A Rational Software Whitepaper
November, 2003

Achieving Quality by Design

Table of Contents

Best Practices and Industry Challenges 2

The Business Problem 3

Quality and Cost Control: Everyone's Responsibility 4

Quality by Design: Is It Possible?....6

Technical Challenges for the Software Industry 6

A Multiplicative Effect 8

Testing Individual Components 8

Design for Testability 9

Test Case Capture 10

Design Validation 11

Test Access 12

Trusted Components....12

Built-in Shelf-Test....12

UML: The "L" IS for Language 13

So Let's Talk a little UML 14

Use Cases 15

Best Practices and Industry Challenges

I used to catch bad guys for a living, and that experience gave me valuable insight into developing quality software. 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 you are still wondering 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.

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.

Use-Case Realizations 16

Sequence Diagrams 16

Activity Diagrams 19

Using UML Diagrams for Testing 20

Test Reuse..22

Some Predictions 25

Twenty Year's Behind, but Catching ...... Up..26

References 27

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 proactive 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 report[1] 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 progresses[3]. 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 phase[5] of a project, after the Construction phase has been completed[6]. 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.

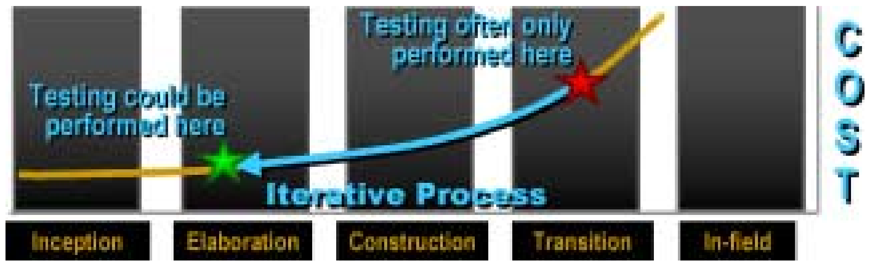

Figure 1: Problems Found Late in the Development Lifecycle Are Much More Costly to Fix.

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.

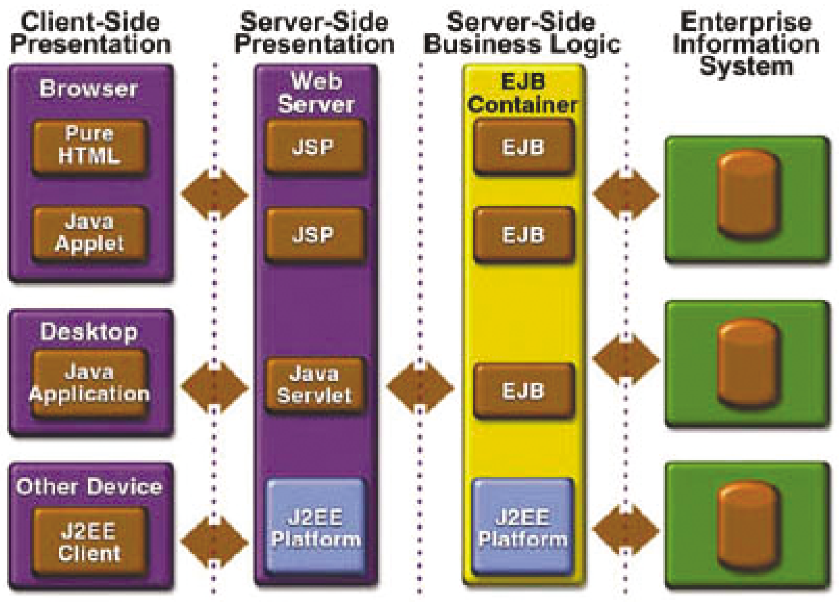

Figure 2: A Typical N-Tier Architecture.
A .NET version would have the same fundamental structure but different names for its components.
(Source: http://www.java.sun.com)

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 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.

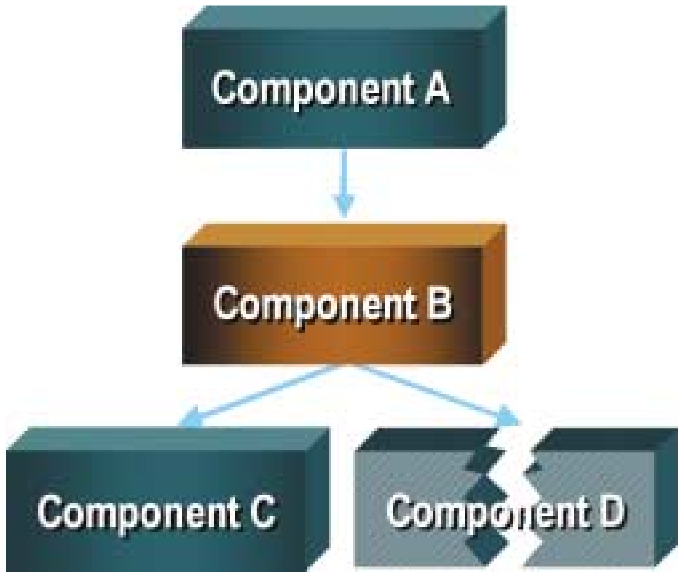

Figure 3: How Can You Test Component B Before Building Components A, C, and D?

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. Later in this paper 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 paper:

- *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 cases[8] 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), 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 AssemblyDesign 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.

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. Next 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 Quality by Design.

UML: The "L" Is for Language

The "L" in UML is for language. Language is all about communication. In Figure 5, we cannot decipher "Bow wow ruff woof…" because we do not speak the same language as our canine pals. This is similar to the problem faced today by developers and testers _ and communication problems between these groups are notorious for delaying and derailing projects.

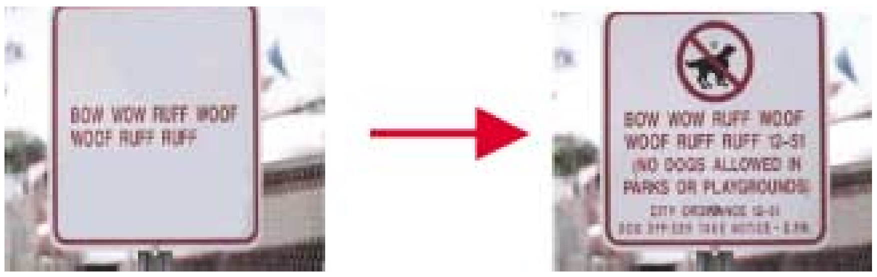

Figure 5: Translating for a Wider Audience

UML, which has become the de facto language of software development, manages a layer of abstraction to improve communication. What does that mean? It means that UML is to software code what CAD drawings and CAE models are to the underlying engineering equations they represent. You can describe a circle with the formula x2 + y2 = r2; however, it is easier for most people to understand what you're talking about if you draw a picture of a circle.

UML provides both pictures and a common language that everyone can understand. In Figure 5, the translation of “Bow wow…” consists of both a diagram _ the no dogs symbol _ and words expressed in a common language, in this case English. That is what UML is all about: translating concepts expressed at a low level that is decipherable only to a few into higher-level, abstract concepts that the whole team can readily understand and use.

A very specific benefit of using UML is that it provides a powerful communication vehicle. It is an industry standard, managed by a third party organization, and with minimal introduction, testers can work from UML diagrams and derive benefit from them. The use of UML also allows testers to participate early in the software development process. If you use an XP (Extreme Programming) approach to development, for example, there’s a requirement to specify and construct tests before the code is written. You can use UML to do this! Even better, test teams can use UML diagrams and extensions to describe their requirements. Yes, the requirements of the test team can and should be considered. Test teams can help architects and developers design test points (a.k.a. design for testability) into their applications, which will trim the test cycle bottleneck and speed up the overall project. A secondary, but significant, benefit of using UML is that it gives you the ability to improve test coverage by mapping typical UML assets like use cases, class diagrams, and sequence diagrams to test activities like test cases, test designs, and test implementations.

So Let’s Talk a Little UML…

The basic, and most intuitive, UML diagram is a use case, which specifies interactions between users and the system. I am a big fan of use cases because in describing interactions between the user and the system, they also convey a set of test cases that need validation. That conveyance can be inferred; however, I argue that it should be explicit and part of the design process. The UML also uses two other types of diagrams to detail use cases:

Dynamic diagrams, such as sequence diagrams and state charts, to specify behavior. Static diagrams, such as object diagrams and component diagrams, to specify organizational structure.

Let’s take a closer look at how these diagrams work together to create an effective system model.

Use Cases

What is a use case anyway? The formal definition in the OMG (Object Management Group) Unified Modeling Language Specification, V1.3 is: “The specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system.” That’s fancy talk for a description of one or more actions that a user would take with respect to a software application. It is a specific way of using the system from the user’s perspective. Use cases can be used to verify that all requirements have been captured and that the development team understands those requirements.

A use case can be graphical or textual, but ideally it is both. The graphical part of a use case is represented in a use-case diagram or use-case model like the one in Figure 6. This use-case diagram shows that a professor can login, select different courses to teach, and submit grades. The use case can be detailed further with textual information or additional diagrams. Each line in this diagram represents a dialog or a sequence of actions, which is documented by a flow of events, pre- and post-conditions, and optionally non-functional requirements for that use case. So use cases are graphical but mostly textual, because we typically rely heavily on text and written work for communication. Use-case diagrams also create opportunities to automate the management and versioning of use cases, using automated tools.

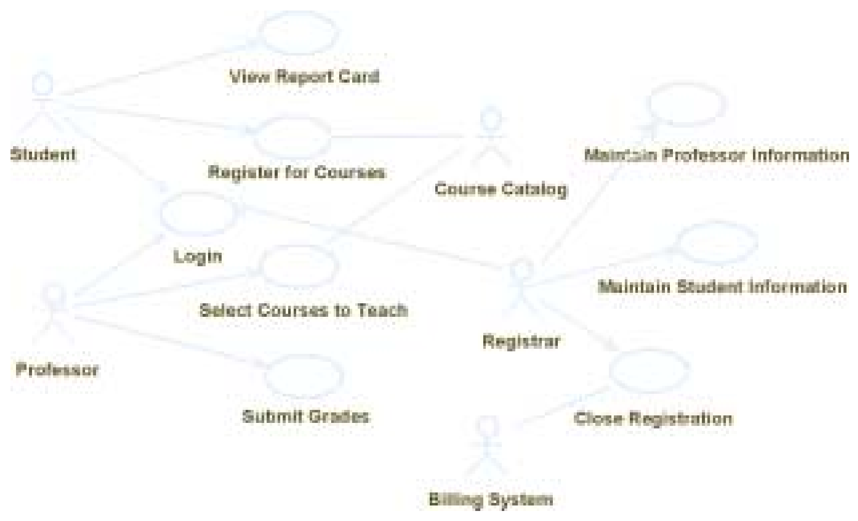

Figure 6: A Simple Use-Case Diagram

The real power of use cases is that they save us time and help reduce errors early on, because they define how real people use the system. And multiple people within an organization can leverage them to work simultaneously on separate tasks. A tester, for example, can easily translate a use case directly into a test case at the same time a developer is building code for that use case. (For more on this topic, see Jim Heumann's article in the June 2001 issue of The Rational Edge (www.therationaledge.com), "Generating Test Cases from Use Cases.")

Essentially, use cases provide a common understanding of the system that is to be designed, built, and tested. With use cases, test teams can validate that the system became what the designers expected; they provide a medium through which to check and ensure that what got built is what you expected. User cases have been around for a long time, but God bless Ivar Jacobson for elevating their visibility and shining a light on the true power and usefulness of use cases with his Objectory Method[10], which is now legendary in software planning and development.

Use-Case Realizations

When a student registers for a course, a lot of details go on behind the scenes. You can realize, or implement, a use case with sequence diagrams, behavioral diagrams, and collaboration diagrams, which incorporate some of these details. Collectively, elements of a use-case realization describe how a particular use case is carried out within the design model, with respect to collaborating objects. A use-case realization ties together the use cases and the classes of the design by specifying what classes must be built to implement each use case. For our discussion, the term class can be used interchangeably with terms like object, component, code-block, or function. The number and types of supporting use-case realizations will vary, depending on what you need to provide a complete picture of the use case and the project's guidelines.

Sequence Diagrams

Sequence diagrams are one way to realize a use case. A sequence diagram represents a time-based flow of a use case. In Figure 7, the Actor (or user, in this case called Ed) interacts with the Authenticator object by invoking the logon method and passing the UserId and Password arguments (which happen to be strings). The user then goes to his shopping cart by calling the retrieveCartSession method from the ShoppingCart object. And finally, the user goes to check out by calling the checkout method from that same object. This sequence of events is a very common use-case realization for Web-based e-commerce applications. And doesn't it specify, quite precisely, the same sequence of events a tester would have to go through to validate that the application can correctly log users on, retrieve existing shopping carts, and proceed to checkout? Of course it does…but it also provides another layer of information usually hidden to the tester: It shows which objects are performing each step! Let's assume the tester had this diagram at his disposal. If he finds a defect when he hits the checkout page, he could augment the defect report by pointing to the object and function call that caused the error. Wouldn't that make for faster and easier reproducibility of bugs!

A sequence diagram shows what the actor is doing, what components he is interacting with, and what parameters are getting passed. Note that the user (actor) is an important part of the diagram to explicitly model what elements communicate with the "outside world."

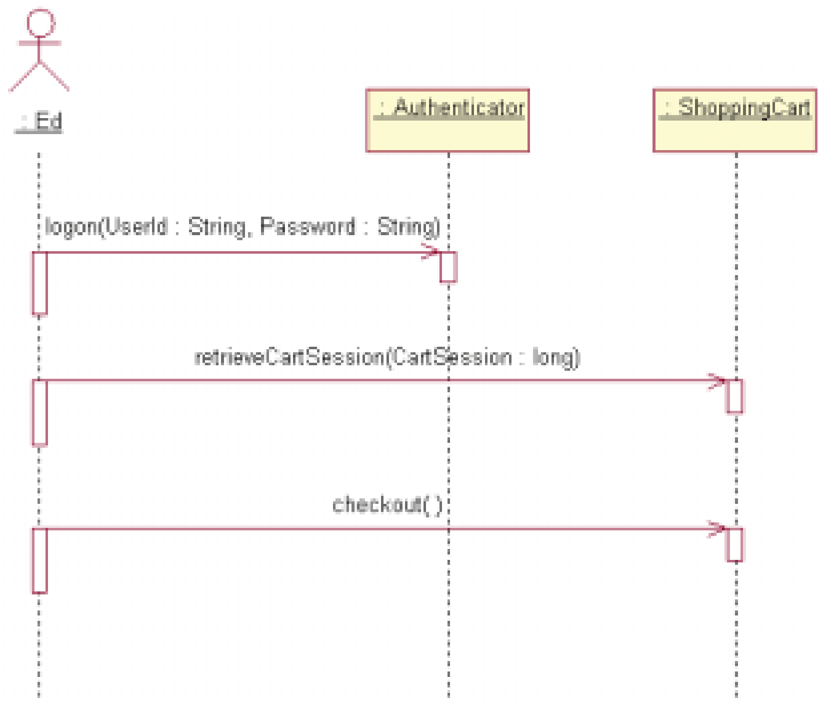
: Ed
: Authenticator
: ShoppingCart
logon(UserId : String, Password : String)
retrieveCartSession(CartSession : long)
checkout( )

Figure 7: A Sequence Diagram

Sequence diagrams are a great asset for testers. They provide exactly the kind of information that testers typically never get to see, because they detail a layer that is rarely accessible by testers. Fundamentally, these diagrams provide a picture of what the testers must test and validate.

A tester going through a use case at the GUI level can see what is going on underneath the covers. Because sequence diagrams are time-based, if the system generates an error at any point, the tester can correlate that with the underlying code. He never sees the code, but he doesn't have to. UML is handling that layer of abstraction between the low-level code and the high-level GUI. This allows him to annotate the defect that he submits and reference the sequence diagram. He can say, for example, "The error occurred in step 3, so the problem might be with the checkout function in the ShoppingCart object." This is something that the tester could not do before he had access to the kind of information provided in sequence diagrams. Reusing assets in this way is a fundamental tenet of Quality by Design, and UML facilitates this quite nicely with sequence diagrams. I know this piece of advice may seem too "Twentieth Century," but even if you don't share the electronic diagram, print it. Passing on a reusable asset in paper form is better than not reusing the asset at all. And your testers will thank you for it (we hope).

Collaboration Diagrams

A collaboration diagram is just an alternate view of a sequence diagram. You can use one, you can use the other, or you can use both. The decision is up to you because they are just different views of the same thing. A collaboration diagram organizes objects based on interactions instead of time. Figure 8 shows the collaboration of objects to support the Create Schedule sub-flow in the Register for Courses use case. The student is interacting with this course form, and the form interacts with the RegistrationController, which in turn interacts with three other objects. This kind of diagram provides an organizational snapshot of a use-case realization.

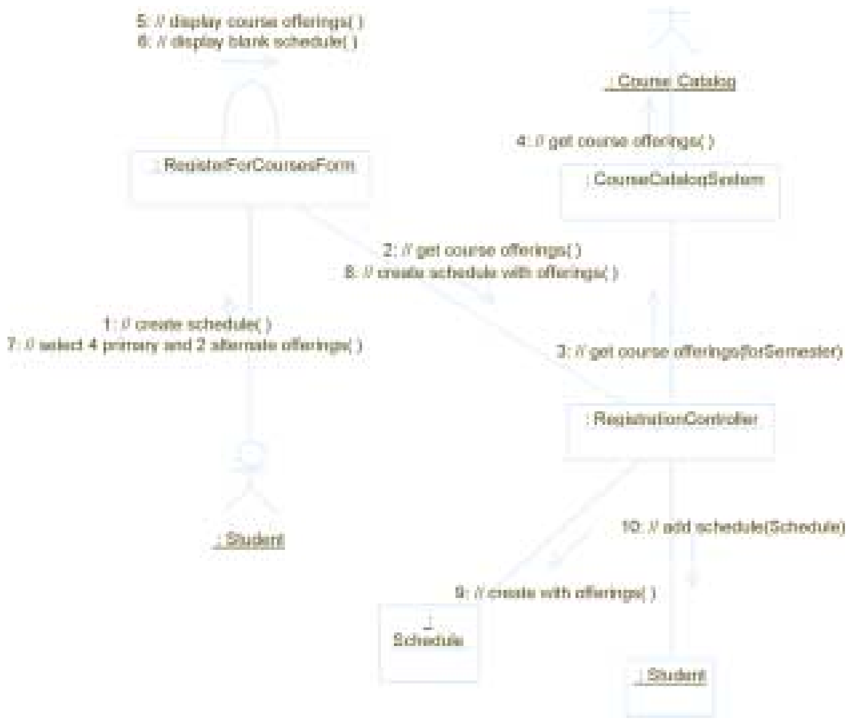
5: // display course offerings( )
6: // display blank schedule( )
: Course Catalog
4: // get course offerings( )
: RegisterForCoursesForm
: CourseCatalogSystem
2: // get course offerings( )
8: // create schedule with offerings( )
1: // create schedule( )
7: // select 4 primary and 2 alternate offerings( )
3: // get course offerings(forSemester)
: RegistrationController
10: // add schedule(Schedule)
: Student
9: // create with offerings( )
Schedule
: Student

Figure 8: A Collaboration Diagram

Activity Diagrams

An activity diagram (Figure 9) is another good behavioral diagram that can augment a use case. I like activity diagrams for two reasons. First, because I came from a procedural programming world (Pascal, Fortran), and activity diagrams are similar to flow charts, which work well in those environments. Actually, activity diagrams are not much more than flowcharts that can handle parallel processes. Activity diagrams illustrate a specific flow through a use case to explain what is happening. Second, activity diagrams include something called swimlanes, which define activities in concurrent threads and who is doing them. Swimlanes provide a view of what the actors are doing versus what the system is doing throughout a use case.

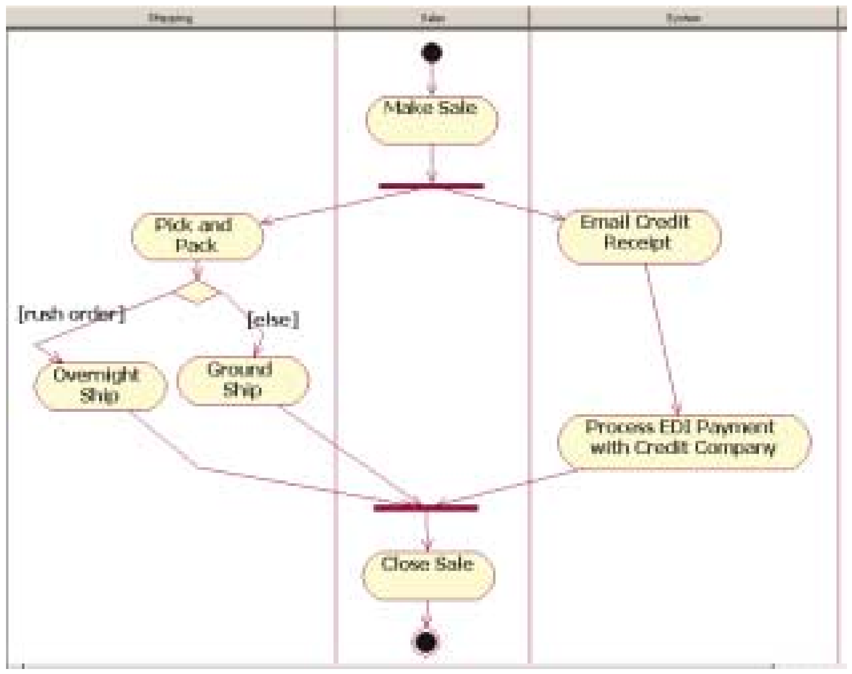

Figure 9: An Activity Diagram with Swimlanes (Vertical Lines)

In Figure 9, the activity diagram tells us that once an order is placed by a sales rep, the system (application) and shipping department can act in parallel. The shipping department gets the goods and packs them in a box, then makes a decision to ship overnight or ground. At the same time, the application can e-mail the credit card receipt and begin processing an EDI (electronic data interchange) transaction with the credit company to start the funds moving. The activity diagram itself visually explains what is happening in this use-case realization, while the swimlanes show which entity performs which actions. These diagrams can be generated very early in the project, because they generally model behavior or business logic. What a tester can take from such a diagram is information that can be used right away to start test planning, because this activity will soon be a test-case realization. The tester can begin to design tests to verify this behavior, including expected results, acceptance criteria, etc.

Using UML Diagrams for Testing

Now, how do these UML diagrams benefit testers and lead to Quality by Design? Well, we have seen that they are designed to help developers and analysts understand both the problem space and the solution space. In the problem space are things like use cases, requirements _ anything that is outside the actual application but still has an impact on it. On the solution side, UML diagrams handle the design and realization of an application: with things such as logical views, organizational views of an application's structure, and deployment views. All along the way _ from requirements to design to realization _ UML diagrams manage levels of abstraction for system definition. As I wrote above, they can give non-programmers a view of the code that is understandable and concise; they interact to provide a comprehensive understanding of the entire system.

But UML diagrams can also manage levels of abstraction for system test. By linking test assets directly to a system's architecture, you can use UML to build in quality from the start. With UML diagrams at their fingertips, testers can create test cases while developers are still working in the problem space, define test designs in the design phase, and manage test scripts in the realization phase.

What exactly are these testing elements I just listed? For the sake of discussion, let's use these definitions:

- *A test design is a description of how to test a system. It includes what and where to test, which data to use, expected results, and the steps needed to implement the design. Test designs can be driven from logical views, which represent the organizational structure of an application. Designs convey intent. Thus, questions like "What is this test supposed to do?" can be answered with test design. So driving designs from organizational UML assets like class diagrams is a logical way to proceed.*

- *A test case is a description of a test, independent of the way a given test is designed. Test cases can be mapped directly to, and derived from, use cases. You can also derive test cases from system requirements. For example, the requirement, "All transactions must process in eight seconds or less" is easily translated into a system performance test case. And much of this work can occur early and iteratively _ at system design and requirements gathering (which happens throughout the project).*

- *Test implementations or test scripts are specific instances of a test design that can be executed against your system. Test implementations can be mapped directly to system implementations and driven from test cases and designs.*

An important concept to retain here is that a test design implies intent. A test design describes how you intend to validate a specific use-case realization.

Test Reuse

A co-worker of mine uses this quote in a presentation he delivers, and I love it for its applicability to testers.

The fox knows many things, but the hedgehog knows only one thing.
-Archilocus, Fifth Century BCE

Testers are foxes, as testing consultant Brian Marick[11] observed in his keynote speech at a Software Testing Analysis and Review (STAR) conference a couple of years ago. This was a shrewd observation. Typically, testers are not specialists (hedgehogs). They have to know a little bit about a lot of things: configurations, network protocols, the development language, and so on. Software development organizations can and should take advantage of this fact. How? By trying to identify and reuse patterns and fault models in their test cycles and then capturing that information for reuse. And testers are generally very good at identifying fault patterns and breaking things.

A pattern, of course, is a proven, reusable solution to a recurring problem. Test patterns can be applied in a number of ways, including Asset reuse across projects and various stages of test. Context matching and finding fault models, etc.

Reuse Across Projects. Let's visit the mechanical design world for a minute. When I was a mechanical engineer, we used a system called "Ideas." Whenever I started a new design, there was a little piece of software running in the background that did a compare every time I made a physical change to my model. It did not do anything too complicated, just simple compares of aspect ratios. If I were designing a bracket, for example, it would compare the height and width and angles with earlier, existing designs. If the system recognized that a similar design had been used or built before in this project, it would interrupt to tell me, "You're designing something that looks very much like this other thing. Do you want to take a look at it to see if you can reuse it?"

It was the most awesome piece of software I ever used. Everything was stored in a central repository that was constantly scoured as I worked with my model. Imagine being able to tell when similar tests have been applied to a particular context before, and then being able to retrieve those assets immediately, making them available (and useful) to your current staff. When software developers figure out how to do that with software designs or tests, they will take the IT world a quantum leap forward. In the meantime, you can use UML to decorate or extend use cases to include details about certain test cases or implementations _ details that will help a tester assess whether it can be re-used or applied to the particular job at hand.

Reuse Across Stages of Test. There are ways to leverage tests for reuse today, and we can benefit from them right away. Developers often create unit tests to validate specific components before integration testing. Why not leverage these unit tests across similar components, or reuse them during the integration testing phase? If you have a test script that was used in the unit test phase, go ahead and run it again as you are doing your sequence testing or scenario testing. If you have validated that your component performs a specific function just fine in isolation, then you will want to verify that it will continue to perform that function when it has to pass data to other components and receive results back from them.

The same idea applies to system testing. A tester does not necessarily have to know how to interpret information that a unit test tool will deliver. But why not use tools that allow the tester to acquire metrics like code coverage while they are performing their normal GUI testing? Then they can simply attach those results to their regular reports as they submit them. The testers do not have to be able to interpret what the code coverage results mean; but if they find a defect, they can provide more information so that a developer can understand and debug that problem as quickly as possible.

Things like memory leaks, system crashes, and missed lines of code happen all the time. And testers find them _ sometimes by accident _ but they typically cannot capture good metrics on these problems or easily reproduce them. Even more significant, testers usually cannot point to the offending function call or line of code that caused the problem. Using “white box” testing tools in conjunction with manual or automated regression testing can yield tremendous benefits. What if you found a crash and could detail your bug submission with the exact line of code that caused the crash? Or what if you were able to attach a code coverage report to your test results that showed the checkCreditCard function got missed during your “place an order” test case? That is powerful stuff! Imagine the ease with which a developer could reproduce those defects. Achieving this level of test quality is easy to do. There are plenty of tools on the market for this kind of testing. Don’t be afraid of them _ Use them! You will be happier, and so will your development team.

Context Matching. Context is one of many test-specific elements of pattern templates as defined by Robert V. Binder in his book Testing Object-Oriented Systems: Models, Patterns, and Tools. Binder suggests thinking about context by asking ourselves these questions.

In what circumstances does this pattern apply? To what kind of software entities? At what scope(s)?

This…corresponds to the first problem to be solved in test design: given some implementation, what is an effective test design and execution strategy? This section corresponds to the "motivation," "forces," and "applicability" subjects of design patterns.[12]

This is a topic too deep for this paper; however, I strongly suggest reading Binder’s book. His content on context matching and patterns is terrific.

Fault Models. Fault models represent another pattern element. Almost everyone uses fault models at one time or another, though they might not even realize it. Here is an example. Let’s say you go to a doctor and tell her you have a pain in your lower back; it goes all the way to your heel, and you cannot sit or lie in bed without pain. When the doctor says, “You must have a sciatic nerve problem,” how does she know that?

Because she has studied fault models. She has studied documentation that identifies patterns, so she knows where to look for problems, and she recognizes this particular pattern of symptoms.

Similarly, testers know where to look for problems in software. They have a sense of where things can break. If a tester opens a browser to test a Web application and immediately gets a JavaScript error, the first thing he might do is check to see if JavaScript is disabled in his browser. If it is, then it might not be an application problem at all, just a configuration problem. The tester knows where to look because he has encountered the problem before. He recognizes the pattern and knows the fault model. Another simple example of this is when a tester checks boundary conditions everywhere because he knows these areas often contain errors.

Fault models are great; but if a tester keeps his knowledge about them to himself, then it is a waste. Testers need to capture that knowledge by taking the time to document fault models. Sharing that knowledge will increase team efficiency in the long run. Return, for a moment, to our doctor/patient example: If you walk in to see a doctor who is just out of medical school, you wouldn't expect to spend five days going through a battery of tests for the doctor to diagnose strep throat, because the doctor has (hopefully) learned from well-documented fault models. Good doctors can easily recognize patterns that lead to accurate diagnoses, and good testers can do the same for your software.

Test Case Generation. Just as with photographic film, test cases can be imaged from the "negative" of a design or model. If the design is clear, then patterns can be recognized and reused. Patterns allow you to automate the test case generation process.

A good example of this comes from the EJB (Enterprise JavaBeans) world. Imagine you are testing an EJB application, and there is a certain bit of logic that has to happen every time you try to instantiate a bean on a certain Web server. The model shows you that it does happen every time, so you create a test once in a header file or in a callable routine and then simply reference it in every test script. You can recognize a pattern in the design and use that knowledge for test case generation: This is yet another example of leveraging your design model to improve quality.

Some Predictions

With such clear advantages, why aren't more development teams taking advantage of these opportunities to leverage the UML and other tools for early testing and reuse and to achieve Quality by Design? In most cases, it is either because their process does not recognize these opportunities or provide the mechanisms to exploit them, or because they do not really use any process at all. A recent Gartner Group summit identified four states of software development, mapped by "speed of development" vs. "quality." Their "Retro Time" quadrant showed where the software industry was ten years ago: comparatively slow development and lower quality. Their "Chaos" or low-quality, high-speed quadrant showed where a lot of e-business companies are right now _ putting out a lot of bad software really fast. Their high-quality, low speed quadrant might be military applications or airlines flight tracking systems; these have to be extremely accurate, yet it might take years to develop just one release. Gartner's fourth quadrant, Utopia, is where we all want to be: putting out high-quality software quickly. The question is, "Where are you, really?"

Even if you are not living in Utopia, there is still hope. As one industry analyst group tells it:

There are some bright spots on the horizon. Vendors are beginning to address software quality issues with methodologies and tools targeted both at higher levels of design abstraction and for use earlier in the design cycle. As a prime example… Rational Unified Process® (RUP®) provides process support for incorporating testing information into Unified Modeling Language (UML) models.[13]

If you were lazy and skimmed through the whole paper just to get to this one quote, don't worry _ it's a good summary of the previous content. I think this quote is very important because it highlights the need to use a process that has support for incorporating test information during the design phase. Whether you subscribe to the Rational Unified Process or some other process, it is vitally important that the process ties quality and testing to the design activity.

Twenty Years Behind, but Catching Up

The 1960s saw the advent of integrated circuits (ICs) and assembly language programming. In the 1970s we progressed to large-scale integrated circuits (LSICs) and third and fourth generation languages (3GL and 4GL). By the 1980s the hardware industry had developed design for testability standards, but the software industry was far behind _ about twenty years behind, as it turned out. But I believe that design for testability standards in the software industry are really going to take off in this decade.

As another industry analyst sees it:

As applications become more complex, spreading over multitiers, across varying networks, and including different client configurations, application architecture testing will become increasingly mandatory.

- Uttam Narsu, GIGA Group

I really love this quote, too _ especially the last line. In fact, I like it so much that I want to leave you with this thought. Mr. Narsu does not say that application architecture testing will be "nice" or "cool"; he says it will be mandatory. That is the essence of Quality by Design. Validate your design! Test your model!

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.

Footnotes

1 The Standish Group International, Inc., CHAOS Chronicles. 2001. 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 paper 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.

10 Ivar Jacobson, Object-Oriented Software Engineering: A Use Case Driven Approach. Addison Wesley Object Technology Series, 1994.

11 Brian Marick, founder of Testing Foundations, is also the author of The Craft of Software Testing. Prentice Hall, 1997.

12 Robert Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools. AddisonWesley, 2000.

13 From “The Changing World of Software Quality,” IDC, 2000.

IBM software integrated solutions

IBM Rational supports a wealth of other offerings from IBM software. IBM software solutions can give you the power to achieve your priority business and IT goals.

- *DB2® software helps you leverage information with solutions for data enablement, data management, and data distribution.*
- *Lotus® software helps your staff be productive with solutions for authoring, managing, communicating, and sharing knowledge.*
- *Tivoli® software helps you manage the technology that runs your e-business infrastructure.*
- *WebSphere® software helps you extend your existing business-critical processes to the Web.*
- *Rational® software helps you improve your software development capability with tools, services, and best practices.*

Rational software from IBM

Rational software from IBM helps organizations create business value by improving their software development capability. The Rational software development platform integrates software engineering best practices, tools, and services. With it, organizations thrive in an on demand world by being more responsive, resilient, and focused. Rational's standards-based, cross-platform solution helps software development teams create and extend business applications, embedded systems and software products. Ninety-eight of the Fortune 100 rely on Rational tools to build better software, faster. Additional information is available at www.rational.com and www.therationaledge.com, the monthly e-zine for the Rational community.