Softw Syst Model (2013) 12:665–668
DOI 10.1007/s10270-013-0383-z

EDITORIAL

Why it is so hard to use models in software development: observations

Robert France · Bernhard Rumpe · Martin Schindler

Published online: 11 October 2013

In a previous editorial, we asked what modeling contributes to the development process. We concluded that models are not so much of value for themselves, but exist to improve certain properties of the product, such as quality or maintainability, or of the process, such as cost-efficiency and predictability.

In this editorial, we would like to report on specific findings that are based on the reported and own experience with some concrete modeling tools and frameworks of different types (without naming them) and draw some conclusions for further tool improvement.

The basic idea is simple: create a model in a source language (e.g., a class diagram or a state machine), take a code generator, and generate the code for it in a target language. The idea behind is that the model is much simpler and thus easier to write and assess than the resulting code. Some of the reasons are that a model can abstract from the implementation details as well as from the technological platform as well as from unnecessary details of the application domain. We can "measure" this by comparing the size of the model with that of the resulting code. And this indeed works and helps.

But there are a lot of practical problems. Let us classify and briefly discuss some of them:

Measurement of speedup: Comparing LOC of the generated code with the number of syntactic concepts (or something similar) used in a model is not easy: the size of the generated code can be impressive. But how much of it do we actually use? How much would we actually write, if written by hand? How much effort goes into the development or adaptation of the generator? How much more familiar are we with the target language compared to the modeling language and the code generator? Do we know a good metrics for speedup through the use of modeling tools? Does speedup only occur in similar follow-up projects?

Integration of generated and handwritten code: One-shot generators allow us to modify the generated code. This is typical for code frames derived from class diagrams. Better tools try mapping back and forth using round-trip engineering. This, however, seems to work to a very limited extend only, especially when the conceptual distance between the languages is small and appears to be somewhat brittle if unexpected modifications occur. A third approach is to not expose the generated code to the developer. While this is similar to the compiler approach, where assembler code is not seen any more today, this approach does not work yet: first, handwritten code usually must know about the generated code. So developers have to inspect at least parts of the generated code. This code inspection could be reduced by generating "interfaces" for all APIs to be available and to describe the generated code by some JavaDoc-like mechanism. Second generation becomes useful when the generator knows something about the technology stack being used. It ideally encapsulates this stack such that the normal developer does not have to know about it anymore. This requires the generation process often to be adapted by certain key developers that thus need to be able to understand the generated code.

Selection of the target language and their concepts: The more elaborate the target language and its frameworks, the better. Generating Java is much easier than generating Assembler.

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

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

M. Schindler
e-mail: schindler@se-rwth.de

But how many concepts should be used? Using visibilities, type infrastructure or even generics in the target code can become a nightmare when some concepts of the source language, such as dynamic extension of the attribute section or type adaptation, do not fit. Encoding these concepts can become tough when the underlying infrastructure should be retained. This is why some generators use reflection when targeting Java: this is easier and more systematic for the generator, but lacks understandability. If the generated code is invisible for the developer, it need not obey any coding guidelines. If the generated code is "correct by generation" anyway, it need not deal with encapsulation (visibility) issues or the underlying typing system at all. It could directly create untyped, generic code or use reflection when needed, quite like assembler, and may be mimic its own typing structure. But what if the code is not correct or the generator needs to be adapted?

Modularity of the generated code: Encapsulation of the generated code is necessary to prevent the developer having to inspect it (at least if he/she is only using it) and allows us to repeatedly generate if the model changes. We discussed this in detail in the editorial in issue 12–3 because we feel this is a very important and for modeling languages still not a very well-understood concept.

Architecture of the System: Let us assume that generated code remains unchanged. How do we organize the system: traditionally, systems are defined in subsystems sharing common data structures and communication and are well structured in packages. Furthermore, architectural styles such as the layer architectures allow us to organize the application core independently of any GUI and the persistence. Keeping the architecture clean when using code generation seems to require a lot of discipline. But is this still an appropriate architecture or do we need to come up with a different approach? Please note, we discuss the dynamic object architecture at runtime, the class dependencies at compile and generation time as well as the package structure. The last two directly relate to the question, whether handwritten code can depend on generated code and vice versa.

Generic versus generative codes: Developers tend to avoid generating clean and powerful solutions but to introduce generic, handwritten solutions instead. Introducing a generation process is generally more work than just coding a generic solution. With today's languages—here, we refer, e.g., to Java—generic solutions often become quite handy. Mechanisms, like subtyping polymorphism, generics, and powerful and dangerous reflection, allow developers to do quite efficient things, with simple concepts like design patterns up to Web frameworks with reflective examinations of plain old Java objects (POJOs). When the languages, frameworks, and tooling infrastructures become ever more generic, do we then need generation at all? The state pattern is a direct explicit representation of automata in code. Is it readable and can we omit generating code from state machines? Can we explicitly encode more modeling languages?

How to organize the development process (methodical part): The classic activities of software development need to be rethought if the coding activity does not happen anymore. How do we ensure quality? What needs to be tested? What are testing metrics? Ideally, the "executable" model replaces the source code roles and should therefore be used as target for quality assurance. Instead, our testing techniques mainly focus on the code and therefore test developers (a) have to understand the generated code and (b) much more tests are necessary to cover the generated code than to "cover" the source model. However, this is necessary anyway, if the generator is adaptive (see above).

How to organize the development process (technical part 1: automation): Automation is the key for speeding up the development process. First, to be able to repeatedly execute the generation, the generation process needs to be fully automated, e.g., through a script, an Ant configuration, or a traditional makefile. While those scripts nicely cooperate, e.g., with C compilers, modern languages like Java come with semi-intelligent compilers that handle dependency management on their own. This collides with dependency management for source generation. Furthermore, there is neither a logical rule nor an explicit connection between the model and their resulting generated artifacts (which can be seen only by examining the code generator). Therefore, it is not externally derivable what needs to be generated again when parts of the models change. Thus, efficient scripts are much harder or even impossible to write. In practice, this leads to heavy and complete regeneration or a generation that needs to be controlled manually instead of a regeneration on demand only. The first seriously slows down the development. The second is very error-prone, as people tend to forget some manual generation steps.

How to organize the development process (technical part 2: dependencies): Another problem is that the dependencies are not only to be maintained for the generation of files, but also for temporary cleanups. A full cleanup of all generated classes in each generation step completely spoils agile development speed. Dependency management needs to be part of a generator. This must also work, e.g., in scenarios, where many classes are generated from a class diagram. When a class description is removed from the diagram, the regeneration needs to remove the old (generated) class too, but the diagram itself does not contain any information about the old class anymore. Thus, dependency management needs to be

intelligent and needs to orchestrate all generation and compilation processes.

How to organize the development process (technical part 3: version control): Do not put generated files under version control! When you want all developers to be able to regenerate, then conflicts on files generated under version control cannot be avoided. One solution could be to put the generator under version control or offer it as commonly and automatically usable Web service such that each developer is able to regenerate locally and generated files need not be version controlled. A second best solution is that only one developer is allowed to regenerate. This seriously complicates the development, when all developers are allowed to adapt the models. We also saw an alternative, where the generation process is done through the version control system: (1) check in the source model, (2) a demon in the back does the generation, and (3) if successful (4), the generated code is checked in, too. That also is a misuse of the version control as it affects other developers before the code is being cleared. So, not putting generated code under version control is the only useful possibility in team-oriented development projects.

Generation using templates: yet another language: In order to make generation adaptable, template languages such as FreeMarker seem to be the appropriate tool. This introduces a third language to be dealt with. Developers now need to master models, target language, and FreeMarker templates. Templates to some extent look like the target with extensions. FreeMarker uses callbacks to Java. Java also has to provide the underlying metamodel (abstract syntax) of the model. The template language also needs control elements (loops, if-then-else, template inclusions) as well as handling of the data (typically strings, lists, maps). FreeMarker would allow complex computations, but they should not be written there to keep templates and therefore the code generator readable and understandable. Instead, these computations should be provided by the underlying metamodel. This enforces the developer to become a tool smith and also define some Java code for the metamodel. But is there a simpler yet powerful alternative to templates or similar scripting languages when generating source code?

Organizing templates: The metamodel, extra Java functionality on top of the metamodel, and the templates exhibit a complex interplay that has to be mastered. Templates easily become rather complex, in particular when alternative cases or iterations need to be handled. Thus, templates need to be structured into hierarchies of subtemplates. Accordingly, the interplay between templates and the Java-based metamodel becomes complex. A plug-in mechanism to integrate calculations is also necessary to allow such adaptive definitions. This introduces lots of Java and template files and hence asks for a precise management and documentation of the templates and plug-in calculations.

The organization of templates needs to cope with several dimensions: (a) a template operates on one node of the metamodel of a certain type. (b) The result may be a complete file or a part to be included in another file. (c) The generated artifacts must conform to some concept (e.g., a nonterminal or metamodel class) of the target language, e.g., a subtemplate can generate a complete class file, but can also produce a statement, a block, an expression, an attribute or method definition, an import statement, a package, or a list of those. (d) Finally, subtemplates often assume context information to be given, e.g., an expression template expects certain variables. How to manage knowledge about template and their purposes?

Finally, there definitively is demand for a library of such templates and plug-ins.

Quality management for templates: Assuming, we have an adaptive development process: how do we keep the quality management simple? Some FreeMarker-based projects demonstrate the complex structure of templates as discussed above. Template engines like FreeMarker do not know anything about their target language and can, thus, produce syntactical nonsense. Control structures in templates make it undecidable whether this can happen. Therefore, the correctness of the generated result is checked after generation by the compiler, which all too often finds illegal syntax, missing variables, or methods, etc., in the generated code. This is too late and does not help to speed up the development. Furthermore, tracing erroneous code back to the originating template and template configuration is complex. "Debugging templates" in the generation process could help when adapting templates.

Organizing generator libraries: We need various libraries of complete or extensible generators for specific target platforms, technologies, and purposes. But how do components of such a generator library look like? How do extension mechanisms look like? Which documentation and set of configuration parameters is needed for each generator to be easily reusable and adaptable for different software projects? And finally, a combination of generators could be used for one software project as well.

Testing (or debugging) generated code: If the generation process was successful, debugging of the product is another important issue. If the error is identified and comes from the generated code: how can it be traced back to the template or the model source? Modularity of the generated code could help a lot here. Furthermore, it is always better to run automated tests than to trace manually using debuggers. Model-based debuggers are widely missing. It is a good approach

to test the generator and its templates on a well-defined but small set of source models before using it on real project models. Automatic tests can then be reused when adapting the generator. Another set of tests can be used for the generated application, assuming that the generated code is already unit-tested and only integration tests are necessary. And by the way, tests can be generated too, using appropriate test models (different from the source models).

Integrated development environment (IDE): IDEs such as Eclipse provide a tremendous amount of comfort to the developers. Editors can make use of highlighting, auto-completion, quick navigation to definitions of methods/classes used, etc. Compilation is done automatically and incrementally in the background. Refactoring steps simplify larger restructurings as well as consistent renaming operations. Programming language infrastructures like for Java are very well elaborated with their typing, visibility, and import mechanisms to control dependencies. All this is widely missing for modeling languages, making model-based development much more tedious. Eclipse in particular does not know the difference between a generated and a handwritten Java class, and therefore, searching facilities do find a great number of code that could otherwise be hidden. Refactoring does not apply to source models but to the generated code, thus enforcing the modelers to adapt their models manually. Even worse, using the automatic refactoring support of current IDEs will change the generated code without notifying the developer who might not be aware of the necessary manual adaption of the models. In this case, the next generator run will override the refactoring, which leads to compile errors in the generated code. Integrated model/code management would allow refactoring beyond language borders. The same would hold for auto-completion that allows developers to include elements from the source models in other source models as well as handwritten code and vice versa. This would also prevent many errors coming from the missing generated code at the beginning or after cleanup.

It would already help, if Eclipse knew the difference between handwritten source and generated code. Currently, when a generation run takes place, the developer manually has to tell Eclipse that code has changed (Refresh). Eclipse should also relax its warnings on generated code, as it is common to add potential imports, methods, or variables that are not used in the code in certain configurations.

Agile development only became possible when the appropriate tooling based on modern languages became available. Modeling tools still miss this agility for a variety of reasons discussed above. Some of these problems come from the man-made situation that IDEs currently assume a solely code-based development process, not at all allowing the integration with modeling tools. This sometimes leads to the invention of tooling workarounds, which in the long run only creates more problems.

However, many of the above-mentioned problems only seem to be pragmatic at first sight, but looking deeper into them, most of them require a very deep and precise theoretical solution that needs to be conceptually developed by researchers.

If you know solutions for some of these problems, please send your paper explaining it to SoSyM or tell us directly. If you experience similar or completely different problems, please also drop us a note (e.g., Bernhard.Rumpe@sosym.org).