

▶ **Achieving Quality By Design** **Part I: Best Practices and Industry Challenges**

by [Ed Adams](#)

Testing Evangelist
Rational Software

I used to shoot people for a living. That experience gave me valuable insight into developing quality software. Before you call America's Most Wanted to turn me in, let me explain. Years ago I was a mechanical design engineer working on non-lethal weapons systems. One weapon I helped design was a perimeter-weighted net that could be fired as a ballistic projectile at a subject from a shotgun mount. The purpose was to restrain but not harm the target. As part of our field testing, I would take some fellow engineers out into a field and "shoot" gun mounted canisters containing these nets at them, and then we would see if they could escape.



OK, so now you know I was not a hit man, but you still may wonder just what that experience has to do with software development. Well, before we went out to shoot nets at our coworkers, we had already done a tremendous amount of work in the design phase of the project. We performed extensive testing *before* we constructed a prototype to test on live people.

In the mechanical design world, there is an established process for assessing the quality of your design before you build it:

- You model the application -- in this case the net and canister propulsion system -- typically using a computer aided design (CAD) system.
- Once you have a model, you test it using computer aided engineering (CAE) tools. With these kinds of tools you can put a load on a beam, put some flow through gas pipes, or stress test a net.

- ▶ [subscribe](#)
- ▶ [contact us](#)
- ▶ [submit an article](#)
- ▶ [rational.com](#)
- ▶ [issue contents](#)
- ▶ [archives](#)
- ▶ [mission statement](#)
- ▶ [editorial staff](#)

- Model test results are analyzed and any needed design changes are made. Then the improved design is fed back into the stress test workflow, and you assess the new design. You repeat this process on the model until it passes the requirements placed on it. Wash, rinse, repeat.
- Only **after** you test your model and verify that your design is architecturally sound, do you start building the prototype -- or the Beta to follow the analogy into the software world. When we built our weapon, we already *knew* that it was going to work. At that point we were just doing fine tweaks; there were no costly design changes or architectural changes late in the game.

When I started working in the software industry I was amazed to find that there was no analogous process; and I thought everyone in the software world was absolutely mad. Developers were building Beta software -- or in some cases release software -- directly. Few organizations were modeling their applications at all, and nobody was able to test designs to ensure they were architecturally sound. So when I moved from the mechanical design world to the software world, I stopped shooting people. Now it was I who was scared to death.

That is why I feel Quality by Design is such an important topic. Quality by Design is a software development solution that uses a very specific process and set of best practices and tools to build in and measure quality at every stage of the software development lifecycle. It is a pro-active approach to software development and testing. More specifically, it is proactive from a *quality* standpoint, not just from a construction or development standpoint. A key factor, as we will see in Part II of this series, is the adoption and use of a design tool -- just like the ones I used to construct my non-lethal weapon systems.

The Business Problem

What was the world's first software project? Well, think about the biblical story of the Tower of Babel. This was not really a software project, but it had a lot in common with today's software projects. A lot of people worked on it, as it was the largest engineering project of its day. There was a goal - - build a tower -- but it was not very well defined. Everyone was using different terminology, different methods, and different tools. Each group was confident (cocky even) that its piece would be the best. Very little integration work was done. Lastly, nobody was serving as project coordinator for the entire project. What happened? The project imploded; the CEO - the big guy upstairs - got angry and canceled the whole thing. And as a result, now almost nobody knows how to speak Babylonian.

According to Standish Group's last report¹ on the subject, nearly three out of four IT projects meet a fate similar to the Tower of Babel's. Why are so many of these projects canceled? Development teams are using different tools and different terminologies that make it difficult for them to communicate and focus on a single goal. It is also very difficult to measure quality and capture metrics along the way, or even agree on what metrics to capture.

Compounding these issues, software developers are being squeezed by opposing forces. On one side there is constant pressure for faster time to market; while at the same time the cost of failure has increased dramatically. Because most applications are used directly by customers (not just internally), we cannot afford to release low quality software. And, as if this situation were not bad enough, all these pressures are multiplied because today's applications are much more complex than they were even just a few years ago.

Quality and Cost Control: Everyone's Responsibility

To help speed development and shorten time to market, many organizations use component-based, modular designs. But many are finding that testing costs are the Achilles' heel of modular designs. A veritable explosion results when many scenarios on several modules must all be tested. Which leads me to one of my favorite quotes on this subject:

"Unless designers can break through the system-testing cost barrier, the option values ... might as well not exist."²

The reason I like this quote so much is that it identifies system testing as a potentially huge cost barrier. It is almost *always* a huge cost barrier in terms of both time and dollars. And, more important, it states that it is the *designers* who must break through that system testing cost barrier, not the testers. Not to sound too clichéd, but quality and cost control are not the responsibility of just the test team; they are the responsibility of everybody on the team. The engineers, architects, and designers that are creating the fundamental structure or architecture are the people who can have the greatest impact on reducing system-testing costs. Without their help, there are no good options for overcoming this system-testing cost barrier; and this is particularly true for modular or component-based designs.

Numerous studies have shown that the cost of fixing a defect rises exponentially as the software development lifecycle progresses³. Forrester Research published an interesting report called "Why Most Web Sites Fail."⁴ It quantifies the time and money required to fix a site when it goes down, and sorts the results by cause of failure. The longer it takes to find a defect, the more expensive it is to fix. This follows intuitively from the fact that defects found late in the lifecycle will be more difficult to fix, and more likely to cause serious schedule delays, particularly if they reveal architectural problems that may require an extensive redesign to correct. Postponing testing until all components are assembled into a completed system is a risky proposition -- one that is likely to end in failure...or a really low quality release.

Typically, most testing is done during the Transition phase⁵ of a project, after the Construction phase has been completed⁶. As cost-conscious developers, our objective is to move testing earlier in the software development lifecycle, to start performing tests and finding defects in the Elaboration phase or during design, when they are easier to fix. By doing this, we can lower system testing costs later in the lifecycle.

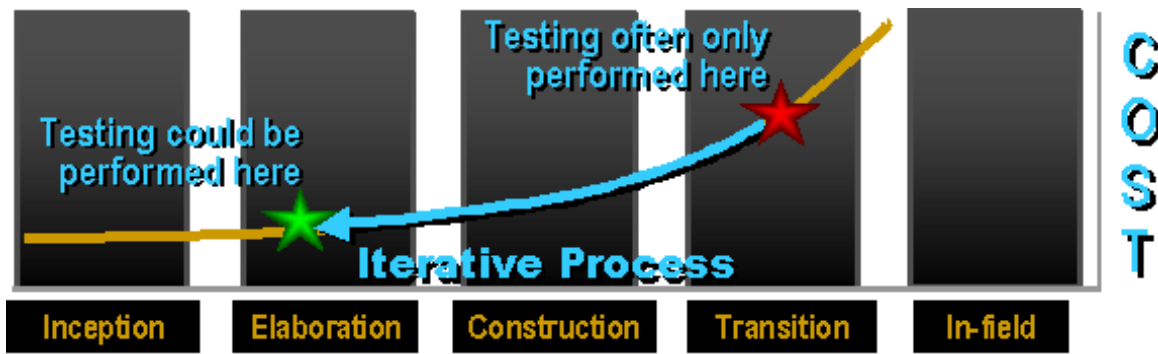


Figure 1: Problems Found Late in the Development Lifecycle Are Much More Costly to Fix.

To put it differently, we should strive to make software testers into checkers or validators instead of defect finders. Right now, developers write code and pass it on to someone else to do the majority of the testing. We need to get away from that practice, and move toward a process in which designers and developers verify quality. Then, all testers will have to do is look at the final application and say "OK, that's right, that's right, and that's right" -- and the entire development process will be vastly more efficient.

An analogy can be made to the manufacturing industry, where parts are designed for manufacture (DFM) or assembly (DFA). The pieces are designed to assemble easily *and* they are checked before being passed on to the assembly line. This is a key concept. For example, there are specifications in Design for Manufacturing that address symmetry of hardware components. If someone is assembling a system, he does not have to worry about orienting the component correctly, because it will work even if it is rotated 90 degrees one way or the other before it is installed. By considering manufacturing issues in the design stage, the manufacturing stage is made easier.

Design for Testability is the same concept applied to quality. By considering testing issues in the design stage, the testing stage is made easier. Design for Testability is a well-established practice in a number of industries, including mechanical design and integrated circuit manufacturing to name just a couple. However, it is still not yet well established in software development.⁷ I'll revisit Design for Testability a little later on.

Quality by Design: Is It Possible?

After considering the disappointing failure of the Tower of Babel, you might begin to wonder if Quality by Design is even possible on large-scale projects. Let's consider another example: integrated circuits. The setup costs of manufacturing a new integrated circuit are quite high -- they include reserving time at the plant, creating masks for each layer of silicon and metal in the circuit, and so on. So engineers use CAD/CAM and CAE tools to test their designs fully before they are ever sent to the plant to be manufactured. They are able to do this because they capture sufficient information in the design phase to allow them to assess the quality of the design and validate the design before they build the first chips. They do

not spend huge amounts of time and money to build a complex chip, and then put probes on it, only to find that the half the circuit was not wired to ground (sound familiar to anyone testing "ready to go" software applications?). Instead, they tested the model to validate the design. When the first chips roll off the line, testers do not expect to find any bugs; they expect to test the chip and say "OK, that's right, that's right, and that's right." And that is exactly the approach we need to take in the software development world as well.

In the software industry we are finally beginning to realize that we too have a huge cost associated with the actual creation of the final application. As a result, it is becoming easier for us to see the significant benefits that can be achieved by testing the model as much as possible before we build it. Let's expand on this area by discussing some specific and common problems in today's applications.

Technical Challenges for the Software Industry

Now that we have the business problem in focus, let's review software's recent evolution as a prelude to discussing the technical challenges associated with quality by design.

In the good old days there were two-tier architectures. There was a fat client, typically a Windows application, built in C++, Visual Basic, PowerBuilder, or some other development environment. The fat client talked to a data server, and both the client and the data server contained some business logic. Releases occurred once every year or two. For the most part, applications were internal releases. If there was trouble, you could get on the PA system and tell everyone to get off the system so you could reboot the server. Life was easy.

Since then, things have gotten a great deal more complicated, especially now that we, as an industry, have moved to the Web. Now we have n-tier architectures. In these systems there is a thin client (commonly a browser or handheld device without much business logic), which talks to a middle layer (usually an EJB or application server, COM server, or Web server). The database is still there on the back end, but now there are two or three additional layers, all with their own business logic, and all communicating with each other via different protocols. With systems this complex, there are many more areas where problems can occur.

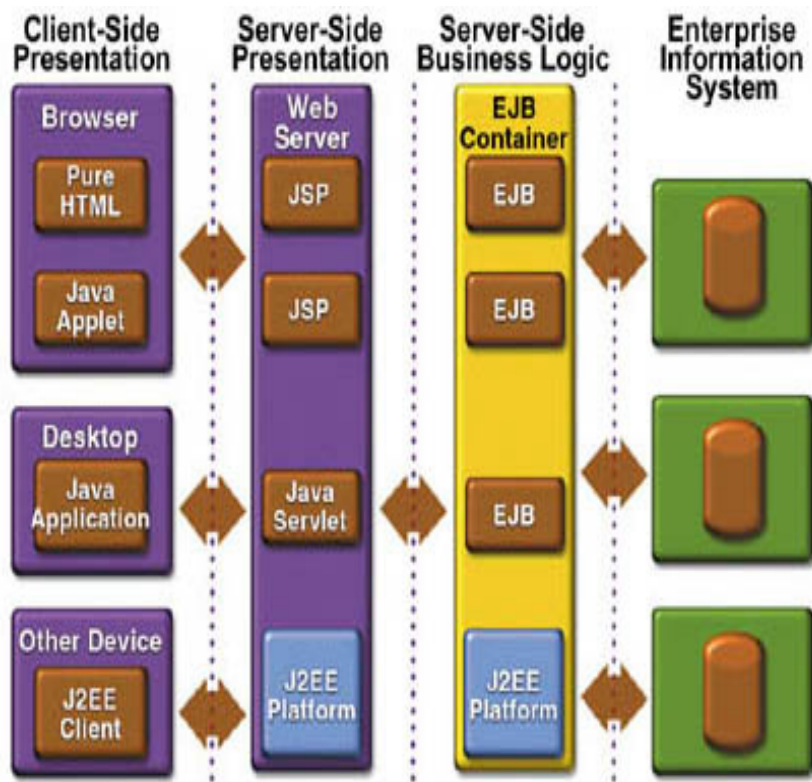


Figure 2: A Typical N-Tier Architecture

A .NET version would have the same fundamental structure but different names for its components.

(Source: <http://www.java.sun.com>)

In the J2EE application model, Web browsers, Java applications, and wireless devices interface with JSPs, which in turn talk to EJBs, which talk to the databases. In the .NET application model, the setup is very similar; it is just not Java. Instead of EJBs there are COM+ components, and instead of JSPs there are ASPs. The point is it does not matter if you're working in the .NET world, the EJB world, or any other world. You still have the same basic architecture, and you still have lots of potential problems to worry about. And it only gets worse.

A Multiplicative Effect

We all like component-based designs. They promote code reuse and parallel development, and they save both time and money. But there is a hidden danger in these designs. Even a system assembled from very high-quality components can have an unacceptably high likelihood of failure.

As an example, pick your favorite metric for measuring reliability. Assume that all the components in your system are fairly reliable -- 85, 90, or 95 percent -- as individual components. When they are combined into one system, you calculate the overall reliability by multiplying the individual reliability metrics for each component, not by averaging them. Consider a simple system with seven components all rated at 95 percent reliable. The overall system reliability is $.95^7$ or less than 70 percent. For many organizations that is well below minimum standards. And keep in mind that this is a very simple example with only seven components. If you have a system with twenty components, you can be in real trouble.

Testing Individual Components

As any tester knows, testing EJB or COM applications can be difficult because there is no GUI, and therefore no direct access to the components you need to test. Yet you need to test the logic and the use cases to validate them, which puts you (or more likely a developer) in the position of having to write a lot of test code yourself -- if there is time and talent enough to even do it. Consider the system in Figure 3. If you need to test Component B, then you need Components A, C, and D. If those components are not yet developed, then you will need to write a test driver to emulate Component A, and to build stubs for Components C and D so you can pass them data and test the expected results or exceptions. You have a major challenge on your hands. Creating all that code is expensive, and most of it cannot easily be leveraged on different projects -- so it just gets thrown away. This development effort takes resources away from the *real* development project and increases the overall cost of quality and development. But you cannot risk leaving Component B untested; you simply have to do it.

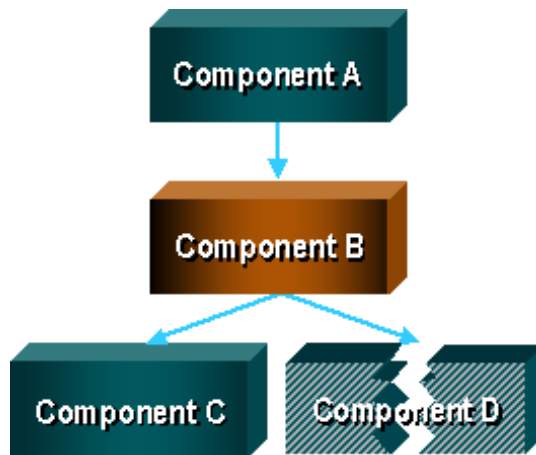


Figure 3: How Can You Test Component B Before Building Components A, C, and D?

So now that we understand both the business problem and the technical challenges faced by the software industry, let's talk about solutions. We'll look first at Design for Testability, an approach used in other engineering environments, and how it can be applied in software development. Remember the discussion of Design for Manufacture above? Well, this is a similar approach. The concept is to construct test access points into the design -- making it easier to validate. In next month's *Rational Edge* we'll see how the Unified Modeling Language and test reuse can support Quality by Design.

Design for Testability

Design for Testability is one facet of a good Quality by Design approach that is employed regularly by integrated circuit manufacturers, mechanical designers, and engineers in many other industries. By considering and addressing testing issues in the design phase, Design for Testability lowers overall development costs because it greatly simplifies the testing phase.

There are five key aspects of Design for Testability I'd like to consider in this article:

- Test case capture
- Design validation
- Test access
- Trusted components
- Built in self-test

Let's consider how each of these can be applied in the software world to establish our own quality by design standards.

Test Case Capture

This is an easy one. If you are responsible for a design or application, then you need to provide some way to capture the use cases⁸ that you are trying to test.

"A use case is a snapshot of one aspect of your system. The sum of all use cases is the external picture of your system, which goes a long way toward explaining what the system will do. ...A good collection of use cases is central to understanding what your users want."⁹

The design model must be instrumented so that it can be measured. For example, in doing the blueprint of a house, a use case requirement might be that the garage must be able to hold two average size cars. As a tester, you have to verify that your design meets this requirement. When you translate this into a test case you need to ask, How can I do that *on my model*? How do I instrument my model (the blueprint) to make sure the test case can be captured and allow someone to validate that requirement? On a blueprint it is easy, because there are physical measurements or aspect ratios. You just ensure the measurements are in the blueprint -- specify the width and height of the garage door in feet -- and that allows a tester to test whether this use case requirement can be met. Using the design tool, you can easily translate that specific use case requirement into a test case. You can even use the design tool to generate the test and validate the use case. Great concept.

Test case capture means that test cases can be *realized* by elements in the model. In this example, the model element we are trying to test is the size of the garage door opening. We can easily test that because the model allows us to determine the width and height of the opening and we have accurately decorated the model to facilitate test case capture and assessment.

Design Validation

Design validation is my favorite Design for Testability topic. Ultimately, it means measuring quality in the model. You can validate your design by

testing the model, and testability is a key aspect of quality by design. Again referring to the mechanical design world, consider the case of a turbine blade assembly (see Figure 4). This component has been modeled with a design tool. Now we are ready to test the design. Using the standards for turbine blade assemblies (to ensure we are in compliance with given standards...and by the way, that's another article series to watch for in upcoming issues of *The Rational Edge*), we are able to emulate a load being placed on this model. Design tools like this CAE system (and the Unified Modeling Language in the software world; more about that in Part II of this series) can be used to generate tests. That, in turn, gives you the power to validate your design and make changes to it when early tests fail.

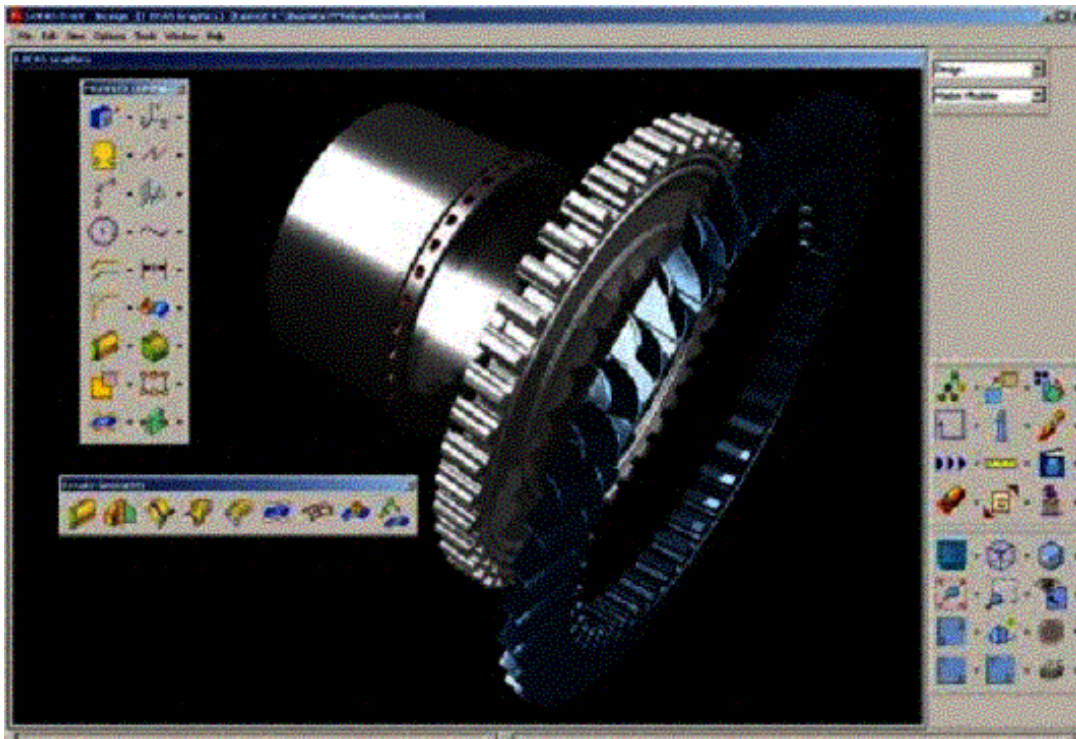


Figure 4: Turbine Blade Assembly

Design tools can be used to generate tests that validate the design before it is built.

When you can test your model, get results, and find out whether it will pass your requirements, that is design validation. It ensures that you are building in quality from the beginning: instead of validating an as-built system, you are validating the as-designed system.

Test Access

Test Access means that you must provide test points or "hooks" in your model that will allow testers to do their jobs. These are interfaces that can be understood by people other than the designer. These interfaces must be designed so that they can accept data passed to them, but also so that they can vary boundary conditions, preconditions, and postconditions. The crucial point is that test access hooks allow *others* to determine whether a component works or not. Without these hooks, others have to figure out ways to get to the components, which can be costly, if not impossible. Taking the time to include the hooks up front saves a great deal of time

later -- for developers and testers alike.

Trusted Components

The vast majority of today's development work is component based; typically, a project either includes third-party components or is built by a distributed team, with different groups working on different components. Either way, the components eventually have to be integrated into the extended system. The idea behind trusted components is that each component must meet a set of auditable quality standards to ensure it will work -- *before* it is integrated into the larger system. Some EJB vendors, for example, certify that their components will work on specific application servers, under specific conditions; you can check a specific set of standards to audit the component. The trusted components requirement means that even (perhaps *especially*) if you do not have control over a component's source code, you still have to ensure that it works. If you implement a process to do this, and adhere to it strictly, then you can rest assured that every component you use for your final system integration is up to standard. That is what trusted components is all about.

Built-In Self-Test

Years ago I bought my first laser printer. Whenever I turned it on, it performed a self-test, proactively exercising certain functions to make sure that it was working the way it was designed to. I did not prompt it to do the testing; it ran through these functions itself, every time. Why not have the same kind of self-test on software applications? I have seen some applications (even an Operating System) that does some rudimentary self-testing upon startup. In every example, the application occasionally uncovered problems that helped me troubleshoot what would have been a much bigger problem. Again, the key concept here is early detection. Software bugs are like a disease. Catch them early and they are almost innocuous. Prevent them with self-tests, and your application will live a longer and healthier life.

Components must be responsible for ensuring and reporting their own quality, and they must provide public interfaces to allow inspection. For example, if I have a component in my Web application that validates credit cards, then every twelve hours that component should go through a self-test to make sure that it is still working correctly. If it suddenly stops validating credit cards for some reason, then I may be selling a whole bunch of stuff that I can never collect money for, and I certainly would want to know about that as soon as possible. With a built in self-test, the component is responsible for its own quality, and it would alert me if there were a problem.

Stay Tuned

Design for Testability is only part of the story; as software developers we have other tools and other methodologies that we can use to ensure the quality of our applications during the Inception phase. In next month's issue, I'll talk about how the Unified Modeling Language and test reuse can be applied to this problem, and I'll offer some predictions for the future of

References

- Carliss Y. Baldwin and Kim B. Clark, *Design Rules: The Power of Modularity*. MIT Press, 2000.
- Alfred Crouch, *Design-for-Test for Digital IC's and Embedded Core Systems*. Prentice-Hall, 1999.
- Martin Fowler, *UML Distilled*. Addison Wesley, 2000.
- Forrester Research, Inc., Cambridge, MA, "Why Most Web Sites Fail," September 1998.
- Philippe Kruchten, *The Rational Unified Process: An Introduction, 2e*. Addison Wesley, 2000.
- Dean Leffingwell and Don Widrig, *Managing Software Requirements: A Unified Approach*. Addison Wesley, 2000.
- The Standish Group International, Inc., *CHAOS Chronicles*. 2001.
-

Footnotes

- ¹ The Standish Group International, Inc., *CHAOS Chronicles*. 2001.
- ² From Carliss Y. Baldwin and Kim B. Clark, *Design Rules: The Power of Modularity*. MIT Press, 2000.
- ³ Including Dean Leffingwell and Don Widrig, *Managing Software Requirements: A Unified Approach*. Addison Wesley, 2000.
- ⁴ Forrester Research, Inc., Cambridge, MA, "Why Most Web Sites Fail," September 1998.
- ⁵ The phases I refer to in this article are defined in *The Rational Unified Process: Inception, Elaboration, Construction, and Transition*.
- ⁶ Philippe Kruchten, *The Rational Unified Process: An Introduction, 2e*. Addison Wesley, 2000.
- ⁷ For more information on Design For Testability, I recommend the first two chapters of Alfred Crouch, *Design-for-Test for Digital IC's and Embedded Core Systems*. Prentice-Hall, 1999. The book focuses on embedded systems, but these chapters give a nice overview of DFT concepts.
- ⁸ A use case is a description of an action that a user would take on an application. It is a specific way of using the system from a user-experience point of view. Use cases can include both graphical representations and textual details.
- ⁹ From Martin Fowler, *UML Distilled*. Addison Wesley, 2000.



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

Thank you!