Interview with Brian Marick on How to do Good Testing

by Mark Johnson
The Software QA Quarterly

Q: What is your background and how did you get into the software testing business?

After I graduated from the University of Illinois I went to work for a startup. As is common with "fresh-outs" from college where they don't know a lot about your abilities, they said, "We'll put him on the testing team. At least there he can't do a whole lot of damage." So I was on the testing team for about 6 months when the second typical thing happened. They looked at the project schedule and realized there was no hope of getting the thing done on time, especially if these testers kept finding bugs, so they took the entire testing team and put them on the development team. So I became a developer.

I stayed a developer for about 7 years or so, doing mostly compilers and operating systems. Then I moved on to line management and training projects for new hires. During the time I was a developer there was always a heavy testing component to my work. After awhile, I decided to concentrate on doing the testing.

Then I did some joint research at the University of Illinois, looking at some ideas in testing techniques. But the ideas didn't work out, so I didn't end up with a Ph.D. I did end up with a framework that made sense of how I was already doing testing, plus a coverage tool (GCT) that I could use in my real job.

After awhile I decided being a consultant would be more interesting than having a steady job, so that's what I do now. I still try to spend time writing code and tests, because I have a dread of becoming one of those consultants who talk about software development but haven't actually done any for a decade. I have to admit that recently I haven't gotten my hands as dirty as I would like, though.

Q: What would you view as the ideal software testing situation?

I don't think there is any real "ideal" situation. As far as who should do the testing, I think it is most cost effective to have the developers of the code do certain types of testing, and then have an independent test group for other types of testing. If you have an independent group trying to do all the testing, they don't have time to do anything well. So the optimal way to divide up testing is to assign a large part of the testing effort to the original developers, and have them backed up by an independent test team. The independent test team is typically testing the entire system. They are looking for problems that the developers have a hard time finding, such as interactions between major subsystems, or certain global issues such as race conditions or memory usage.

As far as the process of testing goes, you really need to have the independent testers involved at the beginning of the project. They will be looking at the first version of the system specification, the user's manual, or the prototype, however your process captures the requirements before implementation. During this period, they are constructing initial test designs or test design information from those documents before any code is written. This is really a form of inspection or review process. In addition to test design information, they will find defects. It is a lot cheaper to find defects at this stage.

So the system testers are involved at the beginning, producing this test design information, which is essentially lists of things that need to be tested. These are what I call test requirements. These are not complete tests, because it is premature to write complete tests, but they are the groundwork upon which you will build the complete tests.

Q: Could you explain what you mean by "test requirements?"

The idea of a test requirement is actually relatively simple. For example, say you have a program that determines the square root. If you give it a number greater than or equal to zero, it gives you the square root. If you give it a number less than zero, it gives you an error indication. Any tester who has read *The Art of Software Testing* , by Glenford Myers, is going to immediately think of boundary conditions. You want to test the value zero, and you want to test the negative number that is as close to zero as possible. Those are two test requirements.

In a more complicated program, you will come up with a long list of things you want to test. There will be tests of all sorts of error conditions and all sorts of normal situations. You want to identify all these situations to test. They don't have specific values assigned to them yet, though. For example, if the program gives money to someone, as in a banking transaction, one test requirement is that you try to give the person a negative amount of money, another test requirement is that the person you are trying to give money to doesn't exist, etc. You have these lists of test requirements. They don't specify exact values, such as the name of the person who will be getting the money.

The next step of testing is to choose input values that satisfy these test requirements. A single test may satisfy several test requirements. For reasons that are hard to explain without a blackboard to draw examples on, it is a good idea to write your tests so they are reasonably complicated, with each one satisfying a number of test requirements. This is actually more expensive than simply writing one test case for each test requirement, because it takes more time to think of these more complex tests. Although it is easier to write one test case per test requirement, those simple test cases are not as effective at finding defects.

Here is a test requirement example. You are testing a hash table insertion routine. For an insertion routine, you might have these test requirements, among others:

- insert a new entry
- insertion fails - element already present
- insertion fails - table is full
- hash table is empty before insertion

These are test requirements, not tests, because they don't describe the element being inserted. That's an irrelevant detail at this point. If you dive into writing tests too soon, it's just like diving into writing code too soon. Important things, in this case test requirements, get overlooked in the rush. Note that in this example, you can satisfy two requirements (the first and fourth) in a single test.

Q: Could you explain more about the creation of these test requirements?

The system test people should begin creating test requirements from the external description of the system, as part of reviewing it. I like to have them also create a few actual tests, although they can't run them at this point, because the act of trying to build a scenario of the use of the system reveals places where the pieces don't fit together well. They are not just reading the system specification, they are trying to use it to perform the same tasks the users eventually will. This is an idea I got from a paper written a long time ago by David Parnas and David Weiss about directed inspections. But I wouldn't have the system testers create too many actual tests, because the system specification is subject to too much change between when it is being initially reviewed and when the code has been written and the system tests can be run.

Now you have the completed system specification, and the test requirements that say in general terms what the system tests will do. The test requirements are something the developers will want to know about, because their code will have to handle them. As the developers start their design process, they should also be finding and recording additional test requirements. When they do a design review or inspection, the list of test requirements, from both the system testers and the developers, makes a nice checklist. So, if the developer is doing the square-root function, they can ask the question, "What is the design going to do if a number less than zero is given to it?" In this way, the test requirements can help keep design defects from being turned into code. At each stage, as they go through design and coding, the developer can find more test requirements.

Q: What are Test Requirement Catalogs and how are they used?

I do a lot of my testing based upon catalogs of test requirements. Take for example the POSIX library routine "malloc." Either malloc returns data, or it returns an error. This leads to two test requirements for any use of malloc. Test if the caller handles returned data, and also test if it handles a returned error. Put these test requirements in your catalog for testing use of the POSIX library. Now whenever someone calls malloc, they know to think about those two test requirements. You can do this with all sorts of function calls, other operations, and data structures such as linked lists, arrays, etc. In my book, I provide such a catalog of standard test requirements. In addition, you can create special purpose catalogs. For instance, you might create a test requirements catalog for VHDL data structures, if that is what your program processes.

Q: OK, once the code is created and you have the lists of test requirements, how do you start creating tests?

You start creating either manual or automated tests. During planning the project, the decision needs to be made as to whether automated, repeatable tests, or manual tests will be done. This is a trade-off. In the short term, automated tests are more expensive, but in the long run they are cheaper. It is an explicit risk decision: here are the costs, here are the tradeoffs, here are the advantages and disadvantages of both. And you make a decision.

If you are doing automated tests, you want to make those tests as cheap as possible. An automated test requires a driver that feeds values to a module and checks the results. But what's a module? People talk about "unit testing," which usually means treating each subroutine independently. That becomes a maintenance nightmare. There are too many drivers. As the system evolves, all the changes done to the code tend to break the unit tests or drivers. Eventually, people give up maintaining them and they are abandoned.

So, to do automated tests, you should test subsystems as a chunk. If I am doing the lexical analyzer for a compiler, I would test the whole thing together. I would write tests that satisfy the test requirements from all the subroutines in that subsystem, driving the tests through the subsystem interface. This is doing unit level test design, but doing implementation at a larger level because it is cheaper. There can be problems with this, such as trying to figure out how to drive a unit deep inside the sub-system from this interface. For instance, how do you get a negative value to that square root function way down inside the subsystem? But, my feeling is that it is better to pay the cost here, in trying to deal with these problems, than to pay the cost of writing and maintaining unit test drivers.

This isn't to say that if you have written the square root program, you can't fire up the debugger, give a "-1" to the function, and see if it works. This is simple, quick, and easy, go ahead and do it. What we are talking about here is creating the repeatable test suite.

Now, there is one other thing that can be done to reduce the cost even more. In certain situations, you can test the subsystems via the system interface. This is great, because the system testers are going to be writing the system driver anyway. So the subsystem tester can use this driver for free. Or, the subsystem tester may be able to pass their test requirements on to the system testers, and have the system testers include them in the system tests. This can be a net win because the system tester already knows the system driver, and can probably write tests more efficiently.

Q: What about the situation where you don't have a good specification to work from?

Well, you do what you can. Even with no specification, you can derive test requirements from the code, by trying it out and seeing what it does. Obviously you will not do as effective a job of getting test requirements, but you can still do a good job. You don't have to give up. For example, if you crash the program doing something that a user could do, then you have gained something.

Q: OK, you are done with your test design. How do you go about improving your tests?

Your intent in defining the test requirements was to have one test requirement for each defect in the code. In the perfect world, you would have every test requirement finding a defect, and every defect in the system would be found by a test requirement. In reality, human beings don't do perfect things.

One easy way to improve your test design is to have another person sit down with you and go through your tests. With a relatively small amount of the second person's time, you can usually find some things that are missing. This is a good thing to do, but it is not sufficient.

What you would like is a tool that would look at your test requirements and the program, and then spit out the ten different test requirements that you missed. Such a tool does not and cannot exist. So you do a series of approximations. A simple approximation is to run all your tests, and then look at the program and see which lines of code have never been executed. Because, if there is a defect in a line of code, and that line of code has never been executed, you are certainly not going to find the defect. A really good job of test design should exercise all the lines of code. Knowing that some line of code was not executed tells you that you did not do a really good job, and it tells you that you under-tested the features associated with that line of code.

For example, say you are testing a networking subsystem, and you find some lines of code that have never

been executed. When you look at the unexecuted code, what you find is a whole bunch of code that looks like "If some error condition occurs, then handle the error." What you conclude from this is that you haven't tested error handling very well. For a networking protocol module this is a problem, because a lot of what a networking module does is handle errors. Now you go back and think about the types of errors this code is going to have to handle, and you design tests for error handling. This is the basic use of coverage. It is a way to point you back to some area in your test design that you didn't do well enough.

The danger of coverage is that people fall into the trap of just trying to execute the lines of code. What happens is you take the list of code line numbers that haven't been executed and just pick test cases to make sure you execute each line of code. The problem with this is that a good test suite will execute all the lines of code, but just executing all the lines of code doesn't mean your test suite is good. There are certain types of defects that are easy to miss simply by executing the lines of code. What you really wanted to know was "Have I covered 100% of the test requirements I should have?"

Q: What are the different types of code coverage and how do they help with understanding the completeness of test requirements?

People have invented a lot of different types of code coverage measures. Here are several that I find useful. Stronger than line coverage is branch coverage, where you exercise every branch in the code in both directions. So, for an "IF" statement, you want that branch to go in both the true and false directions. This also corresponds better to your test design. Your test design shouldn't be "How do I execute all the lines of code?" It should be based on thinking about the different input cases this code will have to handle. These input cases correspond more precisely to the branches in the program than they correspond to the lines. Therefore branch coverage gives you a more precise mapping back to your test design, which is better.

A little bit of an extension to branch coverage is multi-condition coverage. This deals with the situation where a branch has a compound condition. For example, for the condition is "IF a OR b", you could have one test where "a" and "b" are false, and another test where "a" is false and "b" is true. This satisfies branch coverage because you have taken the "IF" both ways. But it is kind of odd that "a" has always been false. What this means is you could take the "a" and lop it right out of the program, changing the "IF a OR b" to "IF b", and the test would work exactly the same. The branch would go in the same directions for these tests. So what this shows you is that you have tests that are not sufficient to detect when you lop out a chunk of the program. This should make you nervous. This is where multi-condition coverage comes in. It requires you to have tests where both "a" and "b" individually take on the values true and false. This is a little stronger than branch coverage, it doesn't cost much more than branch coverage, and I've known it to find problems that branch coverage did not. And it corresponds better than branch coverage to the type of test design you are doing.

So you continue to add on more types of coverage that more closely correspond to good test design. Next is relational coverage, which corresponds to testing boundary conditions. The reason for testing boundary conditions is that sometimes the programmer uses "less than" when they should have used "less than or equal to." It is perfectly easy for a coverage tool to measure these boundaries. So every time it sees a "less than" in the code, it will measure whether you have tested the two boundaries that the "less than" makes. This is an example of a test design technique and a coverage measure that correspond nicely.

The next kind of coverage is loop coverage. There are certain types of problems that will only be detected if you iterate a loop zero times. For example, a "WHILE" loop where you hit the "while," it is false, and so you never enter the body of the loop. There are other problems that are only detected when you iterate the body of the loop more than once, so you go around the loop at least twice. And there are even problems that can only be found if you go around the loop exactly once. I think this last case is the least valuable. However, I have personally seen an extremely embarrassing defect caused by this case which led me to say "This program is garbage!" In fact, the only case where the program failed was when you went around the loop once. It actually worked for almost any normal use of the program, with iterations other than once through the loop. This gives you a nice rule for your test catalog, by the way, which states that for loops, you should test them for zero, one, and many iterations.

The final type of test coverage I like is called interface coverage. This is based on the observation that many times the faults in a program are due to the programmer not anticipating specific results from a function call. The "C" programmer's favorite is calling "malloc," which allocates additional memory, if there is any available. There are a lot of programs with calls to malloc that assume that memory is always available. When they run out of memory, it leads to a core dump because they haven't checked for the error case. Or in UNIX, a lot of programs assume that if you write to disk it will succeed, but in fact it could fail. One person told me about

using a backup program that, while it was writing its archive, did not check for the disk filling up. So if the disk did fill up, it would blithely continue on writing, believing all the writes were succeeding. Of course this is not the type of thing you want your backup program to do! These defects motivate one of the test design techniques for developer testing: Enumerate as test requirements the different distinct return values of the function being called, and make sure that the code doing the call can handle each return value. A coverage tool can measure whether all return values have been exercised.

We have these types of coverage, and they correspond partially or wholly to the test design process. If we really had perfect coverage measurement, we wouldn't have to do any test design, we would just let the coverage tool tell us what we need to do. But even then a coverage tool would not be sufficient to tell us everything we need to know. This is because there is no way for a coverage tool to find faults of omission, where the program should do something but doesn't. These are exactly the faults that escape testing and are found by customers. The coverage tool works on the code of the program, but the fault of omission is the code that ought to be there but is not. A lot of what you do in good test design is try to find faults of omission, and coverage tools are of relatively little help.

Q: I have read in some of your published papers that it should be easy for someone to get 100% coverage. On the other hand, I know people who when they first run a coverage tool on their tests get numbers more like 50% or 30% and are very discouraged. What should they do?

This is a good point. Sometimes this situation is simply that the system test suite has a low coverage level. If it is the case that testing is divided up between subsystem and system level testing, the system test suite shouldn't be trying for 100%. This would be duplicating testing work that should be done at the subsystem level. In this case, the 100% comes from the combined results of subsystem and system testing.

When I talk about 100% coverage being an achievable goal, I am speaking particularly about sub-system level testing. I'm also assuming that you have a reasonable amount of time. My definition of reasonable is that in realistic commercial testing situations, I can get 100% coverage without being given more testing time than you would normally expect. So it is not that I have a 1000 line program that I have been testing for the last 20 years. Of course, you might not be given enough time to get 100% coverage. In that case, you won't. But you can still do the best job you can and use coverage in a useful way.

Q: So, if your time is limited, and you can only focus on system level testing, what should you do?

If you are just doing system level testing, it should be entirely driven by risk. By this I mean first focus on the areas of the system or features of the system where there is the highest probability of faults. For example, if you have some areas with a few minor changes, and other areas with completely new code, the completely new code has probably got more defects because it has more lines of new code. This makes it somewhat more risky. Second, consider the severity of possible failure. Code that can crash the system is riskier than code that cannot. Third is visibility to users. If this is code executed very, very seldom, only by system administrators, then any failures will be less visible than the log-in code, where if it doesn't work, everyone will see it.

The job of the system tester is to take the time that is allowed, look at the system, figure out what the highest risks are, and test those particular areas. In this situation, you do not expect to get 100% coverage because you have identified parts of the system as not worth testing heavily. There will be low coverage, but this won't be uniform. You'll have what Rick Conley of Veritas calls "black holes." Some of these black holes will be areas you decided not to test because they are low risk. In a major testing effort you will usually find one or two cases where there are big chunks of code that you didn't want to miss. What coverage does is focus your attention on unexecuted areas of the code. You look at those and say "Did I expect this? Yes. Did I expect that? Yes. Did I expect this? No! I didn't expect this at all." Now you go and test that part that was unexpectedly unexercised. The actual coverage number is not important. What matters is how well the numbers match your reasonable expectations, given the constraints under which you are working.

Q: What if you have a low level of measured coverage, are doing subsystem level testing, but have limited time?

To a large extent, you are in the same situation. If you don't have enough time you can't expect 100% coverage. In your planning, if you think you will not have enough time, you want to create a flexible enough plan so that you don't run out of time without accomplishing anything, or with having only tested the routines from A to D because you have been testing in alphabetical order.

The risk based testing that I outlined earlier covers this. First, you want to do the same sort of risk analysis. For subsystems, you'll find that generally the visibility in a particular sub-system is pretty much uniform. The severity is more or less uniform, too. But the probability of errors varies more across the routines of the subsystem. Some of the routines, or groups of routines, will have been tricky to write, or you will have found problems as you were writing them. If you found problems as you were writing the code, there are probably more defects that you didn't find. You should plan your testing to hit those high risk routines. When you are creating test requirements, you will want to spend more effort on the high risk routines, less effort on the medium risk routines, and you probably won't even bother to write down test requirements for the low risk routines.

Remember that these test requirements also have value during the design process because they are helping you identify problems. In fact, if you just wrote test requirements and used them in inspections of your high and medium risk routines, it would still be worth the time spent creating the requirements, even if you never used them to create tests.

Maybe you don't have time to create an automated test suite. You can run tests manually, and make them more simplistic than you would otherwise, still using the test requirements for your high and medium risk routines. If you get through all the high risk routines, then you test the medium risk ones, and continue on testing the lower risk areas until you run out of time.

When you finish testing your high risk routines, you should measure your coverage, and it should be 100% for those areas. If you don't get 100%, it is probably because you made a mistake in test design, or a lot of times you made a mistake in implementing the tests. There is a typo in your test input, so you are not really testing what you thought you were testing. Those are really easy to fix. So you get some low-cost benefit from coverage that way.

Generally speaking, the idea should be to measure your coverage and match it against your expectations. You will find you have missed a little bit, and then you will go back and bring your actual results up to your expectations. Once you have gotten good at testing, and you are comfortable using coverage, using coverage will take 4% or less of your total testing effort. It really amounts to double checking your work.

Q: In the case where someone has enough time for doing good testing, what do you mean by 100% coverage?

First we need to define what types of coverage. One problem with the types of coverage that I mentioned earlier is that there is no one coverage tool that measures all of them. There is at least one tool that measures each of them. So the types of coverage that I measure are branch, multi-condition, relational, and loop.

Another thing to define is "feasible" coverage. The idea is that a lot of programmers write defensive code. They will put in sanity checks to test for impossible situations. If the impossible situation occurs, they will print a message warning the user and exit gracefully. Most of the time programmers will be right about what is impossible, so you can't exercise all the code with your testing. Maybe you find that 10% of the branches are impossible situations. So you throw them out. You do this by running your tests and measuring your coverage. Say you get 90% coverage. You look at where the remaining 10% is located and ask yourself "Is this because of a mistake in my test implementation, is it a mistake in my test design, or is this coverage condition truly impossible?" If it looks like code that is impossible to reach, then you accept that you will not be able to test it. So you might decide that you can only achieve 94% coverage on this piece of code, but that 94% is 100% of the feasible code.

In the future if you are running these tests again on this code, you remember that you could only get 94% feasible coverage. If someone has made changes to the code, you will have to go back and rethink whether the previously impossible code is still impossible. Of course, you want to be reasonable in doing this and trade off the risk of missing something against the time consumed.

Next quarter we conclude our interview with Brian Marick. We will cover:

- *Testing Object Oriented Software*
- *Maximizing the benefits of testing efforts and how this can improve time to market*
- *People issues and how to get developers started doing unit and subsystem testing*
- *Where is testing going? What is under control today, what are the challenges for tomorrow?*

Interview with Brian Marick on How to do Good Testing: Part 2

by Mark Johnson
The Software QA Quarterly

Last quarter, in the first part of our interview with Brian Marick, we focused on software test design using test requirements and the evaluation of testing effectiveness using test coverage measurement.

In the final part of our interview we will talk about:

- Brian's new book, *The Craft of Software Testing*
- Testing object-oriented software: what is similar and what is different
- The development life cycle, increasing developer testing, and how software testing can actually improve time to market
- Brian's vision of the future for software testing

Q: Before we talk further about testing, I see that you have a new book out since the first part of this interview was written. Is your title a take off on Glenford Myers' book The Art of Software Testing?

Yes, it is partly to say that since the late 1970s testing has moved from an art to a craft, and also an homage to Myers' book, which I think is still the best introductory book on the what and why of software testing. So I called my book *The Craft of Software Testing* . Some people object to the word "craft," thinking I'm suggesting being sloppy and ad-hoc or somehow unscientific. I'm not. A craft is a disciplined profession where knowledge of the type published in refereed journals isn't yet sufficient.

I wrote this book because I really felt there was a need for one that provided a cookbook, step-by-step process for software testing. Whenever you read a book, then sit down and try to apply it to your problem, there's always that annoying period where you have to fill in the gaps the author left unexplained. Sometimes the gaps are huge, the "obvious details" are often the hardest problem. I wanted to give all the details. Of course, I didn't entirely succeed, and of course the way you really learn a craft is to watch someone else do something, then do it yourself and have them comment on your attempts. So even my book is no replacement for being able to work with and learn from experienced professionals.

You could say the book is a very detailed expansion of the types of things we have covered in this interview.

Q: OK. So far we haven't talked about testing Object-Oriented Software. Do you see it as different than testing other software?

I'd like to preface this by saying that everything we have talked about up to now I have personally used on "big software" and I know it works. My ideas on Object-Oriented software make sense, but I have only tried them on small programs. I believe that they are right, but I'm not as confident as if I had years of experience using them. Think of them as having been inspected but not tested.

There are three issues in testing object-oriented software. First, how do you test a base class? Second, how do you test external code that uses a base class? Finally, how do you deal with inheritance and dynamic binding?

Without inheritance, object-oriented programming is object-based programming: good old abstract data types. As an example of an abstract data type, take a hash table. It has some data structure for storage, like an array, and a set of member functions to operate on the storage. There are member functions to add elements, delete elements, and so on.

Now, the code in those "member functions" is just code, and you need to test it the same way you test other code. For example, a member function calls other functions. Some of them are other member functions, some of them are external to the class. No difference. Whenever code calls a function, you can derive test requirements to check whether the code calls it correctly and whether it handles all possible results.

There is one slightly funny thing in the code: that sort-of-global variable, the array the code maintains. But

that's not really any different than an array passed to the code as an argument. You test it in the usual way: arrays are collections, and you find test requirements for them in a test requirement catalog.

That seems too simple: just testing classes like any other code. What about the relationships between the member functions? What about the state machine the class implements? Relationships are largely captured by test requirements for the shared data and called member functions, secondarily by having more complex tests that exercise sequences of member functions. Those sequences are partially driven by the state machine.

Now look what we have: test requirements for the object as a data type, and also for the member functions that control the object. We've used them, along with other requirements, in testing the class code itself. But these requirements are also useful for testing external code that uses the class. If some code uses a hash table, it should be tested against the hash-table-as-collection requirements, against the requirements for the member functions it calls, and perhaps against the hash-table-as-state-machine requirements.

What we really should have is a catalog entry telling anyone using a hash table how to test their code to see if it misuses the hash table. These cataloged test requirements are a form of reusability: reuse of test requirements. I hope that as vendors provide more and more class libraries, they will also provide catalogs of test requirements for their objects. This will be a good thing, it will save you from having to think of all these test requirements yourself, you can just look them up. The vendor will tell you how to test your code for likely misuses.

When you get into object oriented software, you also get inheritance. The hope is that you won't have to retest everything that is inherited. Using the hash table example again, say you create a class that inherits 80% of its functions from the original hash table, and adds 20% of its own functions, either by adding new functions or by overriding hash table functions. The hope is that you would only have to test the 20% of added or changed code. Unfortunately, this generally doesn't work. The problem is that if you override member function 'A', it may be used by member function 'B' of hash table, which has not been changed. Because something that 'B' uses has changed, 'B' now needs to be retested, along with 'A'.

You want to minimize the amount of testing you have to do. My approach is this: you have the class catalog for the base hash table class, with its test requirements. And you have the new or changed functions in the derived class. Now you need to analyze the differences between the base class and the derived class and use the differences to populate the derived class's class catalog. I have a long procedure for doing this analysis, which considers all sorts of special cases, so I won't try to go into it here. What you end up with is a parallel inheritance structure. On one side, there is the base class and the derived class. On the other side, there is the base class catalog and the derived class catalog. The derived class catalog inherits some test requirements from the base class catalog. It also contains new test requirements based on the differences between the derived class and the base class. The new test requirements are what are used in testing the derived class by itself. The entire test catalog for the derived class is used when testing external code that uses the derived class.

If your object-oriented design is done well, for example following Bertram Meyers' book on object-oriented design, so that you have a good inheritance structure, most of your tests from the base class can be used for testing the derived class, and only a small amount of re-testing of the derived class is required. If your inheritance structure is bad, for example if you have inheritance of implementation, where you are grabbing code from the base class instead of having inheritance of specification, then you will have to do a lot of extra testing. So, the price of using inheritance poorly is having to retest all of the inherited code.

Finally there is the issue of dynamic binding. The issue here is that if you have a variable in your code that is a hash table, you don't know if it is the base hash table or the derived hash table. This means you have to test it with both cases. There is yet another kind of test requirements catalog that I use to keep track of this.

You can see that object-oriented programming requires considerably more bookkeeping during testing. I don't think there is a way to get around this extra bookkeeping. I should also point out that there are lots of other theories right now on how to test object-oriented programs.

Q: How do you see testing as fitting into the overall development life cycle?

I believe the idea of test requirements as separate from actually creating the tests to be a powerful idea. This is because the test requirements can be created well before the actual tests. When you have the first deliverable from your project, say a prototype, a user's manual, or a specification of some sort, you can create

test requirements from it. At this point you can't implement any tests, in fact you might not even be able to design the inputs for any tests because you might not have the complete product specification yet. But you can be writing down test requirements, and asking the question, "Will this design handle the test requirements I'm coming up with?" This means that the test requirements can actually give you something to test your specification against.

So, you want to have the system test personnel involved from the beginning, defining the system test requirements. They can give you a lot of feedback on whether the specified system will actually work. And this means the system testers will have time to plan how they will automate the tests for this product. Automation is really essential to getting a good long-term job done.

Then the product is passed on to a group of developers who will design and implement it. The design might be informal or rigorous. As they are doing their designs, they can use the system level test requirements to evaluate their subsystem designs, by asking the question "This is how the system level tests will exercise my subsystem. Will it work?" They can also be creating their own subsystem level test requirements for use in reviews or inspections before they get to coding. And then as they do the coding they can identify still more test requirements for use in subsystem testing.

It may be the developers who do the subsystem level testing using the test requirements they captured during design and coding. Or, it could actually be more cost effective to pass the test requirements identified by the developers to the system testers. This is because the system testers are much better at producing repeatable test suites than are developers. And it is sometimes easier to create a repeatable test using the system level interface than by taking a module of code and testing it in isolation. So, to reduce costs, you still do the test definition at the code level, but you actually implement the tests at the system level. However, this may not be effective for very low level code that is hidden way down in the system and communicates with hardware, for example. You need to see if tests for this low level code can effectively be written at the system level. If not, you will need to test this low level code at the subsystem interface level.

There are a couple of more things the developers and system testers should cooperate on. Once the tests are implemented, they should sit down together and review the coverage. Together they will be able to find more test design mistakes than they would individually. The other thing they should do together is to look at customer bug reports. Customer found bugs do a very good job of telling you where your testing is weak. The only problem with them is that they come a little bit late. But they can be used for improving the testing for the next release.

Q: I have found that some developers don't focus much on testing their work. How would you recommend going about getting developers to do good testing?

There are matters of corporate culture that have a very real effect on something like this. So, a general prescription may work well for some cultures but not work well for all cultures. There are some things that you can do to build a strong developer testing climate in your organization. You need to think about the things that you can accomplish given your organization's culture, and your developer base, and then tailor your approach to that. An error that a lot of people make in quality assurance is that we tend to be evangelical. Like anyone else, we tend to think that our work is the most important part of the effort. So we tend to go to developers and want to convert them immediately into replicas of ourselves. What happens then, of course, is you try to make large and drastic changes. This will then either succeed entirely, or fail entirely. Unfortunately, it is much more likely to fail. So the single most important thing you can do is to realize that you have to make this change gradually. For the next release of your product, you want a bit more developer testing. And over time you want to continue to improve it. Basically, you want to spend a moderate amount of money looking for a moderate improvement, instead of betting everything on one roll of the dice.

The other thing you need to do is to consider the typical personality of developers. Developers tend to be fairly skeptical and critical. They will not take it on faith that what you are talking about is the right thing to do. You have to convince them, and in their own terms.

So the way that I approach improving developer testing is to start them doing things that will seem most appealing to them, but may not be the most valuable overall. For example, I believe that improving test design is the real key to improving testing effectiveness. But it takes time before you can get people to where they want to do test design. Instead I start people out with coverage. Have them do whatever testing they currently do, and give them a coverage tool to measure the results of their tests. They will see a lot of unexercised code, so they will write tests to exercise that code. They will not write the best possible tests to

exercise that code. But it is a start. It gives them two things: they get instant gratification because they know right away what their tests are doing. And they get some idea of where to stop, because they have gotten to a nice number. For instance, you could say, "Using the debugger, get 100% branch coverage." With manual testing using the debugger, this is usually not a hard thing to do. This means you have gotten 100% coverage, but with a set of tests that will never be run again.

Next you want to introduce test implementation tools and make these tests repeatable. When you do this, you want to make it possible for the developers to create repeatable tests with very little work. For example, to repeatably test a GUI based product, you would want to get a GUI testing tool. There are two styles of GUI testing. One is the capture/replay method and the other is programmatic, where you write your tests as if they were programs, with commands which cause mouse button presses, movement, etc. Of these two styles, the method of writing test programs has much better long term maintainability. But from the standpoint of getting people eased into creating tests, the capture/replay method provides quick satisfaction. The results from capture/replay will not be as good as they could be, but you will end up with a result, which is the starting point. Over the course of time, you can work on demonstrating the benefits of the programmatic method and get people changed over to it.

Once you have done the up-front work of getting developers started doing more testing, measuring their coverage and creating repeatable tests, other forces will make them want to improve their testing. They will be getting bug reports back from system test and customers and wondering how they could have caught them. They will want to know how they can get higher coverage faster. Now you can start talking about test design as a method to catch more bugs up front, and to create higher coverage with less effort. So you can start teaching test design techniques, tailored to their expectations. They will want something that they can go and apply rapidly, not a long drawn out class on everything they could do. As their test design becomes more sophisticated, you can add more advanced techniques.

Overall there are two key things to support this process. First, you need to be willing to spend money on tools. Software development is incredibly tool poor, compared to hardware engineering. Essential tools are test coverage tools and test automation tools, both test drivers and entire test suite managers. You will also need local gurus who know the tools well to help people use them. The second area is to provide training. I remember being given a copy of *The Art of Software Testing* and being told to "Go test!" It took me years to figure out how to really do testing. I have heard of a study from HP where they looked at two groups of engineers. One group had been trained in the use of a design tool, and others had been simply given the tool and the manuals. After a year, most of the people who had been trained were still using the tool. Very few of the people who had not been trained were still using the tool. And, of all the people who were continuing to use the tool, the trained people were using most of its capacity, while the untrained people were using a very limited subset of its features.

Q: I have heard you say that testing can improve time to market. What do you mean?

This is back to the discussion of the testing process, where you create your test requirements in advance. If you are creating test requirements early in the development cycle, you will also be finding a lot of defects early in the process. It is much cheaper and faster to find problems in the earlier phases. Fixing requirement or design problems after the code is written is expensive, creates lots of problems, and takes lots of time.

The other thing that can improve time to market is automating testing. For the first release, automating testing will actually slow things down because time will have to go into creating the infrastructure. But, once this is in place, releases 2, 3, 4, etc. will go much faster. There are places where the test cycles are very long because they have a whole bunch of people doing manual testing. If they took the time to automate their tests, they could greatly reduce this, because in the future they could use computers to run the tests instead of people.

Q: Where do you see software testing going?

There are really two types of testing. One is based on the product, testing of the code, basically the kind of testing we do today. I think this type of testing can be made quite straightforward, a fairly rote procedure. It should not be a big deal, just something you need to do, that you know how to do, that takes some time, but you get it done and then you move on to other things.

Then there is the other type of testing that I think is considerably more interesting. It is asking "Is this the right product? Does it do the right thing for the customer?" This is sometimes done at the system level. It is a form of testing the specification against the requirements. Unfortunately, the requirements are often implicit rather

than explicit. This is the direction that I want to see testing move.

Q: When you said that testing based on the product is a fairly rote procedure, do you mean it could be made fully automated?

There are certain aspects of creating test requirements that could be done automatically. You could run the source of your program through a tool that would look at structures and techniques used and select test requirements from test catalogs. If your design was captured in a stylized form, it could be run through a tool that would generate certain test requirements. There are some tools that do this sort of thing for both code and designs.

The problem is that these tools cannot create as good a test design as even a moderately trained, moderate interested human being. So you still need to go beyond these tools. Omissions, forgotten special cases, and mistaken assumptions still need to be discovered. An automated tool would do a poor job of finding these because it is basing its test requirements on the code or on the stylized design. So I see a need to continue to have a human being involved. As part of my structured approach to test design, I have a stage where I put the code aside for a few days. Then I come back to it and try to be inspired about what things I might have missed, what new test requirements are needed. The additional tests I find will be a minority of the total test requirements. But this minority is very important.

The creation of the actual tests is not yet very automatable. It is not very hard to generate a whole bunch of inputs, but creating their matching expected outputs, so you know if the program passed or failed, is generally hard. There are tools that will generate input test data based on a specification-like language, but they do not also generate the expected results. Therefore a tool that creates 10,000 tests is not very useful, if you then have to go through and manually figure out the expected outcome for each one.

Q: Where do you hope to spend your time in the future?

Testing is about understanding the sources of human errors and how they manifest themselves. How they manifest themselves in code is quite straightforward. We should get those out of the way. How they manifest themselves in defining a product is the really interesting thing to me. This is where I want to move my investigations of testing.

Brian's book is:

The Craft of Software Testing
ISBN 0-13-177411-5
553 pages, $47
Prentice Hall, 1995
1(800)947-7700

The Software QA Quarterly
Ridgetop Publishing, Ltd.
P.O. Box 379, Silverton, OR 97381-0379
Phone/FAX 503-829-6806

E-mail: SQA_Quarterly@ridgetop.com

Return to Marick's Corner | Return to the ST Labs Home Page