* and *7* elegantly model that these tracks need to be mapped to two places, labeled *4* and *5*, before this rule becomes applicable. The right-hand side of the rule specifies that between these two places a new transition, labeled *8*, should be generated.

In summary, it can be stated that the AToM^3 exercises have brought students in contact with the most essential techniques in model-driven development.

4 Lessons Learned

This section summarizes the lessons that were learned from designing and teaching the first edition of the course. In general, student satisfaction (as measured during a formal questionnaire) exceeds the expectations of the instructors. Nevertheless, several points for improvement are considered for future editions. The first part of this section presents the best practices that were applied in the design of the course. The second part discusses the outlook for the second edition. The section concludes by discussing the short- and long-term industrial applicability of the course's learning outcomes.

4.1 Best Practices

The instructors of the course have paid attention to the following didactical aspects:

Feasibility Study. A key factor that contributed to the success of the new course was the use of a sufficiently tested toolsuite. All solutions to the lab sessions were prepared before planning the complete course structure. This ensured that as long as students used the evaluated AToM^3 release, teaching assistants could guide them around known bugs and limitations. It should be noted that AToM^3 was not just the first and best tool selected for this course. Instead, during the course preparation phase, other tools were actually found unsuitable or incomplete for use by undergraduate students.

Expert Supervision. As a constructive guide to the adoption of formal methods, Antony Hall complemented his objections against misbeliefs about formal methods with a list of hints about training in formal methods [10]. Hall essentially confirms that after theoretical training in general discrete mathematics and a specific formal language, practical workshops are indispensable. Moreover, supervision by at least one tutor is said to be essential. Hall also reports about productivity problems in the context of non-user-friendly tools. By allocating two teaching assistants during the course of all AToM^3 sessions, the time that the students under supervision spent on figuring out the AToM^3 user interface specifics was significantly smaller than the time the teaching assistants had spent on that issue.

Work Incrementally. By building upon the results of previous lab sessions, students have been able to construct a realistic toolsuite within a very limited time frame. The result has convinced students that the techniques they have learned can be applied on realistic problems. By defining intermediate milestones for the construction of the graph grammar for animating Petri-Nets, students have been able to demonstrate their progress at the end of each lab. This approach resulted in student satisfaction weeks before the delivery of their complete toolsuite.

Start Small. Because the Petri-Net toolsuite was used as a partial evaluation for the exam, teaching assistants had to ensure that students produced a solution as individually as possible. This was achieved by starting the AToM^3 sessions with the development of the self-contained Robustness Diagram editor. Students should be – *and have been* – able to generalize their expertise from this small exercise to the larger ones.

Illustrate Applicability. Formal methods can help future system users understand what kind of system will be built by modeling functionality before it is realized. However, that requires that the formal model is made accessible to such a user [10]. Instead of claiming the applicability of Petri-Nets by means of natural language explanations, the course relied on the customer-oriented Railroad language. The translation between the Railroad and the Petri-Net languages provides concrete evidence that formal methods can be economically integrated into the requirements elicitation process.

Examples First. Another driving force for letting students define the Railroad language is that this exercise motivates the need for inheritance in the language for metamodeling. From such examples, it becomes straightforward to motivate why the Meta Object Facility (MOF), OMG's standard language for metamodeling, resembles class diagrams.

Problems First. Before referring to the papers on more powerful model transformation languages, the lab exercises expose the limitations of the simplistic AToM^3 approach. More specifically, the Petri-Net animation grammar illustrates that the execution order of graph grammar rules does not necessarily correspond to the order in which these rules are defined. By experiencing this problem in the lab sessions first, students understand why in Story Driven Modeling [7], graph transformation rules are embedded in an activity diagram.

4.2 Planned Improvements

The following list summarizes the future work on the course:

Provide Tool Feedback. AToM^3 was originally developed for research purposes. Applying it in a classroom context revealed several bugs and usability issues. Constructive feedback is collected from students and teaching assistants. This feedback may influence future releases of the tool.

Consider Other Tools. Since the tool landscape is rapidly evolving, new tools may provide the metamodeling, concrete syntax definition and model transformation features required to construct an equivalent of the Petri-Net toolsuite. Therefore, the maturity of alternative tools needs to be evaluated frequently.

Provide Reusable Integration Components. The motivation of the students can be increased by providing so-called *technical projectors*: for example, one could develop a Python component that generates a file compliant with a popular Petri-Net analysis tool from instances of an AToM^3 metamodel for Petri-Nets and vice versa. Such a component would close the remaining gap between the Petri-Net verification part of the course and the model-driven engineering part, without reinventing the wheel.

Extend Railroad Case Study. A feedback loop should be developed from the analysis of a Petri-Net that is generated from a Railroad model by means of the graph grammar. By illustrating students how such analysis results should be interpreted, the case study would practically show how model-driven engineering can help improving the safety of software.

Prepare Follow-Up Courses. The curriculum at the University of Antwerp also includes two new graduate courses related to Model-Driven Engineering. These courses need to be designed such that students can apply their AToM^3 expertise in tools based on actual MDA standards such as QVT [15] and industrial-strength frameworks such as the Eclipse Graphical Modeling Framework (GMF [8]).

4.3 Industrial Relevance

This section briefly discusses the short- and long-term applicability of this course in an industrial context.

In 2006, the computer science curriculum at the University of Antwerp has been completely redesigned. One novelty is that graduate students need to choose between an industrial, educational or research profile. Since the course being discussed in this paper takes place in the third year of the bachelor program, it should still fit into the three profiles. As Section 2.2 illustrates, this leads to subtle trade-offs. Coming back to the design of the course objectives, the learning outcomes that were initially planned would be directly applicable in industry: many organizations are struggling with the complexity of today's middleware and use code generators such as AndroMDA to tackle this issue.

The revisited learning outcomes may seem less applicable in the short term. However, the acquired insights are directly applicable for companies that need technical advice in the selection of an MDA tool. By gaining practical experience with model transformation, graduates should be able to look beyond marketing labels and investigate a tool's adaptability. Moreover, companies from e.g. the automotive industry demand expertise in the customization and integration of custom visual modeling tools [1, 18].

5 Conclusions

This paper presented a new course that combines traditional lectures and exercises on formal modeling and verification with a series of practice-oriented lab sessions on emerging model-driven engineering techniques. The course is focused on the definition and integration of languages rather than on the use of the languages as such. By incrementally developing an integrated case study within controlled lab sessions, students encounter problems in practice before reading and debating related papers. The permanent evaluation of the course indicated continuous student satisfaction, despite the theoretical nature of the background material and frequent failures of the supportive tool.

Acknowledgements

This work has been sponsored by the Belgian national fund for scientific research (FWO) under grant "Formal Support for the Transformation of Software Models". The authors wish to thank Denis Dubé for providing technical support during the preparation of the AToM^3 lab sessions.

References

[1] C. Bock. Visuelle domänenspezifische sprachen - der schlüssel zur modellgetriebenen entwicklung von mensch-maschine-schnittstellen? http://software-families.org/, 10 2006.

[2] M. Bohlen et al. AndroMDA Model Driven Architecture framework. http://galaxy.andromda.org/docs-3.2/, 2007.

[3] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.

[4] E. Commission. ECTS - european credit transfer and accumulation system. http://ec.europa.eu/education/programmes/socrates/ects/, 6 2007.

[5] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[6] J. de Lara and H. Vangheluwe. Using atom^3 as a meta-case tool. In *International Conference on Enterprise Information Systems*, pages 642–649, 2002.

[7] I. Diethelm, L. Geiger, and A. Zündorf. Systematic story driven modeling, a case study (Paderborn shuttle system). In *Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE'04*, Edinburgh, Scottland, 2004.

[8] Eclipse. Graphical Modeling Framework. http://www.eclipse.org/gmf/, 2007.

[9] K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer. Generation of visual editors as Eclipse plug-ins. In *ASE'05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 134–143, New York, NY, USA, 2005. ACM Press.

[10] A. Hall. Seven myths of formal methods. *IEEE Software*, 07(5):11–19, 1990.

[11] A. Kleppe, J. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Object Technology Series. Addison – Wesley, 2003.

[12] T. Mens and P. Van Gorp. A taxonomy of model transformation. In *Proc. Int'l Workshop on Graph and Model Transformation (GraMoT), Satelite Event of GPCE*, 2005.

[13] M. Minas. Generating visual editors based on Fujaba/MOFLON and DiaMeta. In H. Giese and B. Westfechtel, editors, *Proc. Fujaba Days. Technical Report tr-ri-06-275 - University of Paderborn*, pages 35–42, Bayreuth, Germany, 2006.

[14] O. Muliawan, H. Schippers, and Pieter Van Gorp. Model driven, Template based, Model Transformer (MoTMoT). http://motmot.sourceforge.net/, 2005.

[15] Object Management Group. *MOF QVT Final Adopted Specification – ptc/05-11-01*, 2005. http://www.omg.org/docs/ptc/05-11-01.pdf.

[16] D. Rosenberg and K. Scott. *Use case driven object modeling with UML: a practical approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[17] Universiteit Antwerpen. Blackboard academic suite. http://blackboard.ua.ac.be/, 6 2007.

[18] J. Weiland. Variantenkonfiguration von modellbasierter embedded automotive software. http://software-families.org/, 10 2006.

From Programming to Modeling: Evolving the Contents of a Distributed Software Engineering Course

J. Cabot[1], F. J. Durán[2], N. Moreno[2], J.R. Romero[3], A. Vallecillo[2]
[1]*Universitat Oberta de Catalunya,* [2]*Universidad de Málaga,* [3]*Universidad de Córdoba*
jcabot@uoc.edu, {duran, vergara, av}@lcc.uma.es, jrromero@uco.es

Abstract

Distributed Software Engineering (DSE) concepts in Computer Science (or Engineering) Degrees are commonly introduced using a hands-on approach mainly consisting in teaching a distributed and component-based technology platform (as Java Enterprise Edition or Microsoft .NET) and proposing the students to develop a small distributed software application with it. Though this approach provides some relevant practical knowledge to the students, we believe that it is not the most appropriate one to teach all the specificities of DSE. Thus, in this paper we report on our experience with the redesign of the contents of an initial DSE course following an MDA-based approach. By raising the level of abstraction we gained modularity, separation of concerns and technology independence, while making the course evolve according to the latest trends in software development methods. Our experience was not free from problems but in general the initiative has been positively evaluated and welcomed by the students.

1. Introduction

The growing adoption of Model-Driven Development (MDD) and Model-Driven Architecture (MDA) approaches in today's software development projects is shifting the focus of existing software engineering methods from code to models, which are now the primary artifacts of the software process.

To prepare our students for this new development context, we believe that current software engineering courses must evolve in this direction. In this sense, the main goal of this paper is to present our experience with the redesign of the contents of a distributed software engineering (DSE) course (DSE understood as the engineering of distributed software [1] and not as the process of distributed development of software).

This course is taught at the Open University of Catalonia (UOC) [2], a fully virtual university founded in 1993 and with more than 5.000 computer science students. Though studying in a virtual university poses some additional problems on the students' learning process of a DSE course, their discussion is out of the scope of this paper. We just want to remark that, in a virtual course, the teaching materials become the primary reference for the student (instead of the lecturer which now adopts more the role of a tutor). Therefore, an adequate and well-explained content selection for the DSE course is a key issue to improve its quality.

In the UOC, the DSE course is the most advanced course in the software engineering area. In previous courses, students learnt the phases of the software development cycle and how to specify and design (in UML) a given software system. Fundamental concepts of distributed systems (as asynchronous communication or clocks) are also reviewed in previous courses. Then, the DSE course addresses the specificities of developing distributed software systems with emphasis on the use of architectural and software components during the specification and implementation of such distributed systems. The DSE course has an estimated workload of 6 ECTS credits [3] (around 180 work hours in total).

The rest of the paper discusses the reasons that motivated us to change the DSE course contents (section 2), justifies the proposed new contents (Section 3) and gives some preliminary conclusions after the completion of a first term using the new contents and materials (Section 4).

2. Reasons for a change

The previous contents of the DSE course consisted of three different modules. The first one provided a brief introduction on distributed systems and a description of the RMI, CORBA and DCOM technologies. The second module was devoted to the study of the component technology, including some basic definitions and their representation in UML. Finally, the third one (by far the largest one) contained a thorough description of the different technologies forming the Java Enterprise Edition (JEE) platform.

Clearly, with those contents, the course was more a programming course than a software engineering course. The contents were more focused on describing a particular technology than on explaining concepts and techniques common to all kinds of distributed systems. After a few terms with those contents, we realized about a number of drawbacks that impaired a correct learning of the DSE concepts and motivated the redesign of the course contents as explained in the next section. Main drawbacks encountered:

- Students were unable to develop complex systems. This requires raising the level of abstraction far beyond their programming-oriented view.
- No methodological aspects about the specification/design of distributed software systems are taught. Even the few ones that do appear in the course are mostly hidden by the technical details of the JEE platform that are regarded as more relevant by the students.
- There was an important lack of architectural concepts. As a result, we found that students always proposed the same solution (the classical 3-tier architecture) to all kinds of distributed systems presented during the course.
- The focus on specific technologies caused that many of the contents became quickly outdated due to the high speed of technology evolution. This implies that teaching materials needed to be continuously updated and, more important, that part of the time students invested on the course became obsolete shortly after the course. Instead, a more abstract view of the area is more stable and offers a more long-term usefulness of the learned contents.
- No distinction between platform-independent and platform-specific concepts was provided. One of the consequences is that, at the end of the course, students tent to believe that they only knew how to develop distributed systems with JEE (and not, for instance, with .NET or other platforms). Their perception was that all acquired knowledge was specific for that technology platform.

3. Component Software Engineering and Distributed Systems: A New Approach

As mentioned in the first section, this course is intended to provide an introduction to the concepts and fundamental methods for the design and development of component-based distributed applications, thus complementing the knowledge acquired in previous courses. In this way, the course explains both the theoretical concepts in the design and development of distributed applications, and the way in which the present technological platforms implement such concepts.

Our main goal is to combine an eminently practical approach with a conceptual frame that makes it independent of the technology so that the problems sketched in the previous section can be overcome. The new approach we follow counts at the moment on an increasing interest from industry, where the complexity of the applications demands manageable, structured and easy to understand designs.

The new contents of our DSE course consist of five modules which are devoted to the following sub-goals:

- **Module 1**: To understand the different concerns and aspects that need to be considered when developing this kind of applications.
- **Module 2**: To know the different architectural styles and how to define the more suitable software architecture according to the particular characteristics of each application.
- **Module 3**: To learn how component-oriented programming can serve as an implementation technique for software systems.
- **Module 4**: To learn how this theoretical frame can be realized on current technological platforms, with special attention to JEE.
- **Module 5**: To understand the similarities and differences between the different technological platforms currently available, and to become conscious that the software development process explained here can be used independently of the final implementation platform since all of them follow the same architectonic principles.

In the following subsections, we will briefly discuss how our students are led to acquire the knowledge and skills required in each module.

3.1 Module 1: Viewpoints to consider in the development of a distributed system

Most current proposals for describing the global architecture of distributed systems are based on the identification and separation of independent viewpoints. Each one of these viewpoints focuses on concrete aspects of the systems, abstracting from the rest, and thus simplifying the design.

It normally happens that the different aspects to consider in a distributed application are intermingled, thus increasing the complexity of its design and implementation. Even worst, since it is unusual to apply any systematic method to handle these concerns separately or to integrate them in a controlled, their

handling is usually rather chaotic, and dependent on the programmer or designer at hand.

The *Software Engineering* community has already come to the conclusion that these problems must be addressed in the first stages. Specifically, at the architectural level, when decisions on structure, general goals and strategies, implementations platforms and system deployment are taken.

In this line, several (international or de-facto) standards have been published (the IEEE Std. 1471 standard, Kruchten's "4+1" views model, or the Reference Model for Open Distributed Processing, known as RM-ODP), which try to settle the basis for the description of the global architecture of software systems, using a modular separation of the design in different perspectives, named viewpoints. We agree that the use of standards is the most effective way to achieve the required interoperability between the different parties and organizations involved in the design and development of complex systems.

So, among all the standard proposals, we have chosen RM-ODP [4] as a framework for teaching the DSE course. RM-ODP is a joint ISO/IEC and ITU-T standard which is currently receiving increasing interest by many large companies and organization. It provides a comprehensive and coherent framework of concepts for the specification of complex large scale IT systems and now it has taken on a new significance in the light of the MDA initiative from the OMG and the wide-scale adoption of Service-Oriented Architectures (SOA). In addition, major companies and organizations are starting to use RM-ODP as an effective approach for structuring their large-scale distributed IT system specifications, mainly because the size and complexity of current IT systems is challenging most of the current software engineering methods and tools. These methods and tools were not conceived for use with large, open and distributed systems, which are precisely the systems that the RM-ODP addresses.

RM-ODP defines five different and complementary viewpoints: enterprise, information, computation, engineering, and technology. Each of these viewpoints is studied separately inside the course. Besides, in RM-ODP, an "abstract" language is defined for each of the five viewpoints. They are abstract in the sense that they define what concepts should be used but not how they should be represented. Several notations have been proposed for the different viewpoints, although we have opted for the general purpose modeling notation UML (Unified Modeling Language [5]). UML is familiar to our students, easy to learn and to use by non technical people, offers a close mapping to implementations and has commercial tool support. Furthermore, the use of UML for ODP system specification is currently being standardized by ISO/IEC and ITU-T [6], which allows us again aligning our course contents with international standards—something specially important in any engineering discipline.

In summary, this first module serves as an introduction to the concepts and mechanisms on which RM-ODP bases the architectural description of distributed systems using independent viewpoints. In addition, it reviews the main international standards related to these subjects that guarantee portability, interoperability and compatibility between applications developed by different enterprises or organizations.

From the five ODP viewpoints, in this subject we concentrate on two of them: the computational viewpoint, which provides the high-level description of the software architecture and functionality of the system in a platform and technology independent manner; and the engineering viewpoint, which describes the concepts and mechanisms used for the distribution of that functionality across different physical nodes, i.e., machines and processes.

3.2 Architectural styles for development distributed systems

The computational viewpoint of a distributed system determines its software architecture by a high level of abstraction description of its functionality in terms of architectural components, interfaces and connectors. Thus, the architectural components encapsulate the basic system's functionality, provided through the components interfaces, whereas the connectors describe how these interfaces are connected to achieve that functionality. In this module, we show the students the advantages of having an independent technology design for this viewpoint, i.e., a design that is not focused on how the architectural components will be later placed and distributed in physical nodes.

In this regard, we put special attention on the importance that the correct choice of an architectural style has in the development process. We present to the student the commonly used architectural styles for the development of distributed systems (e.g., multi layers, client-server and peer-to-peer architectures) though we focus on the three-layer (or n-layer) architecture due to its importance in the development of web applications.

At the end of this module, students are able to find and adapt an architectural style to meet a concrete system requirement specification, taking into account not only its functional but also extra-functional requirements such as performance, scalability, etc.

This is not an easy task because we can have several appropriate architectures that fit well with a software specification. So, to decide which is the best one requires compromising a set of quality criteria, many of which are usually contradictory to each other.

In addition, this second module tries to show the student how to make the architectural design of a system software using UML. This aim requires that the student understands, on the one hand, the role of the architectural design and its relevance in the development process of distributed systems and, on the other hand, the importance of reusing existing architectural solutions to address a new system design with similar characteristics.

3.3 Component-oriented programming as a technique for the implementation of software architectures.

Once we have selected a specific architectural pattern for our application and the description of the software architecture has been made, we need to develop the system. It is possible to use different alternatives, depending on the programming paradigm chosen: structured programming, object-oriented programming, component-oriented programming, aspect-oriented programming, etc. Each of these paradigms has associated a different technology and a series of programming languages, which are more apt for a specific type of systems. This module focuses on one of these paradigms, namely component-oriented programming (COP), currently the most widely acknowledged and used for developing distributed applications.

Note that at the specification level (see previous section) we refer to architectural components instead of referring to software components. The latter ones implement the functionality architectural components defined by the software architecture of a system (i.e., software components *realize* architectural components).

It is also important to note that the COP concepts, mechanisms and processes presented in this module are described in a general way, independently of any concrete platform or implementation technology, in order to separate the concepts specific to this discipline from their implementation in any concrete platform (commercial technologies evolve much faster than their supporting theoretical concepts). Hence, in this module we adopted the definition of "component" (proposed in [7]) that considers that "the specification of a component represents the specification of a software unit and describes both the provided services, as well as the required ones from other components, and the behaviour of any component instance concerning to its specification." These "component specifications" define the abstract components that comprise the system design and refine the previous ones identified in the computational viewpoint specification although there may not exist a one-to-one correspondence among them. That is, an architectural component may be implemented by several different software components interconnected, that jointly realize the services offered by the specification of such component.

RM-ODP does not prescribe any specific development process to define the tasks that must be performed by the software engineer to "transform" the architectural components and connectors into software components. Therefore, in this course we have tried to offer the students a generic vision of the different software development processes that permit us to implement the requirements of the software architecture from these software components. In any case, and for practical reasons, we try to follow the approach by Cheesman and Daniels [7] because it is intrinsically easy, and also wide well known and adopted in practice.

Architectural and software components are described by means of UML 2.0 component diagrams. In addition, for the representation of physical structures, such as DLL, executable files, etc., UML deployment diagrams are used.

3.4 Implementing with JEE

The basic objective of this module is to show the student how the theoretical concepts studied in the previous modules are implemented in a specific platform. In particular, we describe in detail one of the platforms commonly used nowadays to develop distributed systems: Java Enterprise Edition.

The module explains the principles of the JEE platform, its elements and the architectural patterns it provides. The module is focused on the study of multi-layer architectures, as proposed by the JEE application model, and analyzes the components and technologies offered by this particular platform in each layer (the Enterprise Java Beans, the Java Server Pages, etc.).

Once, the student has acquired a basic knowledge of the JEE platform, he/she learns how to transform the platform-independent specification of the distributed software system (obtained as a result of defining the RM-ODP viewpoints described above) into a platform-specific design for the JEE platform. This design is represented using specific UML profiles for describing JEE applications. If not before, during this translation

the student finally realizes that the contents of the first three modules can be used regardless the technology platform where the system is about to be implemented.

The last part of this module gives some recommendations that help in choosing the right JEE technology for each part of the system during the translation from the RM-ODP specification to the JEE design. These recommendations are based on the own professional experience of most of the course tutors.

3.5 Other technological platforms

Finally, this last module introduces alternative platforms to JEE. This module is intended to provide an historical view of the implementation technologies that can be used for the development of distributed applications. The main principles of CORBA, Microsoft .NET and the Web Services implementation platforms are explained here using the same simple distributed application to illustrate all of them. Thus, the skills and knowledge acquired by the student in this module will allow him to establish the main similarities and differences between them.

Additionally, we believe this comparison facilitates that students become familiar with the main keywords and acronyms used by the different platforms and that this improves the student's confidence on his/her abilities to develop distributed systems in any kind of technology platform.

4. Conclusions

After completing a first term with these new course contents, we must say that the preliminary results are quite positive. Though we have not conducted a formal poll, informal opinions expressed by the students show that they quite prefer this new approach. The most cited reason is that now they have a better overall view of the aspects, concerns and problems of distributed software engineering, without losing a practical view (the programming part with JEE). We found this positive feedback somewhat surprising because students tend to prefer practical contents instead of more "abstract" concepts.

Our approach was not free from problems, either. We would like to point out some of the challenges that we think must be addressed in order to successfully teach a DSE course with our proposed set of contents. They are based on our findings after this experience.

- The contents of the course are quite extensive. To facilitate the student learning process additional auxiliary materials (as tutorials, case studies,…) should be provided
- The course requires a broad set of previous knowledge (software engineering, Java programming, databases,…) and thus, the course pre-requisites must be properly defined and enforced.
- Technical assistance should be provided during the programming part. Implementing an application with JEE requires installing and configuring an application server (*JBoss* in our case), an integrated development environment (*Eclipse*) and a database server (*MySQL*). To avoid students losing too much time with these low-level tasks, some kind of technical assistance should be available. We provided a virtual lab with a specific tutor that answers all installing and configuration questions within 24 hours.
- Difficulties in finding tutors able to teach the course. An adequate profile for this course must mix technical skills with deep UML analysis and design capabilities. We have had serious problems in finding tutors with such a profile. An alternative way to tackle this problem would be to split the course into two separate parts, each one with a different tutor.

Finally, we expect that our experiences and observations can be useful to other educators who are involved in teaching *Distributed Software Engineering* (or similar) courses.

5. References

[1] J. Kramer. "Distributed software engineering", ICSE'94, 253-263, 1994

[2] UOC, "Open University of Catalonia", http://www.uoc.edu/web/eng/index.html.

[3] European Commission, "ECTS - European Credit Transfer and Accumulation System", http://ec.europa.eu/education/programmes/socrates/ects/index_en.html.

[4] ISO/IEC, "Reference Model for Open Distributed Processing" ISO/IEC IS 10746 ITU-T Recs X.901 to X.904, 1997 http://www.rm-odp.net.

[5] OMG, "UML 2.0 Superstructure Specification," OMG Adopted Specification (ptc/03-08-02), 2003.

[6] ISO/IEC, "Use of UML for ODP system specifications" ISO/IEC FDIS 19793, ITU-T Rec. X.906, 2007 http://www.rm-odp.net

[7] J. Cheesman and J. Daniels, *UML Components. A simple process for specifying component-based software.* Addison-Wesley, 2000.

Teaching MDA: From Pyramids to Sand Clocks

Ileana Ober
IRIT – Toulouse University
118 Route de Narbonne, F-31062 TOULOUSE CEDEX 9
ileana.ober@irit.fr

Abstract

One of the main issues in teaching is finding the best abstraction that can help the student get a correct picture of reality. This issue is particularly sensitive when teaching model driven development concepts. Surprisingly, we noticed that even experienced students find it hard to understand the MDA philosophy and that their experience in modeling and using modeling tools does not facilitate their comprehension. Experienced students perception of the meta level is disturbed by the lack of interoperability between various commercial tools, which contradicts the beauty of the pyramidal view of model driven development. In this paper we propose a new point of view on the relationships between the concepts handled by the model driven development technology. This new vision, which is centered on models, proved much more illustrative for the state of affairs in MDD industry as confirmed by feedbacks received from students and by their results.

1. Introduction

The curriculum of the Toulouse University induces an early contact with UML and modeling in general, as means to model and develop software. Students get familiar with the language at the undergraduate level and apply it intensively on scholar projects. Additionally, starting the last year of the undergraduate level, and during each year of the graduate level, students get in contact with the "real" world, through various internships in companies that use the techniques learned at school. More and more, during the last years, the internships confront the students with the use of MDA techniques on real size projects. This usage varies from the "simple" use of modeling languages and tools, to the use of model transformation techniques.

At the graduate level we have a course called "Modeling and meta modeling" which aims to revisit MDA techniques in order to structure the information that students have received through various sources at school and during their practical work in internships. It is at this point that we tackle issues as meta-modeling, levels of abstraction, model transformation, etc. In spite of the relatively good background of our students, as proved also by their results in internships, feed-backs of their industrial tutors and various examinations, we have difficulties in finding the arguments and vocabulary that would allow them to easily switch between abstraction levels and to master the rationales of MDA concepts [1].

We make the case that their early contact with industrial projects, and real-life projects, does not necessarily help them, and thus they find it hard to accept some of the MDA purist principles.

Taking as a basis this observation, and using a theory that we developed on the role of metamodels [4], we tried during the last two academic years a new approach in teaching MDA.

This paper overviews two strategies of presenting MDA concepts and discusses on the advantages of the one that proved most successful.

2. Teaching MDA in the spirit of pyramids

When describing the models, meta-models and their mutual relationships, the classical picture used is that of an edge-up pyramid (see Figure 1), with meta-models on top of models. The meaning of this picture is basically that a model corresponds to a single meta-model and that each meta-model may have several models conforming to its definition. This is the classical view of the model ─ meta-model relationship, which was already presented in the first versions of the UML standard. It occurs again in the last versions, and is discussed into more detail in [2].

While this picture is of course accurate and suggestive for the instance-like relationship between model-realization, models and meta-models, our experience shows that it may be confusing for people familiar with some individual MDA concepts, but lacking the maturity to have a synthetic and abstracting overview.

Discussions with the students allowed us to realize that one of the most disturbing (if not The most disturbing) factor in acquiring a synthetic overview is the technological mess at the level of modeling tools, primarily the lack of real interoperability between tools.

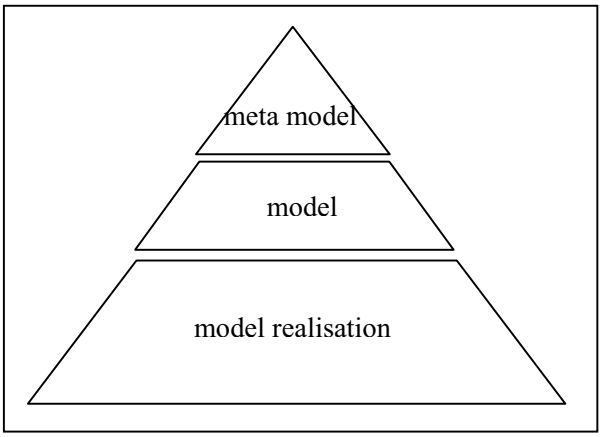


Figure 1 Pyramidal view of MDA relationships

Indeed, students find it hard to understand, for instance, that various UML tools are not actually compatible, especially as they all implement a standard modeling language and there is an allegedly standard interchange format. As teachers, we try of course to find reasons, related to the history of tool development, to the complexity of the XMI standard, to politico-economical issues. It helps students, but some fuzziness still remains.

It seems quite unreasonable in these conditions to argue on the uniqueness of the UML meta-model, and to state that each UML model conforms to the one and only UML meta-model. Our experience shows that this difficulty even more significant for students that have some experience.

The mismatch between reality and its abstraction raises questions about the accuracy and the soundness of the concepts we teach. Therefore we have looked for an abstraction closer to reality.

Our research work on the interoperability between tools based on (ad-hoc) standards [2] gave us the opportunity to reconsider the relationships between the various MDA concepts. As a result we had to drop the pyramid-like view of the model -- meta-model relationships for a sand watch-like view.

3. Sand watches: an alternative view of MDA concepts relationships

The sand-watch view starts from the observation, that in the "real world" models rarely conform to a unique meta-model. Especially when they are dealt with in various tools, the models actually conform to a set of (related) meta models.

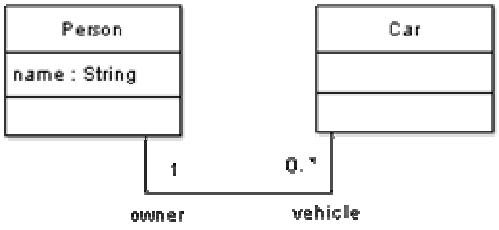


Figure 2 UML model example

Let us consider a very simple UML model, as depicted in Figure 2, and two distorted and simplified "UML" meta-models, such as given in Figure 3 and Figure 4.

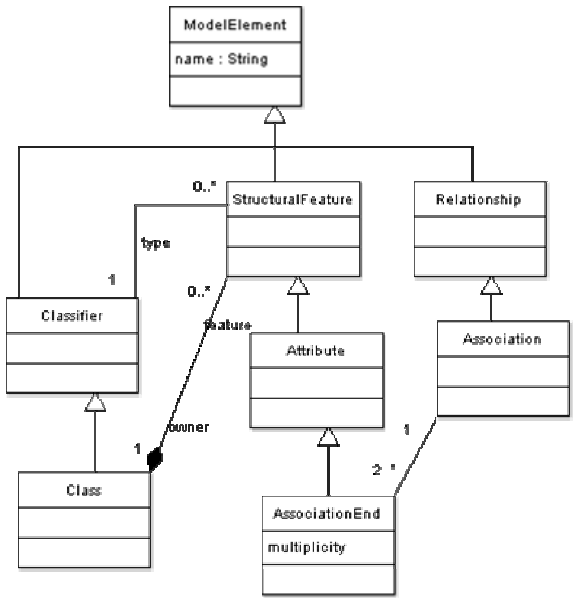


Figure 3 First meta-model candidate

A quick look at the three figures allows us to say that the model in Figure 2 actually conforms to both meta-models. In fact, it also conforms to quite a lot of other meta-models, which are either super-sets of these ones, or "equivalent" to them. We do not discuss here on the equivalence relationship between of meta-models, which is a research topic by itself. We rely on the intuitive notion of meta-model equivalence, since we need it precisely in order to facilitate comprehension.

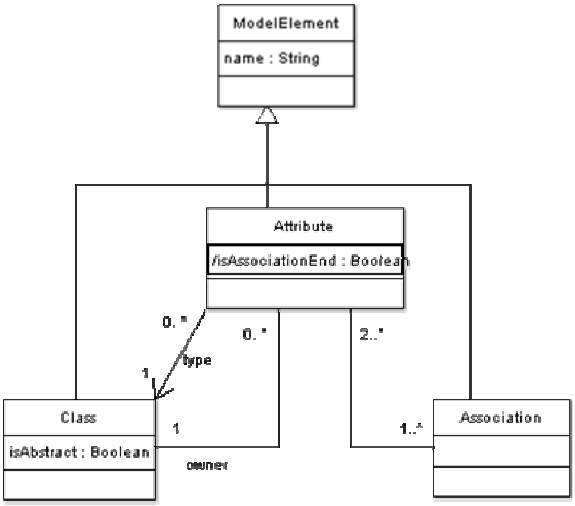


Figure 4 Second meta-model candidate

A UML model that conforms to several meta-models is precisely the situation that exists for instance in the case of two UML tools, offering export-import facilities. Inherently the two tools implement different meta-models, as the UML standard meta-model is not (necessarily) directly used by case tools. Since the two tools support different metamodels, they can manage to interchange some (more or less) simple models, but still cannot exchange any model.

Simple models would conform to both of these meta-models. In their case, the import-export is not problematic. However, for more complex models, that use parts specific to one of the tools, the import-export can be problematic, as such models are not (at least by default) conformant to both meta-models.

Based on these observations we transformed the pyramidal-like view of the models into the sand-watch like view described in Figure 5.

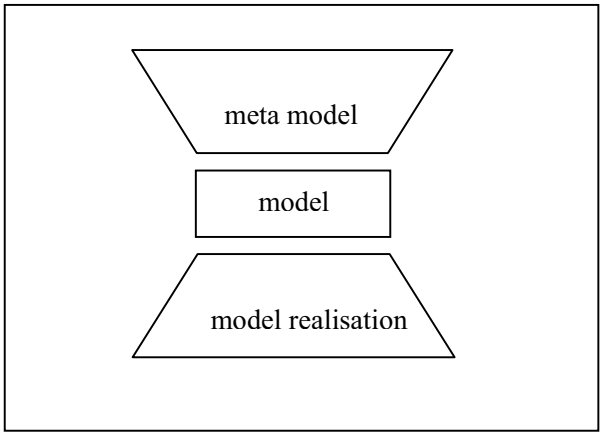


Figure 5 Sand-watch view of MDA relationships

The sand-watch view is centered on the notion of model, basically because the model is the most used concept of MDD. Each model may have a set of (run-time) realizations that conform to it. Moreover, each model conforms to *a set of* meta-models.

The key difference between the two visions is of course that we explicitly admit that a *model conforms to a set of meta-models*. While this is not forbidden by the current view, in which one meta-model has a set of models conformant to it, this is not directly allowed either.

Before discussing how we used our one model conforms to several meta-models vision in teaching MDA principles, we quickly asses the advantages and drawbacks of this sand-watch shaped vision.

The main advantage that we see is that it reflects better the situation existing in the real world. It also naturally raises the issue of compatibility between meta-models which is a research topic of its own, where work still has to be done.

Moreover, the sand-watch vision has the advantage of centering the discussion on models which are the entities the people are the most familiar with.

The only drawback that we see is that since the model is term relative to the viewpoint (e.g. the UML metamodel is a model of its own), the sand-watch view is also relative and may need to be reconsidered, according to the abstraction level that we are focusing on.

We should highlight that we prefer this view not just because of the state of affairs in industrial use of MDA (lack of tool interoperability, variety of UML meta-models, etc) but rather because we feel it reflects better the MDA philosophy. Remember that a few years ago it seemed obvious THE relationship between a model and a meta-model is an *is instance of* relationship. Today we are more flexible about the nature of this relationship and we admit the existence

of a generic *conforms to* relationship between models and meta-models, which can be (but it is not necessarily) an *is instance of* relationship. In the same spirit, we think that the sand-watch view is a generalization of the classical pyramidal view, which removes the unneeded constraint of the uniqueness of the meta-model a model conforms to.

4. Teaching MDA in the spirit of sand watches

During the last two academic years, in the course that teaches the basics of MDA, by presenting its concepts and their mutual relationships, we used the sand-watch vision described in the previous section.

We used this vision in the graduate course *"Modeling and meta modeling"*. This course addresses to students familiar with modeling and with (often strong) practical experience in modeling.

The results were very positive, as students already familiar with MDA concepts and having some industrial experience in modeling, accepted much easier the overall picture and without having the impression that the course is presenting an unrealistic world.

Moreover, and this is the point that made us more confident in this approach, we noticed a much more mature feed-back from students. Spontaneously, students raised issues related to model transformation [3] and model equivalence.

The next topic of the course, that addresses transformation related issues, arrived naturally as an obvious continuation of the MDA concepts introduction, and not as yet another topic within the MDA course.

As the basic information was much easier to transmit and accept, the course was more consistent and we managed to tackle more issues, in particular related to model transformation.

5. Conclusions

Model driven development is a relatively new technological field. Although promising, the technology integration into the industrial practice is slowed down by the lack of properly trained staff [5].

One of the main missions of the academia is to prepare students for the industrial world as it will be tomorrow. The chance of our students is that they do not have to follow the way the technology evolved, and can directly get familiar with a way of thinking that more and more evolves towards abstracting away details, thinking globally, and concentrating to the real problems not to technological small problems (at least not at a wrong time).

In this context, we have to look for the best strategies to present concepts inherently complex. Our aim should be then not only to find the most accessible representations of the concept, but also those that are closest to the realities of the "real life". We make the case that using an alternative representation of the abstraction level hierarchy would facilitate comprehension and the later accommodation to the realities of the industrial best practice.

We had the opportunity to compare the two teaching approaches on similar student populations and the results were very encouraging. Indeed, not only we managed to introduce the MDA specific concepts more naturally, but also the closeness with the situations encountered in practice made our point more trustworthy and raised more valid and more constructive issues. All of these allowed us to have more dense presentations with much better results as shown by the examinations results and feed-backs of their future teams, which we start to get.

6. References

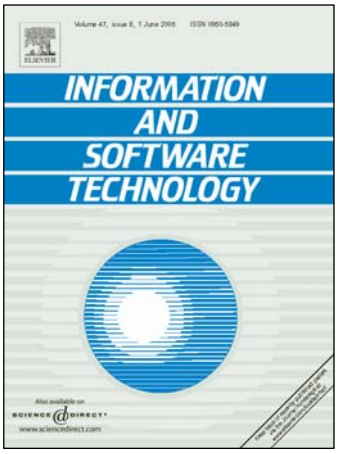
[1] G. Engels, J. H. Hausmann, M. Lohmann, S. Sauer, "Teaching UML Is Teaching Software Engineering Is Teaching Abstraction". *MoDELS Satellite Events 2005*: LNCS 3844, 306-319, 2005

[2] Jean-Marie Favre. *Foundations of Model (Driven) (Reverse) Engineering : Models - Episode I: Stories of The Fidus Papyrus and of The Solarus*. In Jean Bezivin and Reiko Heckel, (eds), *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl, Germany, 2005.

[3] Tom Mens, Krzysztof Czarnecki, Pieter Van Gorp. *A Taxonomy of Model Transformations*. In Jean Bezivin and Reiko Heckel, (eds), *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl, Germany, 2005.

[4] I. Ober, A. Prinz, "What do we need metamodels for?", *4th Nordic Workshop on UML and Software Modelling*, GRIMSTAD, NORWAY, p. 8-28, 2006

[5] Stanley J. Sewall, *Executive Justification for Adopting Model Driven Architecture (MDA)*, OMG Presentation, November 2003, available at http://www.omg.org/mda/mda_files /11-03_Sewall_MDA_paper.pdf



Information and Software Technology

Aims and Scope

Information and Software Technology is the international archival journal focusing on research and experience that contributes to the improvement of software development practices. It covers methods and techniques to more effectively and efficiently engineer and manage software.

Examples of areas covered by the journal include:

- empirical and experimental analyses
- software economics
- project management
- software metrics
- quality management, assurance and control
- software processes and development methods
- requirements engineering
- specification and design
- software architecture and modeling
- components, frameworks and product-lines
- testing and program analysis
- maintenance, reverse engineering and evolution
- configuration management and coordination
- large-scale distributed development

Journal Homepage

The **Information and Software Technology** homepage contains all the information you need on the editorial board, submitting your paper, accessing the editorial management system, accessing articles on ScienceDirect, subscribing to most downloaded article alerts and how to download a free sample copy.

www.elsevier.com/locate/infsof

Co-Editors

M. Shepperd

Brunel University, School of IS, Computing and Maths, Uxbridge, UB8 3PH, UK

Email: martin.shepperd@brunel.ac.uk

C. Wohlin

School of Engineering, Blekinge Institute of Technology, Ronneby, Sweden

Email: claes.wohlin@bth.se

S. Elbaum

Department of Computer Science and Engineering University of Nebraska-Lincoln, Lincoln, USA

Email: elbaum@cse.unl.edu

Submit your paper online

Visit:
http://www.editorialmanager.com/infsof

Most Downloaded Articles on ScienceDirect

Critical success factors for a customer relationship management strategy
Mendoza L.E.; Marius A.; Perez M.; Griman A.C. 49 (8) 01-Aug-07

Group cohesion in organizational innovation: An empirical examination of ERP implementation
Wang E.T.G.; Ying T.C.; Jiang J.J.; Klein G. 48 (4) 01-Apr-06

Data flow analysis and testing of JSP-based Web applications
Liu C.H. 48 (12) 01-Dec-06

PageGen: an effective scheme for dynamic generation of web pages
Al-Darwish N. 45 (10) 01-Jul-03

Software maintenance seen as a knowledge management issue
Anquetil N.; de Oliveira K.M.; de Sousa K.D.; Batista Dias M.G. 49 (5) 01-May-07

An analysis of web services support for dynamic business process outsourcing
Grefen P.; Ludwig H.; Dan A.; Angelov S. 48 (11) 01-Nov-06

An analysis of the most cited articles in software engineering journals – 2000
Wohlin C. 49 (1) 01-Jan-07

Managing the business of software product line: An empirical investigation of key business factors
Ahmed F.; Capretz L.F. 49 (2) 01-Feb-07

Testing Web-based applications: The state of the art and future trends
Di Lucca G.A.; Fasolino A.R. 48 (12) 01-Dec-06

Trust in software outsourcing relationships: An empirical investigation of Indian software companies
Oza N.V.; Hall T.; Rainer A.; Grey S. 48 (5) 01-May-06

Autonomic resource provisioning for software business processes
Pautasso C.; Heinis T.; Alonso G. 49 (1) 01-Jan-07

Evaluation of object-oriented design patterns in game development
Ampatzoglou A.; Chatzigeorgiou A. 49 (5) 01-May-07

This list of most downloaded articles is based on an analysis of usage statistics on ScienceDirect from August 2006 – July 2007

Benefits of Publishing with Elsevier

- Renowned editorial board

All editors are leading researchers in the field. View the full editorial board on the journal homepage.
www.elsevier.com/locate/infsof

- Easy online submission

Submit your paper online to *Information and Software Technology*.
The editorial process is performed electronically, which shortens the refereeing time. Submissions are free of charge.
http://www.editorialmanager.com/infsof

- More services for authors: proofs, online tracking, free offprints

It is easy to check the progress of your paper online through the editorial management system. The corresponding author, at no cost, will be provided with a PDF file of the article via e-mail or, alternatively, 25 free paper offprints.

- Wide online visibility

When your article is published in *Information and Software Technology* it will appear on Science Direct, the world's largest full-text database, reaching over 16 million scientists worldwide.
www.sciencedirect.com

- Online visibility within 3 weeks

Within 3 weeks after acceptance your article will be online on ScienceDirect and accessible by your peers. The article is immediately linkable and citable using the article's Digital Object Identifier.
www.sciencedirect.com

- Free online colour figures

If you submit usable colour figures then Elsevier will ensure, at no additional charge, that these figures will appear in colour on the web regardless of whether these illustrations are reproduced in colour in the English version.

- Liberal copyright policy

Elsevier's copyright statements allow for posting on pre- and post-print servers, as well as personal homepages and institutional repositories, provided a link to the official version on ScienceDirect is also indicated. Please see
http://www.elsevier.com/wps/find/supportfaq.cws_home/copyright for a full overview of Elsevier's copyright policy.

- 30% discount on all Elsevier books

If your work is published in an Elsevier journal you will be entitled to a life-long 30% author discount on all Elsevier, and associated imprints including Academic Press and Morgan Kauffman.
www.books.elsevier.com/authors

Department of Applied IT