Softw Syst Model (2009) 8:163–164
DOI 10.1007/s10270-009-0117-4

EDITORIAL

Why Johnny can't model

Editorial for the SoSyM Issue 2009/02

Robert France

Published online: 24 March 2009

I recently taught an undergraduate C++ course for second year undergraduate students. Teaching this course provided me with some insight into why some students have difficulty grasping the abstraction and modeling concepts introduced in the advanced software development courses we offer to third and fourth year students.

While grading their programming assignments it became evident that students were not thinking of their solutions in an object-oriented (OO) manner. The students were becoming skilled at stringing together C++ statements to produce working programs, but they were having problems thinking about solutions in terms of collaborating objects. Many students used objects more as passive maintainers of data rather than as active participants in a collaborative effort to accomplish functional goals. For example, classes with references to other classes were rare; many classes had only basic get and set methods, and had attributes that uniquely identified objects. What was surprising was the extensive use of globally declared data structures in some of the programs.

If students do not understand how to effectively use OOP concepts to solve problems then it is not surprising that they have problems building good models of OO solutions. Their underdeveloped OOP skills make it difficult for them to distinguish good and bad OO abstractions.

The link between modeling and programming skills should not be undervalued. A good modeler should also be a skilled programmer. A great modeler is invariably an expert programmer. On the other hand, good programmers are not necessarily good modelers. Highly-skilled programmers may rely on mentally-held patterns and abstractions when constructing programs, but using those patterns and abstractions to produce good models is an acquired skill.

Good programming knowledge should not be equated to good knowledge of the syntax and semantics of a programming language. A few students in my C++ class did have good knowledge of the C++ syntax and semantics, but they also had poor OOP skills. Learning how to program is not the same as learning the syntax and semantics of a programming language. Similarly, learning how to model is not the same as learning the syntax and semantics of a modeling language such as the UML.

As an analogy, consider how abstractions are developed and used in mathematics. For example, Category Theory provides abstractions over mathematical structures (e.g., sets) and their relationships (e.g., functions). The developers of these abstractions had in-depth knowledge of the mathematical structures (including in-depth knowledge of their manipulations) they were abstracting over. Few will argue that effective use of these abstractions requires at least a good understanding of the mathematical structures that the theory abstracts over. Attempting to use Category Theory without such knowledge can confuse rather than enlighten.

Given the above, how we can better lay the foundation on which students develop good modeling skills? We should certainly continue to expose students to programming concepts as early as possible in the curriculum. A problem with many introductory programming courses is that emphasis has been more on covering programming language syntax and semantics at the expense of material that addresses how abstractions provided by a programming language can be used to develop good quality solutions. The growing complexity of programming languages is partially to blame for this state of affairs, but I would argue that teaching students how to program trumps the need to expose students to a wide range of program language features. The decision to cover a language

R. France (✉)
Colorado State University, Fort Collins, CO, USA
e-mail: france@cs.colostate.edu

feature in a lower-level programming course should be driven by a desire to expose students to (1) an approach for building good solutions using the feature, or (2) problems that can be encountered when the feature is used inappropriately.

The use of software models to visualize coded solutions can help enhance the learning experience of novice programmers. For example, sequence models can be used to visualize different ways of distributing functionality across classes when discussing on the pros and cons of alternative solutions. Students can also use these models to plan how they will distribute functionality across the classes. The good students I have encountered tend to think first in terms of behavior, and they find it more useful to develop sequence models before developing class models. After they have figured out a "good" way of distributing functionality they are better able to construct a supporting class model.

I am encouraged by the growing number of introductory programming texts that do use models to visualize code. At the same time, I am disappointed by the lack of variety in the models used: Most introductory textbooks limit their use of models to visualizing structure using, for example, UML class models. Very few make good use of sequence models.

Students entering more advanced software development courses should be expected to have a good understanding of programming and familiarity with basic software modeling notation. If we broaden the focus of introductory programming courses as outlined above, we are more likely to have students that are better prepared for advanced courses that go beyond the use of models to visualize code.

Contents in this issue

In this issue, we present seven papers that present interesting research results.

In the regular paper "Redesign of UML Class Diagrams: A Formal Approach" by Piotr Kosiuczenko, a formal method for refactoring OO specifications in a manner that preserves specified properties is described.

The regular paper "Qualifying Input Test Data for Model Transformations", by Benoit Baudry, Franck Fleurey, Pierre-Alain Muller, and Yves Le Traon, tackles the difficult problem of testing model transformations. The authors propose a set of framework for assessing the quality of input models used to test model transformations.

The regular paper, "Use Case Maps as a Property Specification Language", by Jameleddine Hassine, Juergen Rilling, and Rachida Dssouli, describes how use case maps can be used to describe property specifications. Property specification patterns are used to bridge the gap between users not skilled in writing properties in temporal logic and model checking tools.

In the regular paper, "Reusing Semi-Specified Behavior Models in Systems Analysis and Design" the authors Iris Reinhartz–Berger, Dov Dori, and Shmuel Katz, present a reuse approach using OPM, which combines structure and behavior (objects and processes) in a single diagram. OPM enables reusing behavioral modules and organizing their dynamic aspects into complete applications. A set of inter- and intra-weaving rules determines how to define and how to combine reusable generic modules.

In the regular paper "A UML and OWL Description of Bunge's Upper-Level Ontology Model" the author Joerg Evermann goes back to a high-level ontology which was published by Mario Bunge in the 1970s. This is a very general ontology which tries to cover most real world phenomena and therefore is an interesting candidate for a foundation capable of supporting various languages and concepts. The paper formalizes this ontology in UML and OWL.

In the regular paper "Variability Modeling for Questionnaire-based System Configuration" the authors Marcello La Rosa, Wil M.P. Aalst, Marlon Dumas, and Arthur H.M. ter Hofstede, a formal foundation for representing system variability for the purpose of generating questionnaires that guide users during system configuration is presented. The generated questionnaires are interactive, in the sense that questions are only posed if and when they can be answered, and the space of allowed answers to a question is determined by previous answers. The approach has been implemented and tested against a reference model from the logistics domain.

In the regular paper "A UML-based Quantitative Framework for early Prediction of Resource Usage and Load in Distributed Real-Time Systems", the authors, Vahid Garousi, Lionel Briand, and Yvan Labiche, present an approach that supports analysis of resource usage in distributed real-time systems. The prediction is done on models consisting of UML sequence diagrams adorned with additional timing information.