**GUIDELINE: DEVELOPER TESTING**

This guideline describes techniques for getting started with developer testing and characteristics of good developer tests.

**MAIN DESCRIPTION**

**Establish expectations**

Those who find developer testing rewarding do it. Those who view it as a chore find ways to avoid it. This is simply in the nature of most developers in most industries, and treating it as a shameful lack of discipline hasn't historically been successful. Therefore, as a developer you should expect testing to be rewarding and do what it takes to make it rewarding.

Ideal developer testing follows a very tight edit-test loop. You make a small change to the product, such as adding a new method to a class, then you immediately rerun your tests. If any test breaks, you know exactly what code is the cause. This easy, steady pace of development is the greatest reward of developer testing. A long debugging session should be exceptional.

Because it's not unusual for a change made in one class to break something in another, you should expect to rerun not just the changed class's tests, but many tests. Ideally, you rerun the complete test suite for your implementation element many times per hour. Every time you make a significant change, you rerun the suite, watch the results, and either proceed to the next change or fix the last change. Expect to spend some effort making that rapid feedback possible.

**Write and maintain tests**

**Automate your tests**

Running tests often is not practical if tests are manual. For some implementation elements, automated tests are easy. An example would be an in-memory database. It communicates to its clients through an API and has no other interface to the outside world. Tests for it would look something like this:

/\* Check that elements can be added at most once. \*/

// Setup

Database db = new Database();

db.add("key1", "value1");

// Test

boolean result = db.add("key1", "another value");

expect(result == false);

The tests are different from ordinary client code in only one way: instead of believing the results of API calls, they check. If the API makes client code easy to write, it makes test code easy to write. If the test code is *not* easy to write, you've received an early warning that the API could be improved. Test-first design is thus consistent with the iterative processes' focus on addressing important risks early.

The more tightly connected the element is to the outside world, however, the harder it will be to test. There are two common cases: graphical user interfaces and back-end components.

**Graphical user interfaces**

Suppose the database in the example above receives its data via a callback from a user-interface object. The callback is invoked when the user fills in some text fields and pushes a button. Testing this by manually filling in the fields and pushing the button isn't something you want to do many times an hour. You must arrange a way to deliver the input under programmatic control, typically by "pushing" the button in code.

Pushing the button causes some code in the implementation element to be executed. Most likely, that code changes the state of some user-interface objects. So you must also arrange a way to query those objects programmatically.

**Back-end components**

Suppose the element under test doesn't implement a database. Instead, it's a wrapper around a real, on-disk database. Testing against that real database might be difficult. It might be hard to install and configure. Licenses for it might be expensive. The database might slow down the tests enough that you're not inclined to run them often. In such cases, it's worthwhile to "stub out" the database with a simpler element that does just enough to support the tests.

Stubs are also useful when a component that your element talks to isn't ready yet. You don't want your testing to wait on someone else's code.

**Don't write your own tools**

Developer testing seems pretty straightforward. You set up some objects, make a call through an API, check the result, and announce a test failure if the results aren't as expected. It's also convenient to have some way to group tests so that they can be run individually or as complete suites. Tools that support those requirements are called *test frameworks*.

Developer testing **is** straightforward, and the requirements for test frameworks are not complicated. If, however, you yield to the temptation of writing your own test framework, you'll spend much more time tinkering with the framework than you probably expect. There are many test frameworks available, both commercial and open source, and there's no reason not to use one of those.

**Do create support code**

Test code tends to be repetitive. It's common to see sequences of code like this:

// null name not allowed

retval = o.createName("");

expect(retval == null);

// leading spaces not allowed

retval = o.createName(" l");

expect(retval == null);

// trailing spaces not allowed

retval = o.createName("name ");

expect(retval == null);

// first character may not be numeric

retval = o.createName("5allpha");

expect(retval == null);

This code is created by copying one check, pasting it, then editing it to make another check.

The danger here is twofold. If the interface changes, much editing will have to be done. (In more complicated cases, a simple global replacement won't suffice.) Also, if the code is at all complicated, the intent of the test can be lost amid all the text.

When you find yourself repeating yourself, seriously consider factoring out the repetition into support code. Even though the code above is a simple example, it's more readable and maintainable if written like this:

void expectNameRejected(MyClass o, String s) {

Object retval = o.createName(s);

expect(retval == null);

}

...

// null name not allowed

expectNameRejected(o, "");

// leading spaces not allowed.

expectNameRejected(o, " l");

// trailing spaces not allowed.

expectNameRejected(o, "name ");

// first character may not be numeric.

expectNameRejected(o, "5alpha");

Developers writing tests often err on the side of too much copying-and-pasting. If you suspect yourself of that tendency, it's useful to consciously err in the other direction. Resolve that you will strip your code of all duplicate text.

**Keep the tests understandable**

You should expect that you, or someone else, will have to modify the tests later. A typical situation is that a later iteration calls for a change to the element's behavior. As a simple example, suppose the element once declared a square root method like this:

double sqrt(double x);

In that version, a negative argument caused the function "sqrt" to return NaN ("not a number" from the IEEE 754-1985 *Standard for Binary Floating-Point Arithmetic*). In the new iteration, the square root method will accept negative numbers and return a complex result:

Complex sqrt(double x);

Old tests for the function "sqrt" will have to change. That means understanding what they do, and updating them so that they work with the new "sqrt". When updating tests, you must take care not to destroy their bug-finding power. One way that sometimes happens is this:

void testSQRT () {

// Update these tests for Complex

// when I have time -- bem

/\*

double result = sqrt(0.0);

...

\*/

}

Other ways are more subtle: the tests are changed so that they actually run, but they no longer test what they were originally intended to test. The end result, over many iterations, can be a test suite that is too weak to catch many bugs. This is sometimes called "test suite decay". A decayed suite will be abandoned, because it's not worth the upkeep.

Test suite decay is less likely in the direct tests for the function "sqrt" than in indirect tests. There will be code that calls the function "sqrt". That code will have tests. When the function "sqrt" changes, some of those tests will fail. The person who changes the function "sqrt" will probably have to change those tests. Because he's less familiar with them, and because their relationship to the change is less clear, he's more likely to weaken them in the process of making them pass.

When you're creating support code for tests (as urged above), be careful: the support code should clarify, not obscure, the purpose of the tests that use it. A common complaint about object-oriented programs is that there's no one place where anything's done. If you look at any one method, all you discover is that it forwards its work somewhere else. Such a structure has advantages, but it makes it harder for new people to understand the code. Unless they make an effort, their changes are likely to be incorrect or to make the code even more complicated and fragile. The same is true of test code, except that later maintainers are even less likely to take due care. You must head off the problem by writing understandable tests.

**Match the test structure to the product structure**

Suppose someone has inherited your implementation element. They need to change a part of it. They may want to examine the old tests to help them in their new design. They want to update the old tests before writing the code (test-first design).

All those good intentions will go by the wayside if they can't find the appropriate tests. What they'll do is make the change, see what tests fail, then fix those. That will contribute to test suite decay.

For that reason, it's important that the test suite be well structured, and that the location of tests be predictable from the structure of the product. Most usually, developers arrange tests in a parallel hierarchy, with one test class per product class. So if someone is changing a class named "Log", they know the test class is "TestLog", and they know where the source file can be found.

**Let tests violate encapsulation**

You might limit your tests to interacting with your implementation element exactly as client code does, through the same interface that client code uses. However, this has disadvantages. Suppose you're testing a simple class that maintains a doubly linked list:

In particular, you're testing the "DoublyLinkedList.insertBefore(Object existing, Object newObject)" method. In one of your tests, you want to insert an element in the middle of the list, then check if it's been inserted successfully. The test uses the list above to create this updated list:

It checks the list correctness like this:

// the list is now one longer.

expect(list.size()==3);

// the new element is in the correct position

expect(list.get(1)==m);

// check that other elements are still there.

expect(list.get(0)==a);

expect(list.get(2)==z);

That seems sufficient, but it's not. Suppose the list implementation is incorrect and backward pointers are not set correctly. That is, suppose the updated list actually looks like this:

If the function "DoublyLinkedList.get(int index)" traverses the list from the beginning to the end (likely), the test would miss this failure. If the class provides "elementBefore" and "elementAfter" methods, checking for such failures is straightforward:

// Check that links were all updated

expect(list.elementAfter(a)==m);

expect(list.elementAfter(m)==z);

expect(list.elementBefore(z)==m); //this will fail

expect(list.elementBefore(m)==a);

But what if it doesn't provide those methods? You could devise more elaborate sequences of method calls that will fail if the suspected defect is present. For example, this would work:

// Check whether back-link from Z is correct.

list.insertBefore(z, x);

// If it was incorrectly not updated, X will have

// been inserted just after A.

expect(list.get(1)==m);

But such a test is more work to create and is likely to be significantly harder to maintain. (Unless you write good comments, it will not be at all clear why the test is doing what it's doing.) There are two solutions:

1. Add the "elementBefore" and "elementAfter" methods to the public interface. But that effectively exposes the implementation to everyone and makes future change more difficult.
2. Let the tests "look under the hood" and check pointers directly.

The latter is usually the best solution, even for a simple class like "DoublyLinkedList" and especially for the more complex classes that occur in your products.

Typically, tests are put in the same package as the class they test. They are given protected or friend access.

**Approaches for Test Setup**

To successfully run a test, the system must be in a known state. To do this you will need objects or components in memory, rows in the database, etc. that you will test against. The easiest approach is to hardcode the required data and the setup code within the test itself. The primary advantage is that all the information that you need about the test is in one place and that the test is potentially self-sufficient.

Another approach is to define an external data set which is loaded into memory or into the database at the beginning of the test run. There are several advantages to this approach:

* It decouples the test data from the test.
* More than one test can use the same data set.
* It is easy to modify and/or multiply the test data.

There are some disadvantages to this approach:

* Increased complexity for maintaining the external data
* Potential coupling between test cases. When they share a common test data bed it becomes very easy to write tests that depend on other tests running first, thereby coupling them together.

**Coding for Testability**

Add [code instrumentation](http://epf.eclipse.org/wikis/openup/core.tech.common.base/guidances/termdefinitions/code_instrumentation_3060875F.html) for testing and debugging. Pay special attention to the implementation of the observation/control points, such as critical functions or objects, as these aspects might need special support that has to be implemented in the application under test.

**Reviewing Tests**

If a test will be long-lived, ask a person with less inside knowledge of the implementation element to run it and check if there is enough support information. Review it with other people within the development team and other interested parties as needed.

**Characteristic Test Design Mistakes**

Each test exercises an implementation element and checks for correct results. The design of the test-the inputs it uses and how it checks for correctness-can be good at revealing defects, or it can inadvertently hide them. Here are some characteristic test design mistakes.

**Failure to specify expected results in advance**

Suppose you're testing an implementation element that converts XML into HTML. A temptation is to take some sample XML, run it through the conversion, then look at the results in a browser. If the screen looks right, you "bless" the HTML by saving it as the official expected results. Thereafter, a test compares the actual output of the conversion to the expected results.

This is a dangerous practice. Even sophisticated computer users are used to believing what the computer does. You are likely to overlook mistakes in the screen appearance. (Not to mention that browsers are quite tolerant of misformatted HTML.) By making that incorrect HTML the official expected results, you make sure that the test can never find the problem.

It's less dangerous to doubly-check by looking directly at the HTML, but it's still dangerous. Because the output is complicated, it will be easy to overlook errors. You'll find more defects if you write the expected output by hand first.

**Failure to check the background**

Tests usually check that what should have been changed has been, but their creators often forget to check that what should have been left alone has been left alone. For example, suppose a program is supposed to change the first 100 records in a file. It's a good idea to check that the 101st hasn't been changed.

In theory, you would check that nothing in the "background"-the entire file system, all of memory, everything reachable through the network-has been left alone. In practice, you have to choose carefully what you can afford to check. But it's important to make that choice.

**Failure to check persistence**

Just because the implementation element tells you a change has been made, that doesn't mean it has actually been committed to the database. You need to check the database via another route.

**Failure to add variety**

A test might be designed to check the effect of three fields in a database record, but many other fields need to be filled in to execute the test. Testers will often use the same values over and over again for these "irrelevant" fields. For example, they'll always use the name of their lover in a text field, or 999 in a numeric field.

The problem is that sometimes what shouldn't matter actually does. Every so often, there's a bug that depends on some obscure combination of unlikely inputs. If you always use the same inputs, you stand no chance of finding such bugs. If you persistently vary inputs, you might. Quite often, it costs almost nothing to use a number different than 999 or to use someone else's name. When varying the values used in tests costs almost nothing and it has some potential benefit, then vary. (Note: It's unwise to use names of old lovers instead of your current one if your current lover works with you.)

Here's another benefit. One plausible fault is for the program to use field *X* when it should have used field *Y*. If both fields contain "Dawn", the fault can't be detected.

**Failure to use realistic data**

It's common to use made-up data in tests. That data is often unrealistically simple. For example, customer names might be "Mickey", "Snoopy", and "Donald". Because that data is different from what real users enter - for example, it's characteristically shorter - it can miss defects real customers will see. For example, these one-word names wouldn't detect that the code doesn't handle names with spaces.

It's prudent to make a slight extra effort to use realistic data.

**Failure to notice that the code does nothing at all**

Suppose you initialize a database record to zero, run a calculation that should result in zero being stored in the record, then check that the record is zero. What has your test demonstrated? The calculation might not have taken place at all. Nothing might have been stored, and the test couldn't tell.

That example sounds unlikely. But this same mistake can crop up in subtler ways. For example, you might write a test for a complicated installer program. The test is intended to check that all temporary files are removed after a successful installation. But, because of all the installer options, in that test, one particular temporary file wasn't created. Sure enough, that's the one the program forgot to remove.

**Failure to notice that the code does the wrong thing**

Sometimes a program does the right thing for the wrong reasons. As a trivial example, consider this code:

if (a < b && c)

return 2 \* x;

else

return x \* x;

The logical expression is wrong, and you've written a test that causes it to evaluate incorrectly and take the wrong branch. Unfortunately, purely by coincidence, the variable X has the value 2 in that test. So the result of the wrong branch is accidentally correct - the same as the result the right branch would have given.

For each expected result, you should ask if there's a plausible way in which that result could be achieved for the wrong reason. While it's often impossible to know, sometimes it's not.

**Write the tests first**

Writing the tests after the code is a chore. The urge is to rush through it, to finish up and move on. Writing tests before the code makes testing part of a positive feedback loop. As you implement more code, you see more tests passing until finally all the tests pass and you're done. People who write tests first seem to be more successful, and it takes no more time. For more on putting tests first, see [Guideline: Test Driven Development](http://epf.eclipse.org/wikis/openup/practice.tech.test_driven_development.base/guidances/guidelines/test_driven_development_F581182D.html).