**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).

ORIENTACIÓN: PRUEBAS DE DESARROLLADOR  
Esta guía describe las técnicas para comenzar con las pruebas y las características de las buenas pruebas para desarrolladores desarrollador.  
  
Descripción principal  
Establecer expectativas  
Los que encuentran desarrollador prueba gratificante hacerlo. Los que lo ven como una tarea encontrar la manera de evitarlo. Esto es simplemente en la naturaleza de la mayoría de los desarrolladores en la mayoría de las industrias, y tratándolo como una vergonzosa falta de disciplina históricamente no ha tenido éxito. Por lo tanto, como un desarrollador usted debe esperar las pruebas para ser gratificante y hacer lo necesario para que sea gratificante.  
Pruebas desarrollador Ideal sigue un bucle edit-prueba muy apretada. Usted hace un pequeño cambio en el producto, como la adición de un nuevo método a una clase, a continuación, volver a ejecutar de inmediato las pruebas. Si los saltos de prueba, usted sabe exactamente lo que el código es la causa. Esto fácil ritmo, constante de desarrollo es la mayor recompensa de las pruebas desarrollador. Una sesión de depuración tiempo debe ser excepcional.  
Porque no es inusual para un cambio realizado en una clase a romper algo en otro, usted debe esperar para volver a ejecutar no sólo las pruebas de la clase cambiado, pero muchas pruebas. Lo ideal es volver a ejecutar el conjunto de pruebas completa para sus elementos aplicación muchas veces por hora. Cada vez que haga un cambio significativo, que vuelva a ejecutar la suite, ver los resultados, y, o bien continuar con el siguiente cambio o solucionar el último cambio. Prepárese para pasar un poco de esfuerzo por lo que la rápida retroalimentación posible.  
Escribir y mantener pruebas  
Automatice sus pruebas  
Ejecución de pruebas a menudo no es práctico si las pruebas son manuales. Para algunos elementos de implementación, pruebas automatizadas son fáciles. Un ejemplo podría ser una base de datos en memoria. Se comunica a sus clientes a través de una API y no tiene ninguna otra interfaz con el mundo exterior. Las pruebas de que se vería algo como esto:  
  
/ \* Comprobar que los elementos se pueden agregar como máximo una vez. \* /  
// Configurar  
Base de datos db = new Base de datos ();  
db.add ("key1", "valor1");  
// Prueba  
resultado booleano = db.add ("key1", "otro valor");  
esperar (resultado == false);  
  
Las pruebas son diferentes de código de cliente habitual de una sola manera: en lugar de creer los resultados de llamadas a la API, comprueban. Si la API hace que el código de cliente fácil de escribir, que hace el código de prueba fácil de escribir. Si el código de prueba no es fácil de escribir, que ha recibido una alerta temprana que la API podría mejorarse. Diseño de los ensayos primero es, pues, en consonancia con el enfoque de los procesos iterativos 'en abordar riesgos importantes principios.  
Cuanto más firmemente conectado el elemento es para el mundo exterior, sin embargo, más difícil será para probar. Hay dos casos comunes: las interfaces gráficas de usuario y componentes de back-end.  
Las interfaces gráficas de usuario  
Supongamos que la base de datos en el ejemplo anterior recibe sus datos a través de una devolución de llamada desde un objeto de interfaz de usuario. La devolución de llamada se invoca cuando el usuario rellena en algunos campos de texto y pulsa un botón. Prueba de esta llenando manualmente en los campos y pulsar el botón no es algo que quieres hacer muchas veces en una hora. Usted debe arreglar una manera de entregar la entrada bajo control programático, típicamente por "empujar" el botón en el código.  
Al pulsar el botón hace algo de código en el elemento de aplicación que se ejecute. Lo más probable es que el código cambia el estado de algunos de los objetos de interfaz de usuario. Así también debe organizar una forma de consultar esos objetos mediante programación.  
Componentes de back-end  
Supongamos que el elemento bajo prueba no implementa una base de datos. En cambio, es una envoltura alrededor de un verdadero, base de datos en disco. Prueba contra esa base de datos real podría ser difícil. Puede ser difícil de instalar y configurar. Las licencias para que puedan ser costosos. La base de datos podría retrasar las pruebas suficientes de que usted no está inclinado a correr a menudo. En tales casos, la pena "trozo a cabo" la base de datos con un elemento más simple que hace lo suficiente para soportar las pruebas.  
Stubs también son útiles cuando un componente que sus conversaciones elemento a aún no está listo. Usted no quiere que su prueba de esperar en el código de otra persona.  
No escriba sus propias herramientas  
Pruebas desarrollador parece bastante sencillo. Se configura algunos objetos, hacer una llamada a través de una API, comprobar el resultado, y anunciar una falla de la prueba si los resultados no son los esperados. También es conveniente tener alguna forma de pruebas de grupo a fin de que se pueden ejecutar de forma individual o como suites completas. Herramientas que apoyan esos requisitos se llaman marcos de prueba.  
Pruebas de desarrollador es sencillo, y los requisitos para los marcos de las pruebas no son complicados. Sin embargo, si cedes a la tentación de escribir su propio marco de prueba, usted pasará mucho más tiempo jugando con el marco de lo que probablemente esperas. Hay muchos marcos de prueba disponibles, tanto comerciales como de código abierto, y no hay razón para no usar uno de esos.  
No crear código de soporte  
Código de ensayo tiende a ser repetitivo. Es común ver secuencias de código como este:  
  
// Nombre nulo no permitido  
retval = o.createName ("");  
esperar (retval == null);  
// Espacios iniciales no permitidos  
retval = o.createName ("l");  
esperar (retval == null);  
// Espacios finales no se les permite  
retval = o.createName ("nombre");  
esperar (retval == null);  
// Primer carácter no puede ser numérico  
retval = o.createName ("5allpha");  
esperar (retval == null);  
  
Este código se crea copiando un cheque, pegar, luego editarlo para hacer otro cheque.  
El peligro aquí es doble. Si los cambios en la interfaz, gran parte de edición tendrá que ser hecho. (En los casos más complicados, un simple reemplazo global no será suficiente.) Además, si el código es para nada complicado, la intención de la prueba se puede perder en medio de todo el texto.  
Cuando usted se encuentra repetir siempre lo mismo, considerar seriamente la posibilidad de factoring a cabo la repetición en el código de apoyo. A pesar de que el código anterior es un ejemplo sencillo, es más legible y mantenible si está escrita así:  
  
anular expectNameRejected (MyClass o, Cadena s) {  
Objeto retval = o.createName (s);  
esperar (retval == null);  
}  
...  
// Nombre nulo no permitido  
expectNameRejected (o, "");  
// Espacios iniciales no permitidos.  
expectNameRejected (o, "l");  
// Espacios finales no se les permite.  
expectNameRejected (o "nombre");  
// Primer carácter no puede ser numérico.  
expectNameRejected (o, "5alpha");  
  
Pruebas de escritura Desarrolladores menudo errar por el lado de demasiado copiar y pegar. Si usted sospecha que usted mismo de esa tendencia, es útil para errar conscientemente en la otra dirección. Resolver que va a despojar de su código de todo el texto duplicado.  
Mantener las pruebas comprensible  
Usted debe esperar que usted, o alguien más, tendrá que modificar las pruebas más tarde. Una situación típica es que una iteración más tarde pide un cambio en el comportamiento del elemento. Como un simple ejemplo, supongamos que el elemento una vez declarado un método de la raíz cuadrada de esta manera:  
doble sqrt (double x);  
En esa versión, un argumento negativo causado la función "sqrt" para volver NaN ("no es un número" de la Norma IEEE 754-1985 para binaria aritmética de punto flotante). En la nueva iteración, el método de raíz cuadrada aceptará los números negativos y devolver un resultado complejo:  
Sqrt Complex (doble x);  
Pruebas viejos para la función "sqrt" tendrá que cambiar. Eso significa que la comprensión de lo que hacen, y actualizarlos para que funcionen con el nuevo "sqrt". Al actualizar las pruebas, se debe tener cuidado de no destruir su poder bug de investigación. Una de las formas que a veces sucede es lo siguiente:  
  
void testSQRT () {  
// Actualizar estas pruebas para Complex  
// Cuando tengo tiempo - bem  
/ \*  
doble resultado = sqrt (0.0);  
...  
\* /  
}  
  
Otras formas son más sutiles: las pruebas se cambian para que realmente funcionan, pero ya no prueban lo que estaban destinados originalmente para probar. El resultado final, a lo largo de muchas iteraciones, puede ser un conjunto de pruebas que es demasiado débil para atrapar muchos errores. Esto a veces se llama "test suite de decadencia". Una suite decaído será abandonado, porque no vale la pena el mantenimiento.  
Decaimiento conjunto de pruebas es menos probable en las pruebas directas para la función "sqrt" que en pruebas indirectas. Habrá código que llama a la función "sqrt". Ese código tendrá pruebas. Cuando la función sqrt "cambios", algunas de esas pruebas se producirá un error. La persona que cambia la función "sqrt" probablemente tendrá que cambiar esas pruebas. Porque es menos familiarizados con ellos, y debido a su relación con el cambio es menos claro, es más probable que se debilite en el proceso de hacer pasar.  
Al crear código de soporte para las pruebas (como instado anteriormente), tenga cuidado: el código de soporte debe aclarar, no oscura, el propósito de las pruebas que lo utilizan. Una queja común acerca de los programas orientados a objetos es que no hay un solo lugar donde ha hecho nada. Si nos fijamos en uno u otro método, todo lo que descubre es que envía sus trabajos en otro lugar. Esta estructura tiene ventajas, pero hace más difícil para la gente nueva a entender el código. A menos que hagan un esfuerzo, sus cambios es probable que sean incorrectos o para hacer el código aún más complicada y frágil. Lo mismo puede decirse de código de prueba, excepto que los mantenedores posteriores son aún menos probabilidades de tener el debido cuidado. Usted debe atajar el problema escribiendo pruebas comprensibles.  
Coincidir con la estructura de la prueba a la estructura del producto  
Supongamos que alguien ha heredado de su elemento de aplicación. Tienen que cambiar una parte de ella. Es posible que quieran examinar las antiguas pruebas para ayudarles en su nuevo diseño. Quieren actualizar las viejas pruebas antes de escribir el código (diseño de la prueba primero).  
Todas esas buenas intenciones se irán por el camino si no pueden encontrar las pruebas adecuadas. Lo que van a hacer es hacer el cambio, ver qué pruebas fallan, entonces fijar esos. Eso contribuirá a prueba la decadencia suite.  
Por esa razón, es importante que el conjunto de pruebas sea bien estructurado, y que la ubicación de las pruebas sea predecible a partir de la estructura del producto. Más generalmente, los desarrolladores organizar pruebas en una jerarquía paralela, con una clase de prueba por cada clase de productos. Así que si alguien está cambiando una clase llamada "Log", saben la clase de prueba es "TestLog", y saben que el archivo de origen se puede encontrar.  
Deje pruebas violan encapsulación  
Usted puede limitar sus pruebas para interactuar con su elemento de aplicación tal y como código de cliente hace, a través de la misma interfaz que el código de cliente utiliza. Sin embargo, esto tiene desventajas. Supongamos que usted está probando una clase simple que mantiene una lista doblemente enlazada:  
En particular, se está probando la "DoublyLinkedList.insertBefore (objeto existente, de objetos newObject)" método. En uno de sus ensayos, que desea insertar un elemento en la mitad de la lista, a continuación, compruebe si se ha insertado correctamente. La prueba utiliza la lista anterior para crear esta lista actualizada:  
Se comprueba la lista de corrección de esta manera:  
  
// La lista es ahora uno más largo.  
esperar (list.size () == 3);  
// El nuevo elemento está en la posición correcta  
esperar (list.get (1) == m);  
// Comprobar que otros elementos aún están allí.  
esperar (list.get (0) == a);  
esperar (list.get (2) == z);  
  
Eso parece suficiente, pero no lo es. Supongamos que la implementación de la lista es incorrecta y punteros hacia atrás no están correctamente ajustadas. Es decir, supongamos la lista actualizada en realidad se parece a esto:  
Si la función "DoublyLinkedList.get (int index)" atraviesa la lista desde el principio hasta el final (probable), la prueba se perdería este fracaso. Si la clase proporciona métodos "elementAfter" "elementBefore" y, la comprobación de estos fallos es sencillo:  
  
// Comprobar que los enlaces fueron actualizados  
esperar (list.elementAfter (a) == m);  
esperar (list.elementAfter (m) == z);  
esperar (list.elementBefore (z) == m); // Esto fallará  
esperar (list.elementBefore (m) == a);  
  
Pero ¿y si no proporciona esos métodos? Usted podría diseñar secuencias más elaboradas de llamadas a métodos que fallan si el supuesto defecto está presente. Por ejemplo, esto funcionaría:  
  
// Comprobar si copia de enlace de la Z es la correcta.  
list.insertBefore (z, x);  
// Si de forma incorrecta no se ha actualizado, X tendrá  
// Ha insertado justo después de A.  
esperar (list.get (1) == m);  
  
Pero esta prueba es más trabajo para crear y es probable que sea mucho más difícil de mantener. (A menos que usted escribe buenos comentarios, no será del todo claro por qué la prueba está haciendo lo que está haciendo.) Hay dos soluciones:  
1. Añadir la métodos "elementAfter" "elementBefore" y para la interfaz pública. Pero que expone eficazmente la aplicación a todo el mundo y hace que el cambio futuro más difícil.  
2. Deje que las pruebas "se ven bajo el capó" y comprobar directamente punteros.  
Este último es generalmente la mejor solución, incluso para una clase simple como "DoublyLinkedList" y sobre todo para las clases más complejas que se producen en sus productos.  
Por lo general, las pruebas se ponen en el mismo paquete que la clase que ponen a prueba. Se les da protegida o acceso amigo.  
Enfoques para la Configuración de la prueba  
Para ejecutar con éxito una prueba, el sistema debe estar en un estado conocido. Para ello necesitará objetos o componentes en la memoria, las filas de la base de datos, etc., que pondrá a prueba en contra. El método más sencillo consiste en codificar los datos requeridos y el código de configuración dentro de la prueba en sí. La principal ventaja es que toda la información que necesita acerca de la prueba está en un solo lugar y que la prueba es potencialmente autosuficiente.  
Otro enfoque es definir un conjunto de datos externo que se carga en la memoria o en la base de datos al inicio de la prueba de funcionamiento. Hay varias ventajas de este enfoque:  
• Se desacopla los datos de prueba de la prueba.  
• Más de una prueba puede utilizar el mismo conjunto de datos.  
• Es fácil de modificar y / o multiplicar los datos de prueba.  
Hay algunas desventajas de este enfoque:  
• El aumento de la complejidad del mantenimiento de los datos externos  
• Acoplamiento de potencial entre los casos de prueba. Cuando comparten una cama de datos de prueba común se hace muy fácil escribir pruebas que dependen de otras pruebas que se ejecutan primero, acoplando con ello juntos.  
Codificación para Testabilidad  
Añadir instrumentación de código para la prueba y la depuración. Preste especial atención a la aplicación de los puntos de observación / control, tales como funciones u objetos críticos, ya que estos aspectos podrían necesitar un apoyo especial que tiene que ser implementado en la aplicación bajo prueba.  
Revisión de exámenes  
Si una prueba será de larga duración, pida a una persona con menos conocimiento interior del elemento de aplicación para ejecutarlo y comprobar si hay información de apoyo suficiente. Revise con otras personas dentro del equipo de desarrollo y otras partes interesadas, según sea necesario.  
Errores de diseño característicos de prueba  
Cada prueba ejerce un elemento implementación y cheques por resultados correctos. El diseño de la prueba de los insumos que utiliza y cómo se comprueba la corrección-puede ser bueno en defectos que revelan, o puede ocultarlos inadvertidamente. Aquí hay algunos errores de diseño característicos de prueba.  
Si no se especifica resultados esperados de antelación  
Supongamos que usted está probando un elemento aplicación que convierte XML en HTML. Una tentación es tomar algún XML de ejemplo, ejecutar a través de la conversión, y luego mirar los resultados en un navegador. Si la pantalla se ve bien, que "bendiga" el código HTML guardándolo como el funcionario espera resultados. A partir de entonces, una prueba compara la salida real de la conversión a los resultados esperados.  
Esta es una práctica peligrosa. Incluso los usuarios de computadoras sofisticadas se utilizan para creer lo que hace el equipo. Es probable que pasar por alto errores en el aspecto de la pantalla. (Por no hablar de que los navegadores son bastante tolerantes con HTML misformatted.) Al hacer que el HTML incorrecto el funcionario espera que los resultados, se asegura de que la prueba no puede encontrar el problema.  
Es menos peligroso doblemente comprobar mirando directamente en el HTML, pero sigue siendo peligroso. Debido a que la salida es complicada, será fácil pasar por alto errores. Encontrarás más defectos si se escribe el resultado esperado con la mano primero.  
Si no se revisa el fondo  
Las pruebas suelen comprobar que lo que debería haber sido cambiado ha sido, pero sus creadores menudo se olvide de comprobar que lo que debería haber sido dejado solo se ha quedado solo. Por ejemplo, supongamos que un programa se supone que debe cambiar los primeros 100 registros en un archivo. Es una buena idea para comprobar que la 101 no se ha cambiado.  
En teoría, debería comprobar que nada en el "fondo", el sistema de archivos completo, toda la memoria, todo lo que puede llegar a través de la red se ha quedado solo. En la práctica, hay que elegir cuidadosamente lo que puede pagar para ver. Pero es importante hacer esa elección.  
Si no se revisa la persistencia  
El hecho de que el elemento de aplicación le indica un cambio se ha hecho, eso no quiere decir que en realidad ha estado comprometida con la base de datos. Es necesario comprobar la base de datos a través de otra ruta.  
La falta de añadir variedad  
Una prueba podría ser diseñado para comprobar el efecto de tres campos de un registro de base de datos, pero muchos otros campos tiene que ser rellenado para ejecutar la prueba. Probadores utilizarán a menudo los mismos valores una y otra vez para estos campos "irrelevantes". Por ejemplo, siempre vamos a usar el nombre de su amante en un campo de texto, o 999 en un campo numérico.  
El problema es que a veces lo que no debería importar en realidad hace. De vez en cuando, hay un error que depende de una combinación de insumos oscura improbables. Si siempre utiliza los mismos insumos, se pone de pie ninguna posibilidad de encontrar este tipo de errores. Si persistentemente varías entradas, te pueden. Muy a menudo, no cuesta casi nada para utilizar un número diferente de 999 o para utilizar el nombre de otra persona. Cuando la variación de los valores utilizados en las pruebas cuesta casi nada y tiene un beneficio potencial, entonces variar. (Nota: Es aconsejable utilizar nombres de antiguos amantes en lugar de su actual si su actual amante trabaja con usted.)  
Aquí hay otro beneficio. Una falla plausible es que el programa utilice campo X cuando debería tener Y. campo utilizado Si ambos campos contienen "Dawn", la culpa no puede ser detectado.  
La no utilización de los datos realista  
Es común el uso de datos confeccionados en las pruebas. Que los datos es a menudo poco realista simple. Por ejemplo, los nombres de los clientes podrían ser "Mickey", "Snoopy", y "Donald". Debido a que los datos es diferente de lo que los usuarios reales entran - por ejemplo, que es característicamente más corto - puede faltar defectos clientes reales verán. Por ejemplo, estos nombres de una sola palabra no detectar que el código no maneja nombres con espacios.  
Es prudente hacer un esfuerzo leve extra para usar datos realistas.  
El no se dio cuenta de que el código no hace nada en absoluto  
Supongamos que inicializar un registro de base de datos a cero, ejecute un cálculo que debe dar lugar a cero que se almacena en el registro, a continuación, comprobar que el registro es cero. Lo que ha demostrado su prueba? El cálculo no podría haber tenido lugar en absoluto. Nada podría haber sido almacenado, y la prueba no podía decir.  
Suena poco probable que el ejemplo. Pero este mismo error puede surgir en formas más sutiles. Por ejemplo, podría escribir una prueba para un programa de instalación complicada. El ensayo tiene por objeto verificar que todos los archivos temporales se eliminan después de una instalación exitosa. Pero, debido a todas las opciones del instalador, en esa prueba, no fue creada de un archivo temporal particular. Efectivamente, esa es la que el programa se olvidó de quitar.  
El no se dio cuenta de que el código hace las cosas mal  
A veces un programa hace lo correcto por las razones equivocadas. Como un ejemplo trivial, considere este código:  
  
si (a <b && c)  
volver 2 \* x;  
más  
return x \* x;  
  
La expresión lógica que está mal, y que ha escrito una prueba que hace que se evalúe de forma incorrecta y toma la rama equivocada. Desafortunadamente, por pura coincidencia, la variable X tiene el valor 2 en esa prueba. Así, el resultado de la rama equivocada es accidentalmente correcta - el mismo que el resultado de la rama derecha habría dado.  
Para cada resultado esperado, usted debe preguntar si hay una forma plausible en la que ese resultado podría lograrse por la razón equivocada. Si bien a menudo es imposible saber, a veces no lo es.  
Escribir las pruebas primero  
Escribiendo las pruebas después de que el código es una tarea. El impulso es correr a través de él, para terminar y seguir adelante. Escribir pruebas antes de que el código hace pruebas de parte de un bucle de retroalimentación positiva. Como se implementa más código, verá más pruebas que pasa hasta que finalmente todas las pruebas pasan y ya está. Las personas que escriben pruebas primero parecen ser más exitosa, y se necesita más tiempo. Para más información sobre las pruebas de poner primero, véase la directriz: Test Driven Development.