Introduction

For many years computer scientists have attempted to build models to represent the logic of various software components. We might ask why. It is because modeling is a universal way of both managing cost and facilitating exploration and innovation. Architects have always built models to envision complex construction projects; engineers in many disciplines build models as a way of better understanding, visualizing, and simulating the characteristics of their creations without incurring significant expense. More recently, software engineers have significantly increased our ability to model physical objects by incorporating more verisimilitude into the simulation of designs for automobiles, aircraft, and consumer products; almost anything that can be constructed is first modeled using CAD/CAM (computer-assisted design and manufacturing) systems from multiple vendors in this field.

Amazing feats can now be accomplished with sophisticated modeling tools. For example, the latest Dassault Aviation commercial jet, the Falcon 7X®, was designed, simulated, and had its entire bill of materials generated by a CAD/CAM system. It was the first airplane ever designed that flew without any physical models ever being built and discarded. Millions of dollars and significant amounts of time were thus saved. In addition, the completeness of the model makes testing the comfort and safety characteristics of the airplane very simple and quick.

In light of this, the question should be why *not* model complex software applications? Why shouldn't we leverage sophisticated and precise models to eliminate the sometimes tedious and error-prone task of actual coding? While we are at it, why not test the model itself and hence, save time and money in testing the code, because the code, if fully generated, is true to the model? Why not incorporate known patterns and algorithms for scalability and security of applications into the generation of these models and hence, eliminate much of the error-prone relearning that goes on in the actual creation of the code? Furthermore, having an accurate model would make it significantly easier to maintain and modify large, complex programs.

The good news is that we have made significant progress in modeling software. Standardization efforts like UML® (Unified Modeling Language™) by Grady Booch, Ivar Jacobson, and Jim Rumbaugh at Rational® incorporate multiple representations to capture program logic, and interaction models are now commonly used. Such languages are being enhanced in standards bodies like the Object Management Group, Inc. (OMG®). Tools that allow us to create, maintain, and check consistency in these models have been built. Tools have even been created to automatically generate the code in numerous computer languages and verify or generate test cases.

Nevertheless, we have not yet been able to popularize modeling among many developers; they always seem so eager to simply start coding. Modeling often seems to take a back seat. To be fair, modeling presents significant challenges. Physical constraints give real-life models boundaries and well-understood properties; in the modeling of complex programs, these constraints are lacking. As a result, it is difficult to create models of sufficient generality and power to be complete and effective, given the infinite space of possible abstractions and logic associated with complex programs and network deployments. The creation of models that are powerful and complete enough to shortcut the tedious and error-prone creation of actual, precise, high-performance software code still eludes us.

On the positive side, as you will see in this issue of the *Systems Journal*, we continue to generate good ideas to capture and make models more powerful in helping to reduce the cost of innovating, securing, and maintaining complex software. One of the major advances in modeling in the last 20 years was the realization

that we tend to code in patterns. As a result, in addition to the compilation of books like Donald E. Knuth's volumes of basic algorithms, we are now starting to compile books of common design patterns. One of the first such compilations was the seminal work *Design Patterns: Elements of Reusable Object-Oriented Software* by John Vlissides, Erich Gamma, Richard Helm, and Ralph Johnson. The abstracted patterns described therein add another dimension of power to our models of software and are being used extensively by IBM and others around the world in designing products and tools.

It is to John Vlissides, who passed away recently at a very young age, that we dedicate this special issue of the *IBM Systems Journal*.

Daniel Sabbah
General Manager, Rational Software
IBM Software Group