

Teaching UML is Teaching Software Engineering is Teaching Abstraction

Gregor Engels, Jan Hendrik Hausmann, Marc Lohmann, Stefan Sauer
Universität Paderborn, Germany
{engels\hausmann\mlohmann\sauer}@upb.de

Abstract

As the Unified Modeling Language (UML) has by now seen widespread and successful use in the software industry and academia alike, it has also found its way into many computer science curricula. An outstanding advantage of teaching UML is that it enables an integration of many crucial concepts of software engineering, far beyond its concrete notation. Most important among these concepts is that of abstraction. We present a course design which demonstrates the use of UML as a vehicle for teaching such core concepts of software engineering. Multimedia elements and tools help to efficiently convey the course's message to the students.

1. Introduction

What shall we teach undergraduate computer science (CS) students? This question has plagued our profession for decades now. Possible answers range over several dimensions from principles (“Programming first!”) over different application fields (“Compiler construction!”), technologies (“Java!”), and paradigms (“OO!”). And for the last five years, another big player has entered the contest for the students' attention: UML [9]. Being widely used in industry by now, proficiency in UML is certainly a valuable asset for every CS student. The widespread academic interest is also a factor in integrating this topic in the curriculum. Evidence for the rapid increase in the importance of UML can be gained from the IEEE/ACM's computing curricula project, which mentioned UML only in passing in 2001 (amongst CRC cards and other data modeling techniques) [11] and which explicitly places UML as a central component in the core course in software engineering (SE) in 2004 [12].

The incorporation of UML in a curriculum can and should not happen by adding a separate “UML” course. UML is nothing but a means to reach an end (the end in this case being the expression of software models). And there should not be courses purely on means in an academic setting. On the one hand, there is clear and present danger that the knowledge gained in such a course is outdated before the graduation day (UML 2, anyone?).

On the other hand, university graduates are expected to know the 'why' and not only the 'how'.

Fortunately, UML has a very natural application context and that is software development. In the context of a stepwise refinement from problem to program, one cannot only showcase UML's various features, but one can also demonstrate how these features are employed in a practical and meaningful context. The teaching of structured development itself (also a fundamental concept which should appear in every CS education) also benefits from employing UML diagrams as they help to make the various stages of development more tangible.

Another benefit of teaching UML to undergraduates goes even beyond software development: UML is a set of notations for modeling. The core of modeling is *abstraction*; and abstraction is the principle which is right at the heart of software engineering and in fact of computer science in general. According to Hoare: “In the development of our understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction.” [5]. However, as Jeff Kramer repeatedly pointed out in a recent series of keynotes [7], abstraction is as hard to teach as it is important. His solution, as well as ours, is the use of modeling. Modeling languages provide a fixed structure in which to express certain aspects of a domain or idea. Thus, the creation of UML diagrams always implies abstracting from the concrete instance and recognizing the more abstract core of the matter.

The notion of abstraction is not absolute, but depends on a particular purpose. Abstractions of the same subject can vary in the aspect they represent and their level of detail. The relations between these abstractions are captured by the notions of consistency and refinement. Decomposition and synthesis are techniques to elicit new abstractions. All of these very general concepts appear naturally in the context of UML-based development. Since UML is a family of diagrams rather than a single diagram, students also very naturally learn about different possible aspects and their interrelations.

In this paper we make the case that teaching UML is teaching Software Engineering and is teaching abstraction

at the same time. As such, teaching UML is not just “another language course” but it can and should play a central role in each undergraduate CS education.

The presentation of the paper proceeds in three sections. We start by laying out the recently remodeled CS program in Paderborn, which implements the points made in this paper. Focusing more concretely on a single course, we show the concept of the course *Softwareentwurf* (translates to software design), which is the fundamental SE/UML course for undergraduate students (Sect. 3). Here, we illustrate the integrated teaching of software development principles along a simplified software development process together with extensive UML modeling. We point out how fundamental concepts can be elicited in this structure. Even the best teaching concepts fail without a proper operationalization. Thus in Sect. 4, we present some of the practices which we have found useful in conveying our intentions to the students. Our experiences with the course are briefly reported in Section 5.

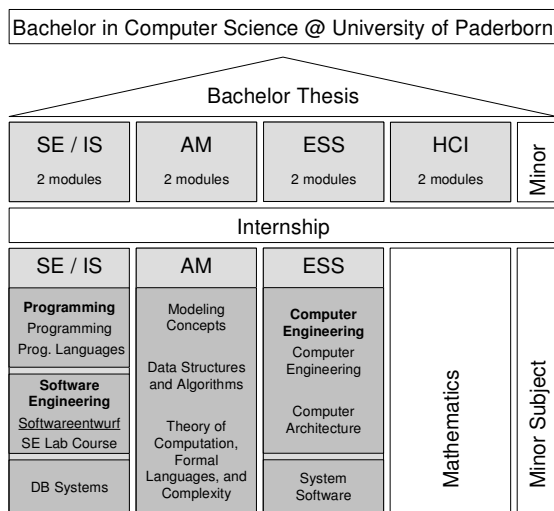


Figure 1: Paderborn CS curriculum

2. The Paderborn CS curriculum

Due to the switch to the Bachelor/Master structure, the Paderborn CS department has recently restructured its academic program.

In the Bachelor CS program, four different thematic columns are integrated (see Figure 1): Software Engineering and Information Systems (SE/IS), Algorithms and Models (AM), Embedded Systems and System Software (ESS), and Human Computer Interaction (HCI).

In addition, the curriculum covers courses in mathematics as well as a limited number of courses in a minor subject (like economics, engineering or natural sciences). This overall structure as well as the concrete courses implementing this structure have not only been accredited by the German ASIIN accreditation agency, but have also formed the basis for a national recommendation for a well-structured CS curriculum.

In the first term of the Bachelor curriculum, CS students have to take a course on object-oriented programming (with an introduction to the Java programming language) as well as a course on foundations of modeling concepts. Topics which are taught in this course cover basic structuring means like sets, trees, terms, graphs, and basic modeling instruments like predicate logic, finite automata, Petri Nets, and ER models.

This emphasis on programming and modeling concepts right from the beginning distinguishes the Paderborn CS curriculum from most other CS curricula of German universities. Intentionally, this choice led to a reduction of general mathematics courses. Rather emphasizing the modeling aspect right from the beginning yields the advantage that CS students are dealing with computer science issues from the first day of their studies and are focusing more on formalisms and abstractions that are directly applicable in CS.

The programming education is continued in the second term by introducing advanced programming concepts (e.g., concurrency), different paradigms (e.g., functional programming), and the wealth of fundamental data structures and algorithms. All of these courses provide the students with solid programming skills but the scope of examples and exercises is limited to the implementation of search algorithms and the like, i.e., tasks which have a clear structure and require neither teamwork nor extensive planning.

In the third term, the students take the course *Softwareentwurf*, which is the running example of this paper. The objective of the course is to introduce the students to the problems of industrial-strength software development and supply them with knowledge about software engineering techniques (especially UML) to overcome these problems. The course is also intended to build on, recall, and expand basic OO principles from the fundamental programming courses.

Based on this introduction to SE, all CS students have to take part in a lab course during the fourth term.

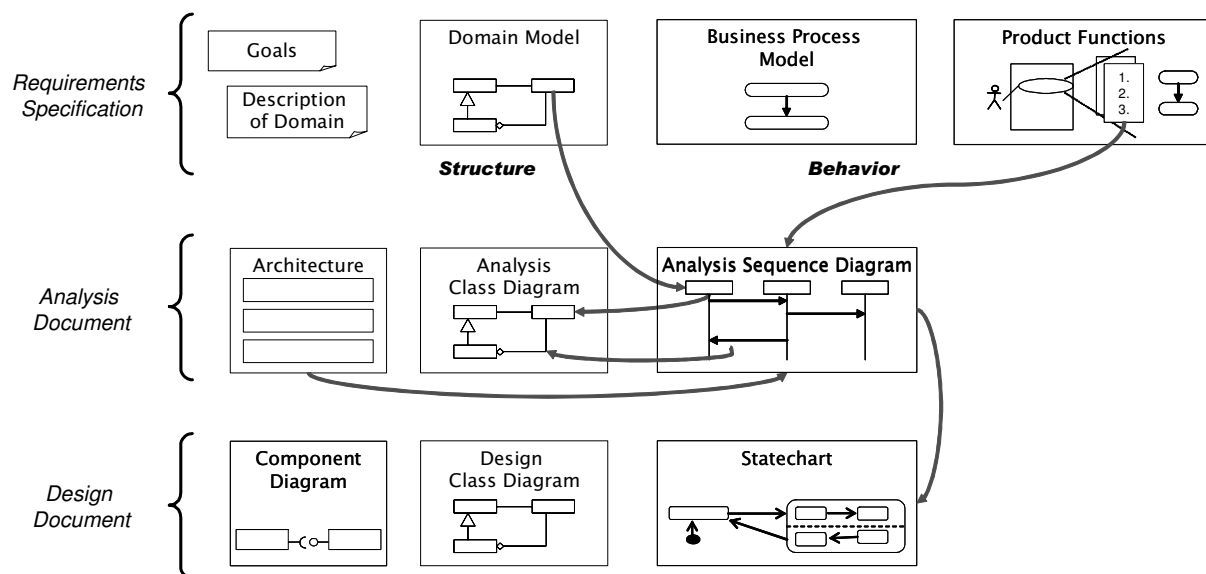


Figure 2: Concepts underlying the Paderborn Softwareentwurf course

In this lab course, a complex software development project has to be solved by small teams of students (cf. [4] for details on this course). The skills and knowledge acquired during the Softwareentwurf course are applied here in a practical setting. Upon finishing this lab, the students have to apply their skills in the industrial reality during a mandatory 8-week internship in a software company.

The overall structure of the program is thus to start from initial programming experience and to continually refine this experience in eliciting more abstract concepts on the one hand and practically applicable skills on the other hand. Teaching UML works towards both of these ends.

3. Embedding UML in a software development process for teaching – the Softwareentwurf course

In this section we focus on the course Softwareentwurf. The presentation here is not intended to completely reflect the contents of the course (nor is the course intended to cover all topics of Software Engineering) but rather to point out how fundamental CS concepts are embedded in this course.

The course is oriented along a simplified software development process which starts with a requirements specification phase and moves over analysis and design phases towards actual implementation. The different models and UML diagrams employed during this process and a selection of their relations are depicted in Figure 2. We omitted all relations which are not mentioned in the following sections.

3.1 Building abstractions – the domain model

The first model which is being built in our development process is the so-called domain model, a class diagram representing concepts and relations in the problem domain of the software system. Domain models help us to thoroughly understand the structure of the problem we face and thus aid us in building the right system for the user's needs.

Creating such a model requires two different kinds of knowledge: *what* is to be modeled and *how* is the model to be expressed. The latter kind of knowledge comprises the actual UML syntax symbols and is the 'easy' part of teaching as students quickly pick up the notations and tools can be used to aid them in drawing diagrams. What to model is actually the hard part as this question involves understanding of object-oriented structures and *abstractions*.

There are actually two steps of abstraction required for eventually achieving the domain class model. At first, situations in the problem domain have to be interpreted in terms of relevant objects and relations (we cannot—in general—assume a generalized description of the domain to be readily available, see Sect. 4.1 and 4.2). Irrelevant details have to be discarded and relevant information has to be notated using an object diagram.

The second abstraction generalizes these instance models towards the type level (Figure 3 shows a slide from the lecture exemplifying this distinction). Its result is expressed by a UML class diagram. Note that the notion of abstraction in these cases is not the same: while an object diagram is a token model, i.e., a projection of an observed situation, the class diagram is a type model, i.e.,

a generalization which only corresponds indirectly to real situations [8].

Experienced modelers actually perform both of these abstraction tasks at the same time, i.e., they can formulate a class diagram directly on the basis of concrete situations, but it is easier for students to separate these tasks.

Except for trivial cases, both abstraction tasks can yield a number of different actual results for a given problem. Typical questions are: “What is relevant?”, “Which entities belong to the problem domain, which are outside of it?”, and “Which level of detail is detailed enough?”. Furthermore, information can often be encoded in a number of ways in a UML diagram. Elements of the problem domain can for instance become classes of their own or they may become part of another class as its attributes. Structural relations can be encoded as object-valued attributes, associations, association classes or classes representing the relation explicitly.

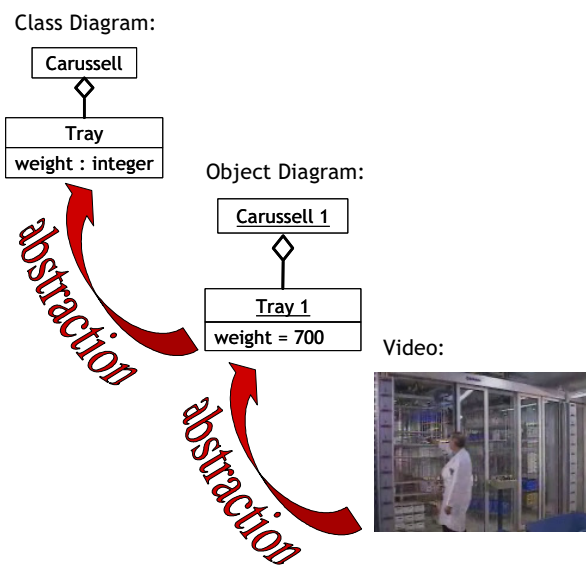


Figure 3: Slide of lecture Softwareentwurf

Students need to be provided with a set of guidelines which help them to decide such questions. Scott Ambler’s UML Style [2] is a collection of such general modeling guidelines, others are suitable for the didactic context only. For instance, we explicitly interdict object-valued attributes since the elicitation of domain structures and their visual representation are the main purpose of the domain model and using object-valued attributes defies this purpose. However, such rules can only be guidelines and students often feel very uncomfortable in facing the unavoidable ambiguities in building abstractions.

3.2 Separation of concerns – the business activities

The documentation of the problem domain also covers the description of important business processes. This is the second modeling perspective of the software’s context description and it is separated from the domain model as another concern. While the domain model represents the structure of the problem domain, the business process model describes the relevant behavior. We use UML activity diagrams to represent the business processes.

The hard part here is once again not the UML syntax (which is fairly easy for basic activity diagrams) but realizing what to model. In the domain model we forced students to concentrate on the structural aspect only. Now, only activities are in the focus of the model. It is often surprising for students that the same domain descriptions which already yielded a detailed domain model also contain the information to build an equally detailed process model which is (at least notationally) completely independent from the domain model. The realization of multiple aspects which transcend most problem situations is very important as it allows for immense reduction in problem complexity. Structure and behavior are actually an easy case as they are intuitively orthogonal, later phases in the process will pose much closer related concerns (e.g., inter-object and intra-object behavior).

3.3 OO at work – the analysis sequence diagram

Moving from the requirements specification to the analysis phase again implies a significant shift in perspective. While the former phase forces the students to think in terms of the problem domain only and discourages all premature notions of solutions, the analysis phase works towards shaping a solution for the given problem. This entails a number of different concepts which coincide in the central diagram of this phase: the analysis sequence diagram.

The first concept that underlies this diagram is *integration*. Following the object-oriented paradigm of integration of structure and behavior, the different concerns of the requirements specification need to be brought together in a meaningful way. In addition, we need to account for the overall structure of the chosen architecture.

Another underlying principle of analysis is *decomposition*: the responsibilities of the software system are partitioned among interacting objects. This relates to architecture again. For example, following a three-layer architecture we can distinguish between Entity, Boundary, and Control classes (as introduced in the Unified Process [6]). Entity classes are derived from the domain model, Control classes represent the execution of product functions (modeled as use cases in the requirements specification), and Boundary classes encapsulate the

communication with external entities. The interaction between these objects is depicted by a sequence diagram.

Building the analysis sequence diagram forces students to recall the basic principles of object-orientation. Distributed data and localized behavior on this data are principles which lead them to a suitable solution. In each step of the construction, one needs to decide which step in the overall task is to be carried out next, which object is best suited to carry out this task, and how can this object be accessed from the current object. Even though the students should be aware of these principles from previous courses, applying them in this abstract fashion is rather challenging.

A further complication is that—at this point of the course—students have become rather familiar with thinking in terms of the type level, i.e., classes. Analyzing a single scenario with a sequence diagram forces them to work on the instance level again. The ability to mentally switch between these levels seamlessly is a very important capability for a software engineer. To quote Donald Knuth on this: “Computer scientists see things simultaneously at the low level and the high level” [13].

The overall effort for students to cope with tying information from three different sources together and eliciting new structures under the OO paradigm is very high. Again, not a unique diagram can be designated to be the only correct solution for a given problem, there is rather a space of equivalent acceptable solutions, differing in the level of detail or in some decisions taken.

3.4 Refinement – how class diagrams change

As emphasized in the last section, analysis works along the scenarios of the system on the instance level. The information distributed over the various analysis sequence diagrams needs to undergo a *synthesis* into a single diagram. For this purpose we use the analysis class diagram. Information regarding objects, their operations and properties are collected from the sequence diagrams and combined in a class diagram. Especially the derivation of associations is not straightforward as associations denote a permanent relationship between objects of two classes (in contrast to temporary relationships by parameter or result passing). Whether an established relationship is permanent can only be decided by taking *all* sequence diagrams into account, i.e., we have a global criterion.

Up to this point in the course, the students could identify UML notations and their purpose by their use in the process. The analysis class diagram breaks this mapping up as its intention differs significantly from that of the domain model, even though both employ the notation of a UML class diagram. Eliciting the connections and distinctions between these two kinds of class diagrams gives rise to the notion of *refinement*. The

refinement notion in this case is that classes of the domain model that are not used in an analysis sequence diagram will not appear in the analysis class diagram; classes that are used might be extended with new information. The classes conceived during the analysis phase (Control and Boundary classes) and their relationships are added, and operations and navigation directions of associations can now be used. Later in the process, a further class diagram will be introduced for gathering the results of the design phase. With this overloading of the class diagram notation, the students realize that modeling notations can be used in a number of contexts and that knowledge about the context is essential for their interpretation.

3.5 A different perspective – statecharts

Introducing the statechart notation during the design phase enforces once more a change of perspective. While sequence diagrams depict system behavior in terms of object interaction, statecharts concentrate on invocation sequences on a single object. These two views on object behavior have close connections, thus the problem of *consistency* needs to be addressed. In the Softwareentwurf course, we use statecharts in the sense of protocol statecharts, i.e., every invocation sequence on an object in a sequence diagram must be permitted by the transitions of the statechart of the object’s class. This notion of consistency facilitates the construction of statecharts from sequence diagrams and vice versa.

3.6 Summary

In this section we have shown how teaching the different diagrams of UML in the context of a software development process can demonstrate a number of fundamental concepts related to abstraction. If students understand these concepts, they acquire knowledge which is neither bound to our particular instance of a process, nor to the use of UML as a modeling notation (as, e.g., pointed out in [3]).

Using models in this way is not without its problems, however. As emphasized above, the necessary creativity and ambiguity in finding the ‘right’ abstraction is an uncomfortable process for students. Keeping track of the different relations between the produced diagrams is also challenging. For these and other problems, practical measures need to be taken to actually achieve the course’s goals. The next section is devoted to such more practical teaching concepts.

4. Getting the show on the road – didactics, exercises, tools and tricks

Planning an educational course has a strategic as well as a tactical level. On the strategic level, the learning targets are determined and the general structure of the

course is derived from them. This has been covered in the previous section. On the tactical level, one needs to address the more detailed issues of how to realize the strategies in a successful way. While general didactic principles can often be applied here, we are aware of a number of issues which are rather specific for the teaching of UML as we perform it. In this section we highlight four of these specific issues and outline our solutions for them.

4.1 Feigning incompetence for requirements capture – the use of multimedia

In teaching requirements engineering, we can identify an inherent obstacle. Most of the problems in gathering the requirements stem from the fact that the requirements are formulated and posed by persons who are unfamiliar with the process of requirements capture, have no knowledge of the technical terminology used in computer science, and cannot distinguish requirements towards a software system from the working context surrounding it. There is furthermore the problem that the input is typically incomplete and can even be contradictory if multiple stakeholders have been interviewed.

Good software engineering teachers, however, are always trying to present consistent, precise and complete information to their students. They are aware of the technological terms and automatically categorize information according to their knowledge. If such teachers do now try to present a requirements engineering problem, they will find it very hard, if not impossible, to keep this knowledge from influencing the presentation in a way as to give students hints on the solution of the problem. In other words, it is impossible for them to feign the required incompetence. For example, consider most textbook examples for descriptions underlying the construction of class diagrams (exceptions notwithstanding). The text usually comprises clear entities with elaborated relationships. (“An elevator serves a number of floors. Each elevator has a maximal load capacity. On each floor there is one calling button.”) Expressing this information in the form of a UML class diagram can be no more than a simple translation and is not necessarily modeling. All necessary abstractions have been already made by formulating the text.

A possible solution for this dilemma is the use of video techniques. By relying on an external medium for the presentation of the problem domain, the lecturer can be sure that no solution hints slip into the presentation. This effect is maximized if videos can be obtained from external sources which are not influenced by the intentions of requirements engineering. Thus the influence of the lecturer is eliminated from both the production and the presentation of the example.

A video does furthermore present the full complexity of realistic work surroundings and the activities carried out there. To handle this audiovisual complexity, to avoid

information overload, and to form the right abstractions are important parts of requirements engineering. If the problem domain is presented by a text instead of a video, this complexity is vastly reduced.

In our Softwareentwurf course, we make heavily use of video techniques to present concrete and realistic problem domains with a sufficient complexity. For example, we were able to obtain a video from a machine manufacturer (originally intended for marketing purposes) which describes the operations of an automated storage system for a hospital. We use this video in the lecture to present the domain as a foundation for the domain and business models (cf. Sect. 3.1 and 3.2).

4.2 Listening is not nearly enough – exercises and examples

A central problem of teaching modeling (and thus abstraction) is that the process of abstraction cannot be formalized, but relies on creativity and intuition. Consequently, no unique model can be designated in practice as the optimal abstraction for a given problem, but there is rather a space of good solutions. Building abstractions can thus not be learned by following a deterministic recipe. It rather requires a certain skill and intuition to detect relevant information, settle on the correct level of abstraction, and choose adequate representations for the information. Such skills cannot be effectively taught in lectures. They have to be based on experience (see also [14]). Thus, exercises are absolutely crucial for the teaching of modeling.

In the Softwareentwurf course we employ two different kinds of exercises: group sessions where the students solve small problems in teams under the guidance of a tutor and home exercises where the students have to tackle more complex assignments over the course of a week. Especially for the (unguided) home exercises, it is essential to provide examples which are complex enough to actually allow for non-trivial abstractions, but which also have enough limitations to prevent students from getting lost in the task. We found board games to be a very good domain for such examples. Board games comprise domain structures and business processes (game rules) which are partially documented in the instructions and can partially be elicited by actually playing the game. Especially this hands-on experience allows the students to engage in much more detailed discussion on the relevance of certain elements. Disagreement in interpreting rules gives rise to sometimes rather different models and sparks debates on the comparability of different solutions. Board games also have a reduced complexity, clear context delimitations, and a high motivational factor for students. A rich variety of examples ranging from the very simple to the very complex can be gained from this general domain.

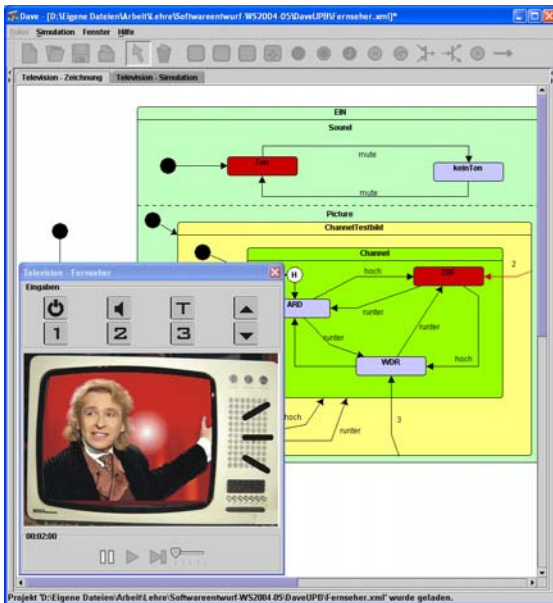


Figure 4: DAVE

4.3 Modeling at hand – how tools can help

Employing modeling tools in the educational setting comprises advantages and disadvantages which need to be carefully weighted. Advantages are that (1) modeling tools allow for incremental development of complex models, (2) changes are facilitated easily, and (3) the resulting models are not only cleaner than many scribbled paper-and-pen solutions, but also better thought out as students are unburdened from looking up notations and can concentrate on the content of the model rather than its form. Experience with modeling tools is also a benefit for students who start working in industry after their graduation.

Unfortunately, modeling tools typically used in industry, such as IBM's Rational Rose or Borland's Together, have significant drawbacks when applied in an educational setting. These drawbacks stem from the fact that professional modeling tools are rather *heavyweight* pieces of software in several aspects. From an organizational point of view, licensing and cost questions need to be addressed as well as support for their installation and operation (in the rather diverse configurations of the students' computers). Other issues stem from the didactical point of view: the tools' user interfaces are quite complex and their rich set of features can distract inexperienced users. Using such tools entails the risk that the students concentrate on the tool usage rather than the particular modeling exercise. Additionally, confusion arises if the modeling tools do not follow the standard UML notation (e.g. great variety can be observed in displaying an association's reading direction).

Heavyweight tools furthermore enforce certain consistency notions in the models, for example object diagrams can only be drawn if an according class diagram is present. The consistency in this case is intended for enabling automatic code generation and does not necessarily conform to the notions of consistency that we introduce in the lecture.

For educational purposes it is desirable to have rather *lightweight* tools that allow the students to concentrate on modeling instead of tool handling. The Dortmund Automaton Visualizer and Editor (DAVE) [10] developed in the eLearning project MuSoft [1] is such a lightweight tool for modeling statecharts. A screenshot of this tool is displayed in Figure 4.

Explicitly designed for eLearning purposes, DAVE not only supports the modeling of statecharts, but also includes a simulation engine. Students can thus get direct feedback on the semantics of their models when they are doing their homework. We observed a significant improvement of the students' understanding of statecharts due to the adoption of DAVE in the exercises while the organizational effort was reasonable.

4.4 Holding things together – on the importance of documents

From previous courses we knew that the students had huge problems with the large amount of different UML notations and their application in a software development process. Diagrams were regarded as separate entities and their interrelations were not comprehended by the students.

As a remedy for this problem, we introduced documents as a further structuring means in our process. The main idea of this document-centric approach is that documents incorporate the different diagrams of a development phase, thus clarifying their relations. We designed templates for the documents which do not only predefine the structure, but also provide guidelines for constructing the models and ensuring their consistency. Document templates are complemented by review guidelines and checklists in order to let the students review and improve the quality of their respective documents. The document structure is also used in guiding the lecture, thus students have now an improved understanding about where to apply which technique (according to our empirical evaluations, the number of students who reported problems with the mapping of lecture and exercise topics was reduced by 70%).

5. Findings

Taking one's own medicine is always bitter. As the students need time to learn the abstract concepts behind software engineering, we needed time to recognize the

abstract concepts behind our teaching of UML as well. The current form of the Softwareentwurf course as presented in this paper is the result of 7 years of teaching UML to more than 2000 students. The process of reflection upon our teaching was fueled and guided by closely monitoring our recipients, the students. This monitoring included direct feedback from the students collected, e.g., by questionnaires as well as our examination of their performance. Obtaining meaningful quantified evaluation results has proven to be hard as different student groups were exposed to multiple different factors which cannot be reliably separated (e.g., changes in the overall curriculum, lecture times, lecturers, examples used etc.).

More important than our own judgement of our teaching success is the fact how well our course prepares the students for the challenges ahead of them, i.e., the lab course and the internship. By using video observations of software development teams in the lab course we could ascertain that the students followed the structured development process that we taught and used the UML diagrams accordingly. When faced with new requirements (in the lab course they have to extend an existing system [4], thus use re-engineering techniques) they were able to adopt the process and the documents and showed (sometimes surprisingly good) ideas how to employ UML diagrams for this new task. We see this flexible handling as an indication that the students' understanding does indeed transcend the concrete notations and methodology that we teach them and extends to the concepts behind it. It is this understanding of principles which will be most beneficial for CS students in the long term, whether they choose to continue with the Master program or begin a career in industry.

6. Conclusions

In this paper we have argued that—from an educational perspective—UML is not “just another modeling notation”, but an excellent vehicle to demonstrate some of the very core principles behind computer science in general and software engineering in particular. Our Softwareentwurf course implements this idea of combining the usually competing goals of eliciting abstract concepts and training practical skills. In our experience, the combination of these goals is beneficial for learning and acquiring both concepts and skills since the abstract concepts become more tangible and the practical skills become more transferable by this combination.

For educators who teach UML and software engineering, we hope to have provided some new insights on the strategic and some useful tips for the tactical planning of their own course.

References

- [1] Alfert, K., Doberkat, E.-E., Engels, G., Lohmann, M., Magenheim, J., Schürr, A., “MuSoft: Multimedia in der Softwaretechnik”, in *Software Engineering im Unterricht der Hochschulen*, pp. 70-80, dpunkt, 2003.
- [2] Ambler, S.W., “The Elements of UML 2.0 Style”, Cambridge University Press, 2005.
- [3] Bruce, K. “Thoughts on computer science education”, *ACM Computing Surveys* 28(4):93, 1996
- [4] Gehrke, M., Giese, H., Nickel, U.A., Niere, J., Tichy, M., Wadsack, J.P., Zündorf, A., ”Reporting about Industrial Strength Software Engineering Courses for Undergraduates”, in *Proc. 24th Intl. Conf. on Software Engineering (ICSE)*, pp. 395-405, ACM Press, 2002.
- [5] Hoare, C.A.R. “Notes on data structuring”, in O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare (eds), *Structured Programming*, pp. 83-174. Academic Press, 1972.
- [6] Jacobson, I., Booch, G., Rumbaugh, J., “The Unified Software Development Process”, Addison-Wesley Professional, 1999.
- [7] Kramer, J., “Abstraction - Is it Teachable? The Devil is in the Detail”, in *Proc. 16th Intl. Conf. on Software Engineering Education and Training*, p. 32, 2003.
- [8] Kuehne, T., “What is a Model?”, In J. Bezivin, R. Heckel (eds), *Language Engineering for Model-Driven Software Development*, Dagstuhl Seminar Proceedings 04101, 2005.
- [9] OMG, “UML 2.0 Superstructure Specification”, ptc/04-10-02, 2004.
- [10] Pleumann, J., “Erfahrungen mit dem multimedialen, didaktischen Modellierungswerkzeug DAVE“, in *Proc. DeLFI 2004*, LNI P-52, pp.55-66, 2004.
- [11] The Joint Task Force on Computing Curricula, “Computing Curricula 2001 Computer Science – Final Report”, IEEE Computer Society, Association for Computing Machinery, <http://www.sigcse.org/cc2001/>, 2001.
- [12] The Joint Task Force on Computing Curricula, “Software Engineering 2004 Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering: A Volume of the Computing Curricula Series”, <http://sites.computer.org/ccse/>, 2004.
- [13] Woehr, J., “An Interview with Donald Knuth - DDJ chats with one of the world's leading computer scientists”, *Dr. Dobb's Journal*, April 1996.
- [14] Burton,P.J., Bruhn, R.E.,”Using UML to facilitate the Teaching of Object-oriented Systems Analysis and Design”, *Journal of Computing Sciences in Colleges* Vol. 19 , Issue 3, pp. 278 – 290, 2004.