Electronic Communications of the EASST
Volume 52 (2011)

7th Educators' Symposium @ MODELS 2011: Software Modeling in Education (EduSymp2011)

Teaching Programming Students how to Model: Challenges & Opportunities

Robert France

2 pages

Guest Editors: Marion Brandsteidl, Andreas Winter
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Teaching Programming Students how to Model: Challenges & Opportunities

Robert France

Department of Computer Science
Colorado State University
Fort Collins, CO80532, USA
france@cs.colostate.edu

Abstract: Computer Science students who have one or more years of basic programming experience and little or no exposure to abstractions above the code level, often struggle to grasp modeling concepts and practices in more advanced software development courses. In this talk I discuss some of the factors that contribute to this situation and propose a learning process in which the abstraction gap between models and implementations is gradually widened as student modeling skills are developed

Keywords: model-driven development, modeling-in-the-small, modeling-in-the-large, teaching software modeling concepts

1 Overview of Talk

The focus on programming abstractions in early programming courses may be a contributing factor to the difficulty some programming students have with mastering software modeling concepts. Students that have invested significant effort in learning how to think about solutions using only abstractions provided by programming languages, may find it difficult to let go of those "low-level" abstractions and think about problems and solutions in more abstract, but still rigorous terms. This view of the source of student difficulties is somewhat optimistic. It presumes that students have achieved a level of programming mastery that makes them comfortable with program-level abstractions. In my experience, student difficulties can also stem from their struggle with identifying and using appropriate abstractions, including programming abstractions. While students may have developed some mastery of programming language syntax, they may not yet have grasped how to program. Most educators would agree that learning how to use a particular programming language is relatively easier than learning how to program. If students have difficulty formulating solutions using programming abstractions, then they are likely to struggle with identifying and using more abstract concepts that can be used as the basis for building well-designed programs (i.e., programs built using good programming abstractions).

Its interesting to explore the source of difficulties from a student perspective. Many of the students who discussed their problems with me feel that modeling adds accidental complexity to software development. They often point to the steep learning curves associated with modeling tool suites, and the difficulty of determining the "goodness" of a model they create. On the latter point, students find it difficult to determine whether the abstractions they use in their models are

appropriate or "fit-for-purpose". Students also have difficulty determining what information to include and not include in a model. In some cases students use models as an excuse to think informally about a problem or solution, often omitting critical information needed by a model to fulfill its purpose. Whilst this may be appropriate when using models as sketches, it is not appropriate for more rigorous use of models (e.g., use of models to produce implementations or to formally analyze functional properties).

In addition to the above, students often question the merits of learning modeling techniques. From their perspective, modeling becomes relevant if it makes the programming task easier, but this is not an experience they all have. In addition, modeling becomes relevant if it is viewed as a marketable skill. In my classes, some students took a deeper interest in modeling, not because I convinced them of its merits, but because company recruiters started to ask student job seekers about their modeling background.

In the past I used a waterfall-like approach to introducing modeling concepts, starting from requirements modeling and drilling down to detailed design modeling. This top-down approach to teaching modeling is supported by popular modeling textbooks. It has become clear to me that having students begin their modeling experience at an abstraction level that is far removed from programming abstractions can be counterproductive. Top-down modeling approaches can overwhelm students whose previous experience consists solely of developing small programs with fully specified requirements. In my classes, students now start by developing what I call modeling-in-the-small (MITS) skills, and then move to developing more advanced modeling-in-the-large (MITL) skills. MITS focuses on the use of models to describe to describe program designs. In this case the abstraction gap (between the program design model and the program) is small and thus less challenging for the students to bridge. In the MITS part of the course students are given stable, well-defined requirements of small programs and are required to build program design models from which implementations can be generated. In the MITL section, attention turns to modeling larger systems with problematic requirements statements, and that have architectures consisting of three or more non-trivial subsystems.

The notions of MITS and MITL can be extended across programming courses in a Computer Science program. Introductory programming courses can focus on developing MITS skills hand-in-hand with the development of programming skills, while more advanced software development courses can focus on developing MITL skills.

To support effective development of MITS and MITL skills it would help to have the following available: (1) Modeling patterns and anti-patterns that distill expert modeling experience, (2) A repository of models that illustrate good and bad modeling practices (some are already available in the MDD repository, ReMoDD - http://www.cs.colostate.edu/remodd), (3) Text books that focus on developing modeling skills rather than on covering syntactic and semantic language concepts, and (4) Lightweight modeling tools that tolerate incompleteness and support exploratory design.

I'll conclude with a hypothesis that is based on my personal observations: *A good software design modeler is a good programmer; a good programmer is not necessarily a good design modeler*. Formal empirical studies that test this and other similar hypotheses are needed if we are to effectively develop student modeling skills.