JUnit está diseñado alrededor de dos patrones de diseño principales: el patrón *Command* y el patrón *Composite*.

Un TestCase es un objeto *Command*. Cualquier clase que contenga métodos de testeo debería extender la clase TestCase. Un TestCase puede definir cualquier número de métodos públicos testXXX(). Cuando quieres comprobar el resultado esperado y el real, invocas una variante del método assert().

Las subclases de TestCase que contienen varios métodos testXXX() pueden utilizar los métodos setUp() y tearDown()para inicializar y liberar cualquier objeto común que se vaya a testear, conocido como la "instalación" (material) del test. Cada método de tests se ejecuta sobre su propia instalación, llamando a setUp() antes y a tearDown() después de cada método para asegurarse de que no hay efectos colaterales entre ejecuciones de tests.

Los ejemplares de TestCase pueden unirse en árboles de TestSuite que invocan automáticamente todos los métodos testXXX()definidos en cada ejemplar de TestCase. Un TestSuite es una composición de otros tests, bien ejemplares de TestCase u otros ejemplares de TestSuite. El comportamiento compuesto exhibido por TestSuite te permite ensamblar suites de tests de suites de tests, de una profundidad arbitraria, y ejecutar todos los tests automáticamente para obtener un simple estado de pasado o fallado.

**Paso 1: Escribir un Test**

Primero, escribiremos un test para probar un único componente de software. Nos enfocaremos en escribir test que comprueben el comportamiento que tiene el mayor potencial de rotura, así maximizaremos los beneficios de nuestra inversión en testeo.

Para escribir un test, sigue estos pasos:

1. Define una subclase de TestCase.
2. Sobreescribe el método setUp() para inicializar el objeto(s) a probar.
3. Sobreescribe el método tearDown() para liberar el objeto(s) a probar.
4. Define uno o más métodos testXXX() públicos que prueben el objeto(s) y aserten los resultados esperados.
5. Define un método factoría suite() estático que cree un TestSuite que contenga todos los métodos testXXX() del TestCase.
6. Opcionalmente, define un método main() que ejecute el TestCase en modo por lotes.

Abajo puedes ver un test de ejemplo:

import junit.framework.Test;

import junit.framework.TestCase;

import junit.framework.TestSuite;

public class ShoppingCartTest extends TestCase {

private ShoppingCart \_bookCart;

private Product \_defaultBook;

/\*\*

\* Constructs a ShoppingCartTest with the specified name.

\* @param name Test case name.

\*/

public ShoppingCartTest(String name) {

super(name);

}

/\*\*

\* Sets up the test fixture.

\* Called before every test case method.

\*/

protected void setUp() {

\_bookCart = new ShoppingCart();

\_defaultBook = new Product("Extreme Programming", 23.95);

\_bookCart.addItem(\_defaultBook);

}

/\*\*

\* Tears down the test fixture.

\* Called after every test case method.

\*/

protected void tearDown() {

\_bookCart = null;

}

/\*\*

\* Tests adding a product to the cart.

\*/

public void testProductAdd() {

Product newBook = new Product("Refactoring", 53.95);

\_bookCart.addItem(newBook);

double expectedBalance = \_defaultBook.getPrice() + newBook.getPrice();

assertEquals(expectedBalance, \_bookCart.getBalance(), 0.0);

assertEquals(2, \_bookCart.getItemCount());

}

/\*\*

\* Tests the emptying of the cart.

\*/

public void testEmpty() {

\_bookCart.empty();

assertTrue(\_bookCart.isEmpty());

}

/\*\*

\* Tests removing a product from the cart.

\* @throws ProductNotFoundException If the product was not in the cart.

\*/

public void testProductRemove() throws ProductNotFoundException {

\_bookCart.removeItem(\_defaultBook);

assertEquals(0, \_bookCart.getItemCount());

assertEquals(0.0, \_bookCart.getBalance(), 0.0);

}

/\*\*

\* Tests removing an unknown product from the cart.

\* This test is successful if the

\* ProductNotFoundException is raised.

\*/

public void testProductNotFound() {

try {

Product book = new Product("Ender's Game", 4.95);

\_bookCart.removeItem(book);

fail("Should raise a ProductNotFoundException");

} catch(ProductNotFoundException success) {

// successful test

}

}

/\*\*

\* Assembles and returns a test suite for

\* all the test methods of this test case.

\* @return A non-null test suite.

\*/

public static Test suite() {

//

// Reflection is used here to add all

// the testXXX() methods to the suite.

//

TestSuite suite = new TestSuite(ShoppingCartTest.class);

//

// Alternatively, but prone to error when adding more

// test case methods...

//

// TestSuite suite = new TestSuite();

// suite.addTest(new ShoppingCartTest("testEmpty"));

// suite.addTest(new ShoppingCartTest("testProductAdd"));

// suite.addTest(new ShoppingCartTest("testProductRemove"));

// suite.addTest(new ShoppingCartTest("testProductNotFound"));

//

return suite;

}

/\*\*

\* Runs the test case.

\*/

public static void main(String args[]) {

junit.textui.TestRunner.run(suite());

}

}

**Paso 2: Ecribir la Clase a Testear**

Ahora debes escribir las clases necesarias para pasar el test construido en el paso anterior:

Primero necesitas la clase principal: ShoopingCart.java:

import java.util.\*;

/\*\*

\* An example shopping cart.

\* This class should not be mistaken for a production-quality shopping cart. It's

\* merely provided as an example class under test as described in the JUnitPrimer.

\*

\* @author <a href="mailto:mike@clarkware.com">Mike Clark</a>

\* @author <a href="http://www.clarkware.com">Clarkware Consulting, Inc.</a>

\*/

public class ShoppingCart {

private ArrayList \_items;

/\*\*

\* Constructs a ShoppingCart instance.

\*/

public ShoppingCart() {

\_items = new ArrayList();

}

/\*\*

\* Returns the balance.

\* @return Balance.

\*/

public double getBalance() {

Iterator i = \_items.iterator();

double balance = 0.00;

while (i.hasNext()) {

Product p = (Product)i.next();

balance = balance + p.getPrice();

}

return balance;

}

/\*\*

\* Adds the specified product.

\* @param p Product.

\*/

public void addItem(Product p) {

\_items.add(p);

}

/\*\*

\* Removes the specified product.

\* @param p Product.

\* @throws ProductNotFoundException If the product does not exist.

\*/

public void removeItem(Product p) throws ProductNotFoundException {

if (!\_items.remove(p)) {

throw new ProductNotFoundException();

}

}

/\*\*

\* Returns the number of items in the cart.

\* @return Item count.

\*/

public int getItemCount() {

return \_items.size();

}

/\*\*

\* Empties the cart.

\*/

public void empty() {

\_items = new ArrayList();

}

/\*\*

\* Indicates whether the cart is empty.

\* @return true if the cart is empty;

\* false otherwise.

\*/

public boolean isEmpty() {

return (\_items.size() == 0);

}

}

También necesitarás la clase Product.java:

/\*\*

\* An example product for use in the example shopping cart.

\* @author <a href="mailto:mike@clarkware.com">Mike Clark</a>

\* @author <a href="http://www.clarkware.com">Clarkware Consulting, Inc.</a>

\*/

public class Product {

private String \_title;

private double \_price;

/\*\*

\* Constructs a <codigoenlinea>Product</codigoenlinea>.

\* @param title Product title.

\* @param price Product price.

\*/

public Product(String title, double price) {

\_title = title;

\_price = price;

}

/\*\*

\* Returns the product title.

\* @return Title.

\*/

public String getTitle() {

return \_title;

}

/\*\*

\* Returns the product price.

\* @return Price.

\*/

public double getPrice() {

return \_price;

}

/\*\*

\* Tests product equality.

\* @return true if the products

\* are equal.

\*/

public boolean equals(Object o) {

if (o instanceof Product) {

Product p = (Product)o;

return p.getTitle().equals(\_title);

}

return false;

}

}

Y la clase que generará la excepción ProductNotFoundException:

/\*\*

\* Exception thrown when a product is not found in a shopping cart.

\* @author <a href="mailto:mike@clarkware.com">Mike Clark</a>

\* @author <a href="http://www.clarkware.com">Clarkware Consulting, Inc.</a>

\*/

public class ProductNotFoundException extends Exception {

/\*\*

\* Constructs a <codigoenlinea>ProductNotFoundException</codigoenlinea>.

\*/

public ProductNotFoundException() {

super();

}

}

**Paso 3: Escribir una Suite de Tests**

Luego, escribiremos una suite de tests que incluya varios tests. La suite nos permitirá ejecutar todos los tests como si fueran uno sólo.

1. Define una subclase de TestCase.
2. Define una método factoría suite() estático que cree un TestSuite que contenga todos los métodos testXXX() del TestCase.
3. Opcionalmente, define un método main() que ejecute el TestCase en modo por lotes.

Abajo puedes ver un ejemplo de la suite de tests:

import junit.framework.Test;

import junit.framework.TestSuite;

public class EcommerceTestSuite {

/\*\*

\* Assembles and returns a test suite

\* containing all known tests.

\*

\* New tests should be added here!

\*

\* @return A non-null test suite.

\*/

public static Test suite() {

TestSuite suite = new TestSuite();

//

// The ShoppingCartTest we created above.

//

suite.addTest(ShoppingCartTest.suite());

//

// Another example test suite of tests.

//

suite.addTest(CreditCardTestSuite.suite());

return suite;

}

/\*\*

\* Runs the test suite.

\*/

public static void main(String args[]) {

junit.textui.TestRunner.run(suite());

}

}

**Paso 4: Ejecutar los Tests**

Ahora que ya hemos escrito una suite de tests que contiene una colección de tests y otros suites, podemos ejecutar toda la suite o cualquiera de sus tests individualmente. Ejecutando un TestSuite se ejecutarán automáticamente todos sus ejemplares de TestCasey de TestSuite subordinados. Ejecutando un TestCase invocará automáticamente a sus métodos testXXX() públicos.

JUnit proporciona dos interfaces de usuario uno de texto y otro gráfico. Ambos interfaces de usuario indican cuántos tests se ejecturaron, cuantos errores o fallos, y un único estado de terminación. Deberías poder ejecutar tus tests y ver de un vistado su estado, al igual que lo haces con tu compilador.

El interface de modo texto (junit.textui.TestRunner) muestra "OK" si se pasaron todos los tests y un mensaje de fallo si cualquier de los tests falla.

El interface de modo gráfico (junit.swingui.TestRunner) mestra un ventana Swing con una barra de progreso en verde si se pasaron todos los tests o una barra de progreso roja si falló alguno de los tests.

En general, las clases TestSuite y TestCase deberían definir un método main() que emplee el interface de usuario adecuado. Los tests que hemos definido hasta ahora han definido un método main() que emplea el interface de usuario en modo texto.

Para ejecutar nuestros tests según lo hemos definido en el método main(), utilizamos:

java ShoppingCartTest

Alternativametne, el test se puede ejecutar con el interface de usuario de modo texto utilizando:

java junit.textui.TestRunner ShoppingCartTest

o con el GUI de Swing utilizando:

java junit.swingui.TestRunner ShoppingCartTest

La suite EcommerceTestSuite se puede ejecutar de forma similar.

**Paso 5: Organizar los Tests**

El último paso es decidir donde residirán nuestros tests dentro de nuestro entorno de desarrollo.

Aquí tienes la forma recomendada para organizar tus tests:

1. Crea los tests en el mismo paquete que el código a testear. Por ejemplo, el paquete com.mydotcom.ecommerce contendría todas las clases a nivel de la aplicación y todos los tests para esos componentes.
2. Para evitar la mezcla de código de la aplicación y de testeo en tus directorios de código fuente, crea una estructura de directorio paralela alineada con la estructura de paquete que contenga el código de los tests.
3. Por cada paquete Java de tu aplicación, define una clase TestSuite que contenga todos los tests para validar el código de ese paquete.
4. Define clases TestSuite similares que creen suites de tests de alto y bajonivel en los otros paquetes (y sub-paquetes) de la aplicación.
5. Asegurate de que tu proceso de compilación incluye todos los tests. Esto ayuda a asegurar de tus tests siempre están actualizados con el último código y los mantiene frescos.

Mediante la creación de un TestSuite en cada paquete Java, a los distintos niveles de empaquetamiento, podrás ejecutar un TestSuite con cualquier nivel de abstracción. Por ejemplo, puedes definir un com.mydotcom.AllTests que ejecute todos los tests del sistema y un com.mydotcom.ecommerce.EcommerceTestSuite que solo ejecute aquellos que validan los componentes de comercio electrónico.

El árbol de herencia de tests se puede extender hasta una profundiad arbitraria. Dependiendo del nivel de abstracción que estemos desarrollando en el sistema, podrás ejecutar un tests apropiado. Sólo elige una capa del sistema y testeala!

Aquí tienes un ejemplo de árbol de tests:

AllTests (Top-level Test Suite)

SmokeTestSuite (Structural Integrity Tests)

EcommerceTestSuite

ShoppingCartTestCase

CreditCardTestSuite

AuthorizationTestCase

CaptureTestCase

VoidTestCase

UtilityTestSuite

MoneyTestCase

DatabaseTestSuite

ConnectionTestCase

TransactionTestCase

LoadTestSuite (Performance and Scalability Tests)

DatabaseTestSuite

ConnectionPoolTestCase

ThreadPoolTestCase