MDA Journal

January 2004

David S. Frankel

David Frankel Consulting

df@DavidFrankelConsulting.com

Over the past year, Microsoft has given indications that it takes model-driven approaches to software seriously. Statements emanated from the top of the company about the importance of model orientation. At its recent Professional Developers Conference, Microsoft unveiled a modeling tool for rapidly developing Web services. The company also hired some of the top talent in the model-driven systems field. One of these people is Steve Cook.

Steve had a broad and deep involvement in the discipline of formal specification during his tenure with IBM. He was a key creator of Object Constraint Language (OCL), which became part of UML. He contributed to the Precise UML (pUML) group, and consistently challenged the architects of UML to approach modeling with formality and precision.

In his position at Microsoft, Steve is continuing his work. In this important article, he lays out some fundamentals that Microsoft will follow as it moves increasingly to be a player in model-driven systems.

Steve's emphasis on product lines and domain-specific languages is important. The original work on product lines came out of the Carnegie Mellon Software Engineering Institute, the same outfit that gave us the Capability Maturity Model (CMM). Regarding domain-specific languages, I have pointed out previously on these pages that a general purpose modeling language like UML can only take you so far.

However, although some of us involved with MDA have indicated the importance of domain-specific languages and product lines, Microsoft is doing the industry a service by presenting a thorough integration all of these key concepts, and by expressing a commitment to put the ideas into practice in its tools. Steve references a book about to be released named "Software Factories," which is authored by key people in his group at Microsoft, and to which Steve himself contributed. This book weaves ideas about product lines, domain-specific languages, and models together into an impressive conceptual framework that is more than the sum of its parts.

In this article, Steve explains that Microsoft has come to some negative conclusions about MOF and XMI, two important MDA standards. A response to these conclusions will be published in a future issue of MDA Journal. Nevertheless, these views have to be taken seriously, as they stem from very smart people at a hugely influential company. By diverging from MOF, Microsoft will not be treading exactly along the path of the MDA brand, but the model-driven systems community can manage this divergence, much as the distributed object community managed divergence between CORBA and COM via CORBA-COM interworking standards a short generation ago.

The overwhelming significance of this piece, the Software Factories book, and the corresponding work going on at Microsoft is that model-driven approaches to developing, deploying, and managing computer systems are now percolating at almost every corner of the software industry. It's going to be interesting to watch and even more interesting to take part in.

Until February…

David Frankel

MDA Journal

January 2004

Steve Cook
Software Architect
Enterprise Frameworks
& Tools Group
Microsoft Corporation

Contents

Domain-Specific Modeling and Model Driven Architecture

The Role of Models in Software Development

The development of information systems is getting increasingly complex as they become more and more widely distributed and pervasive in influence. Today's advanced software developer must be familiar with a wide range of technologies for describing software, including modern object-oriented programming languages, XML and its accessories (schemas, queries, transformations), scripting languages, interface definition languages, process description languages, database definition and query languages, and more. Translating from the requirements of a business problem to a solution using these technologies requires a deep understanding of the many architectures and protocols that comprise a distributed solution. Furthermore, end-users expect the result to be fast, available, scalable, and secure even in the face of unpredictable demand and unreliable network connections. It can be a daunting task.

In areas other than software development, such as electronic consumer products (TVs and HiFis), cameras, cars, and so on, we have come to expect a high degree of reliability at low cost, coupled increasingly in many cases with the ability to have items customized to satisfy individual needs. These expectations are met because of advances in industrial manufacturing processes made over many decades. Building a car or a television involves the coordination of a complex chain of manufacturing steps, many of which are wholly or partially automated.

We would like to apply similar principles to the construction of software. The main difficulty in doing so is that we have not yet developed languages for software description that allow effective separation of concerns within the software development process. Although we increasingly use different languages for different tasks – programming languages for writing application logic, XML for transmission of data between application components, SQL for storing and retrieving data in databases, WSDL for describing the interfaces to web-facing components – there are many complexities involved in getting these languages to work effectively together, and none of them directly address the business problem faced by the end user.

This article proposes that the next step towards developing a technology for software manufacturing is the development of *domain-specific languages*, i.e., languages that instead of being focused on a particular technological problem such as programming, data interchange or configuration, are designed so that they can more directly represent the problem domain which is being addressed. We often call such languages *modeling languages*, and we use them to build models of the domains they address.

A model is a computerized representation, where each element in the representation corresponds to an element or concept in the domain. For many

years, models have been important in defining how data is held within IT systems. Today, models are increasingly being used for other purposes, such as modeling of business processes, distributed service implementations, and physical data centers. Models can be attractive because they allow a problem to be precisely described in a way that avoids delving into technological detail. As technology becomes more and more complex, modeling is increasingly necessary in order to be productive. Another advantage of models is that they allow the problem to be described using terms and concepts that are familiar to people who work in the domain of the problem, rather than in terms only familiar to IT experts. We can think of models as a way to help bridge the gap between business and technology, a pervasive problem in today's business environment.

Over the past few years, new modeling technologies have begun to emerge. Notably, the Unified Modeling Language (UML) and related technologies promoted by the Object Management Group (OMG) under the banner of Model Driven Architecture (MDA®) have created a buzz of interest by promising to increase the productivity of software development and the portability of software. On the other hand, we can see parallels between the promotion of MDA and the promotion of Computer-Aided Software Engineering (CASE) tools during the 1980s, and CASE clearly failed to live up to its promises. We should be very skeptical about new claims for model-driven software development unless we can demonstrate new ways of thinking about the problem that will avoid the pitfalls and failure of CASE. Only the emergence of technologies that synergistically combine models, patterns, frameworks, and code into an agile software development process will avoid the CASE trap.

Domain-Specific Languages

If we are going to make it easier for a domain expert to solve problems using models, it is very important that the modeling language clearly represent the problem domain. By modeling language, we mean the definition of the symbols and relationships that are used to construct a model of something in the problem domain. Typically, but by no means necessarily, a modeling language is diagrammatic, and represents a particular model by planar graphs consisting of nodes connected by lines, rather like a flowchart or an entity-relationship diagram.

Although the overall structure of such a model may be reasonably apparent to the casual observer, the devil is in the detail. Models are typically decorated with a lot of symbols and textual elements that must be carefully inspected to see the real meaning. A detailed understanding of how these elements work is necessary to build a complex model successfully. It follows that a significant amount of investment is required in order to become a skilled modeler.

Some would argue that the right approach is to define a general-purpose modeling language and use it for all domains by teaching the domain experts how to use the general-purpose language. Experience with using the Unified Modeling Language (UML) tells us that this is not often successful. We will return to the UML later.

As already stated, we often call modeling languages that are carefully designed to facilitate modeling within a particular problem domain, *domain-specific languages*. Domain-specific languages can be created for numerous problem domains: we know of applications in telecommunications, investment banking, public transport, space exploration, and in many other areas.

However, designing and using a domain-specific modeling language is only a small part of the overall picture of how models can assist software development. There must also be a process by which a model of the domain can be analyzed and validated, as well as transformed, often through several steps, into deployed and executing software. This process involves the development, analysis, and validation of models in several different domains, and tools and processes to transform between one kind of model and another, finishing with the deployed system.

The natural counterpart to a model in the construction of a software system is a *framework*. Such a framework is a body of code that implements the aspects that are common across an entire domain, and exposes as extension points those elements of the domain that vary between one application and another. There are many examples of such frameworks, ranging from the code that supports graphical user interfaces in a personal computer to the code that represents the basic structure and computation of an ERP (Enterprise Resource Planning) application. In each case, the role of a model is to define how to fill in the extension points in a way that suits their particular usage of the framework. In this way we can see the modeling language as a way to make the framework's extension points accessible in a way that suits the user's understanding of the problem.

Another important constituent of the usage of models in software development is the *pattern*. A pattern is essentially a model with some holes in it, together with some rules about how each hole may be filled in by some or all of another model. An effective technology for describing and applying patterns enables large models to be constructed from small ones, and models of one kind to be assembled from models of other kinds.

A further crucial aspect of the use of models, patterns, frameworks, and code within a software development process is that the overall end-result must be *agile*. There must not be any irreversible generation steps or large discontinuities between the model and the deployed code. It must be possible to change any visible artifact within the process and be able rapidly to recreate the end-result. What went wrong with CASE was that it did not make use of frameworks crafted to the domain; instead, it included a large and irreversible code-generation step, such that the code that was generated could not be modified by the developer without invalidating the entire approach, and the developers did want to change the code. Only by designing models, patterns, and frameworks that fit together seamlessly within an overall agile development experience can the pitfalls of CASE be avoided.

The overall process of using models, patterns, and frameworks is quite analogous to what happens in a manufacturing environment for goods and services other

than software. We can think of the software development process by analogy with the more general concept of a value chain, where each participant in the process takes inputs (which might be either physical goods or information assets) from one or more suppliers, uses their particular expertise to add value to these inputs, and passes on the outputs to participants further down the chain. To date, the production of software has rarely been organized along such lines, primarily because the ways for describing software have been limited–when the only way to describe software is the source code, then the only people who can be intimately involved in its production are the programmers. The development of domain-specific models, patterns, and frameworks, organized into software value chains– *product lines,* promises to industrialize the production of software similar to the way in which the production of many household goods was industrialized in the last century[1].

At Microsoft, we firmly believe that modeling is an increasingly important aspect of the software development process, and we will integrate support for modeling into forthcoming releases of Microsoft Visual Studio®. We believe that it is essential to design modeling languages very carefully to suit the skills of their target users: we intend to delight our users by giving them an experience of modeling that is intuitive, agile, productive, and seamless. We are targeting our first modeling products at areas that we believe will give most immediate benefit to our customers. At the recent Microsoft Professional Developers' conference, we announced modeling tools–we call them *designers*–that help the developer to design and deploy distributed service-oriented applications. Details of this announcement, including white papers, can be found at the following URL: http://msdn.microsoft.com/vstudio/enterprise.

Over time, we will expand our range of designers to cover other domains of interest to our customers, such as code visualization, business modeling, and opening up the tool-building environment itself, as well as developing industry partnerships through which other domains will be supported through tools integrated into Visual Studio. Our longer-range vision is to provide integrated support for the entire software development process in many software product lines through integrated patterns, models, frameworks, and tools.

Model Driven Architecture

The phrase Model Driven Architecture (MDA) has achieved some currency within the IT industry as a generic term for describing the use of models within the software engineering process. In fact, the term, which is a registered trademark of the Object Management Group (OMG), refers specifically to a particular approach to model-driven development based on the use of the OMG's modeling technologies, most centrally the Unified Modeling Language (UML) and the Meta-Object Facility (MOF). This section of the article briefly discusses the OMG's Model Driven Architecture; then we move on to focus particularly on the modeling technologies that are involved in the MDA initiative, specifically UML and MOF. In a future article, we will also address the methodology and architecture issues that concern us about MDA.

The essence of MDA is making a distinction between Platform Independent Models (PIMs) and Platform Specific Models (PSMs). To develop an application

using MDA, it is necessary to first build a PIM of the application, then transform this, using a standardized mapping into a PSM, and, finally, map the latter into the application code. According to the OMG's Frequently Asked Questions about the MDA at www.omg.org/mda, "UML is the key enabling technology for the MDA: every application built using the MDA is based on a normative, platform-independent UML model." In this way, it is said, the application's definition is made platform-independent, and thus the application is made portable. This story is highly reminiscent of the Java language's "write once run anywhere" claim. Attempts to create platform-independent frameworks, such as the Swing user-interface component library, necessarily lead to compromises in performance and platform integration. Such compromises have been the primary downfall of many products in the past, and, because of those failures, the industry is now quite skeptical about the claims of MDA, as evidenced by the MDA panel session at OOPSLA 2003.

Nevertheless, an approach like MDA can indeed be helpful for some application areas. Some vendors have shown that building a Web application to run within the J2EE (Java 2 Platform, Enterprise Edition) environment can be simplified by building UML models of the data entities and components involved, and mapping these models into the various J2EE artifacts. However, as we mentioned earlier, for developers to adopt such an approach it is vital for the overall process to be agile and to not introduce unnecessary transitions and barriers such as irreversible code-generation and debugging problems.

Extending the reach of MDA into other areas will be difficult. The OMG's definition of "platform" is vague, and the only real example is J2EE. Using models to assist with creation of J2EE applications is a valid pragmatic step on the road towards using models successfully in software development. In fact, there are very few standardized mappings between platform-independent and platform-specific models; the only ones that exist target the Java platform, although there are non-standard commercial implementations that claim to be MDA-compliant and target other platforms.

In summary, MDA is misnamed: it is not an architecture at all; it is a standardized approach to model-driven development based on abstraction of platform similarities. As promoted by the OMG, it does not address the broader issues involved in using integrated models, patterns, frameworks, and tools synergistically to support software product lines. Furthermore, as we shall see, the fact that the MDA is based on the use of the UML and MOF specifications restricts its usefulness even more.

The Unified Modeling Language

UML is a general-purpose modeling language. The development of the UML started during the early 1990s when it emerged as a unification of the diagramming approaches for object-oriented systems developed by Grady Booch, James Rumbaugh, and Ivar Jacobson. First standardized in 1997, it has been through a number of revisions, most recently the development of version 2, which is currently nearing the end of its finalization process.

UML is large and complicated, version 2 especially so. To understand UML in any depth it is important to understand how it is used. We follow the lead of Martin Fowler, author of "UML Distilled," one of the most popular introductory books on UML. Martin divides the use into UMLAsSketch, UMLAsBlueprint, and UMLAsProgrammingLanguage. For more details, see http://martinfowler.com/bliki.

UMLAsSketch is very popular. Sketches using UML can be found on vast numbers of whiteboards in software development projects. To use anything other than UMLAsSketch as a means of creating informal documentation for the structure of an object-oriented design would today be seen as perverse. In this sense, UML has been extremely successful, and it has entirely fulfilled the aspirations of its creators who wanted to eliminate the gratuitous differences between alternative ways of diagrammatically depicting an object-oriented design.

UMLAsBlueprint raises the bar for UML. Here the objective is to integrate one or more UML models as first-class artifacts within a software development process. Either manually or automatically, UMLAsBlueprint relies on systematically translating a UML model into source code. This means that the UML model has to contain enough information so that the translation can be effective and complete.

When we try to do this, we immediately find problems with UML, because it does not translate very directly into the technologies that we use. For example, a UML class cannot be used directly to depict a C# class because a UML class does not support the notion of properties while a C# class does. Similarly, a UML interface cannot be used directly to depict a Java interface because a UML interface does not contain static fields while a Java interface does. From the point of view of UMLAsSketch, these are non-problems: the sketcher simply adds the desired elements to the sketch and moves on. However, if the UML model is to be used as a first-class development artifact, either the standard must be violated, or uncomfortable and intrusive translations must be introduced into the development process to overcome these mismatches.

UMLAsProgrammingLanguage is an initiative supported by a rather small community, which is unlikely to gain much headway commercially, and which we will not dwell upon in this article.

We observe that for the vast majority of uses of UML–sketching and blueprinting—it would have been much more helpful had the standard been formulated as a set of flexible and extensible diagramming conventions which allowed the appropriate technology mappings to be developed seamlessly without the mismatches described above. Only with seamless and reversible mappings will the concept of a model as blueprint be accepted by the vast majority of developers. The technology of "UML profiles" is intended to offer flexibility in the blueprinting world by permitting a limited level of extensibility to the language, but practice has found that the very limited level of extensibility provided by profiles does not permit a seamless mapping from UML into popular target technologies.

At Microsoft, our approach within our new modeling designers will be to take advantage of UML to the extent that it provides recognizable notation for well-understood concepts. At the same time, where we discover that UML notation is not clear enough, we'll supplement it with our own forms. For example, we will visualize .NET classes in a UML-like way, extended to convey additional information, enhance usability, and make the meaning of the diagram elements more precise. This allows real-time round-tripping—a vital requirement— and ensures that the terminology and concepts of .NET appear in the diagrams. Feedback from our customers is overwhelmingly supportive of this approach. Although the diagrams are not legal UML by the letter of the standard, the meaning of the diagrams is readily apparent to anybody that understands the UML conventions.

For domains that UML does not address, we must create new conventions. For example, the designer that supports the development of models of logical data centers, which we use to model the deployment of distributed applications, uses concepts that are either unsupported or very poorly supported by UML, and so we are creating new conventions for such domains. As domain-specific languages are developed for new domains outside the scope of the UML, we can expect new conventions to emerge in those domains as well.

We may note in passing that the development of version 2 of UML extends the language with a lot of material that originated in another modeling language, SDL (Specification and Description Language), widely used in the telecommunications field. We expect these new aspects of UML to provide diagrammatic conventions, and, to some extent, a standard blueprinting technology, in those domains.

Defining Languages and Interchanging Models

The other central technology promoted by the OMG under the MDA banner is the Meta-Object Facility (MOF). MOF is a more abstract thing to understand than UML, and this understanding is made more difficult by the use of jargon terms like metamodel and even meta-metamodel (although thankfully there is no meta-meta-meta-model). We will do our best to avoid such jargon.

MOF essentially does two jobs. First, it is a domain-specific modeling language designed for defining modeling languages: a MOF model is a definition of an MDA modeling language. Second, it is a mechanism that determines how a model defined in an MDA modeling language can be serialized into an XML document and represented via a CORBA or (using an additional specification issued through the Java Community Process) a Java API. We will examine how well it does these jobs.

A modeling language for a domain has several aspects. It must define the concepts of the domain; it must represent these concepts diagrammatically, or in text; it must define ways by which a user may interact with the language; it must define what a valid model is and is not; and it must define how models will be interchanged. But MOF only defines the basic concepts in a language, and

how models of those concepts are to be stored and interchanged. A MOF description of a language describes little about those aspects that are of direct interest to its user: what models in the language actually look like, or how the user interacts with them.

At Microsoft, we want our languages to be fully integrated into the Visual Studio IDE, including IntelliSense®, toolbox, menus, property grid, and debugging support. We find that defining the way the modeling concepts are stored is a rather minor aspect of the whole job. We also find that the tools we need for integrating the language definition into the overall Visual Studio experience require facilities rather different from those provided by MOF.

In fact, although it is often positioned as a technology for language definition, MOF is primarily a technology for the storage of conceptual models, exposing these models via CORBA or Java APIs, and interchanging them, using XMI (XML Metadata Interchange), another of the OMG's technologies. If the concepts of a language are defined using MOF, then an XML-based serialization for the language can be automatically generated using the XMI approach.

On the face of it, this may seem rather attractive. However, there are problems. First of all, the XML generated is dependent upon the language definition. This means, for example, that the XMI serialization of a model defined in UML1.4 cannot be understood by an implementation of UML2.0, even in those areas of the language that are effectively identical from the user's perspective. Secondly, the definition of XMI itself is subject to change. This means that there is XMI1.0 serialization of a model in UML1.3, an XMI1.1 serialization of the same model, etc. And, thirdly, the definition of MOF is also subject to change, which adds a further term to the set of possible combinations; indeed, the latest proposed version of MOF has two flavors, which are not fully compatible with each other. Thus, although XMI claims to provide interoperability of modeling tools, in practice, unless every tool supports every possible combination of each of the MOF, XMI and UML standards, interchange is problematical. A further problem with XMI, especially with early versions, is that its documents are instances of machine-generated schemas that tend to be verbose and difficult to read; this goes against the grain for most developers who will want to apply the many readily-available technologies for visualizing, transforming, and manipulating XML documents.

We do not think that XMI is the right approach for model serialization. The world of XML itself is becoming very mature, with large numbers of schemas and tools available in the marketplace. We believe that the right approach for serializing a particular modeling language is to purpose-build a schema for that language, and to provide tools that explicitly manage and automatically interpret the mapping between the language and its serialization format. If standardization is appropriate within a particular domain, then this schema should be standardized: indeed, this is already a widespread practice within the industry. Then, if the language definition needs to evolve, the new definition can serialize using an extension of the schema for the old definition, allowing an effective migration path. XMI effectively prohibits this clean evolution and causes a proliferation of incompatible serialization formats, entirely at odds with its interoperability goals.

In summary, Microsoft does not support MOF for the following reasons: 1) it is not yet a single stable standard; 2) using it as the language for designing our tools would have far reaching practical consequences on those tools that we are unwilling to accept; 3) addressing the missing elements of the MOF required by commercial grade implementations (e.g., notation, transactions, events, etc.) will continue to introduce major changes into the MOF specification; and 4) MOF's approach to model serialization fails to meet its goals.

Conclusion

This article has discussed the role of models in software development, in particular the definition and use of domain-specific languages, and their use within software product lines. It has also commented on the suitability of the OMG's MDA technologies within this overall picture. We strongly believe that the industry will increasingly adopt models as first-class artifacts within an agile software development process, and we are building tools and technology to support this development. We see UML as an important step on the road, whose future is primarily as a set of diagrammatic conventions that facilitate communication between developers, and that can be used as inspiration for the design of domain-specific languages targeted at particular problems faced by our customers. We see XML as a key technology for representing and interchanging models, and we expect the progressive standardization of schemas representing domain content to continue apace.

Footnotes

[1] For a comprehensive and detailed discussion of the forthcoming industrialization of software development see the book *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools* by Jack Greenfield and Keith Short.

Author

Steve Cook is a Software Architect in the Enterprise Frameworks and Tools group at Microsoft, which he joined at the beginning of 2003. Previously he was a Distinguished Engineer at IBM, whom he represented in the UML 2.0 specification process at the OMG. He has worked in the IT industry for almost 30 years, as architect, programmer, consultant and teacher, and has focused on modelling languages and tools since the 1980s. He has published a book and many papers and articles on software-related topics.