Pruebas de unidad con JUnit

Cuando se implementa software, resulta recomendable comprobar que el código que hemos escrito funciona correctamente.

Para ello, implementamos pruebas que verifican que nuestro programa genera los resultados que de él esperamos.

Conforme vamos añadiéndole nueva funcionalidad a un programa, creamos nuevas pruebas con las que podemos medir nuestro progreso y comprobar que lo que antes funcionaba sigue funcionando tras haber realizado cambios en el código (*test de regresión*).

Las pruebas también son de vital importancia cuando refactorizamos (aunque no añadamos nueva funcionalidad, estamos modificando la estructura interna de nuestro programa y debemos comprobar que no introducimos errores al refactorizar).

Automatización de las pruebas

Para agilizar la realización de las pruebas resulta práctico que un test sea completamente automático y compruebe los resultados esperados.

- ✗ No es muy apropiado llamar a una función, guardar el resultado en algún sitio y después tener que comprobar manualmente si el resultado era el deseado.

- ✓ Mantener automatizado un conjunto amplio de tests permite reducir el tiempo que se tarda en depurar errores y en verificar la corrección del código.

JUnit

Herramienta especialmente diseñada para implementar y automatizar la realización de pruebas de unidad en Java.

Dada una clase de nuestra aplicación…

- En una clase aparte definimos un conjunto de casos de prueba
  - La clase hereda de `junit.framework.TestCase`
  - Cada caso de prueba se implementa en un método aparte.
  - El nombre de los casos de prueba siempre comienza por `test`.

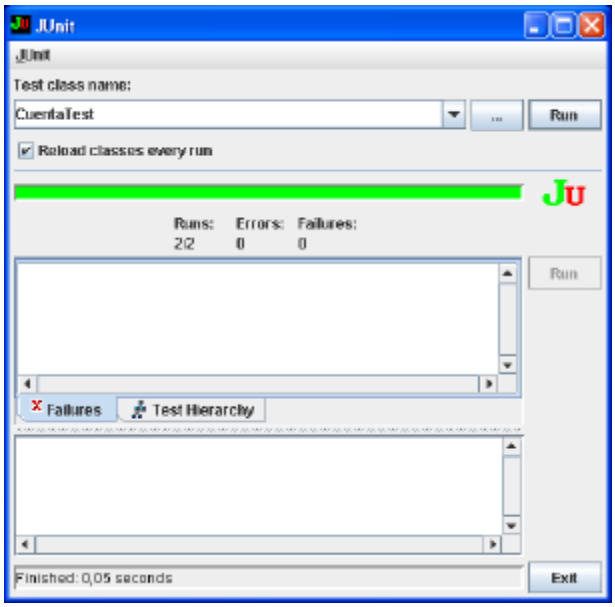
```
import junit.framework.*;

public class CuentaTest extends TestCase
{
 …
}
```

- Cada caso de prueba invoca a una serie de métodos de nuestra clase y comprueba los resultados que se obtienen tras invocarlos.
  - Creamos uno o varios objetos de nuestra clase con `new`
  - Realizamos operaciones con ellos.
  - Definimos aserciones (condiciones que han de cumplirse).

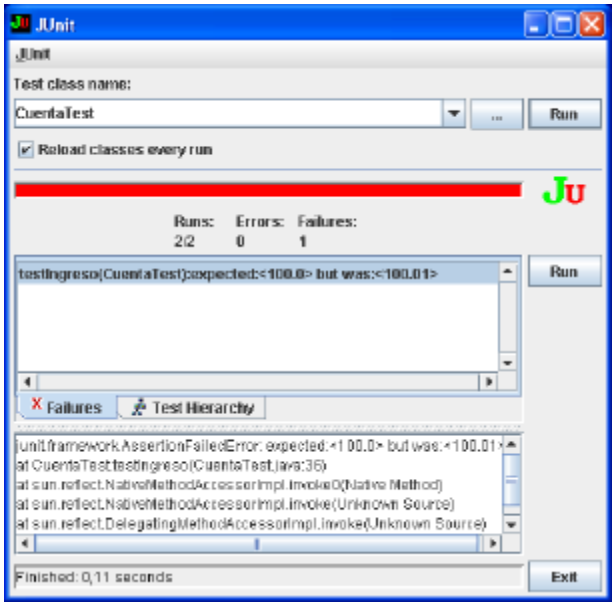
```
public void testCuentaNueva ()
{
  Cuenta cuenta = new Cuenta();

  assertEquals(cuenta.getSaldo(), 0.00);
}

public void testIngreso ()
{
  Cuenta cuenta = new Cuenta();

  cuenta.ingresar(100.00);
  assertEquals(cuenta.getSaldo(), 100.00);
}
```

Finalmente, ejecutamos los casos de prueba con **JUnit**:

- Si todos los casos de prueba funcionan correctamente…



- Si algún caso de prueba falla…



Tendremos que localizar el error y corregirlo con ayuda de los mensajes que nos muestra JUnit.

MUY IMPORTANTE: Que nuestra implementación supere todos los casos de prueba no quiere decir que sea correcta; sólo quiere decir que funciona correctamente para los casos de prueba que hemos diseñado.

Apéndice: Cómo ejecutar JUnit desde nuestro propio código

Para lanzar JUnit desde nuestro propio código,
sin tener que ejecutar la herramienta a mano,
hemos de implementar el método `main` en nuestra clase
y definir un sencillo constructor.

```
import junit.framework.*;

public class CuentaTest extends TestCase
{
  public static void main(String args[])
  {
    junit.swingui.TestRunner.main (
          new String[] {"CuentaTest"});
  }

  public CuentaTest(String name)
  {
    super(name);
  }

  // Casos de prueba
  …
}
```

Ejemplo: La clase Money

Basado en “Test-Driven Development by Example”, pp. 1-87 © Kent Beck

Vamos a definir una clase, denominada `Money`, para representar cantidades de dinero que pueden estar expresadas en distintas monedas

Por ejemplo, queremos usar esta clase para generar informes como…

Empresa	Acciones	Precio	Total
Telefónica	200	10 EUR	2000 EUR
Vodafone	100	50 EUR	5000 EUR
			7000 EUR

El problema es que también nos podemos encontrar con situaciones como la siguiente …

Empresa	Acciones	Precio	Total
Microsoft	200	13 USD	2600 USD
Indra	100	50 EUR	5000 EUR
			7000 EUR

donde hemos tenido que utilizar el tipo de cambio actual (1€=$1.30)

Comenzamos creando una clase para representar cantidades de dinero:

```
public class Money
{
  private int cantidad;
  private String moneda;

  public Money (int cantidad, String moneda)
  {
    this.cantidad = cantidad;
    this.moneda = moneda;
  }

  public int    getCantidad() { return cantidad; }
  public String getMoneda()   { return moneda;   }
}
```

Una de las cosas que tendremos que hacer es *sumar cantidades*, por lo que podemos idear un caso de prueba como el siguiente:

```
import junit.framework.*;

public class MoneyTest extends TestCase
{
  public void testSumaSimple()
  {
    Money m10 = new Money (10, "EUR");
    Money m20 = new Money (20, "EUR");

    Money esperado = new Money (30, "EUR");
    Money resultado = m10.add(m20);

    Assert.assertEquals(resultado,esperado);
  }
}
```

Al idear el caso de prueba, nos estamos fijando en cómo tendremos que usar nuestra clase en la práctica, lo que nos es extremadamente útil para definir su interfaz.

En este caso, nos hace falta añadir un método `add` a la clase `Money` para poder compilar y ejecutar el caso de prueba…

Creamos una implementación inicial de este método:

```
  public Money add (Money m)
  {
    int total = getCantidad() + m.getCantidad();
    return new Money( total, getMoneda());
  }
```

Compilamos y ejecutamos el test para llevarnos una sorpresa…

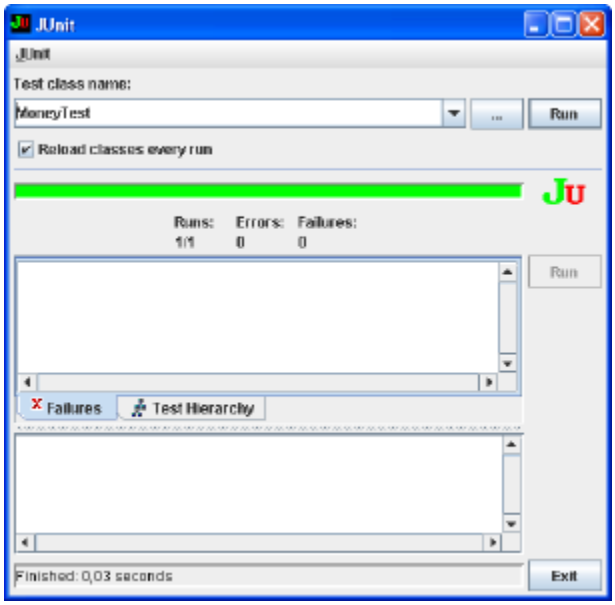
El caso de prueba falla porque la comparación de objetos en Java, por defecto, se limita a comparar referencias (no compara el estado de los objetos, que es lo que podríamos pensar [erróneamente]).

La comparación de objetos en Java se realiza con el método `equals`, que puede recibir como parámetro un objeto cualquiera.

Por tanto, hemos de definir el método `equals` en la clase `Money`:

```
public boolean equals (Object obj)
{
  Money   aux;
  boolean iguales;

  if (obj instanceof Money) {
     aux = (Money) obj;
     iguales = aux.getMoneda().equals(getMoneda())
           && (aux.getCantidad() == getCantidad());
  } else {
     iguales = false;
  }

  return iguales;
}
```

Volvemos a ejecutar el caso de prueba…



… y ahora sí podemos seguir avanzando

De todas formas, para asegurarnos de que todo va bien, creamos un caso de prueba específico que verifique el funcionamiento de `equals`

```
  public void testEquals()
  {
    Money m10 = new Money (10, "EUR");
    Money m20 = new Money (20, "EUR");

    Assert.assertEquals(m10,m10);
    Assert.assertEquals(m20,m20);
    Assert.assertTrue(!m10.equals(m20));
    Assert.assertTrue(!m20.equals(m10));
    Assert.assertTrue(!m10.equals(null));
  }
```

Al ejecutar nuestros dos casos de prueba con JUNIT
vemos que todo marcha como esperábamos.

Sin embargo, comenzamos a ver que existe código duplicado
(y ya sabemos que eso no es una buena señal),
por lo que utilizamos la posibilidad que nos ofrece JUNIT
de definir variables de instancia en la clase que hereda de `TestCase`
(variables que hemos de inicializar en el método `setUp()`)

```
public class MoneyTest extends TestCase
{
  Money m10;
  Money m20;

  public void setUp()
  {
    m10 = new Money (10, "EUR");
    m20 = new Money (20, "EUR");
  }
  …
}
```

Aparte de sumar cantidades de dinero, también tenemos que ser capaces de *multiplicar una cantidad por un número entero* (p.ej. número de acciones por precio de cada acción)

Podemos añadir un nuevo caso de prueba que utilice esta función:

```
public void testMultiplicar ()
{
  Assert.assertEquals ( m10.times(2), m20 );
  Assert.assertEquals ( m10.times(2), m20 );
  Assert.assertEquals ( m10.times(10), m20.times(5));
}
```

Obviamente, también tendremos que definir `times` en `Money`:

```
public Money times (int n)
{
  return new Money ( n*getCantidad(), getMoneda() );
}
```

Compilamos y ejecutamos los casos de prueba con `JUnit`:

Poco a poco, vamos añadiéndole funcionalidad a nuestra clase:
Cada vez que hacemos cambios, volvemos a ejecutar todos los casos de prueba para confirmar que no hemos estropeado nada.

Una vez que hemos comprobado que ya somos capaces de hacer operaciones cuando todo se expresa en la misma moneda, tenemos que comenzar a trabajar con distintas monedas.

Por ejemplo:

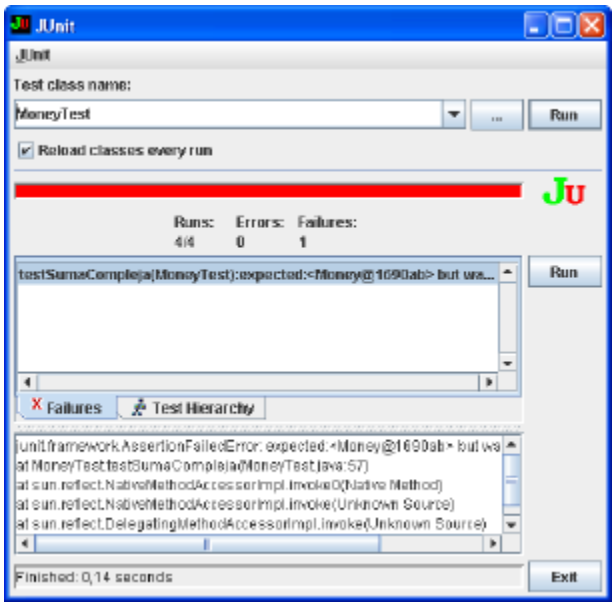
```
public void testSumaCompleja ()
{
  Money euros     = new Money(100,"EUR");
  Money dollars   = new Money(130,"USD");
  Money resultado = euros.add (dollars);
  Money banco     = Bank.exchange(dollars,"EUR")
  Money esperado  = euros.add(banco);

  Assert.assertEquals ( resultado, esperado );
}
```

Para hacer el cambio de moneda, suponemos que tenemos acceso a un banco que se encarga de hacer la conversión. Hemos de crear una clase auxiliar `Bank` que se va a encargar de consultar los tipos de cambio y aplicar la conversión correspondiente:

```
public class Bank
{
  public static Money exchange
         (Money dinero, String moneda)
  {
    int cantidad = 0;

    if (dinero.getMoneda().equals(moneda)) {
       cantidad = dinero.getCantidad();
    } else if (  dinero.getMoneda().equals("EUR")
              && moneda.equals("USD")) {
       cantidad = (130*dinero.getCantidad())/100;
    } else if (  dinero.getMoneda().equals("USD")
              && moneda.equals("EUR")) {
       cantidad = (100*dinero.getCantidad())/130;
    }

    return new Money(cantidad,moneda);
  }
}
```

Por ahora, nos hemos limitado a realizar una conversión fija para probar el funcionamiento de nuestra clase `Money` (en una aplicación real tendríamos que conectarnos realmente con el banco).

Obviamente, al ejecutar nuestros casos de prueba se produce un error:



Hemos de corregir la implementación interna del método `add` de la clase `Money` para que tenga en cuenta el caso de que las cantidades correspondan a monedas diferentes:

```
public Money add (Money dinero)
{
  Money convertido;
  int   total;

  if (getMoneda().equals(dinero.getMoneda()))
    convertido = dinero;
  else
    convertido = Bank.exchange(dinero, getMoneda());

  total = getCantidad() + convertido.getCantidad();

  return new Money( total, getMoneda());
}
```

Volvemos a ejecutar los casos de prueba
y comprobamos que, ahora sí, las sumas se hacen correctamente:

Si todavía no las tuviésemos todas con nosotros,
podríamos seguir añadiendo casos de prueba para adquirir
más confianza en la implementación que acabamos de realizar.

Por ejemplo, el siguiente caso de prueba comprueba
el funcionamiento del banco al realizar conversiones de divisas:

```
public void testBank ()
{
  Money euros     = new Money(10,"EUR");
  Money dollars   = new Money(13,"USD");

  Assert.assertEquals (
     Bank.exchange(dollars,"EUR"), euros );
  Assert.assertEquals (
     Bank.exchange(dollars,"USD"), dollars );
  Assert.assertEquals (
     Bank.exchange(euros,  "EUR"), euros );
  Assert.assertEquals (
     Bank.exchange(euros, "USD"), dollars );
}
```

Comentarios finales: El método `toString`

Para que resulte más fácil interpretar
los mensajes generados por JUNIT,
resulta recomendable definir el método `toString()`
en todas las clases que definamos. Por ejemplo:

```
public class Money
{
  …
  public String toString ()
  {
    return getCantidad()+" "+getMoneda();
  }
}
```

RECORDATORIO: `toString()` es un método
que se emplea en Java para convertir un objeto cualquiera
en una cadena de caracteres.

Teniendo definido el método anterior,
en nuestras aplicaciones podríamos escribir directamente…

```
…
Money share = new Money(13,"USD");
Money investment = share.times(200);
Money euros = Bank.exchange(investment,”EUR”);

System.out.println(investment + "(" + euros + ")");
…
```

y obtener como resultado en pantalla:

```
2600 USD (2000 EUR)
```

