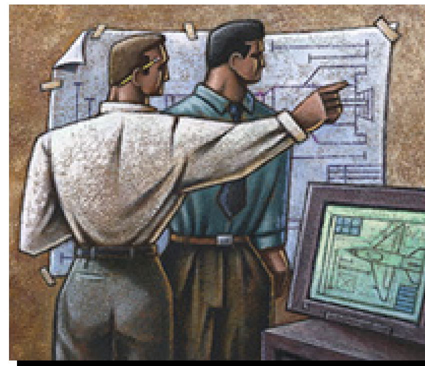
Achieving Quality by Design Part II: Using UML

by Ed Adams with Sam Guckenheimer
Testing Methodologists
Rational Software

In Part I of this article, published in last month's issue of The Rational Edge, *I talked a lot about modeling and its importance to Rational's "Quality by Design" initiative. But so far I have not discussed how modeling is actually done. This article will focus first on the diagrams used for modeling with the UML (Unified Modeling Language), then elaborate further on how using UML models can help software teams achieve Quality by Design.*



UML: The "L" Is for Language

The "L" in UML is for *language*. Language is all about communication. In Figure 1, 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.

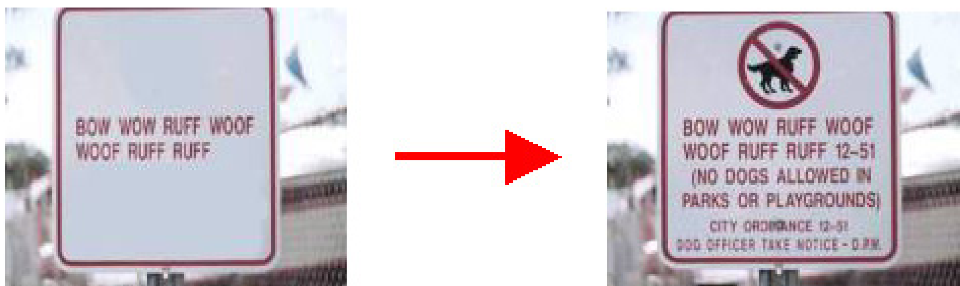


Figure 1: Translating for a Wider Audience

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 $x^2 + y^2 = r^2$; 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 1, 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 2. 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.

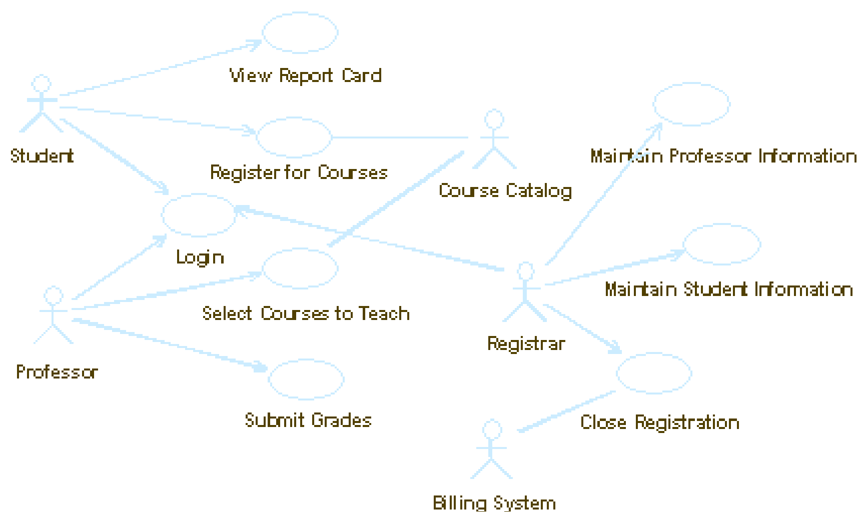


Figure 2: A Simple Use-Case Diagram

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, "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[1], 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 3, 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."

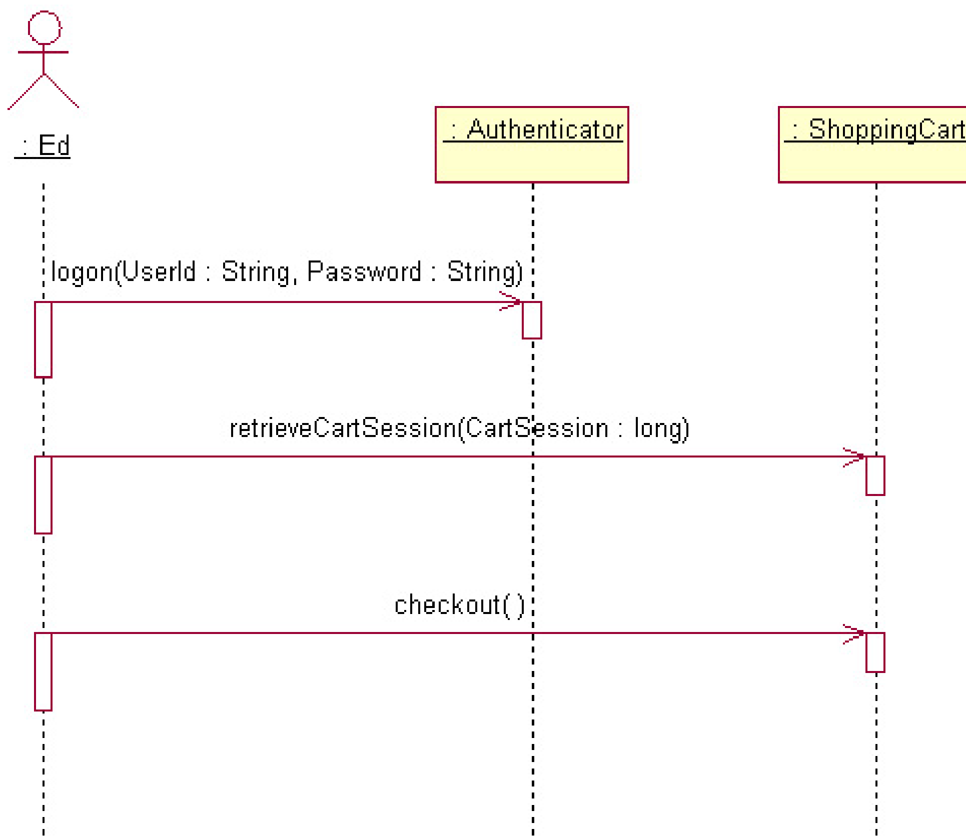


Figure 3: A Sequence Diagram

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 4 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.

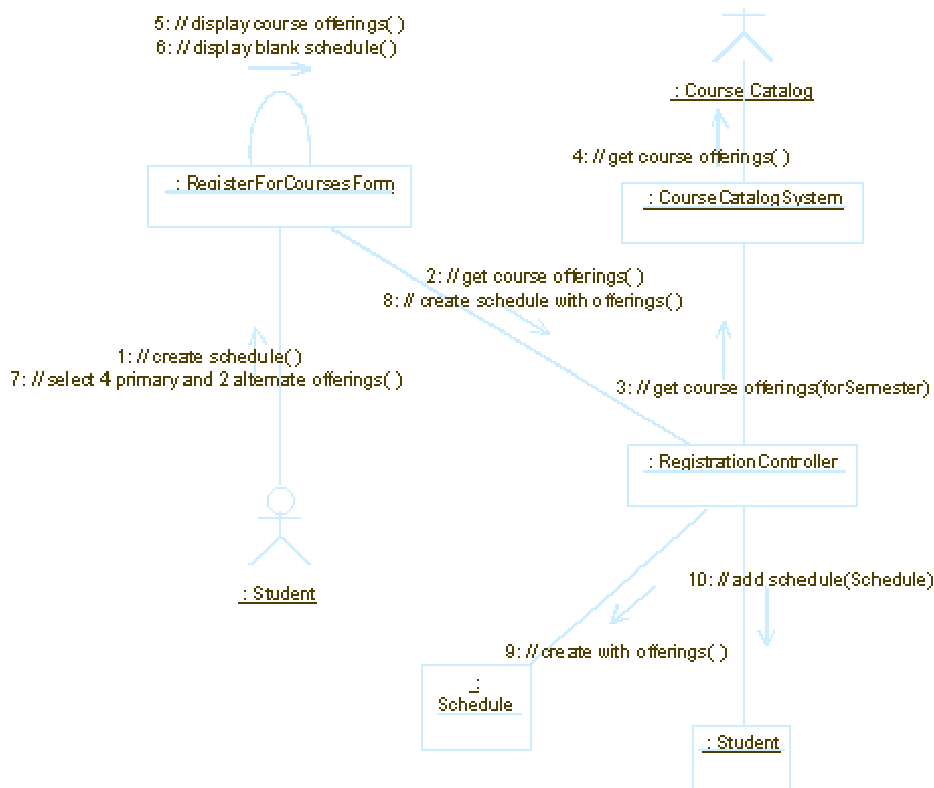


Figure 4: A Collaboration Diagram

Activity Diagrams

An activity diagram (Figure 5) 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.

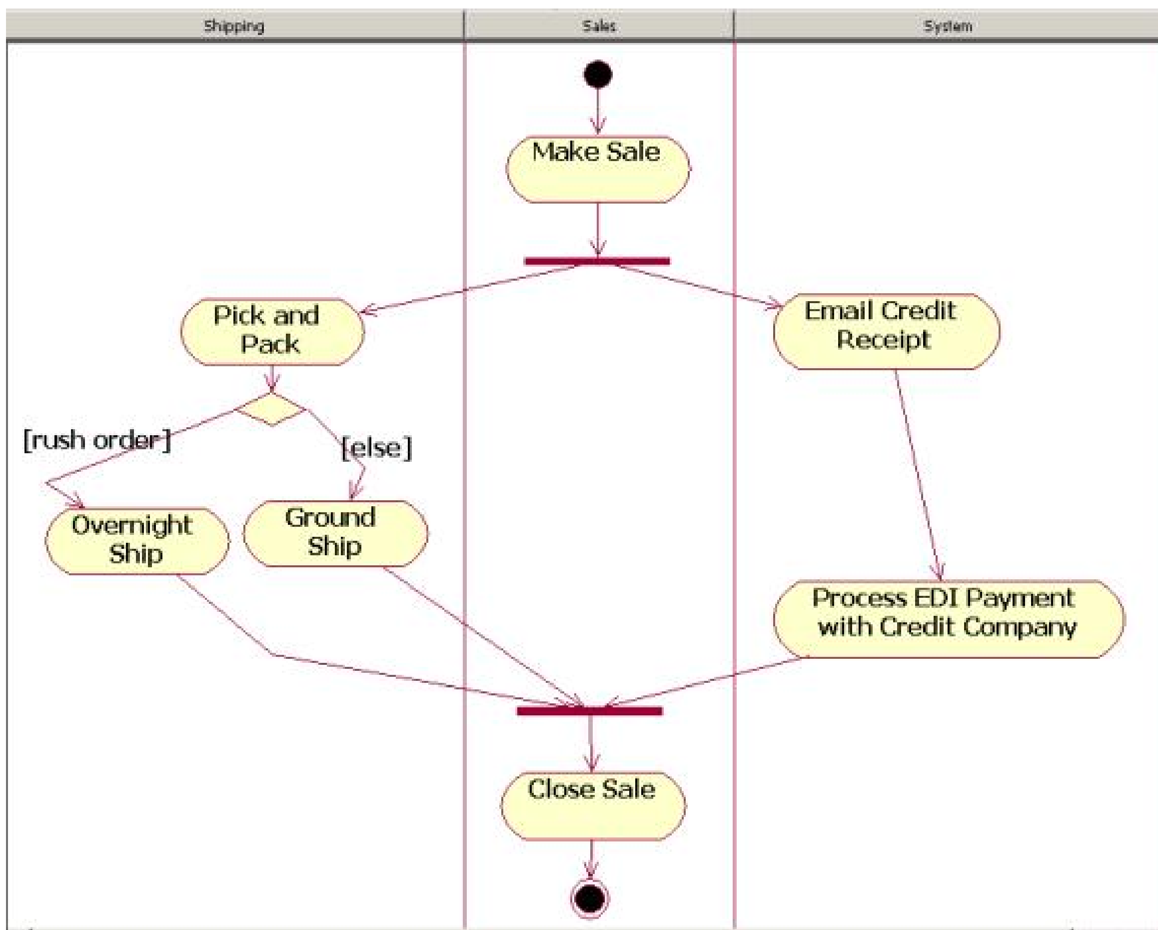


Figure 5: An Activity Diagram with Swimlanes (Vertical Lines)

In Figure 5, 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[2] 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.[3]

This is a topic too deep for this article; 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.[4]

If you were lazy and skimmed through the whole article 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!

Listen to me; I used to shoot people for a living.

Notes

1 Ivar Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Object Technology Series, 1994.
2 Brian Marick, founder of Testing Foundations, is also the author of *The Craft of Software Testing*. Prentice Hall, 1997.
3 Robert Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley, 2000.
4 From "The Changing World of Software Quality," IDC, 2000.

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

Copyright Rational Software 2001 | Privacy/Legal Information