Bad Modelling Teaching Practices

Richard F. Paige, Fiona A. C. Polack, Dimitris S. Kolovos, Louis M. Rose, Nicholas Matragkas, and James R. Williams

Department of Computer Science, University of York,
Deramore Lane, York, YO10 5GH, UK.
{richard.paige, fiona.polack, dimitris.kolovos, louis.rose, nicholas.matragkas, james.r.williams}@york.ac.uk

Abstract. There are numerous excellent textbooks that provide guidance on what and how to teach modelling, founded on sound principles (e.g., mathematics, domain analysis, object-orientation) and engineering practice. There is less guidance available on what and how *not* to teach modelling - i.e., what principles to avoid conveying either directly or indirectly, and what engineering practices should be ignored (e.g., because they misrepresent the practices of modelling). We describe a number of bad modelling teaching practices that we have identified, as a result of substantial individual and team teaching experience to both industry and academia.

1 Introduction

Modelling is now taught widely (and in many cases deeply) in both undergraduate and postgraduate curricula around the world. In some cases, it is taught as a cross-cutting concern – that is, as a topic within a wider subject, such as systems or software engineering; in other cases it is taught as a subject in its own right, e.g., as a course in object-oriented design or Model-Driven Engineering (MDE). As educators who teach modelling, we recognise that we are fortunate, in that we have a wealth of resources at our disposal:

– Substantial legacy research and teaching material on data flow modelling, system modelling, architectural modelling, simulation modelling, object-oriented modelling (including patterns) that forms a principled basis for modern approaches to engineering with models.
– Modern standards for modelling languages (e.g., SysML, UML), constraint languages (e.g., OCL) and model management (e.g., QVT, MOF Model-to-Text), some of which are even used in industrial contexts.
– Powerful, robust and reliable toolsets to support modelling and modelling tasks (e.g., EMF[1], MetaEdit+), as well as model management (e.g., Xtext, AMMA, Epsilon, Acceleo, VIATRA2).

[1] http://www.eclipse.org/emf

– Repositories of examples of models, modelling languages, and operations thereupon (e.g,. the Atlantic Zoo[2], REMODD[3], Eclipse).
– Teaching resources, such as Open Model Courseware[4], as well as excellent textbooks (such as [1] and [2]).
– Best practice guides, which exist for particular domains (e.g., automotive systems, financial systems) and for particular organisations, and which inform rigorous engineering use.

There is substantial information and recommendations on the principles, practices, tools and processes that can be used to successfully teach modelling. Arguably, we have so much positive advice and guidance that it is difficult to know where to start, and how. The modelling educators' community would benefit from *negative advice and guidance* – on practices and principles that have been unsuccessful in teaching and educating modellers.

We have substantial negative experience in teaching modelling at all educational levels: from introductory undergraduate through to experienced professional postgraduate. This paper attempts to distill some of the negative experience into a set of teaching *anti-patterns* or *anti-practices* – things not to do! – along with an explanation of why the experience was negative.

2 Bad Teaching Practices

Our teaching anti-practices range from the highly technical, e.g., how tools should not be used, to the pedagogic. We present them in a fairly arbitrary order, and then synthesise some lessons learned at the end.

2.1 Teach to the specification

A particularly demotivating way to teach modelling is to teach to the language specification. A typical instance of this is a course or set of lectures that works its way through the UML specification (e.g., starting with structural diagrams, then behavioural diagrams, then deployment). In essence, the lectures and the exercises that the students may be asked to carry out focus on the language specification (or metamodel), not how the language is to be used to help solve problems. This applies to more than just visual modelling languages – the same issues arise in teaching the use of constraint languages (such as OCL), model management languages (such as QVT), or metamodelling languages (such as MOF or Ecore).

There is a subtle difference between how modelling is often taught and how programming is often taught: the latter very often emphasises use of integrated development environments, debuggers, and compilers – programming is learned by *programming*, using the real tools that professionals apply. Modelling is very

[2] `http://www.emn.fr/z-info/atlanmod/index.php/Zoos`
[3] `http://www.cs.colostate.edu/remodd/v1/`
[4] `http://www.eclipse.org/gmt/omcw/resources/`

often taught by doing modelling, but with reference to a language specification and with use of paper and pencil, rather than by using the real tools that professional modellers apply. Possible explanations for this are that many tools reveal the innate complexity of modelling languages (i.e., the tools expose metamodel details), and the relative immaturity of modelling tools, at least when compared with programming tools.

Rather than teach to the language specification (and teach the boring and exhaustive details of UML, SysML, etc), we should teach to the problem: what are we trying to solve by modelling, and what support do we need – from our languages, tools, processes and practices – to make that happen? We can then more judiciously choose the modelling language features (or, indeed, the entire modelling language that we require) to fit the problem at hand.

2.2 Teach modelling languages deeply, not broadly

Related to the previous anti-pattern, a traditional and particularly flawed way of teaching modelling using a large language (like UML or OCL) is to delve deeply into its features, constructs, syntax and semantics. We often take great joy in doing this when teaching programming languages – a new API or a new and complex language feature that interacts in weird and wonderful ways with existing features is a thing to be dwelled on and investigated. A thorough exploration of each feature of a large modelling language is the wrong way to proceed.

Such in-depth explorations of modelling language features are suitable and important exercises for established researchers or practitioners (who want to understand language design, evolution implementation, etc), but are not what is needed for novices not yet comfortable with modelling. Students need pathways through large and multi-featured languages, particularly grounded by *feedback* on the usefulness of particular features, and of how those features are to be used when solving real problems.

The feedback issue is an important one: when learning how to program, students can get instant feedback on their attempts to use the programming language from the compiler or interpreter/IDE. Superficial feedback on models can be obtained from some modelling tools, but this typically only involves concepts or properties that are missing from the model (e.g., types on attributes) rather than more sophisticated semantic errors or omissions, and interpreting this feedback typically involves exposing the underlying metamodel of the language.

By connecting the *language features* that are introduced to students to the *problem* that is being solved, it may be possible to provide better (and more concrete) feedback to students as to the efficacy of the language constructs that are being used, and the quality of the models that are being produced, in terms of real problems. Of course, a large language can be thoroughly explored over time, by considering different problems from different domains, but using the language to drive such teaching is a demotivating way to proceed.

2.3 Provide answers, not solutions

When teaching introductory courses, we often encounter students who expect to be given answers to problems. The refrain “is this right?” is heard frequently. Giving in to the temptation to provide an answer to a modelling problem – or a design problem in general – is a terribly bad practice.

Why is this? Modelling is an art; different developers arrive at different models from the same starting point. Sometimes the models are of different systems, but sometimes they are just alternative representations of the same system. This presents teachers with a dilemma: the students want answers to exercises, but there are (too) many possible right answers. What is more, the students need to learn the subtleties of modelling through experience, rather than by imitation of one style of modelling.

Consider a very simple modelling problem: *create a class diagram to specify the structures needed for a system that records students (university id, name, email) that create and edit models (title, url, date).* Here are some issues that arise if the students are given an answer, rather than a discussion or a model answer with explanations.

- A common problem in specification is over- or premature elaboration of attribute types. A student with a good knowledge of OO programming starts from concrete classes, and may need to understand the idea of over-specification. For instance, a specification should not tie down a url or date to a specific format, and should not need to consider the accessibility or visibility of classes and attributes. Similarly, a student who understands abstract data types may create too many classes, including type classes to capture the structure of attribute types. A student with a good theoretical understanding of relational databases might simply capitalise the attribute name to represent an abstract type that has yet to be determined. A student following the commendable maxim of doing no more than is asked would not put in any attribute types, because there is no information on types in the scenario. None of these models is wrong, but not all would match a given answer.
- The scenario states that students “create and edit” models. Even if we assume agreement on two classes (*student* and *model*), there are at least two obvious and visually very different, ways to model this:
  1. an association class with one (Enumeration) or two (Boolean) attributes to record whether the link is a creation link or an edit link, and a date attribute to record when the edit or creation occurred;
  2. two associations, named respectively *create* – to link one model to its one or more student creators; and *edit* – to link one model to its one or more student editors. In this case, the *date* information for models might be represented as an attribute of *model*, where it might be the creation date or the last-update date.

  The models are slightly different: in the first, you can generate an edit log, whereas in the second you can only list the students who have created or edited each model. However, both are valid answers to the scenario as stated.

One approach that overcomes the tendency to see the answer as the only possible answer is to get students to create solutions, and then have a seminar-style presentation and discussion which reveals all the variants that were created and discusses which are valid, which model something slightly different, and which are appropriate for different onward uses (modelling does not stand alone). Another approach is to create models "live", by getting the class to suggest what to add next, and engaging in lively discussion of what is and is not acceptable or conventional. A key skill for the teacher is to be able to explain why different valid answers are appropriate for different situations, bringing in both real life and the different phases and activities of software engineering.

2.4 Choose a serious domain

Students both need and benefit from examples when learning how to model. Examples reinforce the conceptual principles and engineering practice of modelling. These examples must be grounded in reality, otherwise students will not engage with them. Realistic examples, like modelling a library, or a bank, or a traffic light system, are serious, pragmatic, understandable to students ... and seriously demotivating – don't use them during the learning process[5]! A problem domain should be chosen with student engagement in mind.

During the stages of learning and reinforcement, selecting tightly constrained or uninteresting domains can limit the number of students who will engage with the problem. Selecting a more interesting, non-standard, or "fun" domain will allow students to become more engaged. For example, choosing a multi-disciplinary problem from fields outside of typical software engineering, such as archaeology, history, or art may give the students opportunities to do interesting research in order to understand the domain and will generate discussion as they begin to understand how these new concepts can be modelled. Granting students the freedom of their imagination can lead to interesting modelling decisions, and a good problem domain can lead to diversity in the solutions offered by students. Diversity is important and discussing the decisions made and challenges faced to get to each solution enables a more exploratory approach to learning how to model systems. A tightly constrained domain doesn't allow for this solution diversity or design space exploration. Once a student has had practice at modelling open and diverse problems, they should be more comfortable in tackling more realistic problems.

One example that we have used in our teaching is computer games. A particular example of this is the *Super Awesome Fighter* system [3], wherein models of players in a fighting game must be created. Such domains are very often immediately understandable and accessible to novices, allow significant modelling (of structure and behaviours) to be carried out, and lead to discussion.

[5] Though they may be suitable for additional exercises or assessment.

2.5 Teach without prerequisites

Some courses have prerequisites; for example, a course in formal methods often requires a discrete maths prerequisite; a course in compiler design may require an automata theory prerequisite. It is poor practice to teach modelling to students who do not have the prerequisites – after all, modelling can be picked up by anyone, right?

Modelling is an advanced software engineering skill. Developing models that are fit-for-purpose requires excellent analytical skills, an aptitude for abstraction, and careful evaluation of trade-offs. To further develop these characteristics, students need to be able to focus on the domain (e.g., "should we name this type Grade or Registration?") and not on notation (e.g., "how do we represent inheritance?") or fundamentals (e.g., "what is inheritance?!"). Fundamentally, there are prerequisites to studying and applying modelling, and these should not be ignored or deprecated.

The issues that arise when students are unfamiliar with modelling notations and fundamentals are exacerbated when a curriculum also includes metamodelling (i.e., teaching students to define and use their own modelling languages). To construct valid, expressive and succinct metamodels additionally requires students to rapidly switch between at least two levels of abstraction: the modelling language, and example models. An extremely strong understanding of object instantiation is required to be able to teach students the skills necessary to construct metamodels.

Arguably, all of the fundamental theory necessary for understanding modelling and metamodelling can be taught via object-oriented programming. UML class diagrams are a useful precursor to construct metamodels, which are normally expressed in the closely related MOF standard. A good grasp of object-oriented programming teaches the fundamentals and limitations of object instantiation. Even some best practices are transferable: structural design patterns (e.g., composite) and refactorings (e.g., extract class) apply to metamodelling.

An understanding of modelling is of course essential for learning about MDE, model management, and the typical engineering operations that are applied to models. The necessary prerequisites for studying, for example, model transformation are a little less clear to us. Certainly students should have programming experience, though it is not clear whether one paradigm or language is particularly beneficial, given the variety of model transformation languages available today. Some exposure to a template-based language (such as PHP) is helpful for understanding model-to-text transformation languages, as is exposure to a language that supports closures or lambda expressions as many model-to-model transformation language provide the first-order logic methods defined in OCL.

2.6 Teach metamodelling using UML as an example

Once students have a basic grasp of modelling, there is an opportunity to introduce them to metamodelling; this is usually done in order to explain how modelling languages are formally defined. Many approaches to teaching metamodelling start with UML. If students are familiar with UML by the time they are exposed to metamodelling, the UML metamodel may be used as a running example to explain metamodelling concepts and techniques. This is a bad way to introduce metamodelling.

In practice, teaching metamodelling using UML can create substantial confusion, given the structural/naming similarities between UML with MOF/Ecore (consider statements such as “UML classes are instances of the Class UML metaclass which in turn is an instance of MOF Class”). Entity-relationship diagrams are not as confusing, however they are still, both structurally and visually, very similar to class diagrams and as such should be avoided for similar reasons.

So what is an alternative to introducing metamodelling via UML. It would be beneficial to find a domain with little structural and lexical overlap with metamodel-level concepts, and also one in which example-based approaches can be used – viz., small example models that can be used to motivate the development of small metamodels. We are currently using flowcharts to introduce students to metamodelling, as a typical flowchart metamodel consists of concepts such as actions, decisions, transitions and labels.

2.7 Learning to use the tools is trivial

To teach modelling and engineering practice, we benefit from using tools (just as we benefit from using tools to teach programming). Assuming that modelling tools are trivial to learn and employ is our next bad teaching practice.

We often assume that students will be able to pick up, understand and use modelling tools without much difficulty. We often infer from student experience with programming tools (like IDEs and debuggers) that they will have few problems learning, for example, Eclipse-based modelling tools or EMF. But the way users must think about modelling tools differs from programming tools: the former expose students to many different views of a model or modelling problem (as opposed to code, which very often has a single view – the source – possibly extended with debugging views). Also, generic tools such as Eclipse have complexity that is not technically relevant or important to students learning modelling, but cannot easily be hidden from them. This means that it can be dangerous to assume that modelling tools will be straightforward for students to pick up and use. We should acknowledge this in the early exercises we give to students, in the amount of hands-on help we give them in the early stages of courses, and in terms of our expectations – how much progress do we expect students to make early on?

We must also consider how students learn about tools. Increasingly, students rely on resources such as e-books and online video tutorials (e.g., YouTube) to explore new tools, and on developer forums (e.g. StackExchange) for troubleshooting. Due to the somewhat limited use of modelling tools (and especially MDE tools) by mainstream developers, there is a lack of resources for modelling technology, which can hinder the learning process, and even undermine the credibility of tools in the eyes of the students. To address this challenge, tutors need to invest a significant amount of effort to produce or collect documentation and examples that can enable and support the students to learn the tools at their own pace.

2.8 Teach in a vacuum

Modelling can be taught as a pure and self-contained subject – as a theory with laws and rules, with little or no relation with the outside world. This is bad practice, as it ignores engineering context and practices – that is, it ignores the *purpose* of creating models.

In real-life scenarios, modelling principles and tools are used in conjunction with other software engineering approaches to solve complex problems [4]. On the other hand, modelling is typically taught with a strong focus on the technologies comprising this space, without providing adequate references to the "big picture", i.e., the software engineering activities, processes and management practices used simultaneously with modelling (and possibly MDE), as well as the exact problems it tries to solve.

The "big picture" constitutes the context of modelling and MDE. Following [5], context is "the circumstances in which something exists or occurs". Therefore, a modelling course should include explanations on the application scenarios, the related software engineering tasks, and any existing alternatives to modelling. According to [6] context is very important to the human learning process, since it provides meaning and motivation to learners. Moreover, [7] further argues that context and the particulars of that context can provide a powerful motivation for learning.

In addition to providing the real-life context of modelling, educators should position modelling and MDE in the computer science curriculum. More particularly, areas of modelling are conceptually very close to other computer science domains. For example metamodelling has many similarities to database schemata or ontologies. During a modelling module, such correspondences should be highlighted, so that students can gain deeper understanding of the concepts by relating them to already known concepts.

2.9 Code generation is the entry-level drug

A typical question we are asked by students is: once you have models, what should you use them for? Besides talking about communication, evaluation, validating different design options, etc., we often motivate use of modelling to students by talking about code generation as a 'primary use case'. Modelling and MDE are often sold to industrial users with the sole purpose of automatic code generation. This perspective implies that MDE in particular is only applicable to software engineering. Not only is this a needlessly narrow view, conveying it is bad teaching practice. In theory and practice, models and MDE tools and techniques are applied to many more use cases, such as scientific simulation, business intelligence and decision support.

Furthermore, code generation is an arguably over-used tool in the modelling toolbox. After seeing its benefits, many students new to modelling and MDE reach for model-to-text transformation to solve the majority of their implementation problems, without considering alternatives.

A common issue that we see in the classroom (and in the wild) is the use of code generators to bridge large semantic gaps. When the input model and the target code are at substantially different levels of abstraction, large and complex code generators are typically required. Model-to-text (M2T) languages are typically not well suited to complex transformations (reasons for this are minimal support for modularity and incrementality), and as a consequence complex code generators are difficult to develop and even more difficult to test and maintain. As a community, we must be careful to teach techniques that work well in practice: code generators that span large semantic gaps do not work well in practice.

Notwithstanding the downsides of code generation, we note that it does provide a tangible motivation for modelling for some students, particularly those who have industrial experience of developing software for legacy or proprietary systems.

In short, we feel that presenting code generation as a secondary scenario for exploiting models is helpful and sensible, and can help to prevent students from falsely assuming that modelling necessitates code generation, or that MDE is limited to the domain of software engineering.

2.10 Reinforce the concept of silos

We have to compartmentalise when teaching modelling: we have to teach syntax, semantics, tools, best practices, domain modelling etc, and have to break this up in some way. Doing so in a way that reinforces siloed thinking in our teaching is poor practice.

When we teach in software engineering or computer science, we have to draw the boundary somewhere: only some topics are within scope (and feasible to discuss) in a course. But there is substantial potential for harm if we *silo* modelling too much, by minimising reference to related topics. In particular:

- Teaching modelling as a separate subject, e.g., focusing on the languages, rather than as a concept that pervades and underpins engineering.
- Ignoring socio-technical issues when teaching modelling, e.g., the processes – both business and technical – in which modelling are carried out.
- Ignoring team issues: modelling in the wild is a team sport, and receiving feedback from team members on the quality, understandability and usefulness of models is critical. At the same time, learning how to work in a team, and appreciating the communication issues that pervade modelling, is an important skill.
- Teaching modelling as an individual: many people learn by doing and by observing how more experienced practitioners behave. This is very much the case for modelling. So, rather than having *one* person teach modelling to

students, have a *team* of people teach modelling – the students can then see how the team of teachers resolve conflicts, deal with understandability issues, and communicate effectively and constructively to solve problems.
– Teaching modelling without reference to other disciplines: modelling existed well before MDE and software modelling. Reference to other disciplines or domains helps to reinforce both that modelling is an accepted engineering practice, and that software modelling should not be thought of as being in a silo by itself.

2.11 Modelling is for university students

Modelling as a subject is often taught late – perhaps in the second year of a university Computer Science or Software Engineering course. Education (in university as well as at pre-university level) sometimes has a tendency to teach skills, not problem solving. Students are taught to program, solve equations and proofs, do unit testing, etc. There is sometimes a principle stated that a student cannot design or engineer anything until they have the skills to create the product itself. Thus, modelling is either taught as a stand-alone skill, or ignored completely. This is bad practice all around.

In software (and hardware) engineering, modelling can be used to think about and explore a domain, as information for planning and resourcing, and as documentation for any stage of development – as well as the conventional model-driven engineering activities. Encouraging students to draw ideas is helpful, and stimulates creativity. Encouraging students to then make their sketches more formal, or conformant to established notations, seems such an obvious next step that it is surprising that it is not included in all core areas of computer education. Modelling then becomes the basis from which to apply other engineering-related skills – not just programming, but also testing, simulation and visualisation. The meta-issue here is that we should avoid giving the impression that modelling is an academic exercise unrelated to engineering in practice.

2.12 Physical decomposition is all

Very often we model a system: software, a car, an enterprise, a business process. The system of interest may be complicated or complex, involving many different parts that interact in different ways. Decomposition is one of the fundamental techniques that we teach our students for managing complexity and controlling the engineering process. It is a bad practice to teach decomposition superficially: to not provide guidelines to students in terms of identifying purposeful decompositions. We pay lip-service to various poorly understood metrics (like coupling and cohesion), or important architectural styles (like layering). We also refer very often to analogies, particularly where physical and cyber-physical systems are concerned.

In such situations we might teach that *physical decomposition* is the most important way in which a problem can be sensibly broken up, but this is bad practice. Do we teach the consequences? That cross-cutting concerns, such as safety or end-to-end performance can be neglected or treated as afterthoughts? In enterprise architecture, there is consideration of system-level properties and how such properties cut across 'layers' in an architecture, but in many traditional modelling disciplines that use standard languages like UML or SysML, cross-cutting concerns are rarely a consideration. Physical decomposition is easy to grasp and can lead to a sensible solution, as long as cross-cutting properties either don't matter, or can be treated later, when modelling is conveniently out of the way.

2.13 Ignore semantics

Teaching modelling as a purely syntactic effort, focusing on language constructs in detail whilst ignoring semantics, is our final bad practice. Such efforts produce practitioners and researchers who do not use the full power of modelling and modelling languages.

The semantics of modelling languages is an advanced topic in MDE [8]. There are misconceptions and disagreements on semantics even between MDE researchers. Despite the topic's complexity, semantics should eventually be part of a modeller's education, since it is an indispensable part of language engineering. By having an understanding of semantics, students can avoid common misconceptions, for example, that the semantics of a modelling language is its abstract syntax or that a model cannot have more than one interpretation; this in turn reinforces engineering practice, such as that models are a critical tool for communication and discussion. Moreover, an understanding of language semantics can lead to the realisation that ill-defined semantics are the root of ambiguity. Admittedly, studying semantics can be daunting for the mathematics-averse students. Depending on the level of the module, semantics can be taught in a formal or informal way. In any case though, semantics should not be ignored.

3 Observations and Conclusions

What are some of the lessons that we can synthesise from these teaching anti-practices? Are there specific good ideas that we should consider in teaching modelling? We think there are five key points:

- *Integrate modelling in the curriculum.* Modelling pervades engineering and science and it should be taught as such. Using a standard vocabulary for talking about modelling to students (whether at high school, undergraduate level or beyond), demonstrating different types of modelling (e.g., mathematical, data, behavioural), and focusing on its use in problem solving conveys to students that modelling is not a siloed subject. Let's get rid of the standard UML modelling course, and teach modelling as a cross-cutting concern.
- *Problem-based learning.* Modelling for modelling's sake is the remit of the researcher and tool builder. But when learning how and why to model, the subject needs to be grounded in real (but not boring and dull!) problems

that have many possible solutions. This can convey both the engineering importance of modelling, but also the value of feedback in modelling.
- *Preparedness.* In software engineering, modelling is founded on some basic concepts: encapsulation, identity, types, instantiation, properties, references, constraints. Students should have a basic familiarity and some practice with these concepts – for instance, through introductory programming courses – before they are taught to use modelling in anger.
- *Tools get in the way.* We should anticipate that the modelling tools that experts and professionals use on a daily basis – like EMF – get in the way of the novice modeller, and our exercises, laboratory work and assessments should accommodate for this.
- *'Fun' problems.* We should use modelling problems and exercises to support creativity in our students. Fun modelling problems that require students to investigate a topic new to them help convey the joy of research, the flexibility of modelling, the importance of explanations, and the value of models in supporting communication between different groups of stakeholders.

These lessons, and others, form the basis of how we now teach modelling at York to both undergraduates and Masters-level students.

Acknowledgements

This research was part supported by the EU, through the OSSMETER FP7 STREP project (#318736) and the MONDO FP7 STREP project (#611125).

References

1. Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice.* Morgan & Claypool Publishers, 1st edition, 2012.
2. Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.
3. James R. Williams, Simon M. Poulding, Louis M. Rose, Richard F. Paige, and Fiona A. C. Polack. Identifying desirable game character behaviours through the application of evolutionary algorithms to model-driven engineering metamodels. In *SSBSE*, pages 112–126, 2011.
4. John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.
5. Steve Cooper and Steve Cunningham. Teaching computer science in context. *ACM Inroads*, 1(1):5–8, March 2010.
6. Tim DeClue. A theory of attrition in computer science education which explores the effect of learning theory, gender, and context. *J. Comput. Sci. Coll.*, 24(5):115–121, May 2009.
7. D.H. Jonassen and S.M. Land. *Theoretical Foundations of Learning Environments.* L. Erlbaum Associates, 2000.
8. David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72, October 2004.