Model-driven development: The good, the bad, and the ugly

B. Hailpern
P. Tarr

In large software development organizations, increased complexity of products, shortened development cycles, and heightened expectations of quality have created major challenges at all the stages of the software life cycle. As this issue of the *IBM Systems Journal* illustrates, there are exciting improvements in the technologies of model-driven development (MDD) to meet many of these challenges. Even though the prevalent software-development practices in the industry are still immature, tools that embody MDD technologies are finally being incorporated in large-scale commercial software development. Assuming MDD pervades the industry over the next several years, there is reason to hope for significant improvements in software quality and time to value, but it is far from a foregone conclusion that MDD will succeed where previous software-engineering approaches have failed.

INTRODUCTION

Most developers operate by sitting down with their favorite text editor and typing in their program, attempting to compile it, making changes, compiling it, testing it, and so on until the program "works." Sometimes the various reasons for design decisions are captured in comments or other documents. Often, they are lost to posterity. Those rationales and design decisions are, however, critical for the success of a long-lived, ongoing, high-quality programming product. Hence, the standard *laissez faire* approach to programming that many practitioners learned must be replaced by a more disciplined engineering methodology.

Various software-engineering methodologies[1–5] describe processes whereby requirements, architecture, design, implementation, and testing information—along with their interrelationships—can be captured. Why is this information preserved at all? Maintaining this captured data may be a requirement of a customer or mandated for software quality certification. In addition, it may be essential to the development organization, when the development of software extends beyond a single individual developer or development team. It can also be useful or required when teams are distributed geographically (i.e., when requirements are gathered in one city, but code is developed in another). Then this captured data becomes a vital communication link between the teams for many purposes, even as a contract between them. When a

software product takes a long time to develop or has multiple versions over time, then this captured data becomes essential to support the institutional memory as team members leave the project or are required to revisit parts of the software that they have not seen for some time. For large ongoing programming products, capturing and maintaining this data is critical to the success of the product.

It is challenging to convince development teams to create the information in the first place, because it costs time and money that could be used to meet immediate deadlines. It is even more challenging to ensure that the critical information is kept current as changes to the requirements or system are made over time, especially when some information will never be critical and some critical information will "age" and eventually stop being critical. In both cases, the cost of creating/updating the information lies on one part of a team, but the benefit usually accrues to someone elsewhere or "elsewhen." Yet once a development process can rely on the existence of current, accurate information, opportunities for automation abound. Everyone wishes the information were available when questions arise about why some concept was included or excluded or tested or not tested, but collecting and maintaining this data costs time and money.

How then should one describe and preserve the various documents (and other artifacts, such as program comments, test scripts, architectural diagrams) associated with a software project? The simplest answer is to "do what comes naturally." Requirements are often written (text) documents (with bullet points or textual scenarios). Architectures are (unfortunately) frequently just pretty pictures with annotated details of programming interfaces. Programs are almost always source code in some programming language. Test suites are usually embodied by scripts and regression test data. "Bug" (unexpected defect) reports are kept (if at all) in databases or logs. The problem with this simplistic approach is that none of the meta-information associated with these artifacts is captured, and therefore, nothing explicitly relates to anything else, even though the relationships are clearly present. If requirements are documented in unstructured text, what chance does a person (or a system) have of matching them to an architectural element or injecting task automation? What chance is there that someone else will understand the requirement a year later? How well can we understand C code without (or even with) comments? Why was a particular test case included and is it still valid?

An alternative to this multidocument, natural collection of information is to use a "single source" approach, where a given concept is represented only once, in one type of software-engineering electronic artifact, rather than having multiple artifacts per concept. This approach can help reduce the number of types of artifacts and the interrelationships among those artifacts. It does not, however, eliminate the problem described in this section. Interrelationships among concepts (and hence, among artifacts) still exist. Moreover, interrelationships to existing libraries also exist.[6–8]

Model-driven development (MDD) is a software-engineering approach consisting of the application of models and model technologies to raise the level of abstraction at which developers create and evolve software, with the goal of both simplifying (making easier) and formalizing (standardizing, so that automation is possible) the various activities and tasks that comprise the software life cycle. MDD imposes structure and common vocabularies so that artifacts are useful for their main purpose in their particular stage in the life cycle (such as describing an architecture), for the underlying need to link with related artifacts (earlier or later in the life cycle), and to serve as a communication medium between participants in the project (over space or time).

The Object Management Group, Inc. (OMG**) defines a particular realization of MDD using the term *Model Driven Architecture*** (MDA**). Further, they define a special concept of models that distinguishes those models that take into account the details of the underlying hardware and software (platform) and those that do not. OMG defines MDA to be

> based on a Platform-Independent Model (PIM) of the application or specification's business functionality and behavior. A complete MDA specification consists of a definitive platform-independent base model, plus one or more Platform-Specific Models (PSMs) and sets of interface definitions, each describing how the base model is implemented on a different middleware platform. A complete MDA

application consists of a definitive PIM, plus one or more PSMs and complete implementations, one on each platform that the application developer decides to support.[9]

MDA begins with a model concerned with the (business-level) functionality of the system, independent of the underlying technologies (processors, protocols, etc.). MDA tools then support the mapping of the PIM to the PSMs as new technologies become available or implementation decisions change.

MDA represents just one view of MDD, though it is perhaps the most prevalent at present. Others also exist, such as Agile Model-Driven Development,[10] Domain-Oriented Programming,[11] and Microsoft's Software Factories.[12] This paper is about MDD in general. However, due to its prevalence and status as a standardized entity, OMG's MDA is used to exemplify issues throughout this paper. This paper is *not*, however, intended to be a full exposition of the advantages and disadvantages of MDA. It is too early to predict which—if any—of the current MDD approaches will perform best in real-world scenarios.

Thus far, we have defined MDD in terms of "models," relying on the reader's intuition about what models are. We now turn to the question, "What is a model?"

BACKGROUND: TERMINOLOGY AND DEFINITIONS

Because one of the goals of this special issue of the *IBM Systems Journal* is to be accessible to the students of software engineering at large, we define relevant terminology and its implications (we include pseudo-formal notation for this terminology, but it is not essential for the basic understanding of problem definition). Note that among researchers there is no universal agreement as to the precise definitions of the following terms. The reader is encouraged to view these as a consistent set of terminology and indicative of what is meant by many researchers in the field, including the authors of this special issue.

A *model* **M** is an abstraction over some (part of a) software product (e.g., requirements specification, design, code, test, call-flow graph). There is a variety of kinds of models and we indicate those

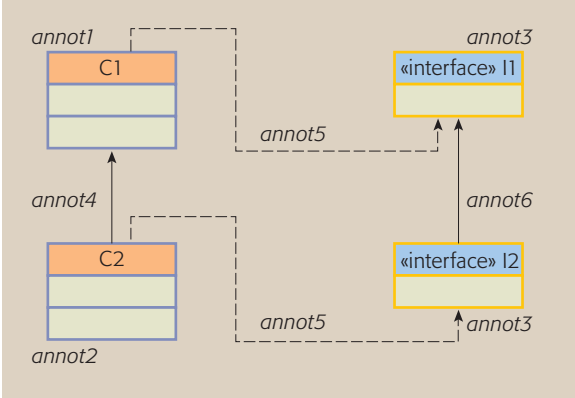

Figure 1
Example UML model (class diagram)

kinds by using subscripts on **M**, such as $\mathbf{M}_{\mathrm{UML}}$ for a Unified Modeling Language**(UML**) model.[13] Of course, a fully formal notation would distinguish among the different levels of abstraction and kinds of diagrams within, for example, a UML model. For the purposes of this paper, this level of detail is not necessary. In our (semi-formal) notation, a model is an annotated graph over a set of model nodes, a set of model edges, an alphabet of labels, and a function annotating nodes and edges ($\mathbf{M} = <\mathbf{N}, \mathbf{E}, \mathbf{\Sigma_M}, \mathbf{\Lambda_M}>$). Model edges are the usual directed edges from nodes to nodes ($\mathbf{E} \subseteq \mathbf{N} \times \mathbf{N}$). The annotation function maps either nodes or edges into labels ($\mathbf{\Lambda_M}:\mathbf{N} \cup \mathbf{E} \rightarrow \mathbf{\Sigma_M}$). A *model element* is a subgraph of **M** (possibly just an individual node). There exists a mapping from each model element to one or more elements of an underlying (uninterpreted) domain. Hence model elements represent (or abstract from) real or conceptual objects. It should be noted that the use of a graph representation supports different kinds of structures that might be used for models, such as tables, stacks, code (modeled using abstract syntax graphs), and structured text, such as requirements documents.

To illustrate, consider the UML class diagram in ***Figure 1***. This class diagram is a model ($\mathbf{M}_{\mathrm{UML}}$)—it represents a partial abstraction of a software system. In this case, **N** is the set of nodes {*C*1, *C*2, *I*1, and *I*2}. **E** is the set of edges {<*C*2, *C*1>, <*I*1, *C*1>, <*I*2, *C*2>, <*I*2, *I*1>}, reflecting the generalization and realization associations in the figure. We imagine a trivial set of annotations ($\mathbf{\Sigma_M}$), shown in black in the figure, consisting of {"annot1", "annot2", "annot3", "annot4", "annot5", "annot6"}. Then, our labeling

($\mathbf{\Lambda_M}$) of the nodes would be $C1$ = "annot1," $C2$ = "annot2," $I1 = I2$ = "annot3". Our labeling ($\mathbf{\Lambda_M}$) of the edges would be $<C2,C1>$ = "annot4," $<I1,C1> = <I2,C2>$ = "annot5," and $<I2,I1>$ = "annot6."

An *artifact* ($\mathbf{A_M}$) is a set of "meaningful" model elements of **M**, for some definition of "meaningful." An artifact represents a complete, consistent, and legal subgraph of **M**. For example, an artifact could represent a complete statement in a programming-language grammar or a legal UML class diagram. In the preceding example, the node $C1$ would be a meaningful artifact, as would the subgraph that includes $C1$, $I1$, and the edge $<I1, C1>$. It should be noted that the definition of artifacts as complete, consistent, and legal subgraphs is only a convenient abstraction. We recognize that, in the real world, people may have to address artifacts that are not complete, consistent, or legal in their modeling notation but that represent meaningful artifacts nonetheless. The abstraction is sufficient for the purposes of this paper.

A *relationship* **R** maps artifacts in one model, M_i, to artifacts in another model, M_j (where i may equal j), with annotations on the edges of the relationship ($\mathbf{R} = <\mathbf{A}_1, \mathbf{A}_2, \mathbf{\Sigma_R}, \mathbf{\Lambda_R}>$). (We note that our definition describes only binary relationships. A complete formal definition would allow for general n-ary relationships.) An essential case for MDD is when the two models are distinct (i.e., $i \neq j$). For example, if $\mathbf{M}_1$ and $\mathbf{M}_2$ are models, with $\mathbf{A}_1$ being the artifacts of $\mathbf{M}_1$ and $\mathbf{A}_2$ being the artifacts of $\mathbf{M}_2$, then **R** represents the relationship edges from artifacts in $\mathbf{A}_1$ to artifacts in $\mathbf{A}_2$, with labels in the alphabet $\mathbf{\Sigma_R}$ assigned by $\mathbf{\Lambda_R}$. Our UML example can be extended to include another model ($\mathbf{M}_{\text{Java}}$) containing a Java** program that corresponds to our UML diagram. The relationship would then contain relationship edges from UML artifacts in $\mathbf{M}_{\text{UML}}$ to the corresponding program artifacts in $\mathbf{M}_{\text{Java}}$ (classes to classes and interfaces to interfaces). We could then annotate these new relationship edges.

As stated above, the models need not be distinct; that is, a relationalship can connect nodes in a model to other nodes in the same model ($M_i = M_j$, for example, a use-def relationship in an abstract syntax tree). We distinguish different kinds of relationships, based on how the relationships are defined or used; for example:

- *Instantiation*—Nodes in $\mathbf{A}_2$ are specific instances of "class/type" nodes in $\mathbf{A}_1$.
- *Refinement*—Nodes in $\mathbf{A}_2$ represent a more detailed description of nodes in $\mathbf{A}_1$.
- *Realization*—Nodes in $\mathbf{A}_2$ represent an implementation of nodes in $\mathbf{A}_1$.
- *Specialization*—Nodes in $\mathbf{A}_2$ are specific instances of "generic" nodes in $\mathbf{A}_1$.
- *Manual*—The relationship was created by the actions of a human being.
- *Generated*—The relationship was created by the actions of a program.
- *Derived*—Nodes in $\mathbf{A}_2$ are a logical consequence of, and generated from, the nodes in $\mathbf{A}_1$.
- *Implied*—The relationship can be deduced by applying a set of rules.

The set of annotations (both at the model level, $\mathbf{\Lambda_M}$, and the relationship level, $\mathbf{\Lambda_R}$) is called *metadata*. Note that "metadata" is generally understood to be "data about data." Hence, one could include the relationships in the metadata, because relationships are links between existing model subgraphs. It all depends on what is the "base" data and what is commentary on the data. Annotations can represent both static and dynamic properties, and both functional and nonfunctional properties.

Given a set of models $\mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_n$ and a set of relationships $\mathbf{R}_{1,2}, \mathbf{R}_{2,3}, \ldots, \mathbf{R}_{n\text{-}1,n}$, a *trace* represents a path through the $\mathbf{R}_{k,k+1}$, so that the destination artifact of one **R** "matches" with the source artifact of another. Thus, a trace represents a chain of relationships across the different models (or artifact representations) through a software product's life cycle (for example, mapping a requirement to its corresponding architectural element, to the code that implements it, and to the test case that validates it). The property of *traceability* (which enables creating or following a trace) is core to the value proposition of MDD. Traceability relies on the essential meta-information that must be communicated among the people, teams, and roles that participate in a large software development process. Participants in the (model-driven) software life cycle must be able to communicate what needs to be done (for example, the architect specifies what the developer is to build) and to determine what caused a particular event to occur or artifact to exist (for example, what requirement resulted in a particular test case that just failed). The ability to *round-trip* across the models in a life cycle embodies the

bidirectional nature of a trace path (that is, the ability to go forward and backward along a trace, and not lose your way).

It should be noted that this recognition of the important nature of traceability is not universally accepted. Some Agile development proponents advocate minimal models and eschew some or all traceability in favor of "traveling light," reducing to a minimum the need to maintain these artifact interrelationships. Whether explicitly represented or not, interrelationships across different artifacts exist. To the extent that these relationships impact the correctness and evolution of the code and the execution of the process, they are critical to understanding and communication among stakeholders.

At a metalevel, the sets of models and relationships (including their annotations) can be constrained to satisfy a set of *consistency specifications*. For example, "every use case must be implemented by (i.e., connected to by an "implements" relationship) a code artifact, and it must be tested by (i.e., connected to by a "testedBy" relationship) at least one test case." Unlike consistency specifications in traditional databases, we do not assume some kind of atomicity or transactional underpinnings which would ensure that consistency is maintained at every observable point. Rather, because of the human nature of the software-development process, the feel is more of long-running transactions, where consistency issues are identified, prioritized, and managed. Inconsistency may persist and must be managed for extended periods of the software life cycle. This process of controlled chaos has been called *inconsistency management*.[14,15]

Once we have a set of models and the relationships between them, we can define *transformations* as the systematic (manual or automated) modification of a model and its set of affected relationships. Hence, a transformation could change a model into a new model, constrained by its current relationships, or it could leave one model unchanged and instead create new models or new relationships based on the existing ones. The term *reengineering* refers to a set of changes that adds to or changes the functionality in the system. When a more systematic, structured set of semantics-preserving changes is engineered, it is termed *refactoring*.[16,17] Keeping track of changes at whatever granularity is appropriate is called *versioning*.

The context of models and relationships also allows us to define *reverse engineering* to be the extraction of a higher-level model from another, lower-level model (or representation). Examples of reverse engineering include extracting architecture from code or extracting requirements from an architecture. The process of reverse engineering can be manual, semiautomatic, or automatic.

WHAT PROBLEM IS MDD INTENDED TO SOLVE? (THE GOOD)

The goals and approaches underlying MDD are not new. The primary goal is to raise the level of abstraction at which developers operate and, in doing so, reduce both the amount of developer effort and the complexity of the software artifacts that the developers use. Of course, there is always a trade-off between simplification by raising the level of abstraction and oversimplification, where there are insufficient details for any useful purpose.[18]

The desirability for more abstract artifacts and more levels of abstraction has a long history. It goes back to the introduction of assembly language as an abstraction over machine code. This was followed

■ Some degree of software complexity is inherent in the difficulty of the problems to be solved ■

by the introduction of third-generation languages, like FORTRAN and COBOL (common business-oriented language), that enabled developers to ignore register allocation and other low-level, machine-specific instructions by introducing higher-level abstractions (such as named variables and structured programming constructs) that are translated to the underlying machine by means of compilation technology. Object-oriented languages, such as Simula, Smalltalk, and C++, introduced additional abstractions—such as abstract data types and objects. In each case, the abstraction had twin effects: higher quality and productivity and the creation of a lingua franca for the users so that there would be a vocabulary closer to the actual problem domain. MDD follows in this tradition and extends it by introducing model abstractions at the various stages of the software life cycle. If the MDD abstractions are to be realized in running code or

instantiated data, they require a process analogous to compilation, where models are transformed to concrete representations.

Analogous to Julius Caesar's observations on Gaul,[19] the MDD community can be divided into three parts,[20] one of which is called the *sketchers*, another is called the *blueprinters*, and the third are those we refer to as the *model programmers*, who support the direct use of modeling languages for development. The sketchers focus on the use of UML (or other modeling notations) to facilitate the understanding of code[21,10]:

> The essence of sketching is selectivity. With forward sketching you rough out some issues in code you are about to write, usually discussing them with a group of people on your team. Your aim is to use the sketches to help communicate ideas and alternatives about what you are about to do. You do not talk about all the code you are going to work on, just important issues that you want to run past your colleagues first, or sections of the design that you want to visualize before you begin programming. Sessions like this can be very short, a 10-minute session to discuss a few hours of programming or a day to discuss a two-week iteration.
>
> With reverse engineering you use sketches to explain how some part of a system works. You do not show every class, just those that are interesting and worth talking about before you dig into the code.[21]

The blueprinters[22] draw the analogy between software architecture and building architecture. They create very detailed design models, which are then handed off to (presumably less expensive) coders to produce implementations. This separation of tasks enables the (generally more expensive) design experts to focus solely on complex design issues. This approach makes the assumption that large development will take place in large organizations containing many different people with many different skill levels, in contrast with small-development organizations made up of only "top guns."

Both the sketchers and the blueprinters maintain a strong distinction between design models and code artifacts. Both groups strongly support modeling.[20] Their notion of MDD assigns a facilitating role to the models. The artifacts promote the development and evolution of code, but are not themselves executable languages that would replace the likes of Java or C#.

The model programmers support the use of UML (or some alternative modeling notation) as a development language with executable semantics,[23] using, for example, action semantics and statecharts. In model programming, the distinction between models and code is obscured. Some form of executable code exists, whether it is realized in a high-level programming language or by direct "compilation" to low-level, executable representations like assembly language. In the former case, the generation to a high-level programming language either produces complete implementations or partial ones, with the programmer left to fill in the blanks. In the latter case, it is not generally manipulated directly by developers. The model-programming camp is typified by the supporters of the OMG vision of MDA.[24] MDA developers work predominantly in UML as their development language. They begin by creating a PIM of their solution in UML (e.g., defining interfaces to domain concepts like ATMs [automated teller machines] and bank accounts), then refine the PIM into PSMs that take into account one or more particular target implementations (e.g., relational tables that store the account information on which the ATM operates). Executable semantics are specified using UML (e.g., activity diagrams). Code (e.g., Java or C#) can then be generated directly from the UML.

WHAT PROBLEMS DOES MDD CREATE? (THE BAD)

The "modest" intent of MDD is to improve software quality, reduce complexity, and improve reuse by enabling developers to work at reasonably higher levels of abstraction and to ignore "unnecessary" details. In practice, however, MDD also raises a number of significant issues.

Redundancy

A central tenet of MDD is that there are multiple representations of artifacts inherent in a software development process, representing different views of or levels of abstraction on the same concepts. To the extent that these are manually created, duplicate work and consistency management are required. A similar problem was found in the software verification work of the 1970s and 1980s, which required two different versions of the same software to be written—one for specification and one for execution.

Rampant round-trip problems

The more models and levels of abstraction that are associated with any given software system, the more relationships will exist among those models. Many of these interrelationships are complex. The round-trip problem occurs whenever an interrelated artifact changes in ways that affect some or all of its related artifacts. For example, if a developer adds a method, *m*, to a class, *C*, in a UML class diagram, the Java code that realizes *C* must be modified to include an implementation of *m* (or at least it must be flagged that an implementation of *m* is needed). In some cases, the change may be propagated automatically—for example, if *C* were an interface instead of a class, it might be possible to automatically generate a method *m* in *C*.

The far worse (and more common) case, however, is when the round-trip problem cannot be addressed automatically. For example, if the change occurs in a method body, human intervention is required to determine the impact of the change on the related use case or business process model. In this case, the structure of the code or model is unchanged, but the semantics underlying the code or model have been adjusted. Is it a change in the desired function? Is it a bug fix? Is it part of a more extensive refactoring of the entire package? Each will have different implications on related artifacts.

The worst forms of the round-trip problem generally occur when changes occur in artifacts at lower levels of abstraction, such as code, because inferring higher-level semantics from lower-level abstractions is much more difficult than generating lower-level abstractions from higher-level ones. Consider the relative difficulty of propagating changes from UML diagrams to code artifacts, compared to the difficulty of propagating any significant changes from code to its corresponding UML diagrams. The problem is magnified when transformation technologies are involved because changes to the generated artifacts may be lost when regeneration occurs. Generation technologies usually generate "bad" variable names, because they lack a programmer's intent. Optimization techniques can reorder, combine, or eliminate details that can be useful for human understanding but are unnecessary to machine execution.

Note that this discussion in this section could imply that round-trip problems only occur in waterfall development methodologies where one stage must be completed, before the next stage occurs.[5] This is not the case. Round-trip problems occur whenever relationships across models are important. The basic problem is that the introduction of multiple, interrelated representations implies the issue of assuring their mutual consistency—a very difficult problem.

Moving complexity rather than reducing it?

Some degree of software complexity is inherent in the difficult problems being solved with software. Other complexity is *spurious*—given an appropriate approach, it need not be present. Differentiating between inherent and spurious complexity can be difficult. As with any development technique or technology, one must determine whether a given MDD approach *reduces* complexity visible to the developer, or whether it simply *moves* complexity elsewhere in the development process. As the number of artifacts increases, the number—and potentially, the complexity—of artifact relationships increases, as does the complexity of the tools that manipulate and visualize them. It remains to be seen if people have an easier time managing a relatively small number of large artifacts with fewer relationships, or if they manage better with a large number of more specialized artifacts, with a correspondingly greater numbers of relationships. The real difficulty of this question becomes obvious when the full life cycle of development is considered. A process may be simple the first time through, but given the complexity that has been "moved," it may be impossible (or prohibitively expensive) to maintain, debug, or change the resulting artifacts in the future.

■ Raising the level of abstraction may lead to oversimplification when there is not enough detail for any useful purpose ■

More expertise required?

Each type of model requires a particular set of skills to produce and evolve effectively. In raising the level of developer abstraction to models, MDD enables specialists to work with abstractions that better suit their tasks and expertise. Conversely, the

interrelationships between multiple types of models, and potentially, different modeling formalisms, suggests that it will be difficult for any given stakeholder (e.g., use case developer, architect, implementor, tester) to understand the impact of a proposed change on all of the related artifacts. They must understand how a change to their artifacts relates to or impacts other related artifacts that could be described in different notations from the ones they use every day. Problems like this have always existed to some extent, but MDD makes them more explicit and harder to ignore. This requirement for cross-discipline understanding is reminiscent of Ambler's concept of "generalizing specialists."[25]

Economic and other realities often dictate that development cannot rely on small, close-knit teams (e.g., offshore outsourcing and open-source development). Hence, large, distributed development teams are created so that different levels of expertise can be exploited based on skill sets at different development sites, such as requirement designers who consult directly with a customer, architects who create common designs to be used throughout an organization, programmers in a "back-office," and testers who may be in yet a fourth location. In the absence of high-bandwidth interactions, such as face-to-face communications,[26] different MDD models can aid in the communication between these different subteams, but it also implies that the different subteams cannot be expert in only their own development genre. Because artifacts resulting from any stage in the life cycle can impact those produced at any other stage, knowledge of different model technologies and terminologies must exist at each site. In the presence of the sorts of transformation technologies that are part of MDD, developers also may have to be fluent in various transformation notations. Transformations may be extremely complex.

MDD LANGUAGES (THE UGLY)

The standardization of modeling notations such as UML is unquestionably an important step for achieving MDD. Standardization provides developers with uniform modeling notations for a wide range of modeling activities. Moreover, standardization efforts (if successful) also open the door to many types of tooling support for creating and manipulating models in novel ways, generating artifacts (such as code) from models, and reverse engineering models from other artifacts. Unfortunately, the development of the UML 2.0 standard is not without its critics. It has been noted by some[27] to have serious problems that may well impede the adoption of MDD.

First, in attempting to address so many disparate needs, UML 2.0 has become enormous and unwieldy. History has not been kind to kitchen-sink languages, as their complexity has tended to impede their successful adoption.[11] The use of UML profiles can help with this significantly by enabling knowledgeable developers to eliminate any parts of UML that they do not need. It remains to be seen whether this mechanism will gain widespread adoption.

UML 2.0 includes a powerful metamodeling facility, Meta Object Facility (MOF**).[28] MOF enables UML to be extended almost arbitrarily. Unfortunately, some of the constructs in UML 2.0 are nearly semantics-free (e.g., use cases). This dearth of semantics complicates the correct usage of UML extensions, reduces their expressive power, and limits the ability of tool vendors to provide reliable, consistent model technologies. As Thomas notes[29]:

> UML 2.0 lacks both a reference implementation and a human-readable semantic account to provide an operational semantics, so it's difficult to interpret and correctly implement UML model transformation tools. For example, key concepts such as Use Cases lack sufficient semantics to support model refinement. Why not provide a simple accessible operational semantic account ... [which] would no doubt point out semantic holes and ambiguities, leading to an improved specification and reducing the time required to build robust MDA tools.

The lack of semantics at the ground and extension levels makes the production of automated MDD tools difficult because the semantics carries the meaning that is essential to enable automation.

The automatic generation of executable code from high-level descriptions faces other challenges as well. In general, the higher the level of abstraction a developer uses, the more choices exist for how to realize the abstraction in terms of executable code. For this reason, design patterns[30] were conceived as, and remain, *architectural* components, rather than specifically *implementation* components. A design pattern represents a solution to a problem in a context. However, the strategy for selecting imple-

mentations can vary widely, depending on the rest of the system requirements. It is unrealistic to assume that automatic generation of efficient and customized implementations could occur for design patterns in general.[31] There is room for some degree of control over the implementation choices (e.g., in the form of "pragmas" that some compilers accept).

■ By enabling developers to work at a higher level of abstraction, MDD aims to reduce complexity and thus improve software quality ■

So long as the set of implementation alternatives is small and so long as people need not modify the generated executable code, higher-level abstractions can be reasonably added as first-class programming constructs. The Eclipse** Modeling Framework (EMF) is an example of a technology that takes this position. It enables developers to program in Java, while using somewhat higher-level abstractions (a small subset of UML class diagram constructs). The wide adoption of EMF demonstrates the value of adding first-class support for what are now commonly used abstractions. However, it also rather pointedly suggests how little of UML 2.0 may be ready for treatment as commonly used, well-accepted abstractions.

The notion of UML 2.0 as a model programming language is predicated on the belief that the use of higher levels of abstraction will make developers more productive than current programming languages.[23,32] Fowler, however correctly makes the following observation:

> The question, of course, is whether this [belief] is true. I don't believe that graphical programming will succeed just because it's graphical. Indeed I've seen (and worked with) several graphical programming environments that failed—primarily because it was slower to use than writing code. (Compare coding an algorithm to drawing a flow chart for it.) Furthermore, even if UML is more productive than programming languages, [it is] hard for programming languages to become accepted. Most people I know don't program for a living in the language they consider to be the most productive. Languages need a lot of things to come together for them to succeed.[23]

Bell[33] offers additional support for Fowler's position by noting the difficulty of using extremely detailed models—the sort required to enable automated transformation:

> Victims of kitchen-sink fever crave the idea of building gargantuan UML models that include all fine-grained design elements in their detailed splendor. Kitchen-sink fever is often accompanied by abracadabra fever in victims who believe that in the absence of code, information can be derived by describing the low-level behaviors of interactions spanning the model's represented subsystems. Victims of kitchen-sink fever typically spend significant amounts of time recovering from the effects of crashes of their modeling tools.
>
> Clinical research has shown that one reason victims of kitchen-sink fever desire all possible artifacts in their models is that they have a poor understanding of the information that can be realistically derived from them. Research has also shown that those infected with this fever have typically never used a gargantuan model.

These and other issues have led Greenfield et al.[34] to argue that although UML 2.0 is a useful modeling language, it is not an appropriate language for MDD. Their Software Factories approach[12] is based, instead, on the use of special-purpose, domain-specific languages (DSLs). This approach shows some promise as well, though as Booch points out,[35]

> the root problem is not simply making one set of stakeholders more expressive, but rather weaving their work into that of all the other stakeholders. This requires common semantics for common tooling and training, so even if you start with a set of pure DSLs, you'll most often end up covering the same semantic ground as the UML.

Clearly, UML—or any other MDD language—faces significant hurdles to demonstrate sufficient value to satisfy the needs of all the different kinds of MDD users.

DISCUSSION

MDD is not the first attempt to solve the "software life-cycle development problem." For example, in

the 1980s, Computer Aided Software Engineering (CASE) was the promised panacea to solve the world's software development problems.[36] CASE systems had suites of tools to facilitate the various stages of the software life cycle. CASE failed. Often the stages were not well integrated (even within the tools of a single vendor)—this is often called the "silo problem"—the processes did not match what developers did or needed to do, and the systems were extremely large and complex. Is MDD fated to meet the same ignoble end?

We believe that MDD has a chance to succeed in the realm of large, distributed, industrial software development, but it is far from a sure bet. The growth of UML-based tools (e.g., Poseidon, TogetherSoft, I-Logix's Rhapsody, Rational* Software Modeler, ArgoUML, and Eclipse's support for both UML 2.0 and the UML-based EMF) suggests that more and more people are finding real use for modeling.

Standards exist so that tools from different vendors have a chance at interoperating. (Of course, standardization alone does not ensure interoperability. For example, different interpretations of the Extensible Markup Language (XML) Metadata Interchange (XMI**) standard[37] have produced non-interoperable tools.) Where those formal standards do not yet exist (such as interchange formats for UML 2.0), open-source tools and environments can drive the community to adopt de facto standards. This community pressure should motivate tool vendors to accept these interoperability standards and formalize them where necessary. Standards (including standardized models, languages, and interchange) are but one step to eliminate the silo problem (which is still with us from the days of CASE), but it is not the full solution. Not only must tools interoperate, but broad support for traceability and inconsistency management between and among different models and artifacts is essential to the elimination of silos.

Technology (both computing power and software environments) has come light years since the 1980s. Almost every personal computer sold today has the capability to run powerful integrated software development environments, such as Eclipse.[38] Generations of students are now taught to develop software using these environments. Tool vendors expect to target their tools to these environments, which means that new tools are designed to be integrated—to work together from scratch. Technology continues to improve the lot of the developer, as illustrated by the papers in this special issue of the *IBM Systems Journal*.

This unprecedented confluence of events means that the stage is set for MDD tools to become stars. The entry barrier for producing and disseminating sophisticated software-engineering concepts and tools is as low as it has been in recent history—new users can easily introduce and exploit extremely complex technology. The need is there—software complexity is at an all time high, and every aspect of modern society depends on the quality of that same software.

ACKNOWLEDGMENTS

We are grateful to Stan Sutton and the anonymous referees for their extremely helpful feedback on this paper.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Eclipse Foundation, Inc., Object Management Group, Inc., or Sun Microsystems, Inc. in the United States, other countries, or both.

CITED REFERENCES

1. K. Beck, *Extreme Programming Explained: Embrace Change,* Addison-Wesley (2000).
2. B. Boehm, "A Spiral Model of Software Development and Enhancement," *SIGSOFT Software Engineering Notes* **11**, No. 4, 14–24 (August 1986).
3. I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process,* Addison-Wesley (1999).
4. D. L. Parnas and P. C. Clements, "A Rational Design Process: How and Why to Fake It," *IEEE Transactions on Software Engineering* **12**, No. 2, 251–257 (February 1986).
5. W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings of the 9th International Conference on Software Engineering* (March 30–April 2, 1987), Monterey, California, ACM, New York (1987), pp. 328–338.
6. D. E. Knuth, *Web System of Structured Documentation*, Stanford Computer Science Report CS980, Stanford University, Stanford, California (September 1983).
7. J. L. Bentley, "Programming Pearls: Literate Programming," *Communications of the ACM* **29**, No. 5, 364–369 (1986).
8. S. W. Ambler, *Single Source Information: An Agile Practice for Effective Documentation* (2005), http://www.

agilemodeling.com/essays/singleSourceInformation.htm.

9. Model Driven Architecture (MDA) FAQ, Object Management Group, http://www.omg.org/mda/faq_mda.htm#whatismda.
10. S. W. Ambler, Agile Modeling (AM) Home Page: Effective Practices for Modeling and Documentation, http://www.agilemodeling.com/.
11. D. A. Thomas and B. M. Barry, "Model-Driven Development: the Case for Domain-Oriented Programming," Companion of the *18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003* (October 26–30, 2003), Anaheim, CA, ACM, New York (2003), pp. 2–7.
12. J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, John Wiley and Sons, Inc. (2004).
13. *The UML 2.0 Specification*, Object Management Group, http://www.uml.org/#UML2.0.
14. R. Balzer, "Tolerating Inconsistency," *Proceedings of the 13th International Conference on Software Engineering* (May 13–17, 1991), Austin, TX, ACM, New York, pp. 158–165 (1991).
15. S. M. Sutton, Jr., "A Flexible Consistency Model for Persistent Data in Software-Process Programming Languages," in *Implementing Persistent Object Bases—Principles and Practice*, A. Dearle, G. M. Shaw, and S. B. Zdonik, Editors; Morgan Kaufman (1991).
16. W. Opdyke, *Refactoring Object-Oriented Frameworks*, Doctoral Thesis. UMI Order No. GAX93-05645, University of Illinois at Urbana-Champaign (1992).
17. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley (1999).
18. J. Spolsky, *Don't Let Architecture Astronauts Scare You* (2001), http://www.joelonsoftware.com/articles/fog0000000018.html.
19. Gaul, Wikipedia, http://en.wikipedia.org/wiki/Gaul.
20. M. Fowler, *UML Mode* (2003), http://martinfowler.com/bliki/UmlMode.html.
21. M. Fowler, *UML As Sketch* (2003), http://martinfowler.com/bliki/UmlAsSketch.html.
22. M. Fowler, *UML As Blueprint* (2003), http://martinfowler.com/bliki/UmlAsBlueprint.html.
23. M. Fowler, *UML As Programming Language* (2003), http://martinfowler.com/bliki/UmlAsProgrammingLanguage.html.
24. Object Management Group, Model Driven Architecture Web site, http://www.omg.org/mda.
25. S. W. Ambler, *Generalizing Specialists: Improving Your IT Career Skills* (2005), http://www.agilemodeling.com/essays/generalizingSpecialists.htm.
26. A. Cockburn, *Agile Software Development*, Addison-Wesley (2001).
27. S. W. Ambler, *Be Realistic About the UML: It's Simply Not Sufficient* (2005), http://www.agilemodeling.com/essays/realisticUML.htm.
28. *Object Management Group, Meta-Object Facility (MOF) Specification, Version 1.4* (April 2002), http://www.omg.org/docs/formal/02-04-03.pdf.
29. D. A. Thomas, "MDA: Revenge of the Modelers or UML Utopia?" *IEEE Software* **21**, No. 3, 15–17 (May/June 2004).
30. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (2004).
31. M. Fowler, *Model-Driven Architecture* (2003), http://martinfowler.com/bliki/ModelDrivenArchitecture.html.
32. B. Selic, *Speed Development with UML 2.0* (2005), http://www.devx.com/ibmrational/Article/27802.
33. A. E. Bell, "Death by UML Fever," *ACM Queue Magazine* **2**, No. 1 (March 2004).
34. J. Greenfield, *Microsoft and Domain Specific Languages* (2005), *Reprise*, http://blogs.msdn.com/jackgr/archive/2004/12/20/327726.aspx.
35. G. Booch, *Microsoft and Domain-Specific Languages* (2005), http://www.ibm.com/developerworks/blogs/page/gradybooch?entry=domain_specific_languages1.
36. Software Engineering Institute, Computer-Aided Support for the Development and Maintenance of Software, http://www.sei.cmu.edu/legacy/case.
37. Object Management Group, XML Metadata Interchange (XMI) (2005), http://www.omg.org/technology/documents/formal/xmi.htm.
38. The Eclipse Foundation, http://www.eclipse.org.

Accepted for publication January 19, 2006.
Published online July 11, 2006.

Brent Hailpern
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York 10532 (bth@us.ibm.com). Brent Hailpern received a Ph.D. degree in computer science from Stanford University in 1980. His thesis was entitled, "Verifying Concurrent Processes Using Temporal Logic." Dr. Hailpern joined the Watson Research Center as a research staff member in 1980, where he worked on and managed various projects relating to issues of concurrency and programming languages. From 1999–2004, he was the Associate Director of Computer Science for IBM Research. Since 2004, he has been the Department Group Manager for Software Technology, where he manages departments researching programming technology, software engineering, and tools for nonprogrammers. Dr. Hailpern is a past Secretary of the ACM, a past Chair of the ACM Special Interest Group on Programming Languages (SIGPLAN), and a Fellow of the ACM and the IEEE. He was the chair of the SIGPLAN '91 conference on Programming Language Design and Implementation and was chair of SIGPLAN's OOPSLA '99 conference. He is currently the co-chair of SIGPLAN's History of Programming Languages Conference (HOPL-III). Dr. Hailpern is an Associate Editor for ACM's *Transactions on Programming Languages and Systems*.

Peri Tarr
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York 10532 (tarr@us.ibm.com). Peri Tarr received a Ph.D. degree in computer science from the University of Massachusetts at Amherst in 1996. Dr. Tarr joined the Watson Research Center as a research staff member in 1996, where she worked on and led various projects relating to issues of software composition, morphogenic software, and aspect-oriented software development. She currently serves as chief architect for Integrated Solution Engineering, a project that addresses various aspects of full life-cycle software engineering. Dr. Tarr is a past program chair of the Aspect-Oriented Software Development conference and is currently the general chair of the SIGPLAN OOPSLA 2006 conference. ■