A technical discussion of software modeling
June 2004

Rational software

The Value of Modeling

Gary Cernosek
IBM Rational Software Design and Construction Products
IBM Software Group

Eric Naiburg
IBM Rational Desktop Products
IBM Software Group

Contents

Executive summary

This white paper discusses how modeling can positively affect software and systems development. The intended audience includes both technical and non-technical personnel associated with the software development process.

Modeling can be an effective way to manage the complexity of software development. It enables communication, design and assessment of requirements, architectures, software and systems. In spite of these virtues, mainstream software development has yet to take advantage of modeling in everyday practice.

This white paper examines how modeling provides not only visual but also textual content, and why the combination is important. It also explains how to model throughout the various phases of the software development life cycle and what modeling types are appropriate for each phase. Although the focus will be on modeling as a discipline in itself, the Unified Modeling Language (UML) will be used as the common means for expressing the models.

Introduction

This white paper discusses the value of modeling in the context of software development. The concepts presented here are not new—savvy software professionals have practiced modeling for years. But in the mainstream software development community, only a fraction of software developers formally model their software. This white paper examines the basics of what motivates the practice of modeling software. For those who are knowledgeable of software modeling, those who know nothing about it or those who know of it but have never embraced it, this white paper intends to explain the benefits and values that this practice can offer.

What is modeling?

For many years, business analysts, engineers, scientists and other professionals who build complex structures or systems have been creating models of what they build. Sometimes the models are physical, such as scaled mock-ups of airplanes, houses or automobiles. Sometimes the models are less tangible, as seen in business financials models, market trading simulations and electrical circuit diagrams. In all cases, a model serves as an abstraction—an approximate representation of the real item that is being built.

Why model?

Why should you model something before you build it? Perhaps you should not. Simple things do not necessarily need a model preceding its construction—such as a simple checkbook register, a currency conversion utility, a doghouse or a simple macro in a word processor that opens up a set of routinely used files. Such projects share all or most of the following characteristics:

- *The problem domain is well known.*
- *The solution is relatively easy to construct.*
- *Very few people need to collaborate to build or use the solution (often only one).*
- *The solution requires minimal ongoing maintenance.*
- *The scope of future needs is unlikely to grow substantially.*

But suppose none of these characteristics apply? Why do some professional disciplines bother to create models? Why do they not just build the real thing right away? The answer has to do with the complexity, the risk and the fact that original practitioners are not always appropriate or even available for completing the task.

Modeling provides architects and others with the ability to visualize entire systems, assess different options and communicate designs more clearly before taking on the risks.

It is neither technically wise nor economically practical to build certain kinds of complex systems without first creating a design, a blueprint or another abstract representation. While professional architects might build a doghouse without a design diagram, they would never construct a 15-story office building without first developing an array of architectural plans, diagrams and some type of a mock-up for visualization.

Modeling provides architects and others with the ability to visualize entire systems, assess different options and communicate designs more clearly before taking on the risks—technical, financial or otherwise—of actual construction.

Why model software*?*
For years, the practice of software development was exempt from many of these modeling issues. By its very nature, software can be easily created and easily changed. Little capital equipment is required, and virtually no manufacturing costs are incurred. These attributes cultivated a do-it-yourself culture—imagine it, build it and change it as often as necessary. There is no "final" system anyway, so why even try to conceive of one before writing code?

Today, software systems have become very complex. They must be integrated with other systems to run the items used in everyday life. Automobiles, for example, are now heavily equipped with computers and associated software to control everything from the engine and cruise control to all kinds of new onboard navigation and communication systems. Software also is heavily used to automate business processes of all kinds—those that are seen and experienced by customers and those that are in the back office.

In many organizations, software development is no longer a cost-center overhead line item—it is an integral part of the company's strategic business processes.

Some software systems support important health-related or property-related functions, making them necessarily complex to develop, test and maintain. And even those systems that are not critical to human health or property can be critical to businesses. In many organizations, software development is no longer a cost-center overhead line item—it is an integral part of the company's strategic business processes. For those companies, software has become a key discriminator in competing in the marketplace.

For these reasons and more, developers need a better understanding of what they are building, and modeling offers an effective way to do that. At the same time, modeling must not slow things down. Customers and business users still expect software to be delivered on time and to perform as expected on demand. To achieve this "fast and good" goal, IBM sees four imperatives for software development: develop iteratively, focus on architecture, continuously ensure quality and manage change and assets.

The same basic reasons why other complex, high-risk systems are modeled also apply to software—to manage the complexity and to understand the design and associated risks. More specifically, by modeling software, developers can:

- *Create and communicate software designs before committing additional resources*
- *Trace the design back to the requirements, helping to ensure that they are building the right system*
- *Practice iterative development, in which models and other higher levels of abstraction facilitate quick and frequent changes*

Why some developers choose not to model software

Despite the many reasons and virtues behind modeling, a great majority of software developers still do not employ any form of abstraction higher than that of source code. Why? As described earlier, sometimes the actual complexity of the problem or solution does not warrant it. Again, if you are building a doghouse, you do not need to hire an architect or contract a builder to produce a set of design specifications. But in the world of software, systems often begin simple and well-understood and then—through the natural evolution of a successful implementation—become more and more complex. In other cases, developers choose not to model because they simply do not perceive a need for it until much too late.

Many will argue that the resistance to modeling software is more cultural than anything else. Traditional programmers are very proficient at conventional coding techniques. Even when unexpected complexity begins to encroach, most developers are comfortable sticking to their integrated development environment (IDE) and debugger and simply working more hours on the problem. Because modeling requires additional training and tools, a corresponding investment in time, money and effort is needed—not at the time of toil, but early in a project's development life cycle. The reason traditional developers are not more proactive in this regard is that they believe modeling will slow them down. The next section intends to help dispel this notion.

When do I model?

Modeling complex applications has several general benefits. Some specific situations in which the modeling effort is worthwhile include:

- *To better understand the business or engineering situation at hand ("as-is" model) and to craft a better system ("to-be" model)*
- *To build and design a system architecture*
- *To create visualizations of code and other forms of implementation*

Modeling is not an all-or-nothing proposition. Models can play a part in the software development process in many ways. Figure 1 illustrates the spectrum of ways to practice model-driven development.

Integrated development environments. In the loosest notion of modeling, IDEs can be considered an entry point into the practice of model-driven development. Modern IDEs offer several mechanisms that raise the level of abstraction in creating and maintaining code. Tools such as language-sensitive editors, wizards, form builders and other GUI controls are not "models" in the more strict sense of the term. Nonetheless, they can raise the level of abstraction above source code, make developers more productive, help create more reliable code and enable a more effective maintenance process. All these attributes are the essence of model-driven development.

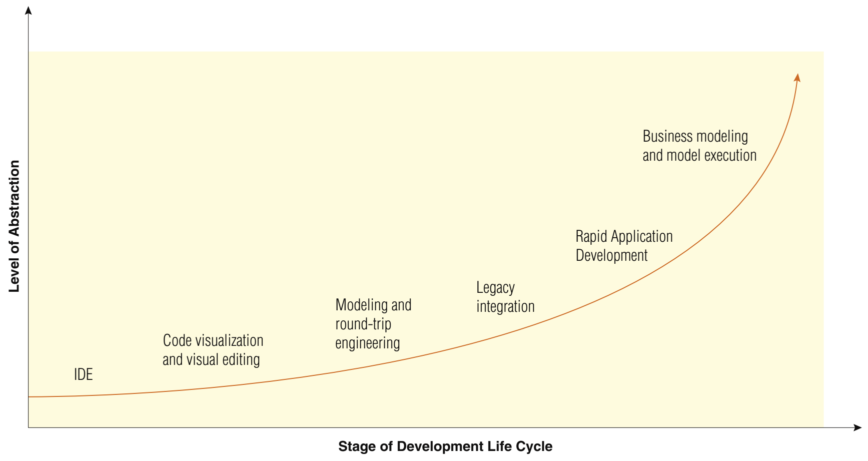

Figure 1. A spectrum of times, places and ways to model.

Code visualization and visual editing. A step above the basic IDE functions is the ability to visualize source code in graphical form. Here, a picture is worth a thousand lines of code, in a sense. Developers have used graphical forms of abstraction above their code for many years. Traditional flow charts are a common method for depicting the algorithmic control flow of code. Structure charts, or even simple block diagrams with arrows, are often used on whiteboards—using boxes to represent functions and subprograms, arrows to indicate calling dependencies and so on. For object-oriented software, boxes typically denote classes and lines between boxes denote relationships between those classes.

Coupled closely with code visualization is visual editing, in which developers edit code through the diagrams instead of through conventional IDE text windows. Visual editing is well suited for changes that have systematic effects on other pieces of code. For example, in an object-oriented system that has a set of classes related in an inheritance hierarchy, certain features of the classes (the field members, methods or functions) may need to be reorganized into different classes (a process called refactoring) as the application evolves. Using conventional code editors to enact such changes can be tedious and error-prone. But an effective visual editor allows developers, for example, to drag and drop a member function from one class to its base class and automatically adjust all code that is affected by such a change.

Code visualization and visual editing are simply alternative methods for viewing and editing the code.

In one sense, code visualization and visual editing are simply alternative methods for viewing and editing the code. Changes to the code are immediately reflected in corresponding diagrams and vice versa. Although some may argue that such depictions do not constitute a "model," the essence of modeling is abstraction and any visualization of code is indeed an abstraction—selectively exposing certain information while suppressing details deemed unnecessary or unwanted. Some practitioners prefer to use terms such as code model, implementation model or platform-specific model (PSM) to qualify such abstractions from other, higher-level forms of modeling that do not have such direct relationships to the code.

Modeling and round-trip engineering. The next step on the modeling spectrum represents the state of conventional model-driven development. Here, visual models are created from a methodological process that begins with requirements and delves into a high-level architectural design model. Developers then create a detailed design model from which skeletal code is generated to an IDE. The IDE is used to complete the detailed coding. Any changes made to the code that affect the design model are synchronized back into the model; any model changes are synchronized into the existing code.

Legacy integration. When developers are ready to integrate systems—whether all legacy or some new systems—they must understand the systems in place, know how the business intends for these systems to work together and prioritize those integrations. Modeling legacy systems does not necessarily mean that the entire system and all its components must be incorporated; however, developers should understand the legacy systems' architectures, how they work and how they interface with others. Understanding what the system does and what other software is dependent on it will help determine suitable steps moving forward.

Several methods can be used to model legacy systems. Developers can reverse-engineer code into models to understand them, manually model them or use some combination thereof.

Rapid Application Development (RAD). The practice of RAD has been around since the early 1980s. The premise is simply to provide highly productive ways to generate and maintain code. RAD is accomplished through easy-to-use, highly graphical features of an advanced IDE. RAD, distinct from both code-centric and model-driven development, raises the level of abstraction above the code, but does not use "models" per se.

Business modeling and model execution. Before the need to develop software is even known, business and engineering analysts often find it useful to create "as-is" models of how their systems work today. From that model, they can analyze what works and what needs improvement. Special-purpose tools can simulate these models along several key variables, such as time, cost and resources. From the analysis, "to-be" models can be built to prescribe how new, improved processes should work. Generally, new software development is needed to implement the new processes, and the "to-be" models serve as key drivers for the ensuing development.

For some application domains, the "to-be" models are specified to such rigor that complete applications can be generated from the models. Modeling at this level of abstraction offers the greatest potential for productivity and integration between the business or engineering problem domains and the technology or implementation domains.

Adopting a standard notation such as UML is an important step in taking a model-driven approach to software development.

How do I model?

The software industry has adopted the Unified Modeling Language as its standard means for representing software models and related artifacts. Software architects, designers and developers use UML for specifying, visualizing, constructing and documenting all aspects of a software system. Key leaders from IBM Rational led the original development of UML. Today, UML is managed by the Object Management Group (OMG), which consists of representatives throughout the world to help ensure that the specification continues to meet the dynamic needs of the software community. Adopting a standard notation such as UML is an important step in taking a model-driven approach to software development.

UML is more than just a graphical notational standard—it is a modeling language. As with all languages, UML defines syntax (both graphical and textual, in this case) and semantics (the underlying meanings of the symbols and text). Having a true modeling language rather than just a standard notation is essential for standardizing the use of UML as well as for helping to ensure that automated tools can properly enforce the rules behind the symbols. UML—a true modeling language—has helped it become the software industry's most recognized and widely applicable modeling standard.

What people are saying about the value of modeling

Like any technology, UML had early adopters that led the charge in discovering its value. Here are just a few comments from IBM Rational® customers about the value that modeling contributed to their businesses:

> *"We are trying to reduce the overall cost of insurance to our members. One of the ways to do that is to reuse information and reuse the assets that we build as we go through our business modeling…Model-driven architecture is really at the core of what we're doing from a business modeling perspective. When we begin projects from a software development perspective without a clear business model, without a clear set of business objectives or business goals, we are finding that the customers don't get what they think they have asked for."*

— Sue Nelson, director of business modeling for Blue Cross and Blue Shield of Florida

> *"I think visual modeling is just a key element in any developer's toolbox. It enables us to bring in specialized expertise, such as security analysis of a product. By having a common modeling technique that everyone knows how to read, we can bring in our company security expert and that person can very easily review the product and point out any potential holes."*

— Nanette Brown, director of applied architecture and quality assurance at Pitney Bowes

> *"Enterprise architecture presents its own very unique modeling challenges. You are modeling at multiple levels. You are modeling with large groups of people and different teams. And the models at each level tend to have to be customized for the individual stakeholder types. [Modeling with UML] provided us with the flexibility [to meet] our unique needs and demands at each level of the enterprise architecture."*

— Frank Armour, president of ArmourIT, LLC

Testimonials like these can show the reduced risks to others who are just getting started with modeling and can ultimately help position modeling closer to the mainstream of software development.

Trends and the future

Ask any software development professional, "Where is the software industry heading?" and you will probably get a wide array of responses. But one trend seems to be quite common:

> *Software development continues to grow in complexity, and developers must work at increasingly higher levels of abstraction to cope with this complexity.*

Modeling software is and will continue to be a key way that developers work at those higher levels. The following specific trends are noteworthy at this time.

Beyond visual modeling. UML has traditionally been associated with a graphical means for depicting software artifacts. While this remains true, there is a growing importance in modeling "under the hood." Meta-modeling is the discipline of having "models of models." The most evident and practical application of meta-modeling can be seen in UML Version 2, which forms the basis for how automated tools share data and interoperate with one another. This applies not only to modeling tools, but also to tools for requirements management, compilers, testing, configuration management and other aspects of the software development life cycle. All of these areas can become better integrated as a result of a common underlying meta-model, such as that afforded by UML 2 and its associated modeling standards.

Unifying software, data and business modeling. This white paper has focused primarily on the value of modeling software. But data modeling and business modeling have been in practice for even longer in some form. The problem is that these types of modeling have traditionally involved entirely different worlds of modeling languages and practitioner cultures. Now the promise for unifying these three worlds is evident—not necessarily with a single modeling language or tool, but through a combination of multiple, open industry standards that are converging.

As business modeling becomes more standardized and integrated with data and software, a model-driven business integration discipline will likely emerge.

Modeling across the life cycle. As standards continue to evolve, modeling will become applicable to an even broader range of activities across the software development life cycle. Applications of modeling are already driving testing and other aspects of quality assurance earlier in the life cycle. And as business modeling becomes more standardized and integrated with data and software, a model-driven business integration discipline will likely emerge.

Domain-specific modeling languages. UML and other modeling languages allow developers to focus on levels of abstraction above implementation details. As shown earlier in Figure 1, modeling includes a wide spectrum of levels, even when removed from actual code. Taken to the highest level of abstraction, a business or domain model focuses not on software, but on the nature of the problem under consideration. Here, the model should use terms and icons familiar to the people and systems of that particular business or application field.

The industry is moving toward domain-specific languages—special-purpose modeling languages dedicated to their respective area of use. More often, however, general-purpose modeling languages—UML, in particular—are extended in standard ways to meet domain-specific modeling needs through innovations such as profiles. Both approaches are consistent with the value of modeling in general: to provide abstractions for specifying problems and solutions in a more productive and effective manner.

The business of software development. Many have called software development a "team sport." A companion statement would be that it is an "international team sport." With today's technology, the software development has no geographical boundaries. The business of software development will likely continue to be distributed and global. Modeling and other higher forms of abstraction will be crucial for helping the practitioner handle the associated complexity.

Model-Driven Architecture (MDA): The next step. MDA is an initiative led by the Object Management Group. While still in its early-adopter stage, MDA can be considered the next logical step in the evolution of modeling and model-driven development technologies. MDA, based on UML and other related standards, focuses on defining models at varying levels of abstraction and on the transformations defined between these different levels. Automated tool support is crucial to the evolution and successful application of MDA.

IBM Project information

Form Number:	**G000-000-000**	Title:	**IBM Project Description**
Announce date: **number**	**00/00/00**	IBM Contact:	**IBM Contact Name/phone**

Agency information: Name of agency

Job Number:	**000000**	Contact:	**Agency contact/phone number**
File name:	**Name of the current page layout file**		
Based on:	**Name of the file this file is based on**		
Version: **system**	**2-00/00/00**	Location:	**Location of job on agency**
Station:	**Station identification**	Operator:	**Initials of opertor(s) rrr/rrr/qqq**
Trim size:	**Width x height**	Output size:	**Width x height**
Output device: **Neg-Film)**	**Name and model**	Output style:	**Type of output (eg., RRED-**
Line screen:	**000 lpi**	Colors:	**List of all plates for output**
Document fonts: **BQBodoni Light)**	**Complete screen font name (eg. Helvetica Regular, Berthold**		

Graphic Data Chart

File Name:	Page #	File Type:	Usage rights	Photographer	Stock House
eb_pos_clr.eps	**1**	**IBM owned**	**All**		
ibmpos.eps	3	IBM owned	All		