Softw Syst Model (2012) 11:1–2
DOI 10.1007/s10270-011-0224-x

EDITORIAL

On the relationship between modeling and programming languages

Editorial for the SoSyM Issue 2012/01: Part 1

Bernhard Rumpe · Robert France

Published online: 18 December 2011

At the MODELS 2011 conference in New Zealand, Colin Atkinson, held a panel on "When will Code become Irrelevant?". The panelists were requested to answer the following six questions:

1. Do you agree with the implied assumption in the panel abstract that "code" is different to "models, at least in the minds of practicing software engineers?
2. If so, do you agree with the premise that code is still the primary artifact, or at least still an important artifact, in software engineering?
3. If so, do you think it matters?
4. If so, when if ever, will the situation change and what will it take to make it happen?
5. What can the research community do to help bring this about?
6. What will the future of modeling look like? Does modeling have a future as an independent activity or will it fade away in importance?

Software developers create and use models for a variety of purposes. For example, the following are three common forms of uses:

- Models as sketches: Developers find it useful to sketch descriptions of requirements, design or deployment concepts on whiteboards or paper when discussing their ideas with other developers or customer representatives. This use of models supports exploratory development of concepts and ideas that may or may not later find their way into more formal models or implementations.
- Models as analysis artifacts: Developers build analyzable models to check specified properties (e.g., consistency and satisfiability properties), to predict implementation qualities (e.g., performance), or to simulate implemented behavior. Included in this category of models are formal, non-executable specifications that can be statically analyzed, and executable models that support more dynamic forms of analysis. These models typically contain only the information needed to analyze target properties, and thus may not include information that is needed to generate full implementations.
- Models as the basis for code generation or synthesis of software artifacts: Models can be built for the purpose of generating implementations, test cases, deployment or software configuration scripts, or other software artifacts. These models must contain all necessary information in a form that allow a generator to mechanically synthesize software artifacts.

B. Rumpe (✉)
RWTH Aachen University, Aachen, Germany
e-mail: Bernhard.Rumpe@sosym.org

R. France
Colorado State University, Fort Collins, Colorado, USA

One of the most important uses of modeling techniques and languages is to build executable models. Executing a model often involves code generation. From a practical standpoint it makes a difference whether the developer's intent is to produce industrial-strength code when executing a model, or to produce code that animates modeled behavior to provide the developer with feedback on the adequacy of the model., One can expect that in the first case the generated code is of better quality (e.g., more robust) than the code generated for solely animation purposes. One can also distinguish between code generation in which modeled behavior is hardwired in the implemented system, and generation of implementations that are configurable at system initialization or even at runtime.

The constituents of an executable model are not necessarily restricted to behavioral models such as Statecharts or activity diagrams. Structural models, such as UML class diagrams and composition structure diagrams can also contribute to the execution of models. These models provide information on the structures that are manipulated by behaviors and thus their elements contribute to the generation of data and information structures in implementations.

Modeling languages that support the building of executable models can be viewed as approximate forms of very high-level programming languages, that is, languages above the current high-level general-purpose programming languages (GPL). We use the term "approximate forms" because we do not consider that current modeling languages can completely replace or mask out GPLs; there may be particular algorithmic elements that probably cannot and should not be modeled in a more abstract way than through statements in a GPL. The term "very high-level" is used to reflect our view that models are more abstract and compact than implementations expressed in a GPL. For example, technical details (e.g., details related to efficient implementations of complex data structures) and domain-specific functionality can be added through intelligent code generators and therefore need not be present in the model used to generate the code.

If we look at today's coding techniques, we can identify quite a number of best practices that are based on abstractions that are more conveniently represented in design models. The state pattern (GOF 1993) provides one such example. Extracting these abstractions from best practices and incorporating them into modeling languages can help promote the use of these best practices. Furthermore, the explicit representation of these abstractions in modeling languages makes it easier to maintain, evolve and reuse the abstractions. The development of modeling languages can thus benefit from an analysis of best coding practices. Below we give our view on why this can be challenging.

First, there is currently no successful tight integration between modeling and programming technologies. Modeling tools are typically heavy weight and not easily integrated with others software development tools. They are heavyweight in that they force developers to describe modeled elements in a detailed form, before code generation can be performed. Furthermore, developers often have to augment the generated code with manually written code fragments, and thus they need to read and understand the generated code in addition to the original model. Another problem is that many tools do not support automated synchronization of models and generated code when changes occur in either the code or the models. If developers manually modify generated parts of the code, the lack of automated synchronization could result in a loss of fidelity between the model and the code implementation. This makes re-generation very difficult, if not impossible. What we propose is needed are lightweight approaches for the integration of models and implementations expressed in a GPL that allows developers to manage both models and GPL programs without directly viewing and manipulating generated code. To our knowledge, current tooling does not support this. While modern IDEs allow developers to effectively manage source code, they do not provide effective support for generating code from models, primarily because they handle generated code in much the same way as hand written code. This frequently leads to problems with refactoring as well as versioning, and can complicate the building process.

Another major problem that the modeling community currently encounters is the lack of modularity. Good modularity allows a developer to encapsulate strongly related design concepts in a single code module (e.g., a procedure or class) and to expose only the information needed by other modules through an interface. This is the major driving force for incremental compilation, reuse, use of code libraries, dividing work amongst team members, and therefore provides the means to scale development projects. With the exception of Matlab/Simulink, today's modeling languages do not support the definition of an "interface" for a model. The lack of interfaces leads to a different handling of models during the development process. Tools typically load a single large model in their workspace (we can be glad that we have enough space to actually do this) and therefore expose developers to all the details of the model. Any change in the model, no matter how small, typically enforces a complete regeneration. Models are thus monolithic entities that expose all details to developers. This makes reuse of models challenging and gets in the way of developing libraries of reusable model fragments. As long as we are not able to deal with models in a modular way, the cost of developing and using models of complex systems may outweigh the benefits of using the models in the development process.

They are other aspects of code management that can be considered when developing effective model management technologies, namely, differencing, merging, versioning, management of variability and refactoring. In summary, we feel there is a need to provide a tighter integration of modeling and programming technologies if we are to see more widespread use of modeling technologies in practice.