Martin Fowler
fowler@acm.org

Testing Methods: The Ugly Duckling

Ever been to the Design Techniques Annual Ball? The hot crowd is the UML group: Class Diagrams, Sequence Diagrams, Activity Diagrams, all wearing sharp and expensive casual business outfits. ER Diagrams go in suits, look a little on the gray side, but are still very popular. (They claim they are just as good as the object techniques, the object techniques are just better dressed.)

CRC cards turn up in sandals, patterns are always trying to turn you into one of them, and formal methods snub everyone else for the slightest crease or stain. In a corner, ignored by all, wearing stained and color-clashing overalls, are the testing methods. Nobody wants to talk to them.

All the design methods get to ride in fancy CASE tools; testing methods are lucky if anyone walks them home. But I've noticed that the parties that end in tears are usually the ones that the testing methods weren't invited to!

Alright, I'm being whimsical, it's late, my hotel room feels boring, and I've got a column to write. But this column is quite serious. You can use design methods until you are blue in the face, but how you do your testing will probably have a bigger effect. Certainly for many of my clients, getting them to use effective testing techniques is more bang for the buck than teaching them UML.

Self-Testing Code

A key technique I use is self-testing code. The principles behind this are that testing is not something to start after you have finished coding, and that test code is as important a deliverable as production code—and should be given the same emphasis. (I should say that this self-testing code is unit test code, written by the developers *themselves*. You should have separately written system test code as well.)

I'm a great fan of incremental development. Every time I try to add new features to software, I stop and ask myself: "What is the smallest piece of new function I can add?" I then focus on adding that feature alone, and I do not move to a new feature until the feature is complete—including the self-testing code for that feature. Development then proceeds by small steps, with the testing code and the production code proceeding in tandem.

It's not unusual to write the test code before the production code. I often find this useful, because it helps me to focus on exactly what this incremental step involves. Essentially, I write the test, and then get the code to pass the test.

Another advantage of writing the test first is that the tests help you concentrate on the interface for the new feature rather than the implementation. You are asking yourself "How will a client use this new feature?" Getting the test to work then gives me closure on the feature. With the test working, I know I did what I set out to do.

But there is a further step, essential to the self-testing technique. Once your new feature test works, you add it to your unit testing code base. There are various ways of doing this. These days I keep a separate set of tester classes in my software. As each feature gets added to the production code, a new test gets added to the test code.

I've been working with this on a major financial system development project. It has over 2000 unit tests, and the unit test code is a quarter the bulk of the production code. Having such a battery of unit tests is valuable, but only if you use it, and to use it you must be able to run it easily.

An essential factor here is that the tests must give a simple indication of whether they pass or fail. A test that produces a number that you then have to check manually against some piece of paper is a stupid test. You should be able to run all 2000 tests and either see "Ok," or "Here is a list of failures."

At the project I just mentioned, they use Kent Beck's Smalltalk testing framework. It has a test panel that goes green if all is well, or red if there are any failures. That way you can happily just hit the button to run the tests and know immediately if everything is fine (Kent and Erich Gamma have ported the framework to Java—you can get the URL from my homepage, http://ourworld.compuserve.com/homepages/martin_fowler).

These self-tests are now a golden asset for future development. Since they are easy to run and interpret, you can run them regularly, even if they take a while. Every time you go off for a meeting or to lunch, run all

Martin Fowler is an independent consultant based in Boston, Massachusetts.

the tests. Or set up a regular job that runs every test at midnight and mails you the result.

If in the future you do something that breaks some other part of the system, the tests will quickly tell you. You'll know it must be something you just did that caused the failure, and that knowledge alone is usually enough to cut out hours of bug chasing. In particular, you should try to run the full suite of tests whenever you integrate.

It amazes me when a programmer checks in code after days of work and just assumes that it will work with whatever anyone else has checked in. If you can, run all tests every time anyone integrates any code. Furthermore, get people to integrate frequently. With incremental development you can easily integrate daily.

Such an approach—what Ron Jeffries calls continuous integration and relentless testing—makes bugs show up early, when they are easier to find and fix. This does wonders in reducing integration time.

When I am developing, I usually will run tests every time I compile. It's clearly not practical to run half-an-hour's worth of tests on every compile, so when I'm doing some development I'll choose a subset, which will run in a few seconds, that focuses on the code I'm currently working on. Maybe I'll do a ten-minute test suite during a coffee break.

These unit tests are a great enabler. I'm a great fan of refactoring: making improvements to the internal structure of the code without changing the external behavior. Regular refactoring is essential to keep the design integrity of a software system. Refactoring without solid unit tests leads to long and discouraging bug chases.

Similarly the tests are essential for performance optimization. Again, without the tests it is hard to tell that a performance enhancement does not introduce a bug.

When I teach self-testing code to developers, a usual first reaction is that "Well this sounds reasonable, but I've got deadlines to make." The crucial realization is that self-testing code actually speeds up writing code. It does this because it makes debugging much shorter.

Early on in my programming life I discovered that I spent much more time and effort removing bugs than I did writing code. If you run tests regularly, bugs tend to show up earlier. If it's not long since your last write of a test, you know which bit of code contains the bug.

After writing self-testing code for a while, developers realize that they are spending less time debugging and thus developing faster. The tests enable them to refactor more easily, thus keeping the system design simpler and allowing them to develop faster.

Introducing Self-Testing Code

How do you introduce self-testing code into your organization? Find some people who are willing to give it a try and have them, well, give it a try. If they can reach the point where they see how self-testing code improves their productivity, they will be the best endorsers of the technique. You may find you need a mentor who practices the technique to pass on to the development staff.

Certainly if I were looking for a mentor, I would be very reluctant to hire one who didn't understand and emphasize testing. Make sure enough testing code gets written. To do this you might use a code coverage tool to see how much of the production code is being exercised.

Failing that, keep an eye on the ratio of test code to production code. Make sure that the tests are being run regularly. If it is practical to enforce running all unit tests before checking in code, do that. If not, make sure there's an automatic run of all unit tests daily when you do your daily build. (You do do a daily build, don't you?)

It's another day now, I'm on an airplane bound for a conference, and I'm not so tired any more. The introduction to this column looks a little cheesy. So I'll refrain from making analogies about smartening up testing techniques with a flashy new outfit. I'll just tell you now, if you don't have self-testing code, start now. It may be the most important thing you do this year! ❊