CODING for the Code

Can models provide the DNA for software development?

Despite the considerable effort invested by industry and academia in modeling standards such as UML (Unified Modeling Language), software modeling has long played a subordinate role in commercial software development. Although modeling is generally perceived as state of the art and thus as something that ought to be done, its appreciation seems to pale along with the progression from the early, more conceptual phases of a software project to those where the actual handcrafting is done. As a matter of fact, while models have been found useful for documentation purposes and as rough sketches of implementations, their ultimate value has been severely limited by their ambiguity and tendency to get out of sync with the final code.

More recently, hopes that modeling might reach its deserved place in the software engineering process have been refueled by so-called MDD (model-driven development) initiatives, most prominently advanced by IBM and the OMG (Object Management Group).[1] The underlying idea is to promote models to the primary artifacts

FRIEDRICH STEIMANN, FERNUNIVERSITÄT IN HAGEN, AND
THOMAS KÜHNE, DARMSTADT UNIVERSITY OF TECHNOLOGY

CODING for the Code

of software development, making executable code a pure derivative. According to this development paradigm, software is generated—with the aid of suitable transformations—from a compact description (the model) that is more easily read and maintained by humans than any other form of software specification in use today. Using a metaphor from biology, such a model would be the construction plan—the DNA—of software, and the transformations the ribosomes of the construction process.

Today, increasing numbers of success stories accompany major releases of software development products that claim to have made MDD a reality, and market pressure will soon force even conservative CTOs to look into this emerging technology. As a colleague recently predicted: "In the future, there won't be any programming jobs in this country. Instead, we will make models and ship them offshore, where programmers will turn them into code."

"False," another colleague responded. "We won't ship the models, but transform them ourselves. However, we will do it all automatically." Take your choice.

In the following conversation, Dr. Con, known as a harsh critic of today's modeling languages and a skeptic of the feasibility of MDD, and Dr. Pro, a believer in the MDD vision, discuss whether MDD is flawed from the beginning or represents the most promising new development paradigm today.

GRAPHICAL VS. TEXTUAL: FORM OVER FUNCTION?

DR. CON MDD starts from a weak basis—graphical models—and aims at the highest possible goal: the delivery of sound production-level code. We know that in theory graphical and textual notations are equivalent (at least in their expressive power), but in practice graphical notations are usually far more cumbersome than textual ones. Admittedly, they aid comprehension if kept simple, but they quickly get convoluted when it comes down to the core of real problems. Why should something as weakly developed as a graphical modeling language help solve a problem all other approaches have failed to do?

DR. PRO Admittedly, textual languages have matured longer, and in today's practice still enjoy better tool support. On the other hand, you cannot deny that graphical models excel in conveying static information: a two-dimensional layout of the structure of a system is much more easily understood than any linear form, in which links (the lines) need to be resolved symbolically. As far as the description of system dynamics, there is certainly no point in having an iconic equivalent of *all* the traditional features of today's programming languages. Yet, surely there is value in expressing behavior in the form of interaction (sequence and collaboration types) and state diagrams. When it gets down to the nitty-gritty, small-grain behavior, graphical notations tend to lose their conciseness, but may be combined with textual behavior descriptions, by having the former serve as navigation aids to the latter.

DR. CON I was not thinking of *all* the features of today's programming languages, only the *most fundamental* ones. Variables and assignment, for example, can only poorly be expressed graphically, since altering a variable's value results in a change of the (dynamic) structure of a system, meaning that lines would have to be redrawn. Statecharts don't really help with specifying possible state changes of this kind, since graphical states are abstract (i.e., detached from variable values and the links that exist between objects). As a consequence, what we find in "visual" modeling is people decorating states with so-called actions that manipulate variable values, but that are more or less Smalltalk blocks in disguise. The statecharts themselves often are degenerate to the extent that all transitions to one state are labeled with the same event—they are in fact flow charts without loops and subroutine calls. This is really one step back from structured programming.

DR. PRO Who says you need variable assignment in modeling? Perhaps this is just too primitive a concept and therefore should be eliminated from modeling altogether. The real challenge is to find powerful transformations that inject the low-level behavior code expressed by means of variables and assignments to the final product, allowing the modeler to stay abstract at all times. Low-level behavior specification, by contrast, is the task of transformation engineers. The modeler's freedom might thus be restricted to choosing from a fixed set of available low-level behaviors; nevertheless, the models are abstract and precise at the same time. This shift—from "programming" to "configuration," from "handcrafting a single piece of software" to "selecting a fitting piece from an existing choice"—is a persistent trend in the history of computer science when it comes to increasing the productivity of software creation.

DR. CON What if your desired behavior is not among the choices offered? Then you are stuck.
DR. PRO The modeler will be stuck—not able to complete the system on his or her own. It is the task of transformation engineers to provide any missing choices that the modeler requests.[2] As a matter of fact, MDD implies new development roles and opens up the way for new development paradigms.
DR. CON So it all comes down to a very high-level language that—no matter whether graphical or textual—can automatically be transformed to some lower-level, deployable specification?
DR. PRO That's right. The usual emphasis on graphical models is not at the core of the MDD approach. What is essential is the idea of using high-level descriptions that are solution- or, if you wish, platform-independent. Ideally, the high-level descriptions resemble *analysis* models much more than solution-oriented *design* models.
DR. CON My analysis models are all lines and boxes, and I suppose most others' are, too. Since you seem to agree that these kinds of models are an insufficient basis for MDD, I would suppose that the *M* in its name is rather misleading.

BEEN THERE, DONE THAT: WHY SHOULD IT WORK THIS TIME AROUND?

DR. CON MDD is not the first attempt at making software construction more productive—code-generation tools and 4GLs (fourth-generation languages), for example, have been around for quite some time. Practice has shown, however, that code generation provides only a first approximation of the desired behavior. If it were powerful enough, we would make the source of the code generator a new programming language and turn the generator into a native compiler, but it doesn't work that way. Instead, we need to make changes to the generated code, partly because it is just one bit off what we need (and what we need precisely cannot be generated), and partly because there is code (e.g., an algorithm) that for good reason is not generated (e.g., because the target language is much better suited for expressing it than the source formalism of the generator). No one would accept a programming language that, in order to produce useful programs, would require hand coding in assembler as an additional exercise.
DR. PRO Some applications of code generation tremendously increase the productivity of the people employing it. It is just a matter of time before this kind of development will be more or less commonplace. Point taken, we still have to learn a number of lessons before we know how to use the generation paradigm optimally. That is why some of the issues you address are relevant, yet not insurmountable. In the future we'll learn to use, for example, parameterization to tweak our transformations so that they have *exactly* the desired effect and do not just approximate it. We will also learn how best to balance

The Role of Models in Software Development

The software industry adopted object orientation based on the promised seamless integration of analysis, design, and implementation, thereby removing the much-decried impedance mismatch between the traditional software development phases. The underlying premise was that the objects discovered in a problem domain and the relationships between them map to classes and their properties in the program, with design and implementation classes merely refining and adding to those found during analysis as the project matures. Although the idea of a purely incremental approach was perhaps a little naïve, today every typical application program has classes that represent ("model") entities of the problem domain with considerable fidelity.

With the adoption of UML (Unified Modeling Language) as an industry standard, the primary artifacts of analysis and design have become object-oriented models. This movement has been supported by the availability of tools allowing the systematic development and maintenance of graphical representations. While most such tools also offer skeletal code generation from designs, full round-trip engineering of models and programs is still hampered by the difficulty of mapping complex behavior and control to readable graphical representations.

MDD (model-driven development) avoids round-trip problems by abandoning the reverse engineering attempt (mapping programs to models) and instead concentrating on the generation of *complete* programs from models.[1] Such a strategy requires the annotation of models with directives steering the generation process. For MDD to work, however, annotations and corresponding transformations must be expressive enough to cover all the subtle details that distinguish a product's success from its failure.

REFERENCE

1. Weis, T., Ulbrich, A., and Geihs, K. 2003. Model metamorphosis. *IEEE Software* 20(5): 46-51.

CODING for the Code

the distribution of behavior in terms of expressing it in the model and/or shifting it to the transformations. We won't generate algorithms from scratch. We will either just select existing ones or compose new ones from building blocks that are known to work together. For example, many sorting algorithms can be expressed by a general approach that is just parameterized with two reduction strategies.[3] With respect to your last sentiment regarding hand coding in assembler, I fully agree. This is not an argument against MDD, though, since the modeler will never have to get in touch with a low-level language. It is the transformation engineer who might.

DR. CON History has shown that the kind of 4GLs to which you seem to aspire work rather well in certain specialized areas such as GUI construction or the games industry, but in general have not (and probably never will) supersede 3GLs. Despite the wide availability and applicability of extremely powerful 4GLs, we all seem to be using Java now, even though this means that we spend most of our time writing the most stereotypical source code. Why is this so? There must be some benefit outweighing the expressive power offered by 4GLs.

DR. PRO First, this is a maturity problem. Models and MDD will become more attractive once more sophisticated tools are available, especially if their usability surpasses that of the first-generation tools we have today. Second, a problem with the 4GLs you mention is their lack of openness and support by multiple vendors. In fact, one benefit of MDD, especially within the framework of standards such as the MOF (Meta Object Facility),[4] is that a variety of different tools can be jointly used to create one piece of software, and that changes made with one tool smoothly propagate to all others.

DR. CON So MDD is really giving a new edge to the 4GL approaches?

DR. PRO It also includes 5GL ideals such as executable specifications, which make validation and verification much easier to realize. Models will thus be not only blueprints for code generation, but also the subject of quality analysis methods and tools. Although the same information is in principle also contained in low-level code, it is practically inextractable since no current parser technology can distinguish low-level realization from high-level specification code,[5] both of which are expressed using the same, often ineffective language. So while MDD may be advertised as a revolution in software engineering, it is in large part really an evolution of previously existing approaches, with a strong emphasis on unification.

DR. CON Perhaps it's worth a try, but I wouldn't hold my breath until it works. I can't see the technology necessary to make computers guess what programmers want to say.

FROM ABSTRACT TO CONCRETE: HOW TO CLOSE THE SEMANTIC GAP

DR. CON How much redundancy is in a program? Given that the most trivial bug can alter program behavior to the point of making it useless, I would assume not very much. Now given that almost nothing of a production-level program can be omitted or changed without violating the specification, for MDD to work, the same information that is represented by the program must be captured by the model. This would imply one of the following:

- The modeling language in use is exceedingly more expressive than any programming language known today.
- Models are (nearly) as complicated as the programs they produce.
- Only a few programs can be generated from models.

Assuming that the first is not the case and that the second is not what we want, we must conclude the third. In other words, if a model were significantly simpler than the program it produces and if redundancy in a program were low, then I would assume that there are far fewer meaningful models than meaningful programs. This implies that many useful programs cannot be generated. After all, programming is a complex matter not because our linguistic means are inappropriate, but because our problems are extremely complex. How can we expect to take out the complexity without losing precision?

DR. PRO You are right in observing that there is no noise in programs. We cannot simply remove or alter pieces of programs and expect them to still work. There is redundancy, however, in that many high-level concepts are repeatedly realized using lower-level features in the same way. Instead of manually creating those repeated patterns of code, we should just specify that we want that particular pattern applied wherever deemed appropriate. In this sense, a modeling language drawing from many such predefined realization patterns, which are enacted by transformations, is indeed "exceedingly more expressive than any programming language known today."[6] That is why your second implication, that models must be "(nearly) as complicated as the programs they produce," does not follow from the use of MDD. Granted, during its infancy MDD will have to live with your third suggestion that "only a few programs can be generated from models." Even initially, however, *few* will equal *very many*, and gradually one will become familiar with the boilerplate code of the currently too-complex applications, and will then be able to capture it through standard transformations guided by model annotations. Regarding your supposition that our linguistic means are sufficient: they have not been in the past, so why should they be today?

DR. CON I question that you can express more with less. What you are saying is basically that for any given total of function points, there will be as many different models as there are programs, so that mapping is not ambiguous. This setting of knobs (the choice or parameterization of transformations) should either not alter the functionality or be part of the specification, hence, should also be found in the model.

DR. PRO Exactly. The ambiguity of the mapping from a model to a program, if any, is a result of a model not needing to get bogged down in realization choices to express the same number of function points as a program. In fact, programs that implement realization patterns in inefficient or even incorrect ways need not be generated and therefore need not be expressible with models. In addition, spelling out all the realization choices makes it difficult to see the functionality for all the realization of it; and they can be platform-dependent and hence subject to change. The actual realization choices—adding

Modeling and Language Productivity

The history of computer languages has been one long quest to increase programmers' productivity. One of the first known computing formalisms, the Turing machine, powerful as it was, was clearly the work of a theoretician: because of its extremely limited symbol and instruction set, writing programs was but an academic exercise. The first practically useful computer languages were designed to directly instruct physically existing machinery, without trying to abstract from it. The second generation, so-called assembly languages, started the process of relieving the programmer from work a mechanical translator could be trusted with: memorizing instruction-set codes and calculating jump distances were no longer a burden on the programmer, but left to automation. Later, programming languages were made to adapt to their particular application domain: so-called high-level (or third-generation) languages such as Fortran and Cobol offered special language constructs for many special purposes, resulting in shorter and/or better readable programs.

Ironically, only after the fourth generation of programming languages offering even more macrolithic constructs, the efficiency of absolute simplicity was rediscovered by a wider audience. Although languages such as Prolog and Lisp were perhaps a little bit too pure to be generally useful, the minimalism of the Smalltalk language was brilliantly compensated for by a library that, in conventional programming terms, is indistinguishable from the language itself: even primitive control structures such as branching and looping are not built into the language, but added as library functions. Together with a very high level of abstraction based solely on the concepts of *objects*, *variables*, and *message sends*, Smalltalk proved to be an excellent starting point for the development and broad dissemination of innovative programming paradigms such as the ample use of associative memory (dictionaries) as both storage and control devices. Its ability to support problem-specific language extensions is still unrivaled in many regards. Not surprisingly, Smalltalk is ranked among the most productive programming languages in Software Productivity Research Inc.'s Programming Languages Table,[1] surpassed only by special-purpose languages such as SQL and Excel.

The modeling community could learn from the Smalltalk lesson and devise a modeling language that shares its philosophy. It could be based on a minimal set of modeling primitives and support greater expressiveness through extendable modeling libraries. The semantics of any model in such a language would then be defined in terms of the library elements it refers to. The direct executability of models that follows should not be seen as an undue binding to some particular realization technology, but as a chance to validate both models and modeling language as early as possible in their respective development processes. Then, perhaps, one day the language productivity list will be led by a modeling language.

REFERENCE

1. http://www.theadvisors.com/langcomparison.htm.

CODING for the Code

realization preciseness to the abstract model—are partly expressed within the model, using a so-called "marking model," and are partly expressed outside the model as "setting of knobs" choices driving the transformation. You may regard the latter as compiler options. Today's compilers also allow some choice of translation strategy (e.g., optimize for code length or execution speed). Model compilers will feature a lot more of these "knobs" since many more solution-specific choices need to be made. In summary, you may create all your different programs from a combination of a model and various transformation choices.

THE NATURE OF TRANSFORMATIONS: ENGINEERING OR ALCHEMY?

DR. CON I think you are mixing up two different kinds of transformations. One goes from platform-independent to platform-specific, the other goes from high-level (abstract) to low-level (concrete). The former adds no real information, whereas the latter has to invent something, or make informed guesses. In fact, I would conjecture that platform-independent to platform-specific transformation is very much like applying a native compiler to source code, or that the platform-independent code could run on a virtual machine leveling all platform-specific differences. Turning an abstract description into a more concrete one, on the other hand, necessarily changes information content: it is a creative process.

DR. PRO At first glance, the two kinds of transformations appear to be different. From an MDD perspective, however, they can be unified. When a compiler translates source code to byte or machine code, it adds information as well. Yet the choices on what to add are so clear-cut that the compiler can make them for you without your advice. If the translation starts from something more abstract than today's source code—namely, a model—there will be more options and choices. Each such choice or option corresponds to a prethought realization pattern, and it is the art of the model compiler to combine these patterns into a functioning piece of software. Thus, it is really the choice that corresponds to creativity (increases information content), not the transformation (which just mechanically implements the choice). Hence, the two kinds of transformations are really not that different.

DR. CON You are basically implying that programming is just about selecting a number of prefabricated behaviors and then combining them. In practice, this is not the case. Just recently I came across a very simple problem: from a document consisting of a list of paragraphs, all paragraphs marked with a given tag had to be printed in their order of appearance; for two selected paragraphs that were nonconsecutive in the original document, the first sentence of the first and the last sentence of the last intervening paragraph also had to be printed to indicate the omission. Although the actual problem was in fact a little trickier than that, the chances that the solution, even of its simplified version, will be readily available in any library are very low, as are the chances that this kind of iterator will ever be needed again (which is why it isn't in the library in the first place). My experience tells me that every non-foobar application comes with countless idiosyncratic problems of this kind, and I would suspect that all attempts to parameterize model elements or transformations so that they can cover every conceivable peculiarity to otherwise stereotypical patterns is doomed to failure. Instead, I would conjecture that with today's programming languages, an optimum has been reached in the trade-off between expressiveness and flexibility. Indeed, only a few of the languages ranked above level 20—the level of Smalltalk and its ilk—in Capers Jones' language productivity list[7] could be considered general purpose. For example, trying to formulate the above selection procedure in SQL would certainly make me wish I had started the project in assembler.

DR. PRO I agree that SQL would be a bad choice, but then SQL is not UML and is not supported by predefined and extendable model transformations either. Regarding your once-in-a-lifetime problem whose solution will not be found in a library, what makes you so certain that it is not expressible by standard iterators with suitable parameterizations? Even if a particular problem cannot be cast in terms of generic solutions, the transformation engineer can always come to the rescue.

DR. CON So what you propose is really a model development kit analogous to the JDK (Java Development Kit) or .NET, and a coordination language that allows the model engineer to put it all together?

DR. PRO If you want to call it MDK that's fine with me as long as you don't confuse the approach with a simple library usage paradigm. MDD is not about including prefabricated parts; it is about applying boilerplate realization strategies automatically. It is about using a core language whose expressiveness can be extended—for example, by using annotations (marks) and associated realization strategies. This way you are neither stuck with some level of language expressiveness, nor do you end up with never-ending featurism for a given language.

DR. CON It seems you are advocating an extensible repertoire of modeling language constructs (various kinds of classifiers, associations, calls, state transitions, etc.), each coming with a set of transformations the modeler can pick from. The transformations of a construct are all functionally equivalent, but differ in the target platform and nonfunctional properties such as efficiency. Whenever I (as the modeler) miss either a language element or transformation rule, I turn to a transformation engineer to have it manufactured for me. Of course, all constructs of the repertoire are designed to go together well—that is, they can be combined freely.

DR. PRO You've got it!

DR. CON Nice vision, but to me it seems that the transformations of MDD are a bit like the weaving of AOP (aspect-oriented programming): either they don't achieve much, or they involve a considerable amount of hocus-pocus.

DR. PRO To quote Arthur C. Clarke, "Any sufficiently advanced technology is indistinguishable from magic." Admittedly though, there are several challenges left for the MDD community to tackle before it can deliver on all promises made so far.

FINAL SPEECHES: WHAT IS THE VERDICT?

DR. CON Models have their greatest value for people who cannot program. Unfortunately, this is not because models are more expressive than programs (in the sense that they are capable of expressing complex things in a simple manner), but because they oversimplify. To make a useful program (i.e., one that meets its users' expectations) from a model, the model must be so complicated that people who cannot program cannot understand it, and people who can program would rather write a program than draw the corresponding model. Even though (over)simplification can be useful sometimes, it is certainly not a sufficient basis for creating a satisfactory end product.

DR. PRO Models are ideal for people who know what they want, but are (and want to remain) unconcerned with realization details. Hence, they are suitable even for people who cannot spell out code (i.e., program). This is fantastic since we need to stop wasting time and money on reproducing boilerplate code and regular realization patterns over and over again—all in an error-prone fashion, often resulting in suboptimal solutions. This does not imply the end of programming, since transformation engineers will still be needed to implement new realization strategies. But, abstraction—as opposed to (over)simplification—is definitely our only known means to master the ever-growing demands on the construction of complex software. Q

REFERENCES

1. OMG. 2003. MDA Guide V1.0.1 (June); http://doc.omg.org/formal/03-06-01.
2. Weis, T., Ulbrich, A., and Geihs, K. 2003. Model metamorphosis. *IEEE Software* 20(5): 46-51.
3. Kershenbaum, A., Musser, D., and Stepanov, A. 1988. *Higher-Order Imperative Programming*. Rensselaer Polytechnic Institute Computer Science Department.
4. http://www.omg.org/technology/documents/formal/mof.htm.
5. See reference 2.
6. France, R., Ghosh, S., Song, E., and Kim, D. 2003. A Metamodeling approach to pattern-based model refactoring. *IEEE Software* 20(5): 52-58.
7. http://www.theadvisors.com/langcomparison.htm.

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

FRIEDRICH STEIMANN, alias Dr. Con, is a full professor in informatics at the Fernuniversität in Hagen, Germany, where he works in the areas of object-oriented development, programmers' productivity tools, and software modeling. He received his diploma in informatics from the Universität Karlsruhe, Germany (1991), his doctoral degree from the Technische Universität Wien, Austria (1995), and his habilitation from the Universität Hannover, Germany (2000). Before returning to academia, he worked as a research engineer for Alcatel and as a software consultant to various small and mid-size companies. He can be reached via e-mail at steimann@acm.org.

THOMAS KÜHNE, alias Dr. Pro, is an assistant professor at the Darmstadt University of Technology. Prior to that he was an acting professor at the University of Mannheim and a lecturer at Staffordshire University (UK). His interests include object technology, programming language design, component architectures, and metamodeling. He received a Ph.D. and M.Sc. from the Darmstadt University of Technology, Germany, in 1998 and 1992, respectively. He can be reached at kuehne@informatik.tu-darmstadt.de.