Quality Software and the Unified Modeling Language

by Grady Booch

Worldwide, there is an insatiable demand for software. On the one hand, that's great news. These are exciting times for the professional software developer, for this is still largely an era of innocence and unbounded opportunity. On the other hand, that's the worst possible news. No amount of heroic programming will ever suffice to meet this demand. Furthermore, as software continues to weave itself deeply into the fabric of society, the stakes have gotten higher. Unfortunately, software bugs are still considered just a normal part of the territory, but now, they may manifest themselves in the fall of a business or even worse, in the loss of a human life.

In addition to this insatiable demand is the almost rabid rate of change in software development technology. Eighteen months on the calendar is an eon in software years. Blink and you will miss the next great shift. Just a year ago, C++ was hot. These days, it's Java and Visual Basic. Middleware such as OLE and CORBA, visual programming, and component-based programming have changed the rules. The great operating system wars still rage on. Frankly, I've given up predicting what might come next.

Although I refuse to predict technological change, I can safely predict the future of software: it's going to be more complex and it's going to be far more distributed. It's going to be far more complex mainly because of demand-pull and supply-push, to use an economic analogy. User expectations for what software can do are exceedingly high. Furthermore, what is possible is far greater now than just a few years ago. The cost of computing has plummeted, yet the cost and complexity of software development have continued to increase. Future systems will be far more distributed for similar reasons. The economics of networking are such that, in the future, the network will be the computer. Additionally, the partly technical, partly social phenomenon of the Web has fueled the drive toward pervasive distribution.

Despite all of the excitement and the rhetoric, one thing does remain constant: Building complex software of quality and of scale is still fundamentally a hard problem. Simply put, this means that, all the latest technology trends notwithstanding, deploying quality software is still an engineering problem. As for any engineering problem, this implies striking a balance between artistry, for the best software is often a thing of beauty, and science.

On the science side, my experience suggests that there are a number of best practices found in common among projects that are successful. Two of those practices that stand out are a focus on architecture, and a focus on an iterative and incremental development process.

A focus on architecture means not just writing great classes and algorithms, but also crafting simple and expressive collaborations of those classes and algorithms. All quality systems seem to be full of these kinds of collaborations, and ongoing work in the area of software patterns is beginning to name and classify them so that they can more easily be reused (Gamma et al. 1995). The best architectures I find have, as Fred Books calls it, "conceptual integrity," and that derives from the project's focus on exploiting these patterns and making them simple, which turns out to be very hard to do.

An iterative and incremental development process reflects the rhythm of the project. Projects in crisis have no rhythm, for they tend to be opportunistic and reactive in their work. Successful projects have a rhythm, reflected in a regular release process that tends to focus on the successive refinement of the system's architecture. This is what Microsoft calls "synch and stabilize," and it's a practice which brings results, for systems of just about any complexity.

A focus on architecture and a focus on process may appear to be simple enough things for a project to institute. In the heat of battle, however, when an unstoppable deadline comes rushing at you, the easy thing to do is abandon these practices. I got into a discussion with a programmer recently, who told me of her company's shift to the use of C++. I asked if they were using any object-oriented analysis and design techniques, and she replied that they didn't have time for that. I then asked her if they were meeting their schedules and target metrics. She said, with some embarrassment, no. Suddenly, I had a Dilbert moment: The cause and effect were just so evident to me, and yet, head down, worrying about the daily blocking and tackling of her project, she just didn't see the connection.

This is not an isolated incident. Some of you may have heard me talk about a couple of horror stories:

This particular project in crisis had written several hundred thousands of lines of C++. A quick review revealed that the team had written lots of code, but not one single class.

Another project was similarly in crisis. A quick review here revealed that the team had written hundreds of thousands of lines of C++, and that they had about the expected number of classes for that size system. However, further investigation revealed that, on the average, each class had about one member function, usually named with some variation of the phrase "do it."

I'm not saying that these projects were clueless, but they both did ignore a pretty fundamental principle: quality software doesn't happen; rather, it's engineered that way.

This is where object-oriented methods come in. I've been living with this technology since the early 1980s. In the years following, characteristic of almost every emerging discipline, there was an explosion of object-oriented methods as various methodologists experimented with different approaches to object-oriented analysis and design. Experience with a number of these methods grew, accompanied by a growing maturation of the field as a whole as more and more projects applied these ideas to the development of production quality, mission-critical systems. Initially, a few methods began to take root, having added value to a number of projects. These methods included ones such as Booch, OMT, Shlaer/Mellor, Odell/Martin, RDD, OBA, and Objectory. By the mid-1990s, a few second-generation methods began to appear, most notably Booch'94, the continued evolution of OMT, and Fusion. By this time, object orientation was decidedly in the mainstream. The important thing about all of these methods is that they attempted to bring about a balance of artistry and science to complex software development.

Given that the Booch and OMT methods were already independently growing together and were collectively recognized as the dominant object-oriented methods worldwide, Jim Rumbaugh and I joined forces in October 1994 to forge a complete unification of our work. Both Booch and OMT had begun to adopt Ivar Jacobson's use cases, and thus it was natural that in the Fall of 1995, Ivar formally joined this unification effort.

Currently, we are focused on what we call the Unified Modeling Language™. UML™ is a third-generation method for specifying, visualizing, and documenting the artifacts of an object-oriented system under development. UML represents the unification of the Booch, Objectory, and OMT methods, and additionally incorporates ideas from a number of other methodologists, most notably Wirfs-Brock, Ward, Cunningham, Rubin, Harel, Gamma, Vlissides, Helm, Johnson, Meyer, Odell, Embley, Coleman, Coad, Yourdon, Shlaer, and Mellor. UML is the direct and upwardly compatible successor to the Booch, Objectory, and OMT methods. By unifying these three leading object-oriented methods, UML provides the basis for a common, stable, and expressive object-oriented development method.

Although the UML itself is intentionally quiet on process, one process that UML must enable is one that is use-case driven, architecture-centric, and both incremental and iterative. In many ways, UML tries to codify the best practices that we and others have encountered in successful object-oriented projects worldwide. Thus, also in many ways, we are not really the "inventors" of anything radically new. Rather, the value that UML brings is that we've observed what works and what doesn't in the world of object-oriented software development, and tried to package that up in the form of a modeling language that scales to systems of complexity.

I have no expectations that the fundamental problems of software development will go away in my lifetime. However, what I do know is that a continued engineering focus will help mitigate those problems, and development such as the UML are one stake in the ground helping to define that engineering focus.