The Future of Software Technology

Interview with Grady Booch
Chief Scientist for Rational Software

Published with permission of CHIPS Magazine, Fall 2002 (Volume XX, Issue IV).

CHIPS' editors had the pleasure of meeting Grady Booch, recognized pioneer in software development, and hearing him speak at the Software Technology Conference (STC) in April 2002. He had the STC mesmerized with his analysis on current and future technology developments. CHIPS asked Grady to share some thoughts with readers.

CHIPS: Rational products are designed following the Capability Maturity Model for Software Development. Why do you think this is important?

Grady Booch: Over the years, Rational has been involved with customers from around the world in literally tens of thousands of software projects for every imaginable problem domain. From that experience, we've observed what has worked and what has not, then codified those best practices in the form of the Rational Unified Process® (RUP®) as well as made them manifest in and encouraged by our tools. Most significantly, we apply those same practices and tools to our own development: architecture [design] first, develop iteratively, test continuously, model visually, manage requirements, and manage change.

The Capability Maturity Model (CMM), also based upon years of experience with real projects, provides a useful means of comparing the maturity of an organization's process. Among other things, the CMM has given the industry a vocabulary with which to name and characterize the various stages of process maturity.

To say that Rational's products are designed following the CMM is therefore not exactly a meaningful statement: the CMM provides a reasonable measure of process maturity, but it does not prescribe a particular process. That being said, the experience of organizations using Rational's process -- and that includes Rational itself -- has demonstrated that they can achieve higher levels in the CMM by applying the best practices that we have collected over the years.
It is also important to note that while the CMM offers a measure of process maturity, that measure is not necessarily related to the measure of project success. We measure success entirely in economic terms: Does the project provide a good return on investment? In other words, is the amount we're spending, in terms of human and capital expenses, giving back tangible value that forwards the mission of its organization? Functionality, quality, and schedule play a seminal role in this equation; missing functionality, poor quality, and missed schedules represent failures in delivering tangible value. Similarly, opportunity cost plays a role: Are these scarce human and capital resources being spent wisely and intentionally on the most important thing to my organization?

These economic measures are of primary importance, because they relate directly to the primary product of any development team, namely, the executable software itself. The CMM is relevant insofar as it relates to the predictability and repeatability of these economic measures: A chaotic process will be neither predictable nor repeatable, but a mature process will be both -- and an exceptional process will predictably achieve better results over time.

Running somewhat counter to the goals of predictability and repeatability is the need for agility: For all kinds of software-intensive systems, the development team must cope with changing technology, changing context, and changing user needs. Among other things, the dot bomb era demonstrated quite soundly that agility and speed without a predictable and repeatable economic model is doomed to failure. Technology does change, the business and environment do change, and user needs do change -- but these forces must be balanced with repeatable and predictable processes that yield a good return on investment both in the short and long term.

The best practices I spoke of earlier -- architecture [design] first, develop iteratively, test continuously, model visually, manage requirements, manage change -- exist because they are proven to help an organization make the proper engineering and business decisions that balance the forces upon the software development organization. By focusing on architecture first, one intentionally attacks the highest technical risks in the system; by developing iteratively, one has the opportunity to reach closure on a regular rhythm and then make intelligent midcourse corrections relative to the current business and technical risks. Testing can then happen continuously with each new release representing a baseline against which the emerging product can be measured relative to the current (and more deeply understood) requirements of the system. Modeling permits the team to visualize, specify, construct, and document the artifacts of a software-intensive system so as to control its architecture and reason about elements that cannot be known at the level of pure code. The management of requirements and the management of change refer to the intentional consideration of changing user needs (for the mere presence of a release will tell the user things they could not have known or asked about initially) and of the artifacts that constitute the developing product itself.

Relative to the CMM, the RUP -- which codifies these best practices -- defines a mature process that is predictable and repeatable as well as
agile. We use these practices ourselves because they work and because, as a user of our own products, we too seek a good economic return on the investment we place in building software. Rational produces a new release of all of its products twice a year, with many more releases made internally. We simply could not develop that much quality software without following our own process and using our own tools. Furthermore, we are getting better and better each year -- the sign of a mature process -- and since those practices are embodied in the RUP and in our tools, the same benefit is made possible to the users of the RUP.

**CHIPS:** With so many challenges to security and a standard architecture for mobile devices, what do you think the design plan should be to overcome these problems?

**Booch:** Building secure systems is not a core competency of the average code warrior, who already has a multitude of skills to balance. Furthermore, individual developers have a difficult enough time making reasonably optimal local decisions, but building secure systems requires making a long series of local design decisions that are globally optimal as well; in short, the problem of security is a classic one of aspect-oriented programming, that of crosscutting concerns. At Rational, for example, in addition to our chief architect (who oversees significant design decisions across all of our products), we also have a security architect. Security is a systemic issue: It must be defended logically in the design of the software itself, physically in the design of the network, and physically in the human processes that interact with a secure software-intensive system.

The presence of wireless connections to a network introduces a completely new set of security issues that further complicate the problems of building secure captive wired systems. First, without an intentional effort to manage the system's topology, adding wireless connections can inadvertently open wide holes in an otherwise secure network. Second, the current standards for secure wireless transmission are still immature. Architecturally, if you have to have wireless connection to a system, the one thing you can do is to form an air gap between the wireless elements of your network and the wired elements. In this manner, you can apply all the usual techniques to make your wired network impenetrable, but defend it from unexpected intrusions via the unwired side by either totally eliminating communication, making communication unidirectional, or vigorously guarding communication between the two elements. It goes without saying (but I'll say it anyway) that active intrusion detection and response is a must in systems with unwired elements.

**CHIPS:** What do you think the next breakthrough will be in software development?

**Booch:** Your question assumes that there have been previous breakthroughs; in my experience, there have been none, nor do I expect any in the immediate future. Now, that may sound terribly pessimistic, but
The history of software engineering has been one of growing levels of abstraction -- we see this in our languages (assembly to FORTRAN to Ada to Java), our methods (structured analysis and design to object-oriented design), and our platforms (programming running on raw iron to basic operating systems to platforms such as .NET and J2EE). This growth has occurred simply as a meaningful engineering response to the growth in complexity in the kinds of software systems we as an industry are asked to create. If you examine every one of these advances, there are no breakthroughs that I can identify (although historians in retrospect may label what they call major events -- but you must remember that history is written by the winners...). Rather, every advance is built upon the experiences of the previous technology. Such is the nature of scientific and engineering progress: We try something, we learn from experience what works and what does not, and we progress by claiming what works and improving upon what does not. Thus, you might say that the UML was a breakthrough. Well, its presence is certainly important, for it has become a part of the mainstream of software development. However, the UML has its roots in developments that began over a decade ago, and it's that decade of use and experimentation that has made it what it is today -- a technology of value that helps organization build better software faster.

Ours is a ruthless industry: It has its fads, but ultimately, those things that do not add value are quickly discarded. Building quality software -- intensive systems that matter -- is fundamentally hard, and just as Fred Brooks observes, I don't see any development in the near future that will change that.

Now, that being said, as Chief Scientist at Rational, my role is to worry about software engineering as it might be three to five years from now. Where possible, we try to invent that future (inventing the future is the best way to predict it, as Alan Kay has observed); otherwise, I stay close to a diverse community of software engineers around the world to track and, where I can, contribute, to their thrusts in this space. A few years ago, this is why I injected myself into the patterns movement. Today, there are a handful of technologies on my radar screen: aspect-oriented programming, model-driven development, collaborative development environments, architectural patterns, and the semantic Web are the ones I’m spending most of my energy on. Will any of these represent breakthroughs? No, I don't think so, although I do think that any one of these elements can nibble away at current points of friction in the way we build software today.

What holds us back the most in building better software faster is not a problem of technology. Computer science tells us some things about the limits of software and how we might best lay down code or design our systems, but ultimately, software development is a human problem. As such, the problems of organization and process (and politics) weigh most heavily upon success in delivering quality software-intensive systems. The cost of software-intensive systems is thus no longer dictated by the cost of
hardware or tools but by each individual's and collectively the team's ability to deliver; improving the productivity of the code warrior and of the team thus can have a large economic impact. The Unified Modeling Language (UML) has gained broad industry acceptance as the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

**CHIPS:** Why is UML important and how does it fit in the Object Management Architecture?

**Booch:** I have been pleasantly surprised at the breadth and depth to which the UML has entered the mainstream of software development. We see its use in virtually every kind of problem domain, all around the world. As I said earlier, ours is a ruthless industry, and the UML would not have the degree of penetration it does today if it did not add value.

The UML is all the things you describe, but from that I'd observe that the UML is important because it addresses several points of pain in building software-intensive systems. First, again as I mentioned earlier, software development is fundamentally hard, and the way we counter that complexity is by abstraction. The UML permits a team to visualize and construct abstractions that transcend the underlying implementation language. As even James Gosling has observed, there are some things you can reason about when staring at raw Java, but many things you cannot (but which you can do in the UML). For example, extracting and/or imposing patterns for security or messaging is far more easily done at the level of models than at the level of code (although those patterns in models most certainly end up being manifest in code). Thus, the UML is important for architectural control.

Second, the UML is important for the individual code warrior for the same reasons, although at a different level of abstraction. In developer's use of Rational's eXtended Development Environment (XDE) for Visual Studio .NET and Eclipse, for example, we see code warriors using UML models extracted from code to better understand and navigate their implementation, and then to use these models or build new ones to push their abstractions into code.

Third, the UML addresses the problem of coordinating the disparate viewpoints of the many stakeholders who contribute to the creation of any nontrivial system. The great thing about the UML is that it provides a single, unambiguous language of blueprints that a system's domain experts and developers can use to communicate with one another. Thus, we see business experts, domain experts, and end users expressing their desired behavior for their system in the UML, with the code warriors, analysts, testers, and even network engineering using the same language to construct the system. Having this common language facilitates the bandwidth as well as the quality of communication among those who must collaborate to build and deploy a software-intensive system.

**CHIPS:** What is the benefit of UML for users? How will using the OO/UML model benefit the Navy's C4ISR enterprise architecture and joint fighting
Booch: The benefit to users is what I express in the third element above: Through the UML's language for use cases (and associated behavioral elements), users have a language whereby they can assert their desired behavior for a system and then effectively collaborate with those who must turn those desires into reality.

This is especially true for frameworks such as C4ISR (and I'd add to that .NET, J2EE, and emerging standards for homeland defense and other government enterprise architectures). The value of an architecture such as C4ISR is that it provides a higher level of abstraction upon which one may build systems. None of these platforms I've mentioned is trivial, and by expressing them in the UML (as has been done for C4ISR) across all the various views different stakeholders may see of the platform, one makes possible both the architectural as well as the implementation benefits of applying the UML. In this way we finally have a relatively seamless and smooth mapping from the highest level enterprise architecture abstraction to the software system abstraction to the implementation in code, and thus a single communication tool from inception to implementation. Ultimately, this translates into helping teams build quality software faster.

CHIPS: Some experts say that the full potential of the Internet for e-business processes has not been realized yet. What important trends in e-business do you forecast? How can the Navy cash in on this still emerging technology?

Booch: The most important trends I see in e-business derive from what we've learned in the dot bomb era: first, we've learned that raw technology without a proper mapping to the human and financial needs of the enterprise is doomed to failure and second, we've learned some canonical architectures for e-business systems that do work.

In the first case, the essence of this lesson is a human one: I think that we as an industry better understand the need to develop any kind of e-business technology with the particular needs of its various human users and financial stakeholders. In the second case, this is a simple matter of having experimented with various Web-centric enterprise architectures and discovering what works well and what does not. For example, throwing together a system with Perl duct tape may be expedient, but results in a brittle system; for this reason, platforms such as .NET and J2EE have emerged, and there exist patterns of use that we better understand (for example, where to build the business rules in a system, how to manage communication among loosely-coupled systems, how to architect the illusion of a session).

As for trends, the rise of the use of XML is certainly a good thing; work on standardizing XML schemas for various domains is also a welcome trend. I'm encouraged by emerging standards for security and agent computing as well.

As for the Navy cashing in on these technologies (can I really comment on that that? I'm a USAFA graduate, you know, and there's this Air Force/Navy game coming up soon, and we know who's going to win...), I
can only advise that first, there must be a good economic model for any investment in this space. Consider the points of pain in your existing human processes, and then and only then consider what technology might do to eliminate that pain. The best way to reduce the risk of software development is to not write any software at all, and thus, rushing in to cash in on a technology is the wrong order of priorities: identify what hurts first, and then build or buy the right technology to cure that pain. Don't underestimate the capacity the Navy and the DoD has in general to influence the commercial marketplace -- helping to direct this emerging technology while learning from the commercial space is far more cost-efficient than trying to invent this stuff by yourself in a cave.

**CHIPS:** When asked at the STC, What scares you the most? You said, the fragility of network security. With so many security risks in the form of information terrorism to the nation's information infrastructure, what steps should be taken to ensure protection?

**Booch:** As I mentioned in an earlier answer, building secure systems is a systemic issue that requires a long series of local design decisions at both the conceptual and physical levels. Unfortunately, the sad reality is that too many platforms are inherently insecure, and even the best laid plans can go astray on top of a wobbly, porous platform. To quote a line from the movie *Network*, I'd say "I'm mad as hell and I'm not going to take it anymore." We individually can respond to the presence of lousy software, simply by not buying it. Unfortunately, we often still do because we don't think we have any control or any other options. Well, as I suggested earlier, the Navy in specific and the DoD in general is not a tiny market, and collectively, a common voice from this community saying that you won't stand for insecure platforms will indeed speak volumes to those who produce those platforms.

I'm currently working with a group responsible for the electric power grids in the Northeast. It's truly humbling and frightening at the same time to gaze at the architecture of the system that will keep the lights on in Manhattan -- it's humbling because what we do here will be invisible to the end user, but ultimately must not fail; it's frightening because I see all sorts of risks (which we know we can mitigate); but I worry about those that we do not see. Every kind of software-intensive system of this size embodies similar risks. Mitigating those risks requires intentional emphasis in understanding the ways in which that system might be breached (and it will be breached) and how it therefore might be defended against.

**CHIPS:** I heard you say that you are a voracious reader; what types of books do you read?

**Booch:** OK, now that you've put security in the front of my mind, I just checked the locks on my doors and engaged the alarm system as well; let me take a deep breath and proceed... I recently finished cataloging the books in my collection; I've close to 2,000 books on software in my
professional library, and my wife and I have over 4,000 books in our private library. I read a lot of software books every year simply to stay up with contemporary and emerging technology; on my desk at this moment are several books I'm reading dealing with business rule modeling as well as a few others -- in a totally different space -- concerning current research in quantum computing and cognitive science. I subscribe to about twenty trade magazines and ACM/IEEE journals as well (my mail carrier hates me, I think, but on the other hand, Amazon loves me). On the personal side, I primarily read a lot of history and biographies (I'm slogging through *Gotham*, a detailed history of New York City, plus I am up to volume two of Churchill's memoirs of World War II, *The Grand Alliance*, and I am halfway through Margaret Thatcher's *Statecraft*), especially those centered around the Middle Ages and the Renaissance (which complements my passion for music -- I play the Celtic harp). I read deeply in the areas of philosophy and spirituality (I just finished *How We Believe* by Michael Shermer). When I need a break from reality, I'll read science fiction (Terry Pratchett and Neal Stephenson are my favorite authors in the genre) or works such as *Interpreter of Maladies* by Jhumpa Lahiri and *Ex Libris* by Ross King. For a typical day on holiday, my wife and I will both consume one or two books per day.

For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided. Thank you!