Ingeniería del Software: Laboratorio

Cuaderno de prácticas. P6 - Pruebas

Contenido

OBJETIVO

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

- Pruebas de caja negra: clases de equivalencia
- Pruebas unitarias con objetos simulados (mocks)

RECOMENDACIÓN DE HERRAMIENTAS

- Netbeans (https://netbeans.org)

DISEÑO DE CASOS DE PRUEBA MEDIANTE CLASES DE EQUIVALENCIA

Un problema importante en las **pruebas de caja negra** es la incertidumbre sobre la cobertura de las mismas. Generalmente, la amplia casuística de los valores de entrada obliga a utilizar técnicas que permitan racionalizar el número de pruebas a realizar. La técnica de las **clases de equivalencia** permite reducir esta casuística de manera considerable. Las clases de equivalencia consisten en identificar un conjunto finito de datos que será suficiente para realizar las pruebas cubriendo aceptablemente los distintos casos posibles. Supongamos, por ejemplo, que se desean realizar pruebas de caja negra sobre el siguiente programa, método o función

```
int compararEnteros(int i, int j){
/* Retorna 0 si i es igual a j, -1 si i es menor que j, y +1 si j es mayor
que i */
if (i%35 == 0) return -1;
if (i==j) return 0; if (i<j) return -1; else return 1;
}
```

El número de posibles combinaciones de enteros que podrían pasarse a esta función es elevadísimo (suponiendo enteros de 2 bytes estaríamos hablando de todas las posibles parejas de valores entre –32,768 y +32,767). Sin embargo, si fuéramos capaces de clasificar los enteros y encontrar un número reducido de clases cuyos elementos fueran equivalentes sería suficiente probar un solo elemento de cada clase, pues el resto de elementos producirá salidas similares a las de su representante de clase. Podríamos de este modo identificar tres clases de enteros: negativos, positivos y cero, para posteriormente probar sólo 9 parejas (positivo-positivo, positivo-negativo, positivo-cero, cero-positivo, etc.).

El método de diseño de casos de equivalencia de caja negra consiste entonces en: (i) identificación de clases de equivalencia; y (ii) creación de los casos de prueba correspondiente.

Para identificar las posibles clases de equivalencia de un programa, método o función, a partir de su especificación se deben seguir los siguientes pasos

1. Identificación de las condiciones de las entradas del programa, es decir, restricciones de formato o contenido de los datos de entrada.
2. A partir de ellas, se identifican clases de equivalencia, que pueden ser de datos válidos o no válidos.
3. Existen algunas reglas que ayudan a identificar las clases de equivalencia:
    a. Si se especifica un rango de valores (p.ej., 1 y 10), se creará una clase para válida (1<=*x*<=10) y dos clases no válidas (*x*<1 y *x*>10).
    b. Si se especifica un tipo booleano, se especificarán dos clases, una válida para cuando la variable booleana es válida y otra cuando no lo es.
    c. Para un conjunto de valores permitidos (p.ej., tipo enumerado), se especificará una clase para cada uno de ellos).
    d. Si se sospecha que hay elementos que se tratan de manera diferente, se pueden crear tantas subclases como sea necesario.
4. Finalmente se identifican las clases de equivalencia correspondientes.
    a. Se asigna un único número a cada clase de equivalencia.
    b. Hasta que todas las clases de equivalencia hayan sido cubiertas por los casos de prueba, se complementarán los casos de prueba tratando de cubrir tantas clases como sea posible.
    c. Hasta que todas las clases de equivalencia no válidas hayan sido cubiertas por los casos de prueba, escribir un caso para una única clase no válida sin cubrir.

Ejemplo: Una función que necesita un código de 3 dígitos entre 100 y 500 un tipo enumerado entre “transferencia” y “tarjeta”, function (code, id). Por lo tanto tenemos:

Parámetros	Clases válidas	Clases no válidas
code	(1) 100 <= code <= 500	(2) Code <100 (3) Code > 500
id	(4) “transferencia” (5) “tarjeta”	(6) otro valor

Casos válidos:

- (1) (4): code=110, id=“transferencia”
- (1) (5): code=200, id=“tarjeta”

Casos no válidos:

- (2) (4): code=010, id=“transferencia”
- (3) (5): code=600, id=“transferencia”
- (1) (6) code=100, id=”cheque”

EJERCICIO PROPUESTO – Clases de equivalencia

1. Identificar las clases de equivalencia para una función que necesite, un código entre 1 y 10, y un nombre (string) entre 2 y 25 caracteres.

EJERCICIO TUTORIZADO – Pruebas unitarias con objetos simulados utilizando EasyMock

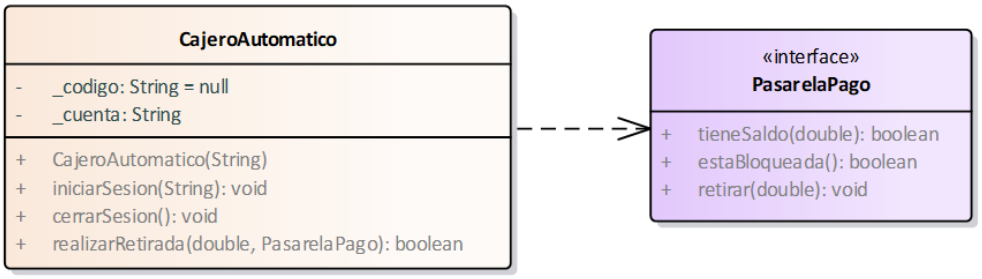
Enunciado

El objetivo de esta práctica es que el estudiante comprenda la utilidad de la prueba con objetos simulados (resguardos-stubs/mocks o conductores-test drivers).

Concretamente, en esta práctica se usarán mocks que son objetos que simulan el comportamiento de una clase que hay que invocar desde la que se está probando, pero con un nivel más avanzado que los stubs, que simplemente devuelven siempre el mismo valor. Se utilizará la librería EasyMock para Java.

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 un parámetro 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.

Primer paso: Diseñar casos de prueba

No se dispone del código fuente de `PasarelaPago`, por lo que para diseñar los casos de prueba, nos podemos basar en el método de los casos de equivalencia.

PasarelaPago	Casos válidos	Casos no válidos
estaBloqueada	(1) estaBloqueada( ) = false	(2) estaBloqueada( ) = true
tieneSaldo	(3) tieneSaldo() = true	(4) tieneSaldo() = false

Casos válidos:

- (1) (3): estaBloqueada()=false, tieneSaldo()=true

Casos no válidos:

- (2) (3): estaBloqueada()=true, tieneSaldo()=true
- (2) (4): estaBloqueada()=true, tieneSaldo()=false
- (1) (4): estaBloqueada()=false, tieneSaldo()=false

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

1. Que se efectúe una retirada de efectivo con suficiente saldo y sin pasarela bloqueada.
2. Que se efectúe una retirada de efectivo sin suficiente saldo y pasarela sin bloquear.
3. Que se efectúe una retirada de efectivo con suficiente saldo pero con pasarela bloqueada.
4. Que se efectúe una retirada de efectivo sin suficiente saldo y con pasarela bloqueada.

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

Segundo paso: Programar la clase y la interfaz

Hay que programar la clase y la interfaz, para lo que se crea con NetBeans un proyecto del tipo “Java with Maven”, al que se añade la clase `CajeroAutomático` y la interfaz `PasarelaPago`.

El código fuente de la clase `CajeroAutomático` podría ser:

```
package p6;

/**
 *   Simula un cajero automático.
 */
public class CajeroAutomatico {

    private String _codigo = null;
    int x;

    /**
     * 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);
        _cuenta = ccc;
    }

    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;
    }
}
```

El código de la interfaz `PasarelaPago` podría ser:

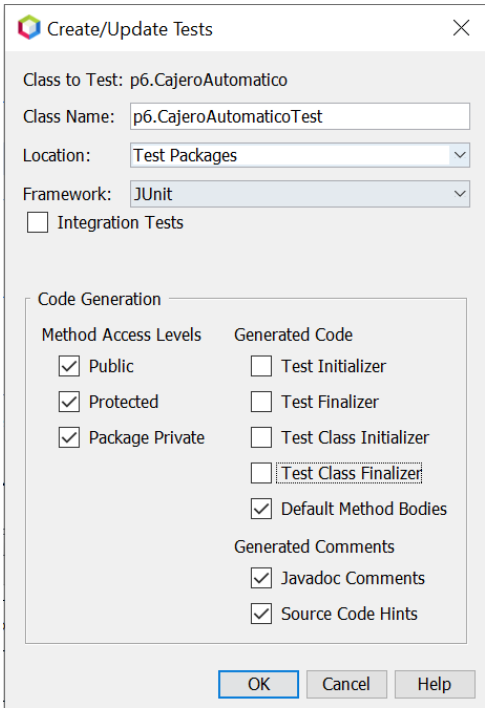
```
package p6;

/*
 * 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.
     */
    public void retirar(double cantidad);
}
```

Tercer paso: configuración de EasyMock

Para usar EasyMock hay que añadir en la carpeta “Test Dependencies” una dependencia a la librería EasyMock. Al ser un proyecto tipo Maven, no es necesario descargarse el archivo jar de la librería EasyMock porque de la descarga se encarga la herramienta Maven incorporada en NetBans, así como de otras posibles librerías de las que dependa EasyMock, como es el caso de una denominada objenesis.

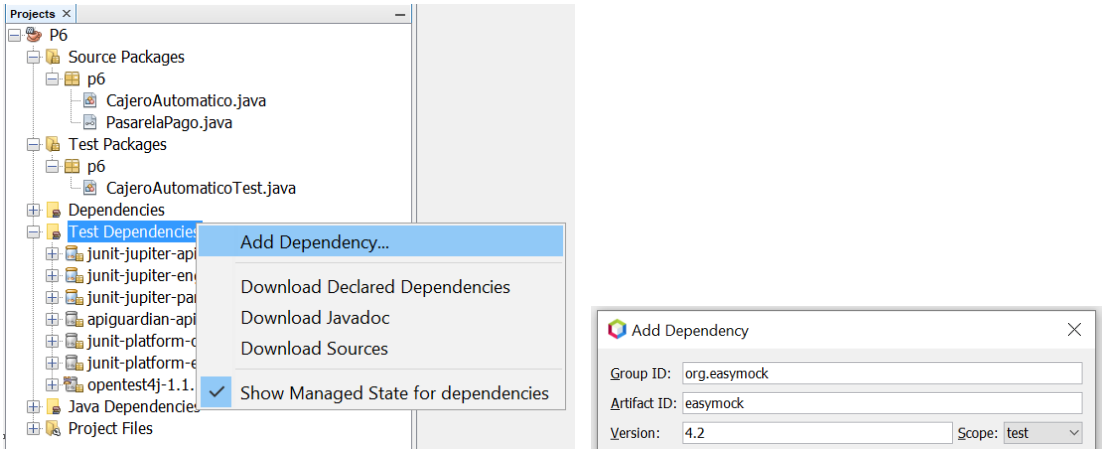
Si la carpeta “Test Dependencies” todavía no aparece en el proyecto, se puede conseguir que aparezca creando la clase de prueba `CajeroAutomáticoTest`.

Se crea como una prueba JUnit.

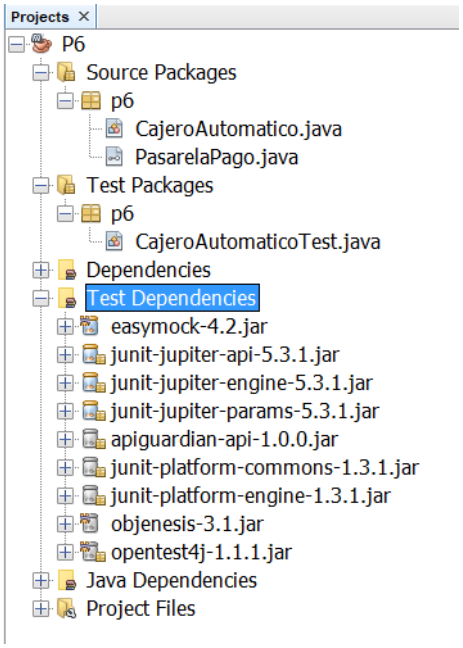


Cuando se trabaja en un Proyecto Maven, los datos para las dependencias de una librería se pueden encontrar en el [Maven Repository](). En la carpeta Test Dependencies, se añade una nueva con estos datos obtenidos de dicho repositorio:

- Group ID=org.easymock, Artifact ID=easymock, Version: 4.2



Y la dependencia easymock-4.2.jar aparece añadida a “Test Dependencies”, junto a otra (objenesis-3-1.jar) que ha detectado Maven que necesita EasyMock para funcionar.



Cuarto paso: Programar un caso de prueba

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

Probaremos el primer caso de prueba, con una pasarela de pago sin bloquear y una cuenta con saldo suficiente. Es decir el caso válido que se determinó al aplicar el método de las clases de equivalencia.

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 5 en el resto del ejemplo.

En la clase de prueba `CajeroAutomaticoTest` hay que declarar el uso de un objeto de la clase `CajeroAutomatico` a probar y un objeto `mock` que simule la pasarela de pago, mediante el siguiente código:

```
public class CajeroAutomaticoTest {

    private CajeroAutomatico unCajeroAProbar;
    private PasarelaPago mock;
```

1) Crear los objetos para la prueba

A continuación se programa el caso de prueba básico, bajo una anotación @Test de JUnit. Comenzando con la creación del objeto que representa el cajero y el objeto `mock`, en este caso invocando el método `createMock` de la librería EasyMock. Habrá que hacer uso de la utilidad "Fix Import" de NetBeans para que añada automáticamente el `import` correspondiente:

```
@Test
public void testRealizarRetirada1() {

System.out.println("realizarRetirada: caso 1 no bloqueada y con saldo");

/* (1) Se crea el objeto de la clase a probar y un mock para simular la
clase PasarelaPago */

unCajeroAProbar = new CajeroAutomatico("1111111111");
unCajeroAProbar.iniciarSesion("1234");
mock = createMock(PasarelaPago.class);
```

2) Grabar las 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.

```
/* (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) Poner objeto simulado en estado de escucha

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

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

replay(mock);
```

4) Programar la prueba del objeto de la clase a probar

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, realizando las comprobaciones mediante aserciones habituales de JUnit. Siguiendo el ejemplo tendríamos lo siguiente:

```
/* (4) Se programa la prueba del objto de la clase a probar */

boolean result = unCajeroAProbar.realizarRetirada(500, mock);
assertTrue(result);
```

En este caso, la llamada tiene que obtener 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, y le pasamos 300 en lugar de 500:

```
Unexpected method call PasarelaPago.tieneSaldo(300.0 (double)):
    PasarelaPago.tieneSaldo(500.0 (double)): expected: 1, actual: 0
    PasarelaPago.retirar(500.0 (double)): expected: 1, actual: 0
```

5) Verificar que se invoquen todas las llamadas al objeto simulado

Si queremos verificar que desde la clase a probar se invoquen todas las llamadas al objeto simulado indicadas en las expectativas grabadas, hay que ejecutar el método `verify()` de la librería EasyMock para que se compruebe que se hacen todas las llamadas esperadas.

```
/* (5) Forzamos a que la ausencia de todas las llamadas previstas sea
un error también */

verify(mock);
```

Podemos comprobar que funciona, accediendo a código fuente de la clase `CajeroAutomatico`, y comentando por ejemplo la línea de código que llamaba al método `estaBloqueado`.

```
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;
}
```

Al ejecutar la prueba aparece un error por no haber usado dicho método, como se había previsto en las expectativas del objeto simulado:

```
Expectation failure on verify:
    PasarelaPago.estaBloqueada(): expected: 1, actual: 0
```

6) Resetear el objeto simulado

Para finalizer el caso de prueba, se utiliza el método `reset()` de la librería EasyMock para "borrar" las expectativas del objeto simulado. En nuestro caso, no es estrictamente necesario invocarlo ya que se creará de nuevo al iniciar la prueba.

```
 /* (6) Se ejecutan instrucciones necesarias de finalizaciónde la prueba
     y se resetea el mock */

unCajeroAProbar.cerrarSesion();
reset(mock);
System.out.println("Fin del caso de prueba 1");
```

7) Código completo de la prueba

El código completo de la clase de prueba es:

```
package p6;

import static org.easymock.EasyMock.createMock;
import static org.easymock.EasyMock.expect;
import static org.easymock.EasyMock.replay;
import static org.easymock.EasyMock.reset;
import static org.easymock.EasyMock.verify;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CajeroAutomaticoTest {

    private CajeroAutomatico unCajeroAProbar;
    private PasarelaPago mock;

    @Test
    public void testRealizarRetirada1() {

        System.out.println("realizarRetirada: caso 1 no bloqueada y con
```

```
saldo");
        /* (1) Se crea el objeto de la clase a probar y un mock para
simular la clase PasarelaPago
         */
        unCajeroAProbar = new CajeroAutomatico("1111111111");
        unCajeroAProbar.iniciarSesion("1234");
        mock = createMock(PasarelaPago.class);

        /* (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);

        /* (4) Se programa la prueba del objto de la clase a probar
         */
        boolean result = unCajeroAProbar.realizarRetirada(500, mock);
        assertTrue(result);

        /* (5) Forzamos a que la ausencia de todas las llamadas
previstas sea un error también
         */
        verify(mock);

        /* (6) Se ejecutan instrucciones necesarias de finalizaciónde
la prueba
        /* y se resetea el mock
         */
        unCajeroAProbar.cerrarSesion();
        reset(mock);

        System.out.println("Fin del caso de prueba 1");

    }

}
```

Quinto paso: Programar el resto de casos de prueba

Hay que programar los otros tres casos de prueba previstos:

- Caso 2: Que se efectúe una retirada de efectivo sin suficiente saldo y pasarela sin bloquear.
- Caso 3: Que se efectúe una retirada de efectivo con suficiente saldo pero con pasarela bloqueada.
- Caso 4: Que se efectúe una retirada de efectivo sin suficiente saldo y con pasarela bloqueada.

El código fuente completo sería:

```
package p6;

import static org.easymock.EasyMock.createMock;
import static org.easymock.EasyMock.expect;
import static org.easymock.EasyMock.replay;
import static org.easymock.EasyMock.reset;
import static org.easymock.EasyMock.verify;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CajeroAutomaticoTest {

    private CajeroAutomatico unCajeroAProbar;
    private PasarelaPago mock;

    @Test
    public void testRealizarRetirada1() {

        System.out.println("realizarRetirada: caso 1 no bloqueada y con
saldo");

        /* (1) Se crea el objeto de la clase a probar y un mock para
simular la clase PasarelaPago
         */
        unCajeroAProbar = new CajeroAutomatico("1111111111");
        unCajeroAProbar.iniciarSesion("1234");
        mock = createMock(PasarelaPago.class);

        /* (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);

        /* (4) Se programa la prueba del objto de la clase a probar
         */
        boolean result = unCajeroAProbar.realizarRetirada(500, mock);
        assertTrue(result);
```

```
        /* (5) Forzamos a que la ausencia de todas las llamadas
previstas sea un error también
         */
        verify(mock);

        /* (6) Se ejecutan instrucciones necesarias de finalizaciónde
la prueba
        /* y se resetea el mock
         */
        unCajeroAProbar.cerrarSesion();
        reset(mock);

        System.out.println("Fin del caso de prueba 1");
    }


    @Test
    public void testRealizarRetirada2() {
        System.out.println("realizarRetirada: caso 2 no bloqueada y sin
saldo");

        unCajeroAProbar = new CajeroAutomatico("1111111111");
        unCajeroAProbar.iniciarSesion("1234");
        mock = createMock(PasarelaPago.class);

        expect(mock.estaBloqueada()).andReturn(false);
        expect(mock.tieneSaldo(500)).andReturn(false);
        mock.retirar(500);

        replay(mock);

        boolean result = unCajeroAProbar.realizarRetirada(500, mock);
        assertFalse(result);

        /* En este caso no se pone verify porque no es obligatorio
ejecutar "retirar" */
        unCajeroAProbar.cerrarSesion();
        reset(mock);

        System.out.println("Fin del caso de prueba 2");
    }


    @Test
    public void testRealizarRetirada3() {
        System.out.println("realizarRetirada: caso 3 con bloqueo y con
saldo");

        unCajeroAProbar = new CajeroAutomatico("1111111111");
        unCajeroAProbar.iniciarSesion("1234");
        mock = createMock(PasarelaPago.class);

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

        replay(mock);

        boolean result = unCajeroAProbar.realizarRetirada(500, mock);
        assertFalse(result);
```

```
        /* En este caso no se pone verify porque no es obligatorio
ejecutar "tieneSaldo" ni "retirar" */
        unCajeroAProbar.cerrarSesion();
        reset(mock);

        System.out.println("Fin del caso de prueba 3");
    }

    @Test
    public void testRealizarRetirada4() {
        System.out.println("realizarRetirada: caso 4 con bloqueo y sin
saldo");

        unCajeroAProbar = new CajeroAutomatico("1111111111");
        unCajeroAProbar.iniciarSesion("1234");
        mock = createMock(PasarelaPago.class);

        expect(mock.estaBloqueada()).andReturn(true);
        expect(mock.tieneSaldo(500)).andReturn(false);
        mock.retirar(500);

        replay(mock);

        boolean result = unCajeroAProbar.realizarRetirada(500, mock);
        assertFalse(result);

        /* En este caso no se pone verify porque no es obligatorio
ejecutar "tieneSaldo" ni "retirar" */
        unCajeroAProbar.cerrarSesion();
        reset(mock);

        System.out.println("Fin del caso de prueba 4");
    }

}
```