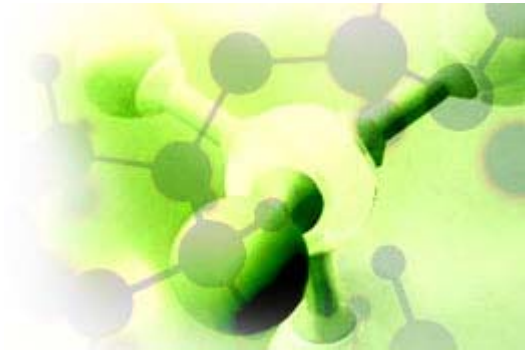
The value of modeling



Level: Introductory

Gary Cernosek, Marketing Manager, IBM Rational
Eric Naiburg, Group Market Manager Desktop Products, IBM Rational

15 Nov 2004

from The Rational Edge: This article discusses how modeling can help development teams manage complexity and enable communication, design, and assessment for requirements, architectures, software, and systems. In addition to explaining how to use models productively throughout the software development lifecycle, it looks at trends affecting the future of 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 scale mock-ups of airplanes, houses, or automobiles. Sometimes the models are less tangible, as is true for financial models, market trading simulations, and electrical circuit diagrams, for example. In all cases, the model is an abstraction -- an approximate representation of the real item to be built. What is modeling's main value? Modeling allows you to visualize entire systems, assess different options, and communicate designs before you take on the risks associated with actually building something. This article elaborates on the benefits of modeling, explaining how it can help development organizations manage complexity in architecting both individual applications and the context in which they must operate. It concludes with a look at industry trends that underscore the demand for modeling.

What should you model?

Should you model everything before you build it? Absolutely not. You probably don't need a model to create things such as a simple checkbook register, a currency conversion utility, a doghouse, or a simple macro to open frequently used word processor 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 person is involved).
- The solution requires minimal ongoing maintenance.
- The scope of future needs is unlikely to grow substantially.

If your project has none of these characteristics, then modeling might be in order. 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. Although most professional architects might build a doghouse without a design diagram, they would never construct a fifteen-story office building without first developing an array of architectural plans, diagrams, and mock-ups to help themselves and others visualize the structure. Modeling helps architects visualize entire systems, assess different options, and communicate designs before they and their clients take on the risks -- technical, financial, and more -- of actual construction.

Why model software?

When computers first appeared, software development projects fit the criteria in the bulleted list above. While not being a trivial effort, the amount and complexity of code in early systems was manageable by informal means. By its very nature, software can be easily created and easily changed. It requires little capital equipment and incurs virtually no manufacturing costs. These attributes cultivated a do-it-yourself culture among developers. "All I have to do is imagine it, build it, and change it as often as necessary," they thought. "There is no 'final' system anyway, so why even try to conceive of one before writing code?"

This is not a good way to approach today's software systems, which have become very complex. In many cases, developers must integrate them with other systems to run the machines we use in our everyday lives. Automobiles, for example, are now heavily equipped with computers and associated software to control everything from the engine and cruise control to new onboard navigation and communication systems. Developers also build software to automate business processes of all kinds -- both those in the back office and those that customers see and experience.

Software systems that support important health-related or property-related functions are necessarily complex to develop, test, and maintain. And some systems are 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 for competing in the marketplace.

To build these complex systems, developers need a better understanding of what they are building. Modeling provides that. But it 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 meet these "fast and good" expectations, software developers can follow four imperatives: Develop iteratively, focus on architecture, continuously ensure quality, and manage change and assets. Modeling is an integral part of all four imperatives. It enables developers to:

- Create and communicate software designs before the organization commits additional resources.
- Trace the design back to the requirements, helping to ensure that the project team is building the right system.
- Practice iterative development, using models and other abstraction techniques to 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 a form of abstraction higher than that of source code. Why? As we noted earlier, sometimes the problem does not warrant one. 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 that are simple when first implemented often become increasingly complex as developers make changes in response to new requirements or to integrate with other systems. In other cases, developers choose not to model simply because they do not perceive a need for it -- until it is much too late.

Many experts argue that the resistance to modeling software is more cultural than anything else. Traditional programmers are very proficient at conventional coding techniques. Even when the complexity level rises unexpectedly, programmers are comfortable sticking with their integrated development environment (IDE) and debugger and simply working more hours on the problem. Why? Because modeling requires additional training and tools, and corresponding investments in time, money, and effort early in a project's development lifecycle. Traditional developers are typically eager to begin coding and believe that modeling will slow them down. In the next section, we will attempt to dispel this notion.

When should you model?

Broadly speaking, modeling complex applications can help software development organizations to:

- Better understand the current state of the business or technical infrastructure ("as-is" model) and craft a solution that directly addresses current business or technical needs ("to-be" model).
- Build and design an efficient system architecture.
- Create visualizations of code and other forms of implementation that enable better communication among all project team members.

However, it is important to remember that modeling is not an all-or-nothing proposition. Software development organizations can pick and choose where to use models in the software

development process, as shown in Figure 1. We will examine each of these modeling opportunities below.

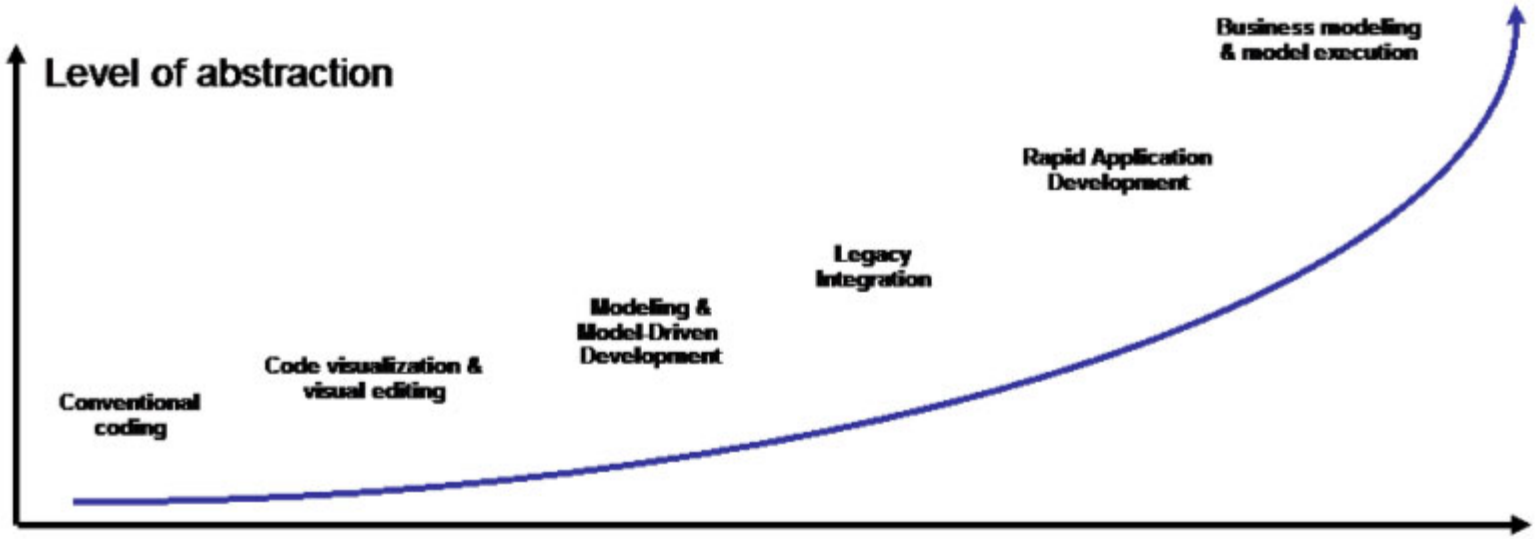


Figure 1. A spectrum of times, places, and ways to model

Conventional coding

An organization's IDE is a logical entry point for the practice of model-driven development because modern IDEs already offer tools that raise the abstraction level for creating and maintaining code: language-sensitive editors, wizards, form builders, and other GUI controls. Like models, these tools help developers to be more productive, create more reliable code, and institute a more effective maintenance process.

Code visualization and visual editing

Modeling goes a step beyond the basic IDE functions by allowing developers to visualize source code in graphical form. Many agree that a picture is worth a thousand lines of code. That is why most developers already use a form of modeling: graphical abstractions such as traditional flow charts to depict algorithmic control flows and structure charts or simple block diagrams with boxes representing functions and subprograms, arrows indicating calling dependencies; and so on. In object-oriented software development, boxes typically denote classes, and lines between boxes denote relationships between those classes.

Coupled closely with these methods of visualizing code is visual editing, which allows developers to edit the code through diagrams instead of through conventional IDE text windows. Visual editing is well suited for changes that have systemic effects on other pieces of code. For example, in an object-oriented system with a set of classes related in an inheritance hierarchy, certain features of the classes (field members, methods, or functions) may need to be reorganized into different classes (a form of 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 to, for example, drag and drop a member function from one class to its base class and automatically adjust all code affected by that change.

Although some may argue that a set of diagrams does not constitute a "model," the essence of modeling is abstraction, and any form of code visualization is also an abstraction. Like models, diagrams selectively expose certain information while suppressing lower-level details. However, some practitioners prefer to use terms such as *code model, implementation model,* or *platform-specific model (PSM)* to distinguish such abstractions from higher-level models that do not have such direct relationships to code.

Modeling and model-driven development

A more advanced form of model-driven development involves creating models through a methodological process that begins with requirements and delves into high-level architectural design. These initial models are one or more levels removed from code or other implementation technologies. Developers then create a detailed design model from which skeletal code can be generated to an IDE, which developers then use to do more granular coding. Any coding changes they make 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 charged with integrating disparate systems—whether the systems are all legacy or a mix of legacy and new—they must first understand those systems' architectures. In addition, to prioritize the integration projects and activities, they must also understand how the business wants the systems to interact. Modeling legacy systems does not necessarily require depicting all components of all systems; developers need just enough information to understand how the architectures work and interface with one another. Understanding what the system does and what other software depends on it will help the team determine what steps to take for the integration and in what order.

Developers use several methods to model legacy systems. They can reverse-engineer code into models to understand the systems, manually model them, or use a combination approach. *Whatever method they choose, adopting a standard notation such as the Unified Modeling Language, or UML, is a necessary step for communicating what they learn about these systems and for effective model-driven development.* We will discuss this in greater detail below.

Rapid application development

The practice of rapid application development has been around since the early 1980s. Its purpose is simply to provide highly productive ways to generate and maintain code. The practice is based on easy-to-use, highly graphical programming features found in advanced IDEs. Rapid application development, in contrast to both code-centric and model-driven development, raises the abstraction level above the code, but it does not use models *per se*.

Business modeling and model execution

Aside from developing specific software systems, business and engineering analysts often find it useful to create "as-is" models of how their systems work today. From that model, they can then analyze what works and what needs improvement. With special-purpose tools, they can also examine the model along several key variables, such as time, cost, and resources. This yields information to create "to-be" models that prescribe how new, improved processes should work. Typically, new software development is needed to implement the new processes, and the "to-be" models can guide that development.

In some application domains, the "to-be" models are specified in enough detail to generate complete applications from the models. Modeling at this level of abstraction offers the highest degree of automation in the development process, resulting in very high productivity and leaving less chance for implementation errors. This level of model-driven development further helps to ensure that the systems you create will truly satisfy the most important business and/or engineering needs.

Why use UML?

The software industry has adopted the UML as a standard 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, which is currently managed by the Object Management Group (OMG). Through that organization, representatives from around the world help ensure that the specification continues to meet the software community's dynamic needs.

Adopting the UML is an important step in taking a model-driven approach to software development because UML is more than just a standard notation system; it is actually a *modeling* language. It defines syntax (both graphical and textual) and semantics (the underlying meanings of the symbols and text). Standardizing on UML can help organizations ensure that their automated tools will all recognize the same symbols and enforce the rules behind them. In turn, that ensures that models will be consistent across the organization and understandable to all.

Trends and the future of modeling

Ask any software development professional, "Where is the software industry heading?" and you're likely to conclude some version of the following statement:

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 method that developers use to work at a high level of abstraction. The following specific trends are noteworthy at this time.

Meta modeling. UML has traditionally provided a graphical means for depicting software artifacts, and developers still use it primarily for this purpose. However, there is a growing trend toward modeling "under the hood" through meta-modeling -- creating "models of models." The most evident and practical application of meta-modeling is reflected in UML Version 2, which provides a foundation for automated tools to 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 lifecycle. An underlying meta-model, as specified in UML 2 and its associated modeling standards, provides a way to improve integration among all of these areas.

Unifying software, data, and business modeling. This article has focused primarily on the value of modeling software, but we should remember that organizations have recognized the value of data modeling and business modeling for a long time. The problem is that the modeling and languages and tools these disciplines use are typically worlds apart. However, there is now a promising prospect for unifying these separate disciplines, not necessarily with a single modeling language or tool, but through a combination of multiple, open, converging industry standards.

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

Domain-specific modeling languages. As we saw in Figure 1, modeling is useful for a wide spectrum of domains and activities, some of which are unrelated to coding. At the highest level of abstraction, a business or domain model focuses not on software, but instead on the nature of the problem under consideration; the model should use terms and icons familiar to the people in that particular business area.

Development organizations for some businesses are adopting domain-specific languages—special-purpose modeling languages dedicated to their respective area of use. However, most development organizations extend a general-purpose modeling language—UML, in particular—to meet their domain-specific modeling needs, using built-in extensions such as profiles. Both approaches deliver on modeling's inherent value: 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." In fact, it is now an "international team sport." With today's technology, software development has no geographical boundaries, and it will likely grow increasingly distributed and global. Modeling and other higher forms of abstraction will be crucial for helping practitioners 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 is considered to be the next logical step in the evolution of modeling and model-driven development technologies. Based on the UML and related standards, MDA focuses on defining models at varying levels of abstraction and on the transformations they define. Automated tool support is crucial to the evolution and successful application of MDA.

Conclusion

The purpose of this article was to define modeling in the most fundamental way and to emphasize the value in applying modeling to software development. Modeling is about thinking and working at higher levels of abstraction. These are tried-and-true techniques that have proven successful for many years across all forms of engineering and technical disciplines. Model-driven forms of software development have already proven effective for the more advanced and progressive organizations in the software industry. Based on the ever-growing complexity of modern software systems, and on several other trends discussed in this paper, we believe that the time has come for modeling to enter the mainstream of the software development community.

If you are interested in learning more about how IBM embraces and supports modeling and model-driven development, check the IBM Rational Website at http://www.ibm.com/software/rational/.

About the authors

Gary Cernosek is currently a market manager for the Rational software brand within IBM Software Group. He is responsible for analyzing and responding to software development market trends, with a focus on software design and development technology, particularly in the visual modeling and model-driven development areas. Previously, he held positions in Rational field sales, field technical training, and customer consulting. Prior to joining Rational, he worked as a software developer in the NASA community on space shuttle and space station systems for more than eight years. He holds a B.S. in electrical engineering from the University of Texas at Austin and an M.S. in computer system design from the University of Houston at Clear Lake, where he concentrated on object-oriented software engineering.

Eric Naiburg is group market manager of desktop products for IBM Rational Software. He is responsible for market strategy, planning and messaging around Rational's desktop products including XDE, WebSphere Studio, Rational's testing solutions and more. Prior to his current position, he was manager of product management, focusing on the IBM Rational Rose and IBM Rational XDE product lines. His focus was to extend the ability of Rational's products to support database design and e-business solutions within the visual modeling tools space and the UML. Eric Naiburg came to Rational from Logic Works Inc., where he was product manager for ERwin and ModelMart.