# Ingeniería del Software: Laboratorio

### Cuaderno de prácticas. P2 – Pruebas

## CONTENIDOS

El presente cuaderno de prácticas debe servir como guía al alumno para la asimilación y estudio de los siguientes contenidos:

* Prueba con EasyMock
* Pruebas Web con HTMLUnit

## RECOMENDACIÓN DE HERRAMIENTAS

* Netbeans(http://netbeans.org/)
* Eclipse (http://www.eclipse.org/).

## EJERCICIO TUTORIZADO – Integracion con EasyMock

### Enunciado

El objetivo de esta práctica es que el estudiante comprenda la utilidad de la prueba con objetos simulados (resguardos o conductores).

Concretamente, se utiliza el framework de prueba con objetos simulados EasyMock para el lenguaje Java.

**Primer paso: configuración de EasyMock**

Pasos a seguir:

1. **Descargar EasyMock y librer** **ías asociadas:**

La página Web para la descarga es: <http://easymock.org/>

(utilizaremos easymock 3.0).

cglib (2.2): <http://cglib.sourceforge.net/>

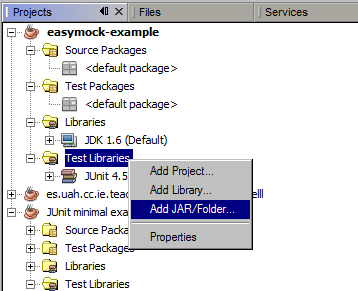
y Objenesis (1.2): <http://objenesis.googlecode.com/svn/docs/index.html>

1. **Descomprimir el fichero .zip de easymock en un directorio.**

La distribución incluye varios ficheros .jar, la documentación y la lincencia de uso.

1. **Añadir el fichero easymock-3.0.jar al classpath de nuestro compilador y máquina virtual Java.**
2. **Añadir cglib-nodep-2.2.jar y objenesis-1.2.jar al classpath de nuestro compilador y máquina virtual Java.**

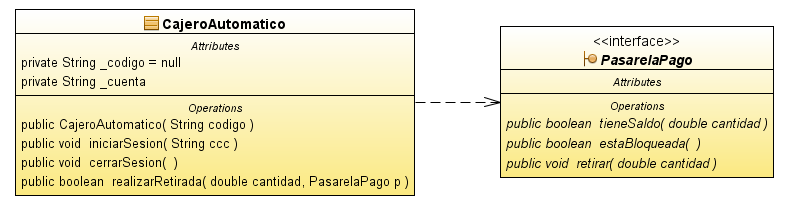
En el caso de trabajar con Netbeans esto es tan sencillo como incluir en la carpeta “Test packages” de nuestro proyecto una referencia al .jar de EasyMock



**Segundo paso: Análisis del comportamiento de la clase a sustituir**

Tomaremos como ejemplo una clase que queremos probar denominada CajeroAutomatico.

Esta clase en el diseño final utiliza otra clase denominada PasarelaPago con la que mantiene una comunicación durante las transacciones de los clientes con el cajero. No obstante, las dos clases han quedado asignadas a diferentes grupos de programadores, que trabajarán de forma independiente. Para facilitar las labores de desarrollo de forma simultánea e independiente, se ha codificado una interfaz Java para PasarelaPago que será sustituida por la clase cuando se haga la integración del trabajo de ambos equipos. El resultado es el que se muestra en la siguiente Figura.



En el diagrama se puede ver que el método realizarRetirada() usa una instancia del tipo PasarelaPago. Más concretamente, la lógica de negocio para la comunicación entre el cajero y la pasarela se ha definido de la siguiente forma:

1. Para cada operación que el cajero solicite a la pasarela, la precondición es que primero se consulte si la pasarela instancia de la pasarela está bloqueada, en cuyo caso se debe abortar la operación.
2. Antes de intentar operaciones de retirada de efectivo mediante la pasarela, se tiene que comprobar que el usuario tiene saldo suficiente.

Esto quiere decir que habrá al menos que comprobar las siguientes posibilidades:

1. Que se efectúe una retirada de efectivo con suficiente saldo.
2. Que se efectúe una retirada de efectivo sin suficiente saldo.
3. Que se efectúe una retirada de efectivo con suficiente saldo, pero habiendo quedado bloqueado previamente la pasarela.

Nótese que la comprobación de bloqueo ha de ser el primer paso en todos los casos.

**Tercer paso: Diseño de los objetos simulados**

Una vez hecho el análisis de la colaboración entre el cajero y la pasarela, tenemos que pasar a diseñar los casos de prueba.

En este caso el objeto simulado (*mock*) será una instancia de tipo PasarelaPago. Nótese que no tenemos ninguna clase que la implemente, luego será el framework EasyMock la que la “simule”. Asumimos que estamos utilizando JUnit version 4.x en el resto del ejemplo.

Lo primero que haremos es inicializar el objeto simulado y el objeto que vamos a probar, mediante el uso de un método inicializador de pruebas JUnit. El código sería el siguiente:

public class CajeroAutomaticoTest {

private CajeroAutomatico classUnderTest;

private PasarelaPago mock ;

@Before

public void setUp() {

classUnderTest = new CajeroAutomatico("1111111111");

classUnderTest.iniciarSesion("1234");

/\* (1) Crea un mock para la clase colaboradora\*/

mock = createMock(PasarelaPago.class);

}

...

La clase que se va a probar es una instancia normal. En este caso, ya se aprovecha la inicialización para comenzar la sesión. Para crear el mock se invoca a createMock. El método proviene de un *import* estático que tendremos en la clase de prueba:

import static org.easymock.EasyMock.\*;

Se le pasa como parámetro una expresión de tipo literal de clase (*class literal*) que se evalúa al objeto que representa a la clase nombrada – en nuestro caso no es una clase sino una interfaz.

A continuación tenemos que escribir los diferentes métodos que trabajarán con los objetos creados en el *fixture*. Habrá que cerrar la sesión al final de cada caso de prueba:

@After

public void tearDown() {

classUnderTest.cerrarSesion();

reset(mock);

}

El método reset() permite “borrar” las expectativas de un objeto simulado. En nuestro caso, no es estrictamente necesario invocarlo en el método de finalización de la prueba ya que se creará de nuevo en la inicialización.

Una vez tenemos los dos objetos, la escritura de un caso de prueba tiene siempre dos partes.

***Primera parte. Grabación de expectativas***

Lo primero que debe hacer el caso de prueba es “grabar” las expectativas del objeto simulado. Es decir, instruirle sobre qué llamadas debería esperar recibir y qué debe dar como valor de retorno para cada llamada.

@Test

public void testRealizarRetiradaBasico() throws Exception {

System.out.println("realizarRetirada");

/\* (2) En estado de "grabación", se le dice al objeto

Simulado las llamadas que debe esperar y cómo

responder a ellas.

\*/

expect(mock.estaBloqueada()).andReturn(false);

expect(mock.tieneSaldo(500)).andReturn(true);

mock.retirar(500);

/\* (3) Ahora, el objeto simulado comienza a esperar

las llamadas

\*/

replay(mock);

...

}

Cuando el mock se crea está en estado de “grabación” y pasa a estado de “escucha” o “ejecución” con la invocación del método replay().

***Segunda parte. Ejecución***

Una vez el objeto simulado está disponible para ser invocado como si se tratase de una instancia “real”, lo único que queda es hacer las invocaciones pertinentes para el caso de prueba, utilizando las comprobaciones mediante aserciones habituales de JUnit. Siguiendo el ejemplo tendríamos lo siguiente:

@Test

public void testRealizarRetiradaBasico() throws Exception {

...

/\* Pasa de grabacion a estado de escucha o ejecución

\*/

replay(mock);

boolean result = classUnderTest.realizarRetirada(500, mock);

assertTrue(result);

/\* Forzamos a que la ausencia de llamadas

sea un error también

\*/

verify(mock);

}

En este caso, la llamada tiene que resultar en un resultado true del método invocado.

Es importante resaltar que si se violase la aserción, lo que tendríamos es un error en la grabación del objeto *mock*. No obstante, cuando se sustituyese el objeto mock por un objeto de la clase realmente implementada, la violación de la aserción indicaría un error en PasarelaPago, por tanto, hay que codificar las aserciones pertinentes aunque mientas utilizamos el *mock* no están revelando fallos en el código del programa sino en el de prueba.

En caso de que el objeto mock no recibiese las llamadas para las que fue instruido, produciría un fallo de JUnit como el siguiente que se produce si cambiamos la cantidad a retirar:

realizarRetirada

Unexpected method call tieneSaldo(3.0):

tieneSaldo(500.0): expected: 1, actual: 0

retirar(500.0): expected: 1, actual: 0)

realizarRetiradaMasComplejo

Nótese que este caso revela también un problema en el caso de prueba. No obstante, si la implementación de realizarRetirada() viola la regla de comprobar el bloqueo, tendríamos lo siguiente:

realizarRetirada

Expectation failure on verify:

estaBloqueada(): expected: 1, actual: 0)

En este caso sí se ha detectado un fallo en el código que está siendo sometiendo a prueba. Esto se comprueba dado que incluimos la invocación a verify() para que se compruebe que se hacen todas las llamadas esperadas.

**Código completo del ejemplo EasyMock tutorizado**

/\*\*

\* Simula un cajero automático.

\*/

public class CajeroAutomatico {

private String \_codigo = null;

/\*\*

\* La cuenta corriente sobre la que se opera en una sesión.

\*/

private String \_cuenta;

public CajeroAutomatico(String codigo){

\_codigo = codigo;

}

public void iniciarSesion(String ccc){

assert(\_cuenta == null);

Cajero

}

public void cerrarSesion(){

assert(\_cuenta != null );

\_cuenta = null;

}

public boolean realizarRetirada(double cantidad, PasarelaPago p)

{

assert(\_cuenta != null );

if (p.estaBloqueada())

return false;

if (p.tieneSaldo(cantidad)){

p.retirar(cantidad);

return true;

}

return false;

}

}

/\*\*

\* Simula la interfaz de una entidad bancaria.

\*/

public interface PasarelaPago {

/\*\*

\* Comprueba si la cuenta tiene saldo suficiente para la retirada

\*/

public boolean tieneSaldo(double cantidad);

/\*\*

\* Una cuenta se bloquea si se ha intentado realizar una retirada de dinero

\* sin saldo suficiente de manera previa.

\*/

public boolean estaBloqueada();

/\*\*

\* Retira dinero de la cuenta.

\* Precondición: la cuenta debe tener saldo suficiente.

\* @param cantidad a retirar

\*/

public void retirar(double cantidad);

}

/\*

\* Prueba con EasyMock de la interacción entre CajeroAutomatico

\* y PasarelaPago

\*/

import static org.easymock.EasyMock.\*;

import org.junit.After;

import org.junit.AfterClass;

import org.junit.Before;

import org.junit.BeforeClass;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CajeroAutomaticoTest {

private CajeroAutomatico classUnderTest;

private PasarelaPago mock;

public CajeroAutomaticoTest() {

}

@Before

public void setUp() {

classUnderTest = new CajeroAutomatico("1111111111");

classUnderTest.iniciarSesion("1234");

/\* (1) Crea un mock para la clase colaboradora\*/

mock = createMock(PasarelaPago.class);

}

@After

public void tearDown() {

classUnderTest.cerrarSesion();

reset(mock);

}

/\*\*

\* Test of realizarRetirada method, of class CajeroAutomatico.

\*/

@Test

public void testRealizarRetiradaBasico() throws Exception {

System.out.println("realizarRetirada");

/\* (2) En estado de "grabación", se le dice al objeto simulado

las llamadas que debe esperar y cómo responder a ellas.

\*/

expect(mock.estaBloqueada()).andReturn(false);

expect(mock.tieneSaldo(500)).andReturn(true);

mock.retirar(500);

/\* (3) Ahora, el objeto simulado comienza a esperar las llamadas\*/

replay(mock);

boolean result = classUnderTest.realizarRetirada(500, mock);

assertTrue(result);

/\* Forzamos a que la ausencia de llamadas sea un error también\*/

verify(mock);

}

/\*\*

\*

\*/

@Test

public void testRealizarRetiradaMasComplejo() throws Exception {

System.out.println("realizarRetiradaMasComplejo");

/\* (2) En estado de "grabación", se le dice al objeto simulado

las llamadas que debe esperar y cómo responder a ellas.

\*/

expect(mock.estaBloqueada()).andReturn(false);

expect(mock.tieneSaldo(500)).andReturn(true);

mock.retirar(500);

expect(mock.estaBloqueada()).andReturn(false);

expect(mock.tieneSaldo(10000)).andReturn(false);

expect(mock.estaBloqueada()).andReturn(true);

/\* (3) Ahora, el objeto simulado comienza a esperar las llamadas\*/

replay(mock);

boolean result = classUnderTest.realizarRetirada(500, mock);

assertTrue(result);

// Intenta retirar demasiado dinero:

result = classUnderTest.realizarRetirada(10000, mock);

assertFalse(result);

// Y despues rectifica, pero ya está bloqueda la cuenta:

result = classUnderTest.realizarRetirada(200, mock);

assertFalse(result);

verify(mock);

}

}

## EJERCICIO Propuesto – Integracion con EasyMock

El objetivo de esta práctica es que el estudiante comprenda la utilidad de la prueba con objetos simulados (resguardos o conductores).

Sobre el ejemplo anterior, continuamos con el siguiente supuesto:

Planteamos una nueva clase denominada GestorPrestamosPersonales, que será colaboradora también de CajeroAutomático. Esta nueva clase aún no está desarrollada, por lo que se especificará su comportamiento mediante una interfaz. El gestor de préstamos permitirá desde el cajero solicitar un préstamo personal (especificando la cantidad y el plazo de amortización) con las siguientes reglas:

* 1. No se puede pedir préstamos de más de 3000 EUR o menos de 1000 EUR
  2. No se pueden pedir préstamos a más de 12 meses ni a menos de 6 meses.
  3. No se puede conceder más de un préstamo dentro del mismo año (por lo que habrá que enviar la fecha actual al gestor de préstamos para que lo compruebe.

Sobre el supuesto anterior, se pide:

1. Codificar la interfaz GestorPrestamosPersonales.
2. Codificar el método solicitarPrestamoPersonal() que se invoca sobre el cajero automático.
3. Diseñar casos de prueba para el nuevo método del cajero, considerando la técnica de clases de equivalencia.
4. Implementar la prueba utilizando un objeto simulado para GestorPrestamosPersonales.

## EJERCICIO TUTORIZADO – Pruebas Web

### Enunciado

El objetivo de de este ejercicio es que el estudiante comprenda la prueba de aplicaciones Web como un tipo de prueba de sistema en la cual se simula un “navegador invisible”, de modo que la prueba es completamente independiente de la tecnología de servidor utilizada para la implementación.

Concretamente, se utilizará el framework HTMLUnit para hacer los ejercicios.

**Primer paso: configuración de HTMLUnit**

Pasos a seguir:

1. **Descargar el fichero .zip de la versión que queramos de HTMLUnit.**

La página Web para la descarga es: http://htmlunit.sourceforge.net/

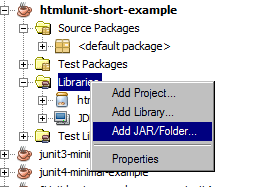
En nuestro caso, utilizaremos [**htmlunit-2.8.zip**](http://sourceforge.net/projects/htmlunit/files/htmlunit/2.6/htmlunit-2.6.zip/download)

1. **Descomprimir el fichero .zip de htmlunit en un directorio.**

La distribución incluye varios ficheros .jar, la documentación y la lincencia de uso.

1. **Añadir el fichero htmlunit-2.8.jar al classpath de nuestro compilador y máquina virtual Java, así como otras librerías necesarias, dependiendo del tipo de prueba que vayamos a realizar.**

En el caso de trabajar con Netbeans esto es tan sencillo como incluir en la carpeta “Test packages” de nuestro proyecto una referencia al .jar de HTMLUnit.



La distribución de HTMLUnit en su directorio /lib incluye muchas bibliotecas que son necesarias dependiendo del tipo de procesamiento que vayamos a hacer. Una lista de esas bibliotecas y su necesidad está en esta página:

[**http://htmlunit.sourceforge.net/dependencies.html**](http://htmlunit.sourceforge.net/dependencies.html)

Se deben incluir también en el proyecto, ya que HTMLUnit depende a su vez de ellas.

**Segundo paso: Un caso de prueba mínimo**

Como primer caso de prueba vamos a probar una funcionalidad del popular buscador Google.

Para ello, definiremos un método testSimpleSearch como caso de prueba JUnit. El comienzo de la prueba necesitará crear un WebClient, que es un “navegador invisible”. Sobre ese cliente podemos obtener cualquier página Web referenciable con una URI invocando a getPage, en el siguiente ejemplo se obtiene la página de Google. La obtención de la página puede producir diferentes excepciones que hay que tratar o relanzar. En el siguiente ejemplo se tratan.

public class TestingGoogle {

@Test

public void testSimpleSearch() throws IOException{

final WebClient webClient = new WebClient();

HtmlPage page = null;

try {

page = webClient.getPage("http://www.google.com/");

} catch (Exception e) {}

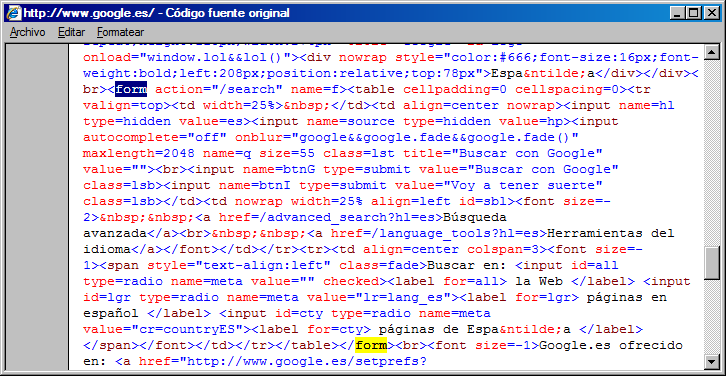
//....

}

}

Una vez se tiene la página, hay que examinar su estructura. A nosotros nos interesa utilizar el botón “Voy a tener suerte” de Google para buscar. Para ello, tenemos que identificar en el HTML de la página su nombre, identificador o algún elemento que nos permita referenciarlo.

En el código de la página vemos que hay un formulario de nombre f, donde se encuentra el campo de entrada y los botones de búsqueda:



Por lo tanto, obtendremos el formulario:

public class TestingGoogle {

@Test

public void testSimpleSearch(){

//...

HtmlForm form = page.getFormByName("f");

//...

}

}

Después podemos apreciar que dentro del formulario hay un campo de entrada con nombre q, que también obtendremos, y escribiremos nuestra cadena de búsqueda en él a través del API HtmlUnit:

public class TestingGoogle {

@Test

public void testSimpleSearch(){

//...

HtmlForm form = page.getFormByName("f");

**HtmlInput input = form.getInputByName("q");**

**input.setValueAttribute("junit");**

//...

}

}

El último paso es “pulsar” el botón. Vemos en el código que el botón en este caso es un input de tipo ”submit”, y el nombre del botón que buscamos es btnI. Obtendremos primero una referencia y después lo “pulsaremos”, produciendo en este caso a la navegación. En el caso de prueba que hemos hecho, la página a la que se navegará deberá ser la de JUnit. El paso final del caso de prueba es comprobar que realmente hemos obtenido esa página. En el ejemplo, lo comprobamos mirando el título de la nueva página.

public class TestingGoogle {

@Test

public void testSimpleSearch(){

//...

HtmlSubmitInput button = form.getInputByName("btnI");

HtmlPage result = null;

try {

result = button.click();

} catch (Exception e) {}

assertTrue(result.getTitleText().equals(

"Welcome to JUnit.org! | JUnit.org"));

}

}

Como hemos visto, la prueba con HTMLUnit se basa en navegar la estructura de la página y simular la navegación del usuario.

public class TestingGoogle {

@Test

public void testSimpleSearch(){

//...

HtmlSubmitInput button = form.getInputByName("btnI");

HtmlPage result = null;

try {

result = button.click();

} catch (Exception e) {}

assertTrue(result.getTitleText().equals(

"Welcome to JUnit.org! | JUnit.org"));

}

}

**Código completo del ejemplo**

import com.gargoylesoftware.htmlunit.html.HtmlPage;

import com.gargoylesoftware.htmlunit.html.HtmlInput;

import com.gargoylesoftware.htmlunit.html.HtmlForm;

import com.gargoylesoftware.htmlunit.WebClient;

import com.gargoylesoftware.htmlunit.html.HtmlSubmitInput;

import java.io.IOException;

import org.junit.Test;

import static org.junit.Assert.\*;

public class TestingGoogle {

@Test

public void testSimpleSearch() throws IOException{

final WebClient webClient = new WebClient();

HtmlPage page = null;

try {

page = webClient.getPage("http://www.google.com/");

} catch (Exception e) {}

HtmlForm form = page.getFormByName("f");

HtmlInput input = form.getInputByName("q");

input.setValueAttribute("junit");

HtmlSubmitInput button = form.getInputByName("btnI");

HtmlPage result = null;

try {

result = button.click();

} catch (Exception e) {}

assertTrue(result.getTitleText().equals(

"Welcome to JUnit.org! | JUnit.org"));

}

}

**Ejercicios propuestos**

1. Sobre el ejemplo anterior, modificar el ejemplo anterior para que la búsqueda se haga en páginas en español, mediante la opción correspondiente que aparece en la interfaz de Google:

