Tool Use in Software Modelling Education (invited paper)

Tool use in software modelling education

Seiko Akayama[1], Birgit Demuth[3], Timothy C. Lethbridge[4], Marion Scholz[2], Perdita Stevens[5], and Dave R. Stikkolorum[6]

[1] Kyushu University,
Japan
[2] Vienna University of Technology,
Austria
[3] Technische Universität Dresden,
Germany
[4] University of Ottawa
Canada
[5] University of Edinburgh
Scotland
[6] Leiden University
The Netherlands

Abstract. An important decision that must be taken by anyone designing a course involving (object oriented software) modelling is what tool support, if any, to use. Options include picking an industrial strength modelling tool, using a tool specifically designed for educational use, or eschewing tool use altogether in favour of pencil and paper. The best answer will depend on many factors, including the prior experience of the students (and staff), the length and organisation of the course, and the learning objectives. Moreover, decisions on tools have an impact on other aspects of course design. In this informal paper, the result of discussion at the MODELS Educators' Symposium 2013, we survey previous work on this question, discuss our own experience, and draw out some key issues that someone designing a new course involving modelling must consider.

1 Introduction

Teaching object oriented design and modelling in a university is important not only because these are important skills that students will need if they pursue careers in software development, but also because this area is a key interface between research and teaching. This double motivation – we might say, vocational and intellectual – for teaching design and modelling is itself a source of challenge for educators. We experience a tension between the desire to train students in the skills they will need after university, and the desire to have them reflect on the nature of design and modelling and the ways in which these activities could be improved with the help of cutting edge research.

One of the decisions that someone (re)designing a course must take is what use, if any, to make of tools that support modelling. This has often been a topic

of conversation among participants at events like the Educators' Symposium at the MODELS conference (EduSymp), and many papers have been written about individual experiences. This paper aims to draw the strands together and discuss the whys of these decisions in a way which will be thought-provoking for people newly planning, or revising, courses. It will do this in two ways: first, by briefly surveying the literature on the topic and providing a collection of references into further reading; second, by discussing our own experiences, decisions, and opinions.

The authors have made very different choices in their own teaching, stemming partly from differences in their environments and partly from differences in their opinions. This paper, accordingly, does not present a complete consensus, far less a recommendation for "How to run a modelling course". Rather, it aims to make explicit some of the factors that need to be considered and the arguments for and against different approaches.

2 Related work

Much has been written about how to educate people in object oriented design (i.e. [7], [3]). Less has been written about education in modelling specifically [15], and still less about the role of tools in that education; typically, tools are mentioned in papers whose focus is elsewhere. The MODELS conference has a long-standing tradition of including an Educators' Symposium in its organisation, and many of the papers we shall mention appeared there.

In this section we focus specifically on the use of modelling tools in education. We shall not attempt to cover related work on other aspects of teaching modelling, nor shall we discuss comparisons of modelling tools that are not education-specific.

The use of tools in modelling education emerged as a widespread concern in the late 1990s/early 2000s, around the same time that UML was being standardised for the first time.

Over the years, some concerns have changed or disappeared. Stevens' 1998 paper [16], for example, mentions as a source of student dissatisfaction the difficulty of running Rational Rose on workstations with 32Mb of RAM! Our experience today is that for models of the sizes typically used in teaching, commodity machines usually run available tools without difficulty.

A major concern that has not disappeared – and is also often mentioned by industrial practitioners [13] – is that UML is a very complex language. A typical modelling course covers only a small subset.

The 2002 short paper [4] describes a tool called QuickUML[1] developed specifically for modelling education at SUNY at Buffalo. Citing students' tendency to use UML features they did not know how to use correctly, the authors explain that their tool supported class diagrams only, with a limited set of features and some skeleton code generation. The latter was motivated by its usefulness in

[1] http://quickuml.softpedia.com/

helping students to get "a feel for how a class diagram translates into real code"; that is, its aim was more to help them understand the model than to contribute to rapid development of real systems. While several tools called QuickUML have web pages, none of them advertise any special educational use and it is not clear that any are related to the tool described in this paper.

The authors of [19], writing in 2005 and reporting on their tool minimUML[2], take a similar approach, implementing a very limited subset of UML to help students get started. They emphasise the need for tools to provide full undo and redo, in order to help students experiment freely. minimUML itself does not seem to be available for download now. Their paper provides a useful comparison of UML tools in use in education at the time, including QuickUML. The others they considered were Violet[3], UMLet[4], Dia[5] and ArgoUML[6]; see Table 1 of [19] and surrounding discussion for more information about their comparison at that time. All are still available today. UMLet, which Stevens has used in the past for teaching purposes, deliberately limits functionality and provides a simple interface. Dia is a general-purpose drawing tool that offers UML; it similarly gives limited functionality and aims for easy learnability. ArgoUML, by contrast, aimed to be a full-featured modelling tool. Stevens has used it too in the past, before switching to UMLet because of the difficulty students experienced learning to use the tool and getting round minor bugs and missing features.

In 2007, [14] described a tool called StudentUML. Neither it nor its apparently planned successor seem to be available today. Like most of its predecessors it focused on a small subset of UML. It aimed to support a student in maintaining correctness and consistency of a set of diagrams. The paper compares it with QuickUML, minimUML, UMLet and one we have not previously mentioned, Ideogramic UML[7][8]. This last was based on gesture recognition to transform rough gestures into formal UML diagrams and thus enable interactive learning.

At EduSymp 2009, Moisan and Rigault reported[12] on experience teaching UML to different audiences, and complained in particular about their experiences with tools (in general: they had not prescribed any one tool to their students). Much of what they say is not specific to education, but they do discuss the choice of what kind of use of UML to teach and how this is affected by tool use; for example, they comment that the tools their students used made it difficult to refactor designs.

In 2011 at CSEE&T, Lethbridge et al presented the use of Umple[8] as a modelling tool [11] in various classes. Umple was shown to improve the grades students achieved, and helped them produce better models. Umple allows mod-

[2] http://minimuml.cs.vt.edu/
[3] http://violet.sourceforge.net
[4] http://www.umlet.com/
[5] https://wiki.gnome.org/Dia
[6] http://argouml.tigris.org/
[7] http://freecode.com/projects/ideogramicuml
[8] http://cruise.eecs.uottawa.ca/umple/

elling both textually and visually, supporting UML class diagrams and state machines. Umple was designed for the educational context and enables students to build complete systems starting with a model and add code to the model to create a complete system. It is lightweight, and can be used on the web, in Eclipse or on the command line. Students can use Umple with C++, Java, PhP and various other languages; they can edit UML diagrams or the textual form of Umple. Umple can also generate yUML, mentioned below.

Also writing in 2011, Kuzniarz and Börstler considered difficulties in the teaching of modelling [9], and commented, like others before them, that using a UML tool designed for professionals can be problematic: "Those tools rarely support easy mechanisms for consistency checking and most of them have a very steep learning curve." They mentioned two tools that had been specifically developed with the needs of education in mind: StudentUML[14], which aimed to help students understand consistency between class and sequence diagrams, by enforcing it during development, and Violet, both mentioned above.

In 2012 the same authors (Kuzniarz and Börstler) reported [2] on an analysis of computing curricula, and on a survey of people involved in teaching modelling that gathered information on 12 modelling courses (81 courses in total). The focus of the paper is not specifically on tools and it says little about them. On the one hand they reported respondents' feelings that better tools for teaching modelling were needed, and on the other, tools appeared in a short list of topics that respondents considered least important.

Stikkolorum et al demonstrated their game for learning software design 'The Art of Software design' [17] which was actually a UML tool-like application with a small subset of UML. In [18] they suggest exploring using such a tool to train students' abstraction skills.

Coming right up to date, relevant work in MODELS 2013 included [1] which argued that modelling, indeed model-driven engineering, should be taught from a relational database point of view. The authors describe their move away from Eclipse MDE tools, which they had found too heavyweight in earlier teaching, towards a home-made framework they call MDELite based on Violet (mentioned above) and two other lightweight UML tools not intended specifically for teaching, UMLFactory[9] and yUML[10].

3 Our experience

In this section we present our own experience in teaching courses that focus on object oriented design and modelling, before drawing out common themes and differences that relate to tool use.

Table 1 summarises the key information about the courses, which are then described in separate subsections named after their authors below.

[9] http://umlfactory.com/
[10] http://yuml.me/

See section	University	Year (UG/PG)	Approx. no. students	Expected work (hrs)	Nature of course	Tool
3.2	TU Dresden, Germany	UG1	350	180	Intro to Software Engineering, OO modelling (analysis and design) in UML, OO programming with Java and design patterns.	Java (BlueJ recommended), Praktomat (e-learning system), modelling with paper and pencil.
3.2	TU Dresden, Germany	UG2	250	180	Software development in a work-sharing project (from analysis to maintenance).	Magic Draw UML, Java (Eclipse recommended), application domain framework (SalesPoint), further development tools (e.g. SVN) .
3.3	U. Ottawa, Canada	UG2	125	240	Introduction to the full spectrum of software engineering with an emphasis on agility, model-driven engineering and design patterns.	Umple, Java, Eclipse.
3.4	U. Edinburgh, Scotland	UG3	50	100	Intro to both modelling in UML and OO design, assuming Java and basic SE knowledge.	None mandated.
3.5	U. for Applied Sciences, The Hague, The Netherlands	UG1	50 30	100 80	Basics of UML Modelling. Requirements Engineering.	Visual Paradigm 10.
3.5	Chalmers and Gothenburg Univ., Sweden	UG2	80	120	Model driven design .	IBM Rational Rhapsody.
3.5	Leiden University	UG2	25	170	Software Engineering	Students' choice: Star UML.
3.6	Kyushu Technical Education College, Japan	UG1	20	147	OO modelling for embedded system.	BridgePoint.
3.6	Tokuyama College of Technology, Japan	UG2,UG3	6	8	OO modelling for embedded system.	clooca.

Table 1. Summary of courses currently taught by the authors

3.1 Marion Scholz

At the Vienna University of Technology, we offer two courses on the topic of modelling, Introduction to Object-Oriented Modelling (OOM) and Model Engineering (ME).

OOM OOM is designed for students in their second semester who have already learned a little bit of programming and object-oriented concepts. The course is awarded 3.0 ECTS[11] points which implies an overall student effort of 75 working hours. Each year, about 400 students attend the course. In OOM we teach modelling basics by introducing syntax and semantics of the following UML 2 diagrams: class and object diagram, sequence diagram, state diagram, activity diagram, and use case diagram. The course consists of lecture videos, demo sessions, a practical part and three exams. In the lecture videos we explain the syntax and semantics of the considered UML 2 diagrams. In the corresponding demo sessions practical examples for each taught diagram type are demonstrated and questions are answered. For the practical part the students are divided into groups of about 30-35 people. Each group meets six times during the semester for so-called "lab sessions" in order to discuss solutions to exercises. Three exams assess whether the students have reached the learning goals, i.e., if they understand the theory and are able to apply it to small real world problems. Additionally, we offer various e-learning exercises in the form of multiple-choice questions the students can voluntarily use to test their knowledge.

Because 75 hours is such a short time, we do not use any tools for modelling, so as to ensure that the students concentrate on the syntax and semantics of UML instead of putting a lot of effort into understanding a tool.

Advanced Model Engineering In addition to OOM, we offer an advanced Model Engineering (ME) course consisting of a lecture and a lab part. The course is obligatory for masters students in the business informatics curriculum and optional for masters students of the computer science curriculum. Overall, about 150 students attend ME each term. More than half of the students are in the computer science curriculum.

The goal of this course is the elaboration of basic concepts of model-driven software development. As the course is attended by masters students only, we presume multiple skills and experiences in object-oriented modelling, object-oriented programming, data modelling, and data engineering, as well as in software engineering. The lecture and the lab are credited with 6.0 ECTS points, which allows us to expect the students to spend about 300 hours of work for this course.

As indicated by its name, during the lecture the different concepts, tools, and practical approaches from the field of model engineering are examined including meta-modelling, model transformation, code generation, and concrete syntax specifications (textual and graphical).

[11] ECTS: European Credit Transfer System. One credit stands for 28 study hours.

In the accompanying lab the students are given practical assignments chosen from the topics of this lecture. The final output of the lab part is a model-driven development environment comprising self-designed metamodels, model transformations and code generation facilities for producing running applications for a particular domain. We emphasise that the students gain practical experience with state-of-the-art MDE frameworks. The frameworks used are EMF for metamodel development, Xtext for developing textual concrete syntaxes, ATL for defining model transformations, and Xtend for developing code generators. To overcome tooling issues, we provide a dedicated Eclipse bundle comprising all necessary plug-ins, as well as tutorials, videos, and forum support for explaining how to use the different plug-ins.

3.2 Birgit Demuth

At the TU Dresden, undergraduate students have to pass two consecutive obligatory software engineering courses:

- Software Engineering (introductory course)
- Software Project Course

The big challenge for both courses is their large-scale character with a high number of students.

Software Engineering Course: At the beginning we expect more than 450 students, however we meet "only" up to 350 students in lectures and in the exercise course. At the end of the course around 250 students take part in a written exam.

We assume basic skills and first experience in procedural programming, especially using the C programming language.

The course is offered for students in their second semester which is 15 weeks long. In each week students attend one lecture (90 minutes) and one exercise (tutorial, 90 minutes). The total effort is assumed as 180 hours including the preparation of the exam.

Basically our teaching approach includes modelling with UML and programming with Java. Students are introduced to object-oriented analysis (OOA) and object-oriented design (OOD) including using selected design patterns as well as to object-oriented programming (OOP) including UML2Java transformation. The main topics in the course are the following:

- Software engineering overview and software development processes
- OO paradigm (thinking in objects/CRC card method, OO programming fundamentals)
- OOA (requirements specification, static and dynamic modelling with UML)
- OOD (software architecture, refined static modelling with UML)
- OOP (implementation of UML models with Java focusing on data structures (Java Collection framework) and on Generics.

- Reuse in OOD (design patterns and frameworks versus class libraries, testing with JUnit)
- Overview to architecture of interactive systems and graphical user interfaces
- Introduction to project planning, preparing the students for the following software project

In teaching UML, we consider use case, class and object, state chart, sequence and activity diagrams.

In our earlier didactic approach we structured lectures and exercises along the lines of the "Waterfall approach". Though we started after an overview of the software engineering discipline with an introduction to the idea of object-orientation, we continued with modelling (OOA and OOD) before we taught OOP with Java and reuse in OOP. It should be noted that we understand the Waterfall approach not as development methodology but as teaching methodology for understanding the software life cycle. Because we observed that students do not really understand modelling until they have practised object-oriented programming we decided to change the didactic approach by reordering the main teaching topics. That way we now introduce the students to OOP with Java at the beginning of the course and motivate them to solve programming tasks of increasing complexity in parallel to the modelling topics of the course. Beginning with teaching the OO basics in Java we already visualise Java code by UML class, object and sequence diagrams. We call this teaching philosophy the "UMLbyExample approach". Programming/modelling examples range from an object-oriented Hello World program (HelloLibrary example [5]) to 3-page Java programs that contain implemented design patterns. The widely used introductory Java program "Hello World" is considered harmful because it communicates nothing about object-oriented thinking. Using the HelloLibrary example we explain the concept of classes, objects, encapsulation and messaging as well as show their representation in Java and UML. Besides first steps in object-oriented programming, we practice the CRC card method to learn object-oriented thinking. Just for inexperienced programmers, this method is a way to teach them the concepts of objects outside of computers. During the OOP lectures and exercises students need only "read" UML diagrams as graphical representations of the semantics of Java classes. So they learn from scratch to "see" OO programs as models.

We recommend Java beginners to work with BlueJ before they later switch to a professional IDE such as Eclipse. Eclipse is used only for programming, not for modelling, but the BlueJ[12] tool shows a first UML sketch of the Java code automatically.

Beyond that, we offer students the possibility to test their Java exercise programs by using an e-learning environment (Praktomat[13]) that we customised and installed for our students. We start modelling with paper and pencil, and on the blackboard. Later we recommend advanced students to work with the

[12] http://bluej.org/
[13] https://praktomat.inf.tu-dresden.de/

powerful UML tool MagicDraw UML[14]. MagicDraw supports several modelling languages (i.e. UML, BPMN, SysML, etc.), can be used for metamodelling and has features such as UML profiles and OCL support.

Basically we expect that our university students are able to familiarise themselves with programming and modelling tools. In lectures we rather focus on teaching concepts than practical skills.

As explained above, we changed our teaching methodology to the UML-by-Example approach, with which we achieved better results in students' understanding of modelling and programming.

A UML tool which provides a more comfortable support for the configuration of modelling functionality (OOA, OOD, OOP, UML subsetting) would be helpful.

The large-scale software engineering course is supervised by one lecturer and requires the assignment of student tutors who attend to student exercise groups of up to 30 students in each group. The lecturer is in charge, supervising the student tutors.

Software project course All students who passed the introductory software engineering course have also to graduate the software project course. Due to the relatively low success rate of the students in the introductory course, typically around 200 students every year (third semester) have to be managed in their software projects.

Basically we only admit students to this course if they have demonstrated, in the introductory course, that they have basic skills in modelling and programming [6].

The student effort is the same as in the introductory course, that is, 180 hours. However, the effort is more concentrated: 15 hours per week distributed over 12 weeks, with a hard deadline for the student software projects.

In the project course students have to implement a middle size application in a work-sharing software development process. While students practice modelling-in-the-small in the introductory course they have to work with models throughout the waterfall-driven development life-cycle in a software project, starting with a textual requirement specification and finishing with a presentation of the tested and deployed application (programming-in-the large and modelling-in-the-large). A further challenge in the students' projects is the reuse of a domain-specific Java framework.

Students need to develop a lot of complex skills. These include technical skills in object-oriented software development, but also social skills, especially how to collaborate with other developers as part of a team working towards a large and complex software system. To acquire these skills, students need hands-on development experiences; we believe these are best delivered through a team-oriented project course. This course must be both sufficiently challenging and achievable within the limited time available. In our special situation (large numbers of students supervised by small numbers of tutors) an important further

[14] http://www.nomagic.com/products/magicdraw.html

requirement is scalability: different projects should be easily comparable while allowing for different tasks for different teams to reduce the risk of plagiarism. The solution that in our experience satisfies all these requirements is to use an application framework for an everyday application domain. We decided to choose sale (web) applications for such a domain.

We have developed SALESPOINT[15] [20], a Java-based framework that underlies most of our project courses in Dresden and at the Universität der Bundeswehr in Munich. SALESPOINT is helpful for teachers to create many different tasks (sales applications) for large-scale courses. The educational background in detail is explained in [20]. Besides this application framework students use state-of-art software development tools of their own choice such as Eclipse, SVN, MagicDraw UML or another UML tool.

SALESPOINT has been used and maintained successfully since 1997. We believe it to be applicable for computer-science undergraduate project courses in a variety of educational contexts.

In an empirical study we evaluated the software project course in 2012 by metrics and qualitative data in detail and reported the results in [6]. We detected significant differences between students of different qualification in their basic skills and their interest in software development. However, we could also show that our teaching approach leads in most cases to a good program quality. An important lesson learned in earlier years was that a hard deadline in a student project course significantly helps project teams to finish their projects successfully.

Comprehensible and well-structured tutorials and documentation of the tools used are an important issue for their acceptance by the students in their development work.

As in the case of our large-scale software engineering course, and due to the limited resources at the university, the project course requires the assignment of student tutors who attend to the project teams of five or six students per group. The challenge for a student tutor is that he has to fulfil two roles: he must be software consultant for the student team as well as customer for the project task. The responsible lecturer is in charge to supervise the student tutors in a close-mesh and intensive way.

3.3 Tim Lethbridge

Course SEG2105 at the University of Ottawa This is a second-year course taught to all majors including as a required course in Computer Science, Software Engineering and Computer Engineering, and as part of a minor taken by students in many other programs of study in engineering, business, sciences and arts. 125 students took course in the autumn of 2013.

Upon entry into the course, students will have normally taken two programming courses with Java as main language, and have received a very basic introduction to UML class diagrams.

[15] http://www.salespoint-framework.org/

The course involves a total of 120 hours of student work, spread over 15 weeks. Students who complete this course may then take more specialized second-year and third-year software engineering courses such as Software Construction, Software Requirements, Software Design, User Interface Design, and Quality Assurance. SEG2105 provides basic coverage of each of these areas. Students in the Software Engineering degree program then take a Project Management course and a full-year capstone course in their fourth year.

Upon completion of SEG2105, the student is expected to be able to:

- Understand the basics of software engineering including requirements gathering, specification, model-driven development, testing and agile processes.
- Design programs using imperative and object-oriented concepts
- Implement designs correctly and rapidly, with increased confidence
- Apply a various design patterns, frameworks and architectures in the design of software
- Use UML effectively
- Conduct performance analysis experimentally
- Use tools for model-driven development, including for analysis of models and generation of code
- Work more effectively on software development both individually and in groups

The educational process is 'hybrid' in that it uses a wide variety of modes:

- Traditional lectures, incorporating asking students oral questions frequently
- Having students watch videos of lectures or demonstrations
- Open ended design projects on the board, led by the professor
- Live use of the MDE tool Umple in the classroom
- Demonstrations of aspects of a live software project (Umple) whose MDE and testing artefacts are all live on the web.
- Having students answer 'clicker' questions frequently to assess their progress and encourage concentration
- Programmed labs where the students follow a series of steps to modify software and solve a software problem
- Design and programming homework
- An open-ended project

The latest best practices are emphasised. Real software projects are demonstrated live. Students are given existing models and software to modify, rather than starting from scratch. A key aim is to convince students of the value of the best practices, so they deeply appreciate them, rather than merely "telling them things".

We use Eclipse as an IDE, but also encourage use of tools on the command line and web-based tools. Umple is used as a tool for drawing UML diagrams, writing models textually, analysing models and generating code. The programming language is Java. Students are encouraged to explore and choose whatever other tools they feel like.

The course has been fine-tuned after 22 years of teaching. In the early days, modelling was taught following examples in various textbooks. Typical mistakes of students were catalogued, leading to development of a curriculum that helped students avoid such mistakes. I wrote a textbook [10] in 2001 for the course that incorporated answers to all the many questions students had posed over the years, and guidance to help readers avoid all the misconceptions and wrong answers I had noticed.

I long ago gave up on commercial modelling tools because they either have too big a footprint, are too complex to learn, don't have the analytic features I would like, or don't generate proper code.

Starting in 2006 we developed Umple to provide exactly the features I believe would most help students learning to model. Over successive years, as Umple was enhanced, a series of new features was made available to students.

I introduce modelling using Umple stealthily, in the very first lecture, not initially emphasizing to students that the tool I am using is called Umple. I start by drawing class diagrams in Umple's web tool, write a main method to instantiate and manipulate the objects, and then compile and execute the resulting system. Students see this just as an extension of the programming they are used to, but at the same time they learn modelling using class diagrams. As the course progresses and students learn to create more complex class diagrams and state diagrams, they always experience the practicality of their models through Umple. They are given feedback by Umple about many aspects of model correctness, but the ultimate test is using the model to create a program that gives the correct result.

In addition to code and diagrams, Umple can generate outputs such as SQL, metrics and (in its latest version) the Alloy formal language. These are used to teach students how their models can be the centre of software engineering. And since Umple is written in itself, students come to appreciate that significant systems can be written in a model-driven manner.

All aspects of the course are assessed in one form or another. The following are the assessments:

- Exam questions (multiple choice and modelling); 2 exams: 60% total
- Lab reports (in groups of two, involving IDE, Java and Umple use): 15%
- Modelling assignments (involving tool use, but they can choose the tool): 10%
- Participation (in labs, and answering questions in class; includes use of tools in certain labs that don't require reports): 5%
- A project in groups of two, including modelling (again they choose the tool) and a live demo: 10%

There are four teaching assistants, each in charge of about 30 students. They run lab sessions and do assignment marking. This semester, two of the teaching assistants are developers of the tools.

3.4 Perdita Stevens

Software Engineering with Objects and Components (at University of Edinburgh) Typically 40-50 students take this course, which is principally a third-year undergraduate course (so students typically aged 19-20) though a few MSc students also take the course and we have a trickle of visiting students.

Prerequisites are: experience programming in a typed object-oriented language (Java, for our own students); the content of an introductory software engineering course which discusses software process engineering overall, introduces very basic UML, and gives students practice building and contributing to a sizeable open-source project.

This is a 10 point course, meaning a nominal 100 hours of student effort over one semester. There are 15 or so lectures, spread over 10 weeks.

The intended Learning Outcomes are that students will be able to: "

1. Design simple object-oriented systems, making appropriate use of available components;
2. Design simple software components, making sensible API decisions;
3. Evaluate and evolve object-oriented software designs, making use of common design patterns if appropriate;
4. Create, read and modify UML diagrams documenting designs;
5. Discuss the use of modelling in software development, e.g. why and how models of software can have varying degrees of formality.

"[16]

The first part of the course is focused on the more technical modelling aspects, teaching UML, though emphasising in class discussion pragmatic issues, especially the benefit and cost in different circumstances of modelling. The second part encourages students to consider quality in design, discusses design patterns, etc. Combining the two strands, the course ends by discussing model-driven development (history, state of the art, future).

The delivery of the course is by a mixture of traditional lectures, in-class group exercises, discussion, on-line videos and self-assessment, and required reading. We aim to put the delivery of factual information into self-study material wherever possible, reserving as much class time for interactive work as possible. There are weekly tutorials (two groups of 10 taken by PhD students, plus a drop-in taken by me). There is no other teaching assistance.

Students do not have to use a modelling tool in the course – modelling is typically done on paper/whiteboard. This works well, compared to similar courses here in the past, in which students (especially the weaker or less experienced ones) have tended to pour huge amounts of effort into learning a tool, to the detriment of the learning objectives. Students who are interested, though, are encouraged to try out tools of their choice, and the course discusses tool use. Encouraging students to try out tools of their choice leads to the class as a whole having a variety of experience, which leads to interesting discussions. We also

[16] http://www.drps.ed.ac.uk/13-14/dpt/cxinfr09016.htm

observe that students who have chosen to try a tool for themselves are able to analyse any deficiencies in it or difficulties with it fairly objectively: by contrast, in courses that prescribe a tool, I have observed that some students blame the course team for any problems, assume that we must have made a bad choice of tool, and imagine that such problems would not arise in industry.

Summative assessment is entirely by exam, principally to encourage collaboration without fear of plagiarism in the formative exercises during the course. The exam is done on paper. Performance in the exam last year was good – almost all students were able to demonstrate competence with modelling, and many also gave reasonable and thoughtful responses to questions aiming to elicit thought about design quality. Feedback from students was also good; they especially liked the "flipped" nature of the course.

3.5 Dave Stikkolorum

UML Modelling (at The Hague University of Applied Sciences) UML modelling is a required first-year course that introduces all the basic diagrams of UML. About 50 students participate in this course. No prior knowledge is assumed, but in practice, they already have taken C programming before the UML course.

The workload of the course is 4 ECTS credits. There are 16 hours of lectures and 16 hours of practicals spread over 10 weeks.

The course covers the basics of UML, i.e. use case diagrams, class diagrams, sequence diagrams, activity diagrams, state machine diagrams. After this course the student should be able to:

- know the basic UML diagrams and its notation
- translate coherent concepts to a diagram
- translate a domain description to a domain model
- create a coherent model, i.e. the different diagrams are consistent with each other.

In the UML modelling course the case of a 'goal keeper' game is used. During the lectures the different UML diagrams will be introduced to the students. Students will discuss possible solutions with each other after they drawn their solutions on the white-board. In parallel they practice their skills in modelling another case in the individual practical assignments with the modelling tool.

The tool we use for the practical assignments is Visual Paradigm for UML[17] (version 10 at the moment of writing).

Assessment is by written exam and practical assignments (with use of the tool).

Requirements Engineering (at The Hague University of Applied Sciences) Requirements Engineering is a required second-year course which is followed by 30 students. Prior knowledge of UML modelling is assumed.

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

The workload of the course is three ECTS credits. We again offer 16 lectures and 16 practical sessions spread over 10 weeks.

We cover problem analysis, requirements (elicitation, specification, validation, modelling as-is systems with UML, modelling to-be systems (concepts) with UML.

In the requirements engineering course we discuss critical papers that reflect on software quality, project organisation in relation to requirements. In the practical assignments students have to improve the specification of a real life software project. UML is used for the specification in the requirements document.

Also in the practicals we use Visual Paradigm for UML.

Assessment is by written exam and practical assignments (with use of the tool).

Technical Analysis and Design (at Chalmers and Gothenburg University) Technical Analysis and Design is a first-year course (second semester). About 80 students participate in the course.

Prior knowledge of object oriented programming is assumed. The work load is 4.5 ECTS credits and spread over 10 weeks. During the course they will have two hours of theory and two hours of supervised group work per week.

In the Technical Analysis and Design course students will be presented with the different UML diagrams (Use Case, Class, Sequence and State) to be used in analysis and design. Design is discussed with the use of well known design patterns. Students will practice modelling with so called 'mini problems' with a problem based approach (short cases). There is a emphasis on the execution of models. Some of the executions are presented in the lectures.

We cover use case based requirements specification, object oriented analysis, and object oriented design.

After the course the student should be able to:

- Understand the role of object oriented artefacts in the software process.
- Analyse a software system and its environment using Object Oriented techniques.
- Design a software system using object oriented techniques.
- Abstract program code by the use of diagrams.
- Use contemporary tools for use case based analysis, object oriented modelling, program visualization, and object oriented program design.
- Implement software using model driven development.

The tool is IBM Rational Rhapsody[18].

Assessment is by written exam plus practical assignments (problem based learning, group work, use of tool).

Software Engineering (at Leiden University) Software Engineering is a second-year course and visited by approximately 25 students.

[18] http://www-03.ibm.com/software/products/en/ratirhapfami

Prior knowledge of programming and algorithms is required to follow the course. Spread over 15 weeks students will follow lectures (two hours per week) and participate in practical sessions (two hours per week). The course is rewarded with six ECTS credits.

The course aims to enable students to function as a professional in a team of Software Engineers, being able to deal with the practical problems that arise as a result of the engineering process utilising a standardised tool set. We cover the following subjects:

- Development processes
- System modelling using UML
- System architecting and design
- Quality assurance and testing
- Software project management
- Empirical research methods in Software Engineering

For the practical sessions no mandatory tool is used. Students can choose their own tool. In the last couple of years StarUML[19] was popular.

In my experience two problems are typically most frustrating for students, across all tools:

- Installation
- Complexity of the tool itself.

Visual Paradigm would be more useful to the courses if it could execute/simulate models in an attractive way (with animations etc.).

IBM Rational Rhapsody would be more useful if it were cross platform, easier to install and less complex.

StarUML seems not to be maintained any more. The latest version (5.0) dates from 2005 and supports UML 2.0.

All three are limited in the feedback they give from a didactic point of view (naming suggestions, important basic syntax). Although a model of course consists of a consistent set of diagrams, when practising, one would like to have the option to make one diagram instead of having to start a whole project.

None of the tools are aimed at students. It is good that they get familiar with tools that are being used in industry, but because of that they are mostly too complex (too much functions for the the initial goal: educate students). Maybe it would be a good thing to be able to set the tools' syntax sensitivity. Then a student could start somewhat informal; when they make progress, the tool could get more complex.

In my experience it is good that students see the effect of their models. Execution of the models can make a difference at that point. IBM Rational Rhapsody can do that. But because of the fact that installation is a problem, a license is needed and one has to use OO programming to program the behaviour, I cannot use it in other lectures.

[19] http://staruml.sourceforge.net/en/

Visual Paradigm is not too complex and offers an academic license. Students can download the cross platform software and use the license key for their personal installation.

3.6 Seiko Akayama

We have studied modelling education using Model-driven development(MDD) for novices. MDD can verify the accuracy of models and generate the source code, which allows a programmer to reduce the development time required to check the software so he or she can focus on the modelling process. Thus, modelling should be taught with MDD because it allows students to acquire modelling skills in a short period of time.

We conducted two trial courses in which we used Executable UML and a Domain-Specific Modelling (DSM) language.

The aim of these courses is to facilitate the acquisition of the minimum skill set required to create an object-oriented model in a short period of time. The educational subjects are software novices.

These courses development systems are embedded software. The aim of the main exercise is to develop an autotransport robot for a fictitious transportation company. The development objective is a vehicle robot developed using LEGO Mindstorms NXT.

Executable UML course at Kyushu Technical Education College We conducted Executable UML course for 20 first-year students. We used BridgePoint [20] as the MDD tool for Executable UML. This course has three classes:

- Programming class (21 hours): fundamentals of embedded systems and developing the auto transport system programs as pair work using C language.
- Modelling class (28 hours): UML, MDD methods, modelling techniques and developing the auto transport system (same system as programming class) as pair work using Executable UML and MDD.
- PBL(project-based learning) class (98 hours): developing software by adding new operations to the modelling class exercise as group work using Executable UML and MDD.

A modelling knowledge test was conducted after the Modelling class and the average scores for the correct answers was 75%. In PBL class, all teams managed to complete 70% of the operations. Some students stated that "It is good that I can check my model using a executable model" and "it is good to be able to try the challenge many times." Thus, they took time to address the model refinement process. We found that this educational program based on MDD gave the students the necessary experience to improve their modelling skills.

However, when students create a model using BridgePoint, they need a long time to learn the action language that is required to define the state actions. Therefore, learners find it difficult to focus on creating state machine diagrams and class diagrams.

[20] http://www.mentor.com/products/sm/model_development/bridgepoint/

DSM language course In order to define actions easily, we have developed a Domain-Specific Modelling (DSM) language for modelling education using the social DSL platform "clooca"[21].

This platform allows making class diagrams and state machine diagrams. A class diagram consists of classes and relations, while state machine diagrams consist of states (including an initial state), event transmission states, events, and actions.

We conducted the DSM language course for 6 second and third-year students in Tokuyama College of Technology. This course is 8 hours. The subjects were well versed with Java and UML. This course covered:

- How to use the MDD tool (clooca).
- MDD methods and modelling techniques.
- Developing the auto transport system (similar system to the one used in the programming class of the Executable UML course) using a DSML and MDD.

One issue was raised when using the MDD method for modelling education, i.e., the students neglected the quality of the model because they were focused on completing the functional aspects that could be evaluated with the MDD. In these course, we addressed this issue by conducting a review with the teachers. It is necessary to consider a supporting method for enhancing the quality of the model from now on.

4 Discussion

In this section we record some of the dimensions on which our courses differ; the designer of a new course will certainly need to consider each of these.

To start with, perhaps, the most basic issue for a paper with this title, and one on which our own preferred solutions differ markedly:
Should we begin to teach modelling using a modelling tool or by pencil and paper?

Our courses differ in whether they introduce students to modelling through working on paper (or whiteboards) or whether they use a pedagogically targeted tool or a mainstream tool. Since industrial practice also differs – some practitioners, particularly some of those in the agile development field, preferring not to use tools even for professional modelling – this is not necessarily a purely pedagogical issue. However, the tension that we, the authors, have found ourselves most aware of in practice is that between, on the one hand, the advantages of tool use, and on the other, the student effort needed to become productive with a tool. Some of us have found tools designed specifically for pedagogy to be helpful.

This is an issue that each course designer will need to consider. Some factors that matter are: the amount of support available to students (e.g., tutorials, timetabled labs); support available for the tool being considered, and especially,

[21] http://www.clooca.com/

the level of familiarity that staff have with it; the timescale of the course; and of course, the intended learning outcomes.

Another fundamental concern is:

How do we divide the conceptual landscape? Are we teaching design, modelling, programming or a combination?

Most courses that involve modelling assume some knowledge of programming, usually in an object oriented language, before the design or modelling course begins. In Section 3.2 we described how an exception to this pattern found, in fact, that it did not work well, and reordered its topics to introduce object oriented programming before modelling. However, several of us have found that illustrating OOP examples with UML models, and thus getting students to read such models before they write them or are formally taught a modelling language, works well. Thus, it is a point of consensus that students should know at least a little OOP before they begin to model for themselves.

Analysis, design and modelling are intertwined activities in real software engineering and conceptually they build on one another. This presents a pedagogical problem which we have solved in different ways, depending largely on local constraints. In Section 3.1 we described a short course OOM in modelling which focused on modelling rather than design, and was then followed by a more advanced course in which these skills were combined. This is one pattern. In Section 3.4 the slightly longer SEOC course, which combines both modelling and design, demonstrates a different approach; just enough modelling is taught to be able to discuss basic object oriented design issues.

This leads on to:

When students first carry out, or at least simulate, a software project that covers the process all the way from requirements to delivering software, what development process should they use? Should it be more rigid or more agile? What other considerations are there?

There is room to disagree here on several aspects. What development process is most important in the world of software that students may later enter? As a separate question, what should we, as responsible academics, be encouraging? Even if we agreed on these, we still have the question of what process will work in an academic context. This is affected by local factors, including the prior experience of the students and the closeness or otherwise of the students' projects to real ones. A high ceremony process performed on a short, simple project, for example, can give the erroneous impression that the ceremony is always useless, when in fact it is simply inappropriate for such a small project. Conversely an agile process may not work well if all the developers involve are very inexperienced. Yet another factor is the presence or absence of other group work in the students' learning experience. For example, at Edinburgh students undertaking the Software Engineering with Objects and Components (SEOC) course are about to spend a significant portion of their time on a group project which, however, is not object oriented and does not involve modelling. Since the benefits and costs of group working are to a large extent independent of the

subject matter, this reduces the potential benefit, and student acceptability, of including a group project in the SEOC course.

A further recurring theme is:

How can we encourage students to produce "good" models, and measure the quality of students' software models?

Our students naturally tend to be more comfortable with programming than with modelling. In the context of the relatively small, quick developments they experience in universities, an error or infelicity in a model tends to be relatively easy to repair at the code level, and there is little incentive for ensuring that the model itself is of high quality. One way to tackle this is to teach using MDD techniques from early on, but this requires a commitment to complex tools. Another approach is to use a tool – Umple is the example discussed in this paper – where code and model are developed simultaneously and have to be consistent.

The quality of a model can be as basic as its adherence to the syntax of the modelling language it is written in, or can refer to the quality of the design that is modelled. Syntax checking is of course an incentive for using a UML tool (although one has to be aware that tools do not always enforce syntax which is genuinely correct, and students can be misled). Our courses differ in the extent to which they mix design considerations with technical modelling ones; this is something for each course designer to consider carefully, in the light of factors such as the prior experience of the students, time and other resources available, and intended outcomes.

5 Conclusions and future work

In this informal paper, we have aimed to record the substance of conversations we had at the Educators' Symposium of 2013, and afterwards, about our knowledge and experience of tool use in teaching object-oriented modelling and related skills.

Our experiences and discussions have given us the impression that educators nowadays still experience a range of difficulties with commercial and non-commercial modelling tools. Features such as code generation or execution of models do not seem to satisfy educators' needs fully. Although educators see the benefits for students in gaining experience with tools for their future professions, the use of pen and paper can often seem a better solution when it comes down to education in modelling, because, for example, of distraction caused by the complexity of tools.

We are aware that we have only scratched the surface of what could be said and only a small number of people participated in our discussion.

It would be interesting to conduct a wider study of relevant courses and of how and why they differ in their tool use. This could yield a set of characteristics for future (educational) modelling tools.

Acknowledgements

We thank the other participants in the Educators' Symposium, especially Kenji Hisazumi, for their participation and their comments before the writing of this paper.

References

1. Don S. Batory, Eric Latimer, and Maider Azanza. Teaching model driven engineering from a relational database perspective. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *MoDELS*, volume 8107 of *Lecture Notes in Computer Science*, pages 121–137. Springer, 2013.
2. Jürgen Börstler, Ludwik Kuzniarz, Carl Alphonce, William B Sanders, and Michal Smialek. Teaching software modeling in computing curricula. In *ITiCSE-WGR '12 Proceedings of the final reports on Innovation and technology in computer science education 2012 working groups*, ITiCSE-WGR '12, pages 39–50, NY, USA, 2012. ACM New York.
3. PJ Burton and RE Bruhn. Using UML to facilitate the teaching of object-oriented systems analysis and design. *Journal of Computing Sciences in Colleges*, 19(3):278–290, 2004.
4. Eric Crahen, Carl Alphonce, and Phil Ventura. QuickUML: a beginner's UML tool. In *OOPSLA '02 Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 62–63, NY, USA, 2002. ACM New York.
5. Birgit Demuth. How should teaching modeling and programming intertwine? In *EduSymp '12: Proceedings of the 8th edition of the Educators' Symposium*, New York, NY, USA, 2012. ACM.
6. Birgit Demuth, Sebastian Goetz, Harry Sneed, and Uwe Schmidt. Evaluation of students' modeling and programming skills. In *EduSymp '13: Proceedings of the 9th edition of the Educators' Symposium*, 2013.
7. Javier Garzás and Mario Piattini. Improving the teaching of object-oriented design knowledge. *ACM SIGCSE Bulletin*, 39(4):108, December 2007.
8. Klaus Marius Hansen and Anne Vintner Ratzer. Tool support for collaborative teaching and learning of object oriented modeling. In *Proceedings of ITiCSE 02*, pages 146–150, 2002.
9. Ludwik Kuzniarz and Jürgen Börstler. Teaching modeling: an initial classification of related issues. In *Pre-Proceedings of the 7th Educators' Symposium@MODELS 2011 – Software Modeling in Education*, pages 65–70, 2011.
10. Timothy Lethbridge and Robert Langaniere. *Object oriented software engineering: practical software development using UML and Java*. McGraw Hill, 2002.
11. Timothy Lethbridge, Gunter Mussbacher, and Andrew Forward. Teaching UML using umple: Applying model-oriented programming in the classroom. In *Proceedings of CSEE&T 2011*, pages 421–428, 2011.
12. Sabine Moisan and JP Rigault. Teaching object-oriented modeling and UML to various audiences. In *Proceedings of EduSymp at MODELS'09*, 2010.
13. Marian Petre. Uml in practice. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 722–731, Piscataway, NJ, USA, 2013. IEEE Press.

14. Ervin Ramollari and Dimitris Dranidis. StudentUML: An educational tool supporting object-oriented analysis and design. In *Proceedings of the 11th Panhellenic Conference on Informatics (PCI 2007)*, pages 363–373, 2007.
15. Carsten Schulte and J Niere. Thinking in Object Structures: Teaching Modelling in Secondary Schools. *Pedagogies and Tools for Learning Object*, 2002.
16. Perdita Stevens. Updating the software engineering curriculum at Edinburgh University. In Paul Klint and Jerzy T. Nawrocki, editors, *Proc. Software Engineering Education Symposium SEES'98*, pages 188–193. Scientific Publishers OWN, 1998.
17. Dave R Stikkolorum, Michel RV Chaudron, and Oswald de Bruin. The art of software design, a video game for learning software design principles. In *Gamification Contest MODELS*, 2012.
18. Dave R Stikkolorum, Claire E Stevenson, and Michel RV Chaudron. Assessing software design skills and their relation with reasoning skills. In *EduSymp '13: Proceedings of the 9th edition of the Educators' Symposium*, 2013.
19. Scott A SA Turner, Manuel A Pérez-Quiñones, and Stephen H Edwards. minimUML: A minimalist approach to UML diagramming for early computer science education. *Journal on Educational Resources in Computing (JERIC)*, 5(4), December 2005.
20. Steffen Zschaler, Birgit Demuth, and Lothar Schmitz. Salespoint: A java framework for teaching object-oriented software development. *Science of Computer Programming*, 79(0):189–203, 2014.