Teaching Modeling: Why, When, What?

Jean Bezivin[(A)], Robert France[(B)], Martin Gogolla[(C)],
Oystein Haugen[(D)], Gabriele Taentzer[(E)], Daniel Varro[(F)]

University of Nantes[(A)], France
Colorado State University[(B)], USA
University of Bremen[(C)], Germany
University of Oslo[(D)], Norway
University of Marburg[(E)], Germany
Technical University of Budapest[(F)], Hungary

Abstract. This paper reports on a panel discussion held during the Educators' Symposium at MODELS'2009. It shortly explains the context provided for the discussion and outlines the statements made by the panelists. The panelists were asked to make their statements along a number of topics relevant to teaching modeling like: Notation, Semantics, Programming, Tooling, Suitability, Metamodeling.

1 Context

In order to set the context for the panel discussion on teaching modeling, we should be aware of the fact that, in the software development process, models can serve different purposes and that models can be used on a number of different levels. The UML (Unified Modeling Language) Reference Manual nicely explains this by stating the following classifications.

Purposes of models

- To capture and precisely state requirements and domain knowledge so that all stakeholders may understand and agree on them,
- To think about the design of a system,
- To capture design decisions in a mutable form separate from the requirements,
- To generate usable work products,
- To organize, find, filter, retrieve, examine, and edit information about large systems,
- To explore multiple solutions economically, and
- To master complex systems.

Levels of models

- Guides to the thought process,
- Abstract specifications of the essential structure of a system,

- Full specifications of a final system,
- Exemplars of typical or possible systems, and
- Complete or partial descriptions of systems.

Having this in mind, the question comes up when and where (in which institution) to teach modeling. There are various possible answers.

- At (high-)school,
- During university studies (before learning programming, after learning programming, parallel to learning programming),
- After obtaining a first degree in programming, and
- During an advanced training for IT professionals.

Teaching modeling also opens up a bunch of questions with respect to the taught contents. When asking "What" is taught at least the following dimensions, which could be considered, come up.

- Graphical versus textual modeling languages,
- "Modern" UML-like formalisms versus "historical" formalisms (like Entity-Relationship schemata, automata, statecharts or petri nets),
- Modeling versus programming,
- Tooling versus modeling with paper and pencil or on the blackboard,
- Modeling as a stand-alone field versus modeling in computer science subfields (like class diagrams in database systems, sequence diagrams in networking, statecharts in embedded systems),
- Modeling versus specification,
- Universal modeling versus domain specific modeling,
- Syntax versus semantics (with different approaches to semantics like textual, informal, formal, programming language-based, graph-based semantics), and
- Models versus metamodels.

The panelists were asked to prepare their statements in light of this context.

2 We Need an Army of Engineers to Implement the MDE Vision (Jean Bezivin)

Towards implementing a vision: Modern software modeling techniques have rapidly evolved since the MDA proposal by the OMG in November 2000. We now understand much more clearly the foundations. Corresponding tools are becoming available, most notably from the open source community. Industrial applications in various fields have demonstrated the practical potential of the approach. It is interesting to notice that this has been an industry-driven evolution, because the original problem was mainly stated by OMG (an industrial oriented consortium) and that research labs and academic conferences came in only in a second phase in order, to propose a more regular conceptual framework and to link this pragmatic quest for more abstraction in software development to general theories of software engineering. The initial problem was put forward by practitioners and progressively reshaped by researchers. Now we are missing the last phase of the process, the education part. We have a reasonable consistent theory; we have the basic tooling support for this theory; we have empirical evidence that the industrial deployment of modern software modeling may bring a significant return on investment for many projects; what we need now is the army of engineers to implement that vision in various branches of industry.

Research? Deployment? Teaching! Unfortunately the engineers in Model Driven Engineering (MDE) are not so numerous. They are few PhD students that can lead innovative projects, but today, the vast majority of professional programmers, educated in the last decade, is excellent at developing object-oriented programming projects in Java, EJB or C#, but much less convinced when it comes to using MDE techniques. We are facing an important paradigm shift, probably more important than the migration from procedural technology to object technology that happened in the eighties. We have not only to educate new students in modeling, but also to re-educate former students in the advantages of transformative domain modeling over practices of direct application coding in high-level general purpose languages. MDE is now accepted in many places. As Tony Hoare stated at ICSE 1996, "Teaching reduces the gap [between theory and practice] and research increases it again". In the never-ending Research-Deployment-Teaching cycles, MDE is now in a position where the teaching question is probably the more acute.

UML versus DSL: So we need to address seriously the teaching of modern software modeling techniques without too much delay. Such a teaching should be modest since we have learned to be cautious in the short history of MDE. For example, we learned in the past that MDE should not be confused with UML, with general purpose languages, with visual languages, etc. There are plenty of advanced and successful deployments of MDE projects that do not use at all UML, or any visual language. On the contrary, there are currently a number of modeling projects using some form of Domain Specific Languages (DSLs) up to a point that it is hard to distinguish between teaching MDE and teaching DSLs. But this situation, as of today, may still evolve in the future.

Metamodeling and model transformation: The first part of a teaching curriculum in software modeling should present the core technology and we see two essential dimensions here: metamodeling and model transformation. This should be presented in such a generic way that it may encompass the most important instantiations of these ideas (e.g., MOF, KM3, ECore; but also Oslo, XML schema, Protege, OWL; QVT, ATL, GReAT, AGG, TGG; XSLT, XQuery).

Applications of MDE: The second part of a teaching curriculum in software modeling should present a broad view of current known applications of MDE in fields like embedded systems, information systems, web engineering, software modernization, etc. There is such a broad range of domains (health care, transportation, business modeling, software processes, banking and insurance industry, human resource management, avionics, automotive industry, etc.) from where we can now draw concrete success stories, that this is becoming an easily documented exercise. Within this large spectrum of case studies, it is quite easy to distinguish the three successive generations of MDE applications: code generation from abstract models (like Java generation from UML), model discovery from unknown systems (like extracting business rules from COBOL legacy code) and the last more general application of model driven interoperability (like connecting different tools by transformation after extraction of their related metamodels). In the last generation transformations are executed at run time while in the first two ones, transformation are executed during the software production and maintenance phases.

Convincing stakeholders about MDE value: But probably the most important sequence in a MDE curriculum is more human and social than technological: how to manage the changes in the enterprise, how to convince all the actors that they are going to be more productive by raising the abstraction working level from base programming code to domain modeling, from direct coding in general purpose programming languages to transformation from standard or custom made DSLs. The non-technical issues are probably the hardest ones to teach.

Building specialized high-level curricula: Nearly ten years after the initial OMG proposal, software modeling seems now quite well accepted. The main challenge we face now is to succeed in building specialized high-level curricula so that engineers will be able to rapidly deploy MDE solutions in industry.

3 Expert modelers are good programmers (Robert France)

Expert modelers are good programmers: There is a misguided notion among some students and practitioners that you can succeed in a software modeling course if your programming skills are weak ("You don't have to program; you just draw pictures!"). By "programming skills" I do not mean working knowledge of programming language syntax; the skill I am referring to is the skill of developing a solution expressed in a programming language using the abstractions provided by the language. The students that hold this notion view modeling as an opportunity for informally thinking about a solution. These notions are often quickly dispelled in a modeling course that emphasizes rigorous modeling of problems and solutions. In such courses students quickly find out that identifying the "right" abstractions requires that they have a good programming background. This should not come as a surprise. How can we develop good abstractions if we do not understand the artifacts (programs) we are abstracting over? I put forward the following hypothesis: An expert modeler is a good programmer. The converse is not necessarily true. A good programmer is not necessarily a good modeler. The ability to abstract over concrete programming concepts is not something that all good programmers can do. It is an additional skill that makes a good programmer also a good modeler. In fact, I'll go further and hypothesize that programmers with good modeling skills produce better quality programs than those with weaker abstraction skills. I would be very interested in hearing about empirical studies that explore (or explode) these hypotheses.

Modeling should be developed alongside programming: If you subscribe to the above hypotheses, what is the implication for educators? It requires that modeling skills be developed alongside programming skills. As students are taught programming concepts, the means to describe abstractions over the programming concepts should also be introduced. Students should learn how to identify and describe abstractions that help them better organize their programs as they are developing their programming skills. This should lead to a greater appreciation of model-driven software development technologies that encode some knowledge about transforming abstractions to their realizations in a program.

Layer of general-purpose abstractions: Some have pointed out that my hypotheses seem to suggest that a good Java programmer must also be a good assembly language programmer (or, a good machine coder!); after all the Java language provides abstractions over these lower level language concepts. The reason why a good Java programmer does not need to have any knowledge of lower level languages is that we have been able to build a complete layer of abstractions above these lower levels with accompanying tools for automatically translating Java program abstractions to lower level abstractions (and vice versa). We do not have such a complete layer of general-purpose abstractions above the level of current high-level programming languages as yet. Such complete layers may appear in the near future in specific domains, thus allowing modelers to become programmers in those domains. In the meantime, modeling and programming skills should be taught hand-in-hand.

4 Models May or May Not Be: Executable, Formal, Complete (Oystein Haugen)

Modeling brings abstraction: We teach modeling because this makes the students better software engineers. We teach modeling to make them understand the value of abstraction. I would like to focus on three dichotomies that explain what modeling is and that should be focused in teaching modeling.

Distinction between model and program: First, we elaborate on the distinction between model and program. Is there a difference, and is that difference important? Models and programs are not very different, but how different they are depends on the kind of model. Programs are always executable. When models are executable some people claim they are programs. When models are not executable they are not programs. Not only executable models are useful, but executable models are particularly useful. Sequence diagrams represent models that are not directly executable, but they are definitely useful. Still the power of execution makes it much easier to understand the model. The model can be explored by those that cannot read its definition. The proof of the pudding is in the eating, and execution is the eating of the model.

Execution concepts versus application domain concepts: Is an executable model equivalent to a program that performs the same execution? That is a matter of definition, but our position is that there is a significant difference, and the difference lies in abstraction. Typically programming languages have general concepts that describe the execution on a computer while modeling languages have concepts closer to some application domain, but they may still be quite general. An example to illustrate the difference is that we find no programming languages with dedicated concepts to describe finite state machines, but you find those in many modeling languages. This does not mean you cannot implement state machines in a programming language, but you have no concepts with which to think in terms of state machines and your programming is more error prone.

Formal and informal models: Second, we consider the distinction between formal and informal models. Again both kinds of models are useful, but formal models have potentials to be more useful. An informal method cannot be executable, but are all executable models formal? Our position is that the execution is the semantics, and the compiler the formal definition of the language. Not all agree to this, and obviously there is much to gain from a more manageable definition of semantics than a compiler. Mathematics and logic are often used for the purpose of defining semantics, but many students of informatics find a mathematical definition more difficult to read than the source of a compiler. Formal definitions can be applied to make executable code, or they can be used as base for analysis such as model-checking or quality metrics. Formal definitions can be transformed into test cases or even user interfaces.

Partial and complete models: Third, we explore the distinction between partial and complete models. An informal model is often partial, but there are partial models that are perfectly formal. Sequence diagrams have a formal semantics, but sequence diagrams can normally not provide a complete model. In fact it would normally be impractical or stupid to provide a complete model through sequence diagram. They are intended to give important, but partial knowledge. Complete models are typically needed to have an executable model, but it is possible to make execution engines that choose randomly between the valid choices.

Modeling brings forth the important aspects: Make no mistake, our categories of models do not cover sloppy and bad models. It is quite possible that an informal and partial model is very useful and fairly precise and even quite unambiguous. On the other hand a formal and complete model may be impossible to understand and extremely difficult to maintain. We should use modeling in informatics for the same reason as they use modeling in every other engineering discipline To abstract away the details to bring forth the important aspects. Therefore modeling should be taught on all levels for all students with formal as well as informal means and using advanced tools or using only pencil and paper.

5 Modeling is Different from Model Engineering (Gabriele Taentzer)

Learn modeling with a standard language such as the UML: Modeling is the process to think abstractly about systems. In software development, requirement elicitation and software design are central modeling tasks. Students should get modeling skills as early as possible. They do not need to learn programming beforehand. Usually, students start to learn programming based on a well-known language such as Java. Similarly, they should also start learning modeling based on a standard modeling language such as the UML. The usage of modeling tools is not important in the beginning, but later on when modeling tasks may become larger appropriate tool support should be used.

Model engineering: Once students know how to model software systems, the study of model engineering concepts and techniques should start. Model engineering comprises the definition of modeling languages including syntactical and semantical aspects as well as supporting tools such as editors, interpreters, debuggers, analyzers, and code generators. Essential concepts for model engineering are metamodeling and model transformations. Model engineering can be compared to the definition of programming languages and compiler construction. However, the concepts used in model engineering are completely different to those used for defining programming languages and their supporting tools. The main reason for this difference might be the visual nature of most models compared to textual programs. While the abstract syntax of programs is usually considered to be tree-like, the underlying structure of a visual model does not necessarily have a distinct tree structure but is considered as graph in general.

Model engineering leads to research topics: Model engineering is a developing discipline with a lot of open issues. They range from layout algorithms for visual models to comprehensive code generation ensuring high code quality standards. Therefore, model engineering is well suited to lead advanced students to current research topics.

6 Improve Use of Language, Not Only Vocabulary (Daniel Varro)

Motivation for extra modeling investment: My main involvement in teaching models is giving a Master level course where mostly UML is used as a modeling language, while models are introduced in the context of popular web technologies especially (like J2EE or Eclipse RAP). At this stage, most of the students can be quite advanced in one or more web technologies, thus they need some extra motivation why they should invest in models.

Improve use of language, not only vocabulary: While most students also learned about UML as a language in their previous studies, they still have difficulties in using UML for their projects. In this respect, a main goal of the course is to improve the use of language, and not only UML vocabulary. For instance, linguistic differences like aggregation and composition are quite irrelevant for them compared to even very simple rules of thumb giving guidance in arranging models in diagrams, like "Large class diagrams do not demonstrate that you worked very hard. They only demonstrate that your are using the language improperly." As a summary, it is very important to give some guidelines to students why certain models are good or bad. Carefully selected model fragments play a crucial role in this.

Models serve as primary means of collaboration: As their homework assignment needs to be carried out in teams, they use models are primary means of collaboration. Instead of arbitrary sketches, models are used within teams to drive database design along the object-relational mapping, stateful behavior of business entities, etc. Moreover, models are also very useful in fixing the interfaces between modules developed by different teams (which will then be realized, e.g., by web services technology). Many experts say that frequently, modeling pays off only from the second project when the team members are already advanced users of the modeling language. The use of models as collaboration means gives students the impression that they are already working on something realistic where it is worth investing in the use of models.

Use modeling tools, not only paper: While models are frequently used as pure sketches, I strongly recommend the use of modeling tools as part of a modeling course. In our case, IBM Rational Software Architect/Modeler provides support for effective groupwork by dividing models into smaller parts where each team member may be responsible for easily identifiable bits of the model. Furthermore, from a didactic point of view, modeling tools help enforce certain well-formedness rules of the UML language. It is extremely important to maintain some consistency criteria along different diagrams, which students frequently find difficult to grasp.