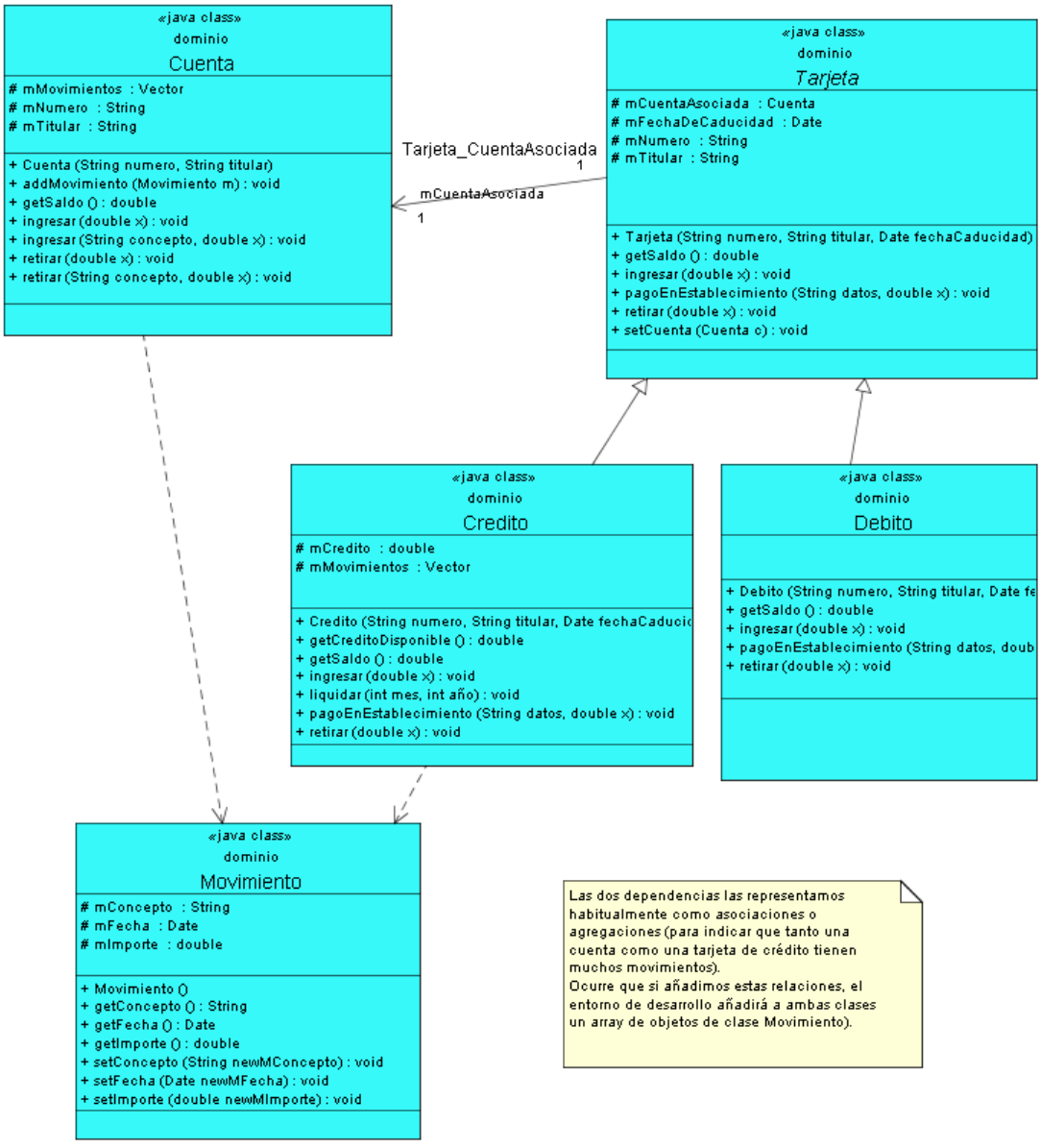
Pruebas de Programas Java mediante JUnit: algunos ejemplos comentados

Macario Polo Usaola
Grupo Alarcos
Escuela Superior de Informática
Universidad de Castilla-La Mancha
Paseo de la Universidad, 4
13071-Ciudad Real
macario.polo@uclm.es
http://www.inf-cr.uclm.es/www/mpolo

Ciudad Real, noviembre de 2003

Este documento contiene tres ejemplos que ilustran la utilización de Junit para realizar pruebas en programas Java.

Los cuatro tres parten del mismo conjunto de clases, que vienen a representar un modelo muy simple de un sistema de gestión bancario. Las clases que forman parte del dominio del problema y sobre las que vamos a relizar las pruebas se muestran en la siguiente figura:



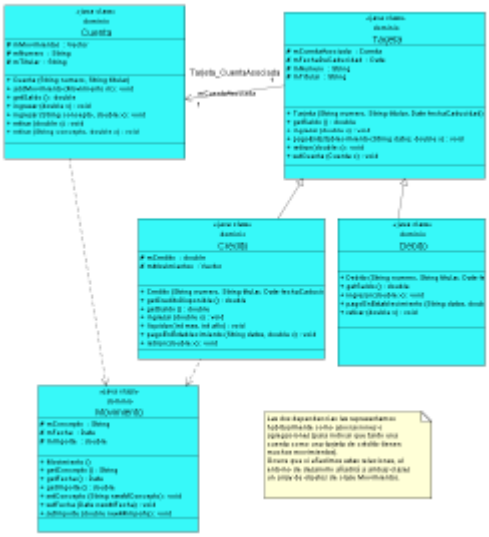
Como se observa, representamos el hecho de que en nuestro banco hay *Cuentas* y *Tarjetas* de *Crédito* y de *Débito*, que están asociadas a una cuenta. Las operaciones que se realizan sobre una *Cuenta* quedan registradas en un *Vector* de objetos de clase *Movimiento*, ocurriendo lo mismo con las tarjetas de *Crédito.* En *Tarjeta,* todas las operaciones son abstractas excepto el constructor y *setCuenta(Cuenta)*, si bien no se señalan como tales porque no lo permite la herramienta utilizada (Oracle JDeveloper 9.0.5).

Los ejemplos confeccionados se entregan en tres proyectos (*Tuto1*, *Tuto3* y *Tuto4*) incluidos en un mismo *workspace* de JDeveloper.

Todas las clases que se van a probar se han colocado en un paquete llamado *dominio*, mientras que las clases de prueba (especializaciones de la clase *TestCase* definida en *JUnit*) se han colocado en el paquete *dominio.test.*

Este documento, los ejemplos y las transparencias pueden descargarse de http://www.inf-cr.uclm.es/www/mpolo

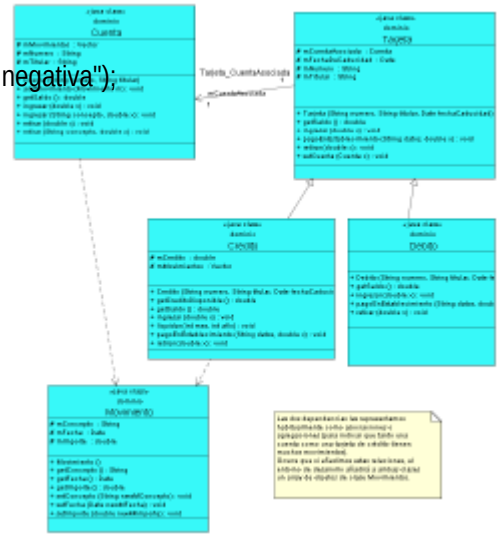
Proyecto *Tuto1.*



Clase Cuenta.

```
package dominio;

import java.util.Vector;

public class Cuenta
{
	protected String mNumero;
	protected String mTitular;
	protected Vector mMovimientos;

	public Cuenta(String numero, String titular)
	{
		mNumero=numero;
		mTitular=titular;
		mMovimientos=new Vector();
	}

	public void ingresar(double x) throws Exception
	{
		if (x<=0)
			throw new Exception("No se puede ingresar una cantidad negativa");
		Movimiento m=new Movimiento();
		m.setConcepto("Ingreso en efectivo");
		m.setImporte(x);
		this.mMovimientos.addElement(m);
	}

	public void retirar(double x) throws Exception
	{
		if (x<=0)
			throw new Exception("No se puede retirar una cantidad negativa");
		if (getSaldo()<x)
			throw new Exception("Saldo insuficiente");
		Movimiento m=new Movimiento();
		m.setConcepto("Retirada de efectivo");
		m.setImporte(-x);
		this.mMovimientos.addElement(m);

	}

	public void ingresar(String concepto, double x) throws Exception
	{
		if (x<=0)
			throw new Exception("No se puede ingresar una cantidad negativa");
		Movimiento m=new Movimiento();
		m.setConcepto(concepto);
		m.setImporte(x);
		this.mMovimientos.addElement(m);
	}

	public void retirar(String concepto, double x) throws Exception
```

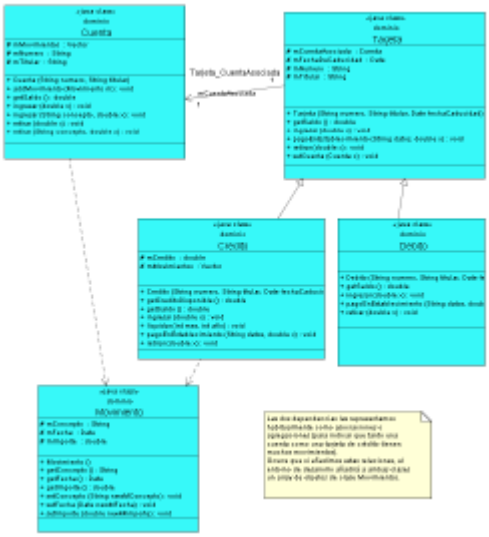
```
	{
		if (x<=0)
			throw new Exception("No se puede retirar una cantidad negativa");
		if (getSaldo()<x)
			throw new Exception("Saldo insuficiente");
		Movimiento m=new Movimiento();
		m.setConcepto(concepto);
		m.setImporte(-x);
		this.mMovimientos.addElement(m);

	}

	public double getSaldo()
	{
		double r=0.0;
		for (int i=0; i<this.mMovimientos.size(); i++)
		{
			Movimiento m=(Movimiento) mMovimientos.elementAt(i);
			r+=m.getImporte();
		}
		return r;
	}

	public void addMovimiento(Movimiento m)
	{
		mMovimientos.addElement(m);
	}
}
```



Clase Movimiento.

```
package dominio;

import java.util.Date;

public class Movimiento
{
	protected String mConcepto;
	protected Date mFecha;
	protected double mImporte;

	public Movimiento()
	{
		mFecha=new Date();
	}

	public double getImporte()
	{
		return mImporte;
	}

	public String getConcepto()
	{
		return mConcepto;
	}
```

```
	public void setConcepto(String newMConcepto)
	{
		mConcepto = newMConcepto;
	}

	public Date getFecha()
	{
		return mFecha;
	}

	public void setFecha(Date newMFecha)
	{
		mFecha = newMFecha;
	}


	public void setImporte(double newMImporte)
	{
		mImporte = newMImporte;
	}
}
```

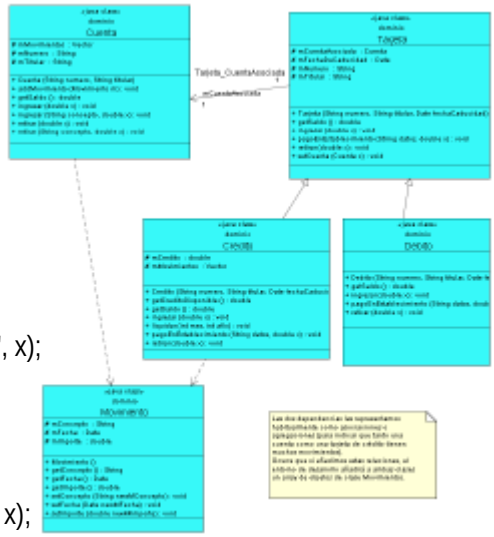


Clase Tarjeta (abstracta).

```
package dominio;

import java.util.Date;

public abstract class Tarjeta
{
	protected String mNumero, mTitular;
	protected Date mFechaDeCaducidad;
	protected Cuenta mCuentaAsociada;

	public Tarjeta(String numero, String titular, Date fechaCaducidad)
	{
		mNumero=numero;
		mTitular=titular;
		mFechaDeCaducidad=fechaCaducidad;
	}

	public void setCuenta(Cuenta c)
	{
		mCuentaAsociada=c;
	}

	public abstract void retirar(double x) throws Exception;
	public abstract void ingresar(double x) throws Exception;
	public abstract void pagoEnEstablecimiento(String datos, double x) throws Exception;
	public abstract double getSaldo();
}
```

Clase Debito (especialización de la clase abstracta *Tarjeta*).

```
package dominio;

import java.util.Date;
```

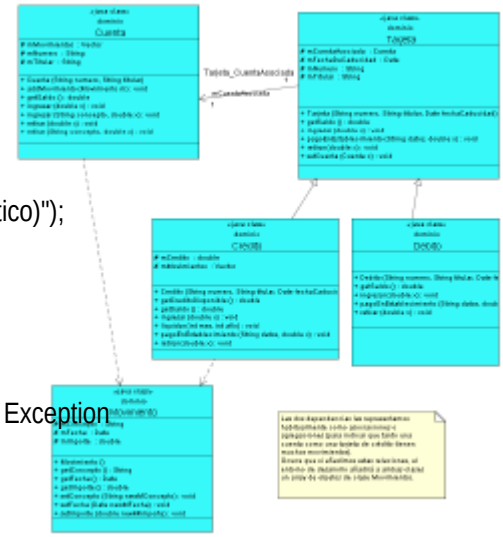


```
public class Debito extends Tarjeta
{
	public Debito(String numero, String titular, Date fechaCaducidad)
	{
		super(numero, titular, fechaCaducidad);
	}

	public void retirar(double x) throws Exception
	{
		this.mCuentaAsociada.retirar("Retirada en cajero automático", x);
	}

	public void ingresar(double x) throws Exception
	{
		this.mCuentaAsociada.retirar("Ingreso en cajero automático", x);
	}

	public void pagoEnEstablecimiento(String datos, double x) throws Exception
	{
		this.mCuentaAsociada.retirar("Compra en :" + datos, x);
	}

	public double getSaldo()
	{
		return mCuentaAsociada.getSaldo();
	}
}
```

Clase Credito (especialización de la clase abstracta *Tarjeta).*

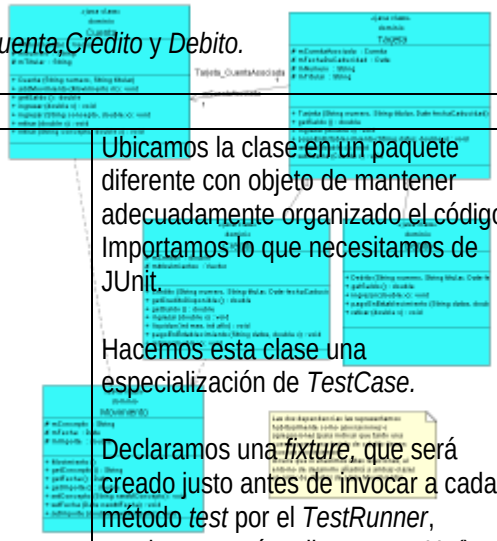
```
package dominio;

import java.util.Vector;
import java.util.Date;

public class Credito extends Tarjeta
{
	protected double mCredito;
	protected Vector mMovimientos;

	public Credito(String numero, String titular, Date fechaCaducidad, double credito)
	{
		super(numero, titular, fechaCaducidad);
		mCredito=credito;
		mMovimientos=new Vector();
	}

	public void retirar(double x) throws Exception
	{
		Movimiento m=new Movimiento();
		m.setConcepto("Retirada en cajero automático");
		x=(x*0.05<3.0 ? 3 : x*0.05);  // Añadimos una comisión de un 5%, mínimo de 3 euros.
		m.setImporte(x);
		mMovimientos.addElement(m);
		if (x>getCreditoDisponible())
```

```
                throw new Exception("Crédito insuficiente");
        }

        public void ingresar(double x) throws Exception
        {
                Movimiento m=new Movimiento();
                m.setConcepto("Ingreso en cuenta asociada (cajero automático)");
                m.setImporte(x);
                mMovimientos.addElement(m);
                mCuentaAsociada.ingresar(x);
        }

        public void pagoEnEstablecimiento(String datos, double x) throws Exception
        {
                Movimiento m=new Movimiento();
                m.setConcepto("Compra a crédito en: " + datos);
                m.setImporte(x);
                mMovimientos.addElement(m);
        }

        public double getSaldo()
        {
                double r=0.0;
                for (int i=0; i<this.mMovimientos.size(); i++)
                {
                        Movimiento m=(Movimiento) mMovimientos.elementAt(i);
                        r+=m.getImporte();
                }
                return r;
        }

        public double getCreditoDisponible()
        {
                return mCredito-getSaldo();
        }

        public void liquidar(int mes, int año)
        {
                Movimiento liq=new Movimiento();
                liq.setConcepto("Liquidación de operaciones tarj. crédito, " + (mes+1) + " de " + (año+1900));
                double r=0.0;
                for (int i=0; i<this.mMovimientos.size(); i++)
                {
                        Movimiento m=(Movimiento) mMovimientos.elementAt(i);
                        if (m.getFecha().getMonth()+1==mes && m.getFecha().getYear()+1900==año)
                                r+=m.getImporte();
                }
                liq.setImporte(r);
                if (r!=0)
                        mCuentaAsociada.addMovimiento(liq);
        }
}
```



Clases de prueba.

Realizaremos pruebas de algunas funcionalidades de las clases *Cuenta*,*Credito* y *Debito.*



Pruebas de Cuenta

```
package dominio.test;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import dominio.Cuenta;

public class CuentaTester1 extends TestCase
{

    Cuenta cuenta;



    public CuentaTester1(String sTestName)
    {
        super(sTestName);
    }

    public void setUp() throws Exception
    {
        cuenta = new Cuenta("0001.0002.12.1234567890", "Fulano de Tal");
    }



    public void tearDown() throws Exception
    {
    }



    public void testIngresar1000()
    {
        try {
            cuenta.ingresar(1000);
            assertTrue(cuenta.getSaldo()==1000.0);
        }
        catch (Exception e)
        {
            fail("No debería haber fallado");
        }
    }

    public void testRetirar1000()
    {
        try
        {
            cuenta.retirar(1000);
        }
        catch (Exception e)
        {
        }
        assertTrue(cuenta.getSaldo()==0.0);
```

Ubicamos la clase en un paquete diferente con objeto de mantener adecuadamente organizado el código. Importamos lo que necesitamos de JUnit.

Hacemos esta clase una especialización de *TestCase.*

Declaramos una *fixture*, que será creado justo antes de invocar a cada método *test* por el *TestRunner*, gracias a que éste llama a *setUp()*.

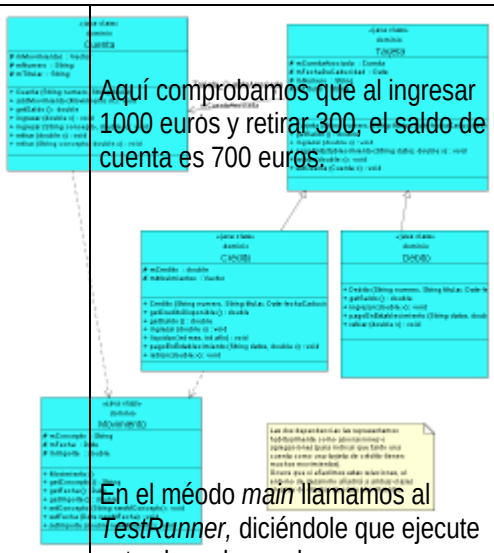
setUp se ejecuta justo antes de cada método *test.* En este caso, creamos una cuenta con ese número y para ese titular.

tearDown se ejecuta justo después de cada método *test*, por lo que aquí podemos escribir código que libere recursos, memoria, etc. En este ejemplo no nos hace falta.

Escribimos un método para comprobar si, tras ingresar 1000 € en la cuenta, su saldo es en efecto 1000 €. Esperamos que no se produzca error, por lo que en el catch lanzamos explícitamente un *fail*.

Con este método comprobamos que al retirar 1000 euros de una cuenta recién creada (y con saldo 0, por tanto), el saldo de la cuenta sigue siendo 0.

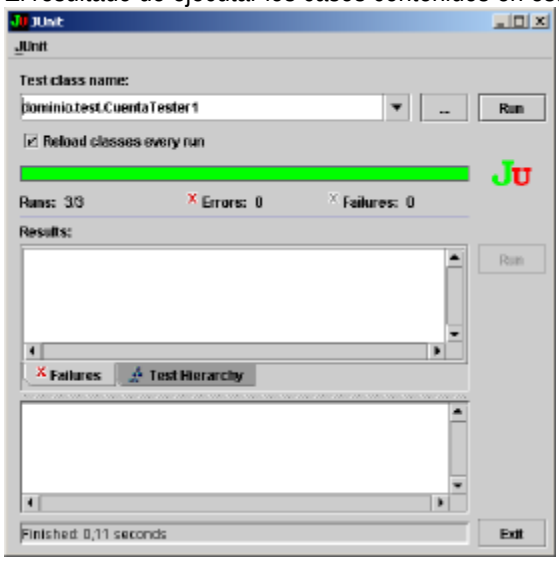
```
    }

    public void testIngresoYRetirada()
    {
        try
        {
            cuenta.ingresar(1000.0);
            cuenta.retirar(300.0);
        }
        catch (Exception e)
        {
        }

        assertTrue(cuenta.getSaldo()==700.0);
    }

    public static void main(String args[])
    {
        junit.swingui.TestRunner.run(CuentaTester1.class);
    }
}
```



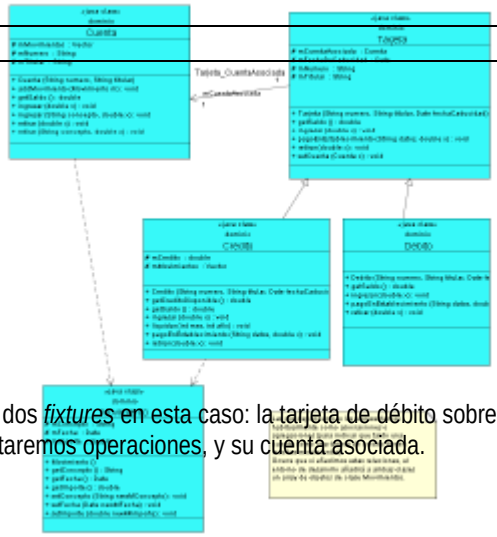
Aquí comprobamos que al ingresar 1000 euros y retirar 300, el saldo de la cuenta es 700 euros.

En el méodo *main* llamamos al *TestRunner*, diciéndole que ejecute esta clase de prueba.

El resultado de ejecutar los casos contenidos en esta clase es el siguiente:



Pruebas de Debito



```
package dominio.test;

import dominio.Cuenta;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import dominio.Debito;
import java.util.Date;

public class DebitoTester1 extends TestCase
{
	Debito debito;
	Cuenta cuenta;

	public DebitoTester1(String sTestName)
	{
		super(sTestName);
	}

	public void setUp() throws Exception
	{
		cuenta=new Cuenta("0001.0002.12.1234567890",
"Fulano de Tal");
		cuenta.ingresar(1000.0);
		Date hoy=new Date();
		long tiempo=Long.parseLong("12096000000");
		Date fecha=new Date(hoy.getTime()+ tiempo); //
Caduca en 4 años
		debito=new Debito("1234567890123456", "Fulano de
Tal", fecha);
		debito.setCuenta(cuenta);
	}

	public void tearDown() throws Exception
	{
	}

	public void testRetirar1000()
	{
		try
		{
			debito.retirar(1000.0);

	assertTrue(debito.getSaldo()==cuenta.getSaldo());
			assertTrue(debito.getSaldo()==0.0);
		}
		catch (Exception e)
		{
			fail("No deberia haber fallado");
		}
	}

	public static void main(String args[])
	{
		junit.swingui.TestRunner.run(DebitoTester1.class);
	}
}
```

Declaramos dos *fixtures* en esta caso: la tarjeta de débito sobre la que ejecutaremos operaciones, y su cuenta asociada.

Creamos las fixtures: la cuenta con esa numeración, de D. Fulano de Tal, y en la que ingresamos 1000 €...

...y una tarjeta de débito que asociamos a la cuenta anterior.

Probamos a ver qué pasa al retirar 1000 euros con la tarjeta. Según el código de la clase *Debito*, el importe se retira de manera inmediata de la cuenta asociada (en la que hay 1000 euros porque así lo hemos dicho en el método *setUp*).

Tras la retirada, el saldo de la tarjeta debe ser igual al de la cuenta asociada (primer *assertTrue*), y además debe ser 0 (segundo *assertTrue*).

El resultado de la ejecución es también correcto.

Pruebas de Credito

```
package dominio.test;

import java.util.Date;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import dominio.Credito;
import dominio.Cuenta;

public class CreditoTester1 extends TestCase
{
	Cuenta cuenta;
	Credito tarjeta;

	public CreditoTester1(String sTestName)
	{
		super(sTestName);
	}

	public void setUp() throws Exception
	{
		cuenta=new Cuenta("0001.0002.12.1234567890", "Fulano de Tal");
		cuenta.ingresar(1000.0);
		Date hoy=new Date();
		long tiempo=Long.parseLong("12096000000");
		Date fecha=new Date(hoy.getTime()+ tiempo); // Caduca en 4 años
		tarjeta=new Credito("1234567890123456", "Fulano de Tal", fecha,
1000.0);  // 1000 € de crédito
		tarjeta.setCuenta(cuenta);
	}

	public void tearDown() throws Exception
	{
	}

	public void testIngresar500()
	{
		try
		{
			double saldoAnteriorCuenta=cuenta.getSaldo();
			double saldoAnteriorTarjeta=tarjeta.getSaldo();
			double creditoAnterior=tarjeta.getCreditoDisponible();
			tarjeta.ingresar(500.0);
			assertTrue(cuenta.getSaldo()==saldoAnteriorCuenta+500);

			assertTrue(tarjeta.getSaldo()==saldoAnteriorTarjeta);
			assertTrue(tarjeta.getCreditoDisponible()==creditoAnterior);
		}
		catch (Exception e)
		{
		}
	}

	public void testRetirar300()
	{
	try
		{
			double saldoAnteriorCuenta=cuenta.getSaldo();
			double saldoAnteriorTarjeta=tarjeta.getSaldo();
			double creditoAnterior=tarjeta.getCreditoDisponible();
			tarjeta.retirar(300.0);
			assertTrue(cuenta.getSaldo()==saldoAnteriorCuenta);
```

Probamos a ver qué pasa si ingresamos 500 euros con esta tarjeta de crédito. Según el código, deben ingresarse en la cuenta asociada a la tarjeta.

Guardo en unas variables el crédito y el saldo de la tarjeta, así como el saldo de la cuenta justo antes de ejecutar la operación.
Ingreso los 500 euros.
Compruebo que el saldo de la cuenta es el saldo anterior más el importe ingresado.
Compruebo que el saldo de la tarjeta no ha variado.
Compruebo que tampoco ha variado el código de la tarjeta.

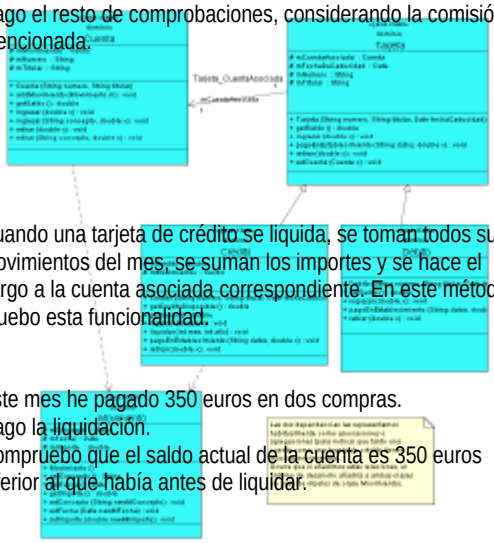
Compruebo qué pasa al retirar 3000 euros. Según el código fuente de *Credito*, la retirada de efectivo con este tipo de tarjeta lleva una comisión del 5% del importe, con un mínimo de 3 euros.

Guardo los datos anteriores a la retirada.

Retiro los 300 €
Compruebo que la operación no ha afectado a la Cuenta (porque es una operación a crédito)

```
                    assertTrue(tarjeta.getSaldo()==saldoAnteriorTarjeta-15);
                    assertTrue(tarjeta.getCreditoDisponible()==creditoAnterior-300);
                }
                catch (Exception e)
                {
                }
        }

        public void testLiquidar()
        {
                try
                {
                        double saldoAnteriorCuenta=cuenta.getSaldo();
                        tarjeta.pagoEnEstablecimiento("Zara", 120.0);
                        tarjeta.pagoEnEstablecimiento("El Corte Inglés", 230.0);
                        tarjeta.liquidar(11, 2003);
                        assertTrue(saldoAnteriorCuenta==cuenta.getSaldo()-
350.0);
                }
                catch (Exception e)
                {

                }
        }

        public static void main(String args[])
        {
                junit.swingui.TestRunner.run(CreditoTester1.class);
        }
}
```

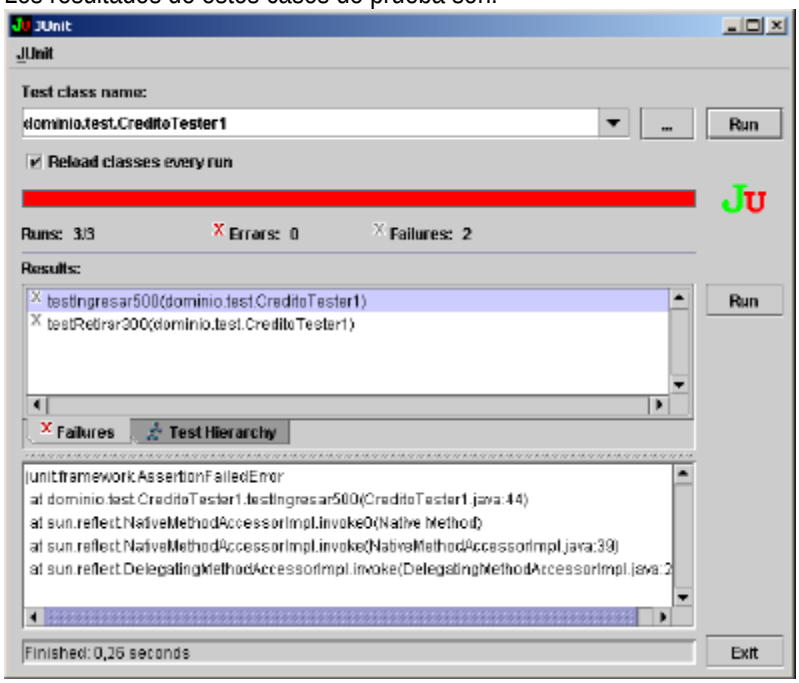
Hago el resto de comprobaciones, considerando la comisión mencionada.



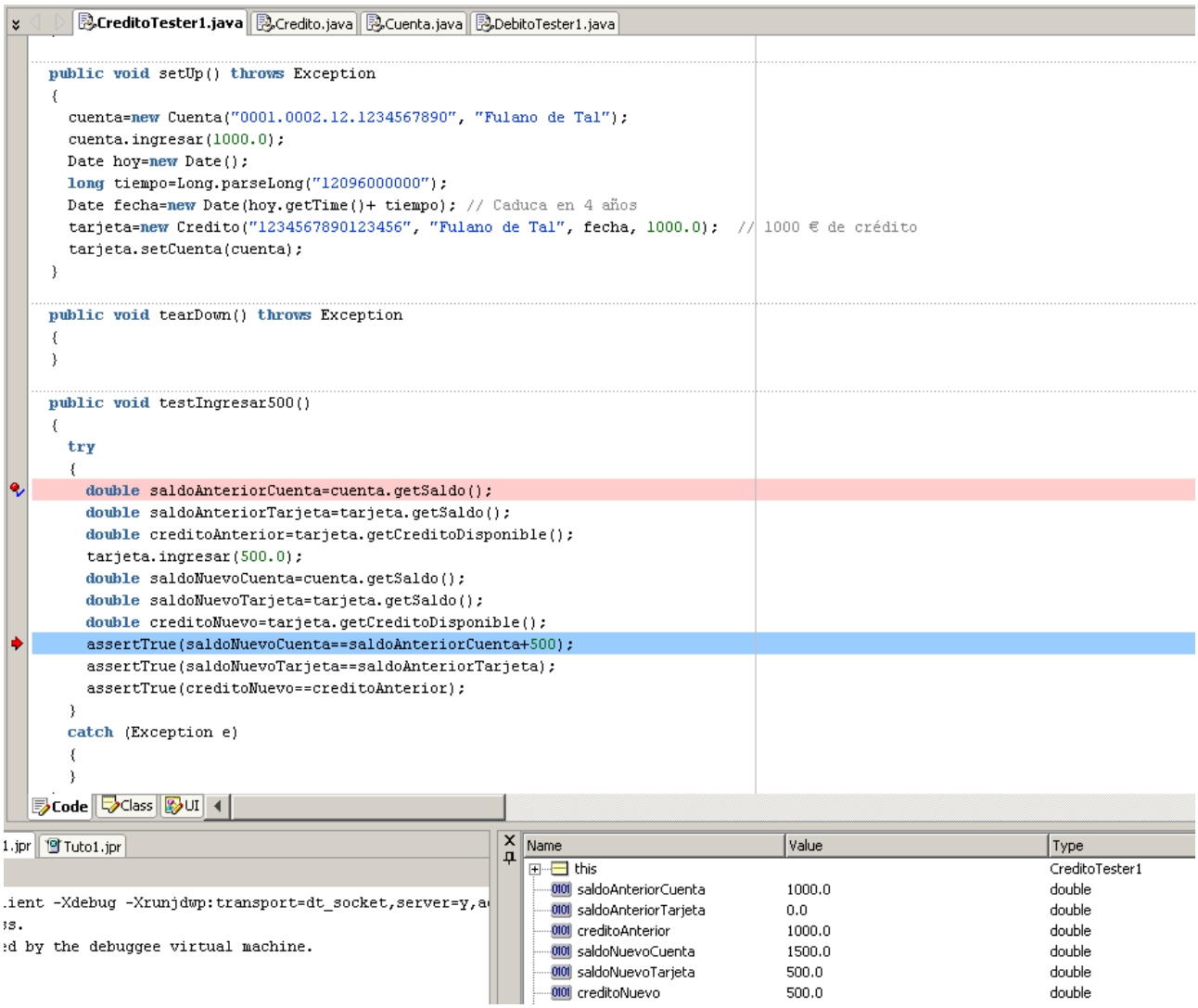
Cuando una tarjeta de crédito se liquida, se toman todos su movimientos del mes, se suman los importes y se hace el cargo a la cuenta asociada correspondiente. En este método pruebo esta funcionalidad.

Este mes he pagado 350 euros en dos compras.
Hago la liquidación.
Compruebo que el saldo actual de la cuenta es 350 euros inferior al que había antes de liquidar.

Los resultados de estos casos de prueba son:

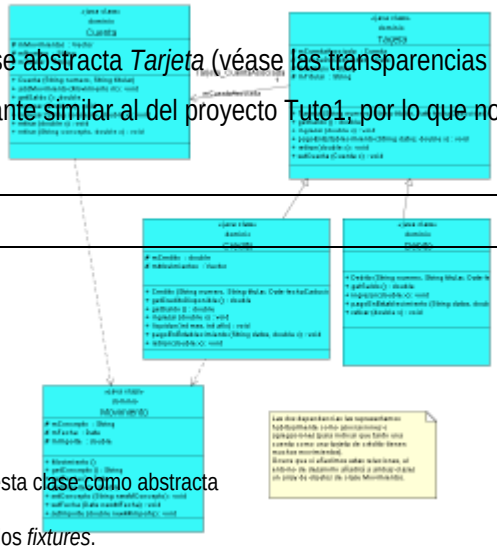


Es decir, funciona el último método (el encargado de hacer la liquidación), pero fallan los otros dos.
Podemos reescribir el método *testIngresar500* para facilitar el proceso de depuración:



En la figura anterior se muestra la nueva escritura que hemos dado al método y los valores de las variables. Se observa que falla el segundo *assertTrue*. Bueno, pues queda como ejercicio para el curioso lector la resolución de este error en el programa (el error ocurre porque el ingreso se realiza en la cuenta asociada, pero también se añade el movimiento a la tarjeta, lo cual es incorrecto).

Proyecto Tuto3.



En este ejemplo crearemos una clase abstracta para probar la clase abstracta *Tarjeta* (véase las transparencias 52 y 53 de la presentación). El código de las clases de dominio es bastante similar al del proyecto Tuto1, por lo que no se ofrece.

Prueba de la clase abstracta *Tarjeta* mediante una clase abstracta	
package dominio.test; import dominio.Cuenta; import dominio.Debito; import junit.framework.Test; import junit.framework.TestCase; import junit.framework.TestSuite; import dominio.Tarjeta; public **abstract** class TarjetaTester1 **extends TestCase** {	Declaramos esta clase como abstracta

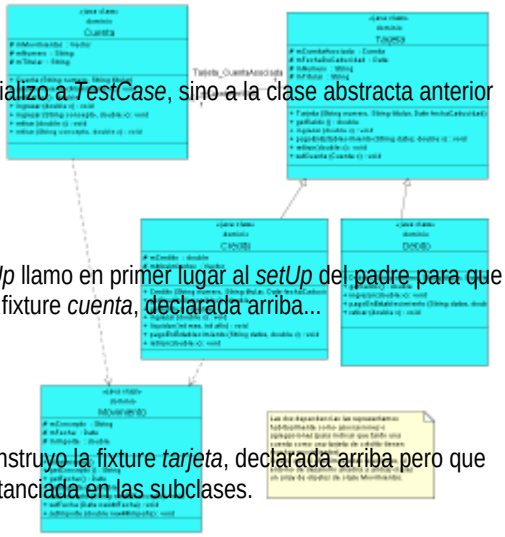
Tarjeta tarjeta;
Cuenta cuenta;
 	
public TarjetaTester1(String sTestName)
{
super(sTestName);
}
 	
public void setUp() throws Exception
{
cuenta=new Cuenta("0001.0002.12.1234567890", "Fulano de Tal");
cuenta.ingresar(1000.0);
}
	Declaramos dos *fixtures*.
	
	La *fixture* cuenta es concreta, así que la creamos; la otra es de un tipo abstracto, por lo que lo crearemos en las especializaciones de esta clase.

public abstract Tarjeta preparaTarjeta(Cuenta c);	Declaramos un método abstracto que me devuelve un objeto de clase *Tarjeta*, objeto que será adecuado para todas las especializaciones de esta clase de prueba.
public abstract Tarjeta tarjetaInicial();	Este método me devolverá la *fixture* tarjeta.

public void testRetirar300()
{
try
{
Tarjeta tarjeta=tarjetaInicial();
tarjeta.retirar(300.0);
 	
Cuenta c=new Cuenta("1234.5678.12.1234567891", "Paco Pil");
Tarjeta esperada=this.preparaTarjeta(c);
 	
assertTrue(tarjeta.getSaldo()==esperada.getSaldo());
}
catch (Exception e)
{
}
}
}
	Tanto con las tarjetas de débito como con las de crédito puedo retirar dinero. Por tanto, puede describirse esta funcionalidad mezclando operaciones concretas y abstractas, mediante lo que sería un patrón Método-plantilla.
	Obtengo la tarjeta inicial, que a veces será de débito y a ves de crédito.
	Retiro 300 euros.
	
	Obtengo la tarjeta que representa el resultado esperado, a la que asigno la cuenta *c*.
	Compruebo que el saldo de la tarjeta sobre la que he ejecutado la operación es el mismo que en la tarjeta esperada.

Prueba de la clase Debito mediante una especialización de la clase abstracta anterior	
package dominio.test; import dominio.Tarjeta; import java.util.Date; import junit.framework.Test; import junit.framework.TestCase; import junit.framework.TestSuite;	

```
import dominio.Debito;
import dominio.Cuenta;

public class DebitoTester2 extends TarjetaTester1
{
	public DebitoTester2(String sTestName)
	{
		super(sTestName);
	}

	public void setUp() throws Exception
	{
		super.setUp();
		Date hoy=new Date();
		long tiempo=Long.parseLong("12096000000");
		Date fecha=new Date(hoy.getTime()+ tiempo); // Caduca
en 4 años
		tarjeta=new Debito("1234567890123456", "Fulano de Tal",
fecha);
		tarjeta.setCuenta(cuenta);
	}

	public void tearDown() throws Exception
	{
	}

	public Tarjeta tarjetaInicial()
	{
		return tarjeta;
	}

	public Tarjeta preparaTarjeta(Cuenta c)
	{
		Tarjeta t=new Debito("9876543210987654", "Paco Pil",
null);
		t.setCuenta(cuenta);
		return t;
	}

	public static void main(String args[])
	{
		junit.swingui.TestRunner.run(DebitoTester2.class);
	}
}
```



Ya no especializo a *TestCase*, sino a la clase abstracta anterior

En este *setUp* llamo en primer lugar al *setUp* del padre para que construya la fixture *cuenta*, declarada arriba...

...y luego construyo la fixture *tarjeta*, declarada arriba pero que debe ser instanciada en las subclases.

Implemento este método, que era abstracto en la superclase

Me construyo una tarjeta ad hoc, de débito en este caso, a la que asocio la *fixture* cuenta declarada y creada en la superclase.

El TestRunner ejecuta los métodos test declarados en la superclase.

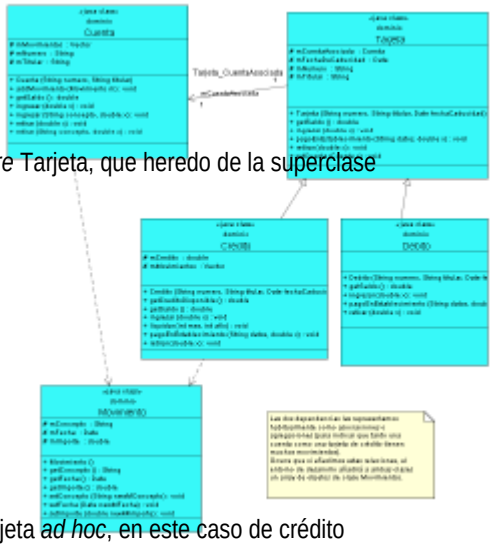
Prueba de la clase Credito mediante una especialización de la clase abstracta anterior

```
package dominio.test;
import dominio.Cuenta;
import dominio.Movimiento;
import dominio.Tarjeta;
import java.util.Date;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import dominio.Credito;

public class CreditoTester2 extends TarjetaTester1
{
	public CreditoTester2(String sTestName)
	{
		super(sTestName);
	}

	public void setUp() throws Exception
	{
```

Extiendo a TarjetaTester1, no a TestCase

```
        super.setUp();
        Date hoy=new Date();
        long tiempo=Long.parseLong("12096000000");
        Date fecha=new Date(hoy.getTime()+ tiempo); // Caduca
en 4 años
        tarjeta=new Credito("1234567890123456", "Fulano de Tal",
fecha, 1000.0);
        tarjeta.setCuenta(cuenta);
    }

    public void tearDown() throws Exception
    {
    }

    public Tarjeta tarjetaInicial()
    {
        return tarjeta;
    }

    public Tarjeta preparaTarjeta(Cuenta c)
    {
        Tarjeta t=new Credito("9876543210987654", "Paco Pil",
null, 1000.0);
        t.setCuenta(c);
        Movimiento m=new Movimiento();
        m.setImporte(-315.0);
        ((Credito) t).addMovimiento(m);
        return t;
    }

    public static void main(String args[])
    {
        junit.swingui.TestRunner.run(CreditoTester2.class);
    }
}
```

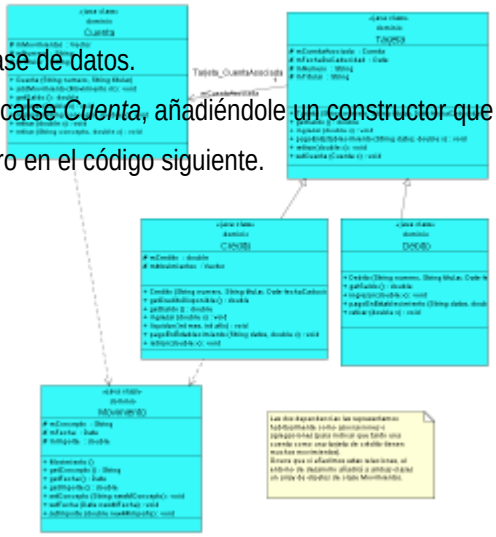
Creo la *fixture* Tarjeta, que heredo de la superclase



Creo una tarjeta *ad hoc*, en este caso de crédito

En este ejemplo, debemos ejecutar de forma separada las dos especializaciones de la clase de prueba abstracta.

Proyecto Tuto4.



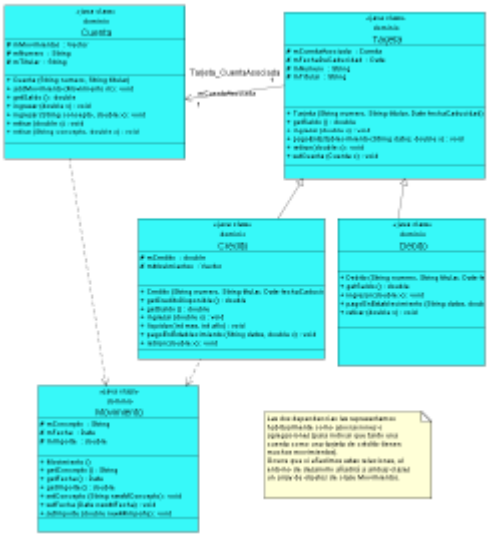
Este ejemplo utiliza objetos Mock para simular el acceso a auna base de datos.

En la capa de dominio solamente se ha modificado el código de la calse *Cuenta*, añadiéndole un constructor que crea instancias de esta clase a partir de la base de datos: véase recuadro en el código siguiente.

Clase Cuenta.

```
package dominio;

import java.util.Vector;
import java.sql.*;

public class Cuenta
{
	protected String mNumero;
	protected String mTitular;
	protected Vector mMovimientos;

	public Cuenta(String numero, Connection bd) throws Exception
	{
		String SQL="Select Numero, Titular from Cuenta where Numero=?";
		PreparedStatement p=bd.prepareStatement(SQL);
		p.setString(1, numero);
		ResultSet r=p.executeQuery();
		boolean enc=false;
		if (r.next())
		{
			mNumero=r.getString(1);
			mTitular=r.getString(2);
			enc=true;
		}
		p.close();
		if (!enc)
			throw new Exception("No se encuentra la cuenta " + numero);
	}

	public Cuenta(String numero, String titular)
	{
		...
	}

	public void ingresar(double x) throws Exception
	{
		...
	}

	public void retirar(double x) throws Exception
	{
		...

	}

	public void ingresar(String concepto, double x) throws Exception
```

```
    {
        ...
    }

    public void retirar(String concepto, double x) throws Exception
    {
        ...

    }

    public double getSaldo()
    {
        …
    }

    public void addMovimiento(Movimiento m)
    {
        ...
    }
}
```



Prueba de la clase Cuenta mediante objetos Mock	
package dominio.test; **import com.mockobjects.sql.MockPreparedStatement;** **import com.mockobjects.sql.MockMultiRowResultSet;** import junit.framework.Test; import junit.framework.TestCase; import junit.framework.TestSuite; import dominio.Cuenta; **import com.mockobjects.sql.MockConnection2;**	Importamos las clases Mock que necesitemos Importamos la *MockConnection2* en lugar de la *MockConnection* normal, que parece ser que no funciona correctamente.

public class CuentaConMock extends TestCase
{
Cuenta cuenta;
MockConnection2 bd;
 	
public CuentaConMock(String sTestName)
{
super(sTestName);
}
	La clase es una especialización de *TestCase* Ponemos dos *fixtures:* una cuenta y una “falsa” conexión a base de datos.

public void setUp() throws Exception
{
bd=new MockConnection2();
}
	En el *setUp* escribimos el código de conexión a la base de datos.

public void tearDown() throws Exception
{
bd.close();
}
	En el *tearDown* el código para cerrar la conexión.

public void testMaterializarCuenta() throws Exception {	En este método hacemos uso de los objetos Mock para probar el funcionamiento del nuevo constructor que hemos añadido a *Cuenta.* Puesto que el constructor crea la instancia mediante la ejecución de una

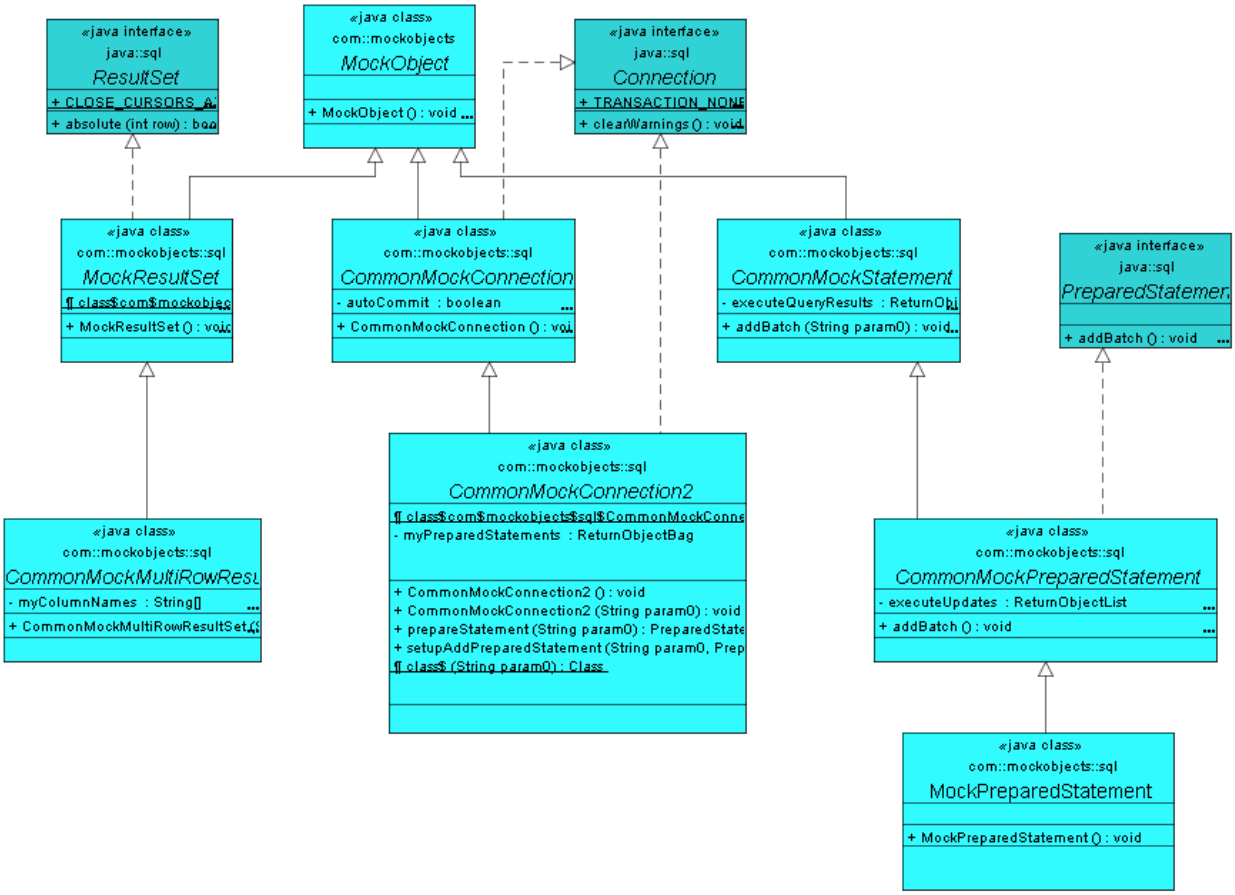
MockPreparedStatement p=new MockPreparedStatement(); bd.setupAddPreparedStatement("Select Numero, Titular from Cuenta where Numero=?", p);	*PreparedStatement* y un *ResultSet*, utilizaremos en el código de la clase de prueba una falsa *PreparedStatement* y un falso *ResultSet*. En primer lugar creamos un *MockPreparedStatement*, al que decimos con qué código SQL se va a rellenar en la clase de dominio. También informamos a la *MockConnection2* de que dicha *MockPreparedStatement* va a ser ejecutada sobre ella.
p.addExpectedSetParameter(1, "0001.0002.12.1234567890");	Decimos a la MockPreparedStatement qué valor de parámetro esperamos que se le pase.
MockMultiRowResultSet r=new MockMultiRowResultSet(); r.setupColumnNames(new String[] {"Numero", "Titular"});	Creo un "falso" ResultSet, al que digo que va a leer dos columnas (Número y Titular: véase el código del nuevo constructor de *Cuenta*).

Object[][] data=new Object[][] {
{"0001.0002.12.1234567890", "Fulano de Tal"}
};
r.setupRows(data);
	Creo una matriz con los objetos que voy a recuperar en el ResultSet (evidentemente, al ser "falsa" la conexión, no hay base de datos, por lo que alimento el ResultSet con los datos que a mí me interesen para esta prueba).

r.setExpectedNextCalls(2);	Le digo al "falso" ResultSet cuántas veces espero que se ejcute su método *next()*: en total dos.
p.addResultSet(r); cuenta=new Cuenta("0001.0002.12.1234567890", bd);	Asigno este ResultSet a la PreparedStatement. Materializo la cuenta pasando la falsa conexión a la base de datos como parámetro.

bd.verify();
}
public static void main(String args[])
{
junit.swingui.TestRunner.run(CuentaConMock.class);
}
}
	Ejecuto el método *verify* sobre la MockConnection2.

Todas las clases *MockX* son especializaciones de *X*: por ejemplo, *MockPreparedStatement* es una *java.sql.PreparedStatement.* En la siguiente figura vemos la estructura de las tres clases mock utilizadas en el ejempllo anterior:


«java interface»
java::sql
ResultSet
+ CLOSE_CURSORS_A
+ absolute (int row) : boo
«java class»
com::mockobjects
MockObject
+ MockObject () : void ...
«java interface»
java::sql
Connection
+ TRANSACTION_NONE
+ clearWarnings () : void
«java class»
com::mockobjects::sql
MockResultSet
¶ classcommockobjec
+ MockResultSet () : voi
«java class»
com::mockobjects::sql
CommonMockConnection
- autoCommit : boolean ...
+ CommonMockConnection () : voi
«java class»
com::mockobjects::sql
CommonMockStatement
- executeQueryResults : ReturnObj
+ addBatch (String param0) : void
«java interface»
java::sql
PreparedStatemer
+ addBatch () : void ...
«java class»
com::mockobjects::sql
CommonMockConnection2
¶ classcommockobjectssqlCommonMockConns
- myPreparedStatements : ReturnObjectBag
+ CommonMockConnection2 () : void
+ CommonMockConnection2 (String param0) : void
+ prepareStatement (String param0) : PreparedState
+ setupAddPreparedStatement (String param0, Prep
¶ class$ (String param0) : Class
«java class»
com::mockobjects::sql
CommonMockMultiRowResu
- myColumnNames : String[] ...
+ CommonMockMultiRowResultSet
«java class»
com::mockobjects::sql
CommonMockPreparedStatement
- executeUpdates : ReturnObjectList ...
+ addBatch () : void ...
«java class»
com::mockobjects::sql
MockPreparedStatement
+ MockPreparedStatement () : void