El concepto de Mockito Stub es uno de los conceptos más habituales a nivel de Test Driven Development cuando usamos Mockito. La idea de Stub no es propietaria de Mockito sino que pertenece a TDD y hace referencia a una clase que simula ser otra pero solo implementa algunos de los métodos de esta última. Aquellos que nos son interesantes a la hora de abordar pruebas unitarias. Vamos a explicar este concepto a nivel básico basándonos en dos clases . La clase Factura y la clase LineaFactura.

Vamos a ver su código:

```
package com.arquitecturajava.mockito;

import java.util.ArrayList;
import java.util.List;

public class Factura {

	private int numero;
	private String concepto;
	private List<LineaFactura> lineas= new
ArrayList<LineaFactura>();
	public int getNumero() {
		return numero;
	}
	public void setNumero(int numero) {
```

```
                this.numero = numero;
        }
        public String getConcepto() {
                return concepto;
        }
        public void setConcepto(String concepto) {
                this.concepto = concepto;
        }
        public void addLinea(LineaFactura lf) {
                this.lineas.add(lf);
        }
        public void removeLinea(LineaFactura lf) {
                this.lineas.remove(lf);
        }
        public Factura(int numero, String concepto, List<LineaFactura>
lineas) {
                super();
                this.numero = numero;
                this.concepto = concepto;
                this.lineas = lineas;
        }
        public Factura(int numero, String concepto) {
                super();
                this.numero = numero;
                this.concepto = concepto;
        }
        public double getImporteConIVA() {
                double total=0;
                for (LineaFactura l:lineas) {
                        total+=l.getImporteConIVA();
                }
```

```
                return total;
        }
}
```

```
package com.arquitecturajava.mockito;

public class LineaFactura {

        private int numero;
        private double importe;
        private String concepto;
        public int getNumero() {
                return numero;
        }
        public void setNumero(int numero) {
                this.numero = numero;
        }
        public double getImporte() {
                return importe;
        }
        public void setImporte(double importe) {
                this.importe = importe;
        }
        public String getConcepto() {
                return concepto;
        }
        public void setConcepto(String concepto) {
                this.concepto = concepto;
        }
        public LineaFactura(int numero, String concepto,double
importe) {
                super();
```

```
                this.numero = numero;
                this.importe = importe;
                this.concepto = concepto;
        }
        public double getImporteConIVA() {
                return importe* 1.20;
        }
}
```

En este caso lo que queremos es construir un prueba unitaria que nos valide el importe con IVA de una LineaFactura. Se trata de una de las operaciones más sencillas. Para ello usaremos un proyecto Maven y añadiremos un par de dependencias.

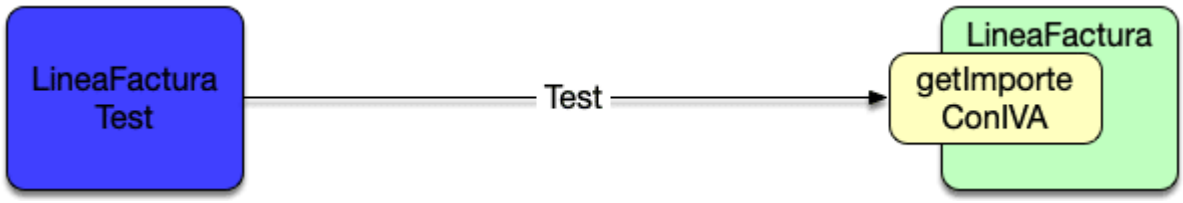
```
        <dependencies>

                <!--
https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api
-->
                <dependency>
                        <groupId>org.junit.jupiter</groupId>
                        <artifactId>junit-jupiter-api</artifactId>
                        <version>5.6.2</version>
                        <scope>test</scope>
                </dependency>
                <!--
https://mvnrepository.com/artifact/org.mockito/mockito-core -->
                <dependency>
                        <groupId>org.mockito</groupId>
                        <artifactId>mockito-core</artifactId>
                        <version>3.3.3</version>
                        <scope>test</scope>
                </dependency>
```

```
	</dependencies>
```

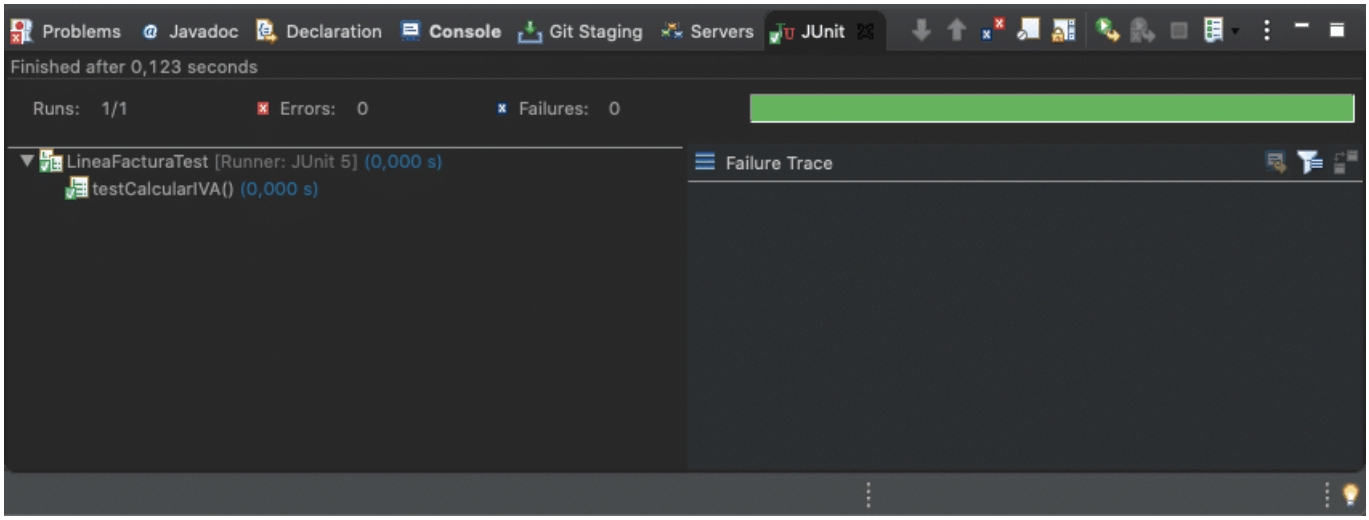
En este caso vamos a utilizar jUnit 5 y Mockito3. Es momento de mostrar el código de la primera prueba unitaria:

```
package com.arquitecturajava.mockito.test;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

import com.arquitecturajava.mockito.LineaFactura;

public class LineaFacturaTest {

	@Test
	public void testCalcularIVA() {
		LineaFactura linea1= new LineaFactura(1,"lampara",30);
		assertEquals(36.3,linea1.getImporteConIVA(),0);
	}
}
```

No tiene nada de peculiar ya que se encarga de comprobar que el IVA que es de un 21 % se calcule correctamente en la clase LineaFactura



El resultado es que el test pasa sin ningún tipo de problema:



Mockito Stub y Test

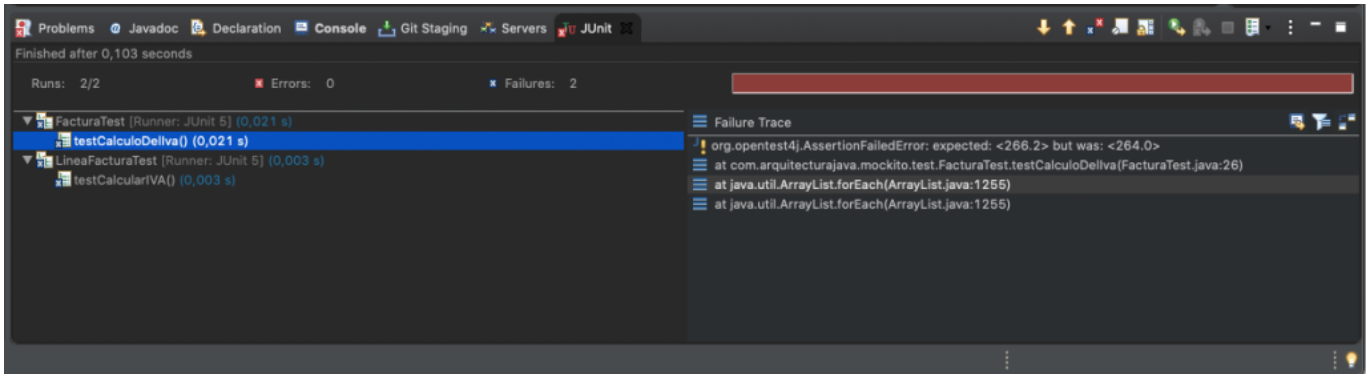
Es momento de realizar un test similar en la clase Factura:

```
@Test
void testCalculoDelIva() {
        LineaFactura l1= new LineaFactura(1,"tablet",200);
        LineaFactura l2= new LineaFactura(2,"lampara",20);
        List<LineaFactura> lineas= Arrays.asList(l1,l2);
        Factura f= new Factura(1,"compra online",lineas);
        assertEquals(266.2,f.getImporteConIVA(),0);
}
```

Si ejecutamos este test todo funcionará correctamente. Ahora bien tenemos un problema que al principio cuesta ver . ¿Qué sucede si nosotros modificamos la clase LineaFactura para que falle? . Por ejemplo cambiamos el método de getImporteConIVA() por lo siguiente:

```
public double getImporteConIVA() {
                return importe* 1.20;
        }
```

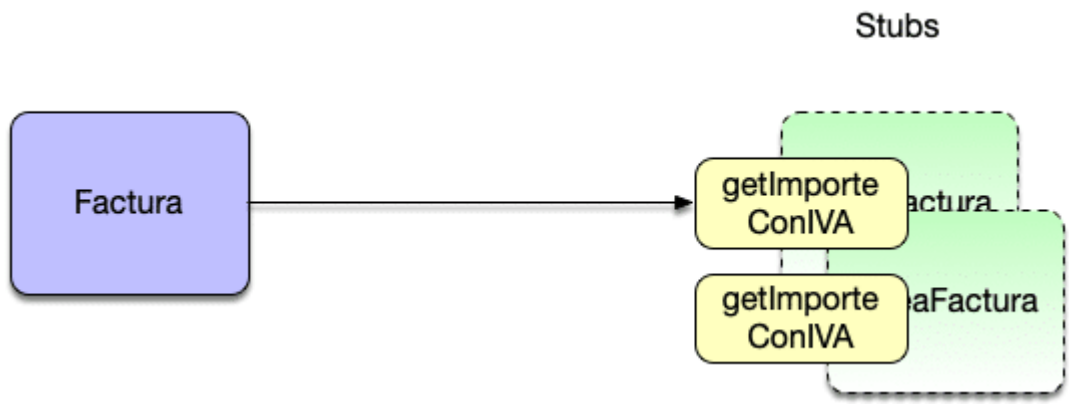
Es evidente que los test van a fallar porque el calculo es incorrecto.



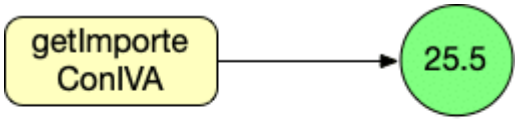
El problema viene al saber que es lo que ha fallado . Recordemos que el test que calcula el importe con IVA de la Factura depende de la funcionalidad de las lineas de Factura . Por lo tanto falla también .¿Qué es lo que tenemos que hacer? . Tenemos que aislar completamente cada test de tal forma al fallar uno no oblige a que el otro falle. Para ello vamos a usar el concepto de stub (o talon) . Un stub hace referencia a una clase que simula ser otra pero que solo tiene implementada una pequeña parte de su funcionalidad . Lo suficiente para gestionar el método al que invocamos. Veamoslo en código con Mockito.

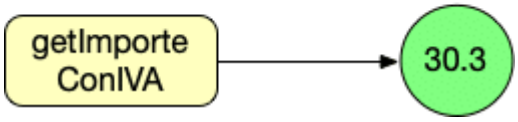
```
@Test
	void testCalculoDelIva2() {
		LineaFactura l1= mock(LineaFactura.class);
		LineaFactura l2= mock(LineaFactura.class);
		when (l1.getImporteConIVA()).thenReturn(25.5);
		when (l2.getImporteConIVA()).thenReturn(30.3);
		List<LineaFactura> lineas= Arrays.asList(l1,l2);
		Factura f= new Factura(1,"compra online",lineas);
		assertEquals(55.8,f.getImporteConIVA(),0);
	}
```

En este caso hemos usado el método mock para generar un Mockito Stub para la primera linea de factura y otro tanto para la segunda . Por lo tanto tenemos en memoria dos objetos prácticamente vacios que simulan ser objetos reales.

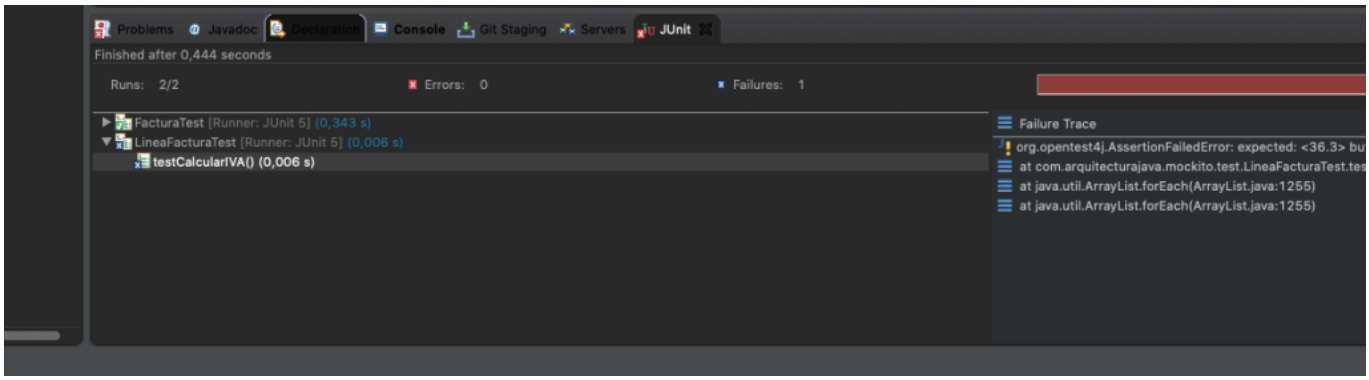


Les hemos configurado con Mockito a través del método thenReturn para que cuando solicitemos getImporteConIVA() nos devuelvan valores fijos en concreto 25.5 y 30.3.





Les añadimos a la lista y los pasamos a la Factura en su constructor. Por lo tanto ahora mismo la Factura contiene dos stubs en memoria. Si ejecutamos los test otra vez las cosas funcionarán mejor. Ya que solo falla el test del método que hemos modificado.



Acostumbremonos a usar Mockito Stub para mejorar el aislamiento de nuestros test.

Otros artículos relacionados

- Mockito
- Java Aserts y sus librerias
- Spring Testing
- Mockito Testing