

Pack de Cursos Gratis para Juniors



Accede ahora

arquitecturajava

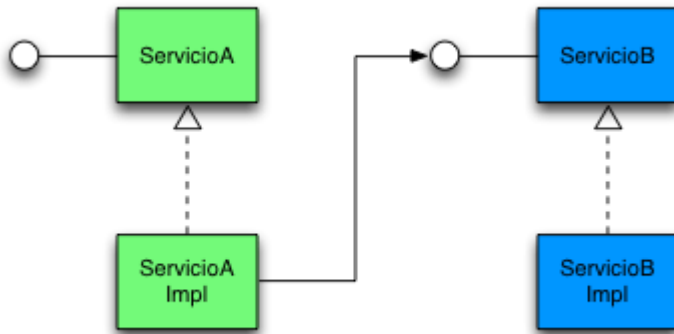
Pack de Cursos Gratis para Seniors



Accede ahora

arquitecturajava

Java Mockito es uno de los frameworks de Mock más utilizados en la plataforma Java. Esto es debido a la gran facilidad de uso que tiene. Vamos a intentar explicar el concepto de Mock y como funciona Java Mockito , para ello construiremos unos test sencillos partiendo de dos clases de Servicio que se encuentran relacionadas. Recordemos que muchas metodologías apoyan generar los test primero ,pero aquí nos vamos a centrar en los conceptos.



En este caso una clase se encarga de sumar dos números y la otra se encarga de multiplicarlos.

```
public interface ServicioA {

    public abstract int sumar(int a, int b);

}

package com.arquitecturajava.servicios;

public class ServicioAImpl implements ServicioA {

    public int sumar(int a ,int b) {

        return a+b+1;
    }
}
```

```
package com.arquitecturajava.servicios;

public interface ServicioB {

    public ServicioA getServicioA();

    public void setServicioA(ServicioA servicioA);

    public int multiplicarSumar(int a, int b, int multiplicador);

    public int multiplicar(int a, int b);
```

```
package com.arquitecturajava.servicios;

public class ServicioBImpl implements ServicioB {

    private ServicioA servicioA;

    public ServicioA getServicioA() {
        return servicioA;
    }

    public void setServicioA(ServicioA servicioA) {
        this.servicioA = servicioA;
    }

    public int multiplicarSumar(int a ,int b,int multiplicador) {
```

```

    return servicioA.sumar(a, b)*multiplicador;

}

public int multiplicar(int a ,int b) {

    return a*b;
}
}

```

Como podemos observar ambas clases disponen de un método que se puede testear de forma aislada ServicioA (sumar) y ServicioB (multiplicar). Vamos a instalar JUnit y Mockito como dependencias de Maven para poder trabajar.

```

&lt;project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"&gt;
&lt;modelVersion&gt;4.0.0&lt;/modelVersion&
&lt;groupId&gt;com.arquitecturajava&lt;/groupId&
&lt;artifactId&gt;mocks&lt;/artifactId&
&lt;version&gt;0.0.1-
SNAPSHOT&lt;/version&
&lt;name&gt;mocks&lt;/name&
&lt;dependencies&

```

```

&lt;dependency&gt;
&lt;groupId&gt;junit&lt;/groupId&gt;
&lt;artifactId&gt;junit&lt;/artifactId&gt;
&lt;version&gt;4.12&lt;/version&gt;
&lt;/dependency&gt;
&lt;dependency&gt;
&lt;groupId&gt;org.mockito&lt;/groupId&gt;
&lt;artifactId&gt;mockito-
core&lt;/artifactId&gt;
&lt;version&gt;1.10.19&lt;/version&gt;
&lt;/dependency&gt;
&lt;/dependencies&gt;
&lt;/project&gt;

```

Construimos los test :

```

package com.arquitecturajava.test;

import org.junit.Assert;
import org.junit.Test;

import com.arquitecturajava.servicios.ServicioA;
import com.arquitecturajava.servicios.ServicioAImpl;

public class TestServicioA {
    @Test

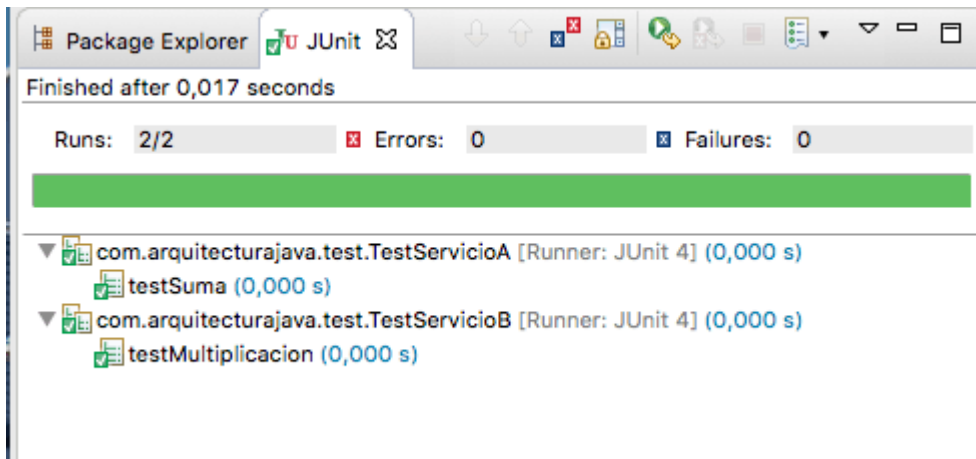
```

```
public void testSuma() {  
  
    int a=2;  
    int b=2;  
    ServicioA servicio= new ServicioAImpl();  
    Assert.assertEquals(servicio.sumar(a, b),4);  
  
}  
}
```

```
package com.arquitecturajava.test;  
  
import org.junit.Assert;  
import org.junit.Test;  
  
import com.arquitecturajava.servicios.ServicioA;  
import com.arquitecturajava.servicios.ServicioAImpl;  
import com.arquitecturajava.servicios.ServicioB;  
import com.arquitecturajava.servicios.ServicioBImpl;  
  
public class TestServicioB {  
  
    @Test  
    public void testMultiplicacion() {  
  
        ServicioB servicioB= new ServicioBImpl();  
        Assert.assertEquals(servicioB.multiplicar(2, 3),6);  
    }  
}
```

```
}
```

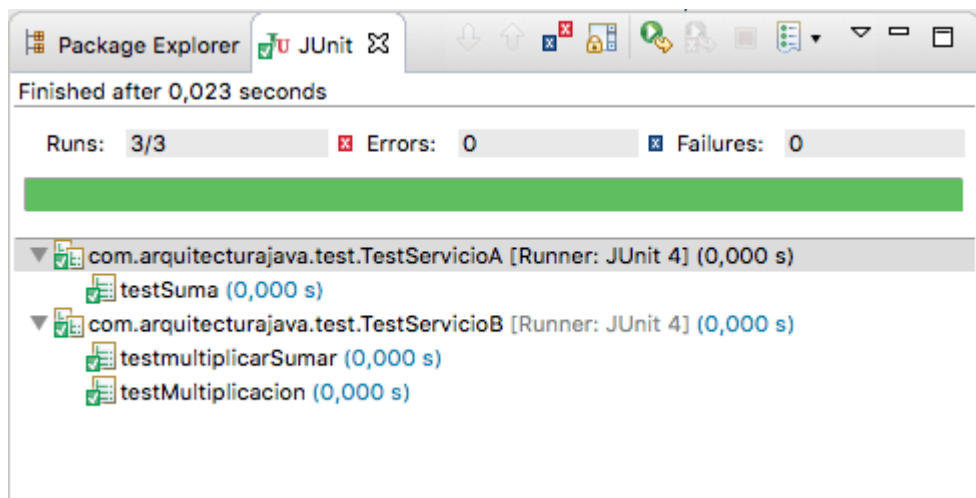
Ejecutamos ambos Test con JUnit :



Ahora bien ¿qué pasará si tenemos que testear el método sumarMultiplicar que tiene el ServicioB?. Este método primero suma dos números delegando en el ServicioA y luego los multiplica.

@Test

```
public void testmultiplicarSumar() {  
  
    ServicioA servicioA=new ServicioAImpl();  
  
    ServicioB servicioB= new ServicioBImpl();  
  
    servicioB.setServicioA(servicioA);  
    Assert.assertEquals(servicioB.multiplicarSumar(2, 3, 2),10);  
  
}
```



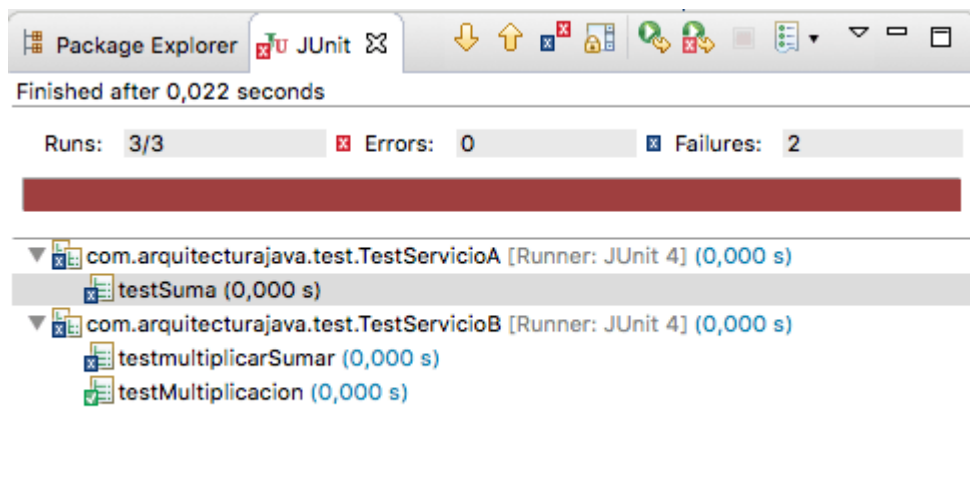
Todo funciona correctamente. Los problemas aparecen cuando yo modifiko el método sumar del ServicioA y le asigno el siguiente código:

```
package com.arquitecturajava.servicios;

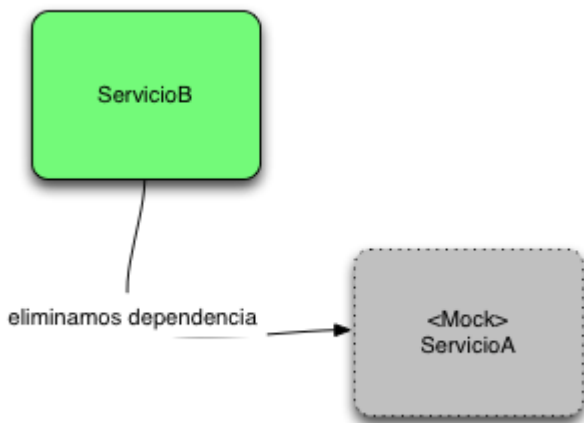
public class ServicioAImpl implements ServicioA {
    public int sumar(int a ,int b) {

        return a+b+1;
    }
}
```

Los test fallarán:



Fallan dos test y no uno solo, esto es un problema importante ya que los desarrolladores no sabrán que esta pasando exactamente. Para evitar este tipo de problemas debemos aislar las pruebas unitarias y una de las opciones es usar Java Mockito y crear un Mock Object. Los objetos Mock nos permiten simular ser un objeto real y eliminan dependencias , permitiendo que los test se ejecuten de forma aislada.



Vamos a ver el código:

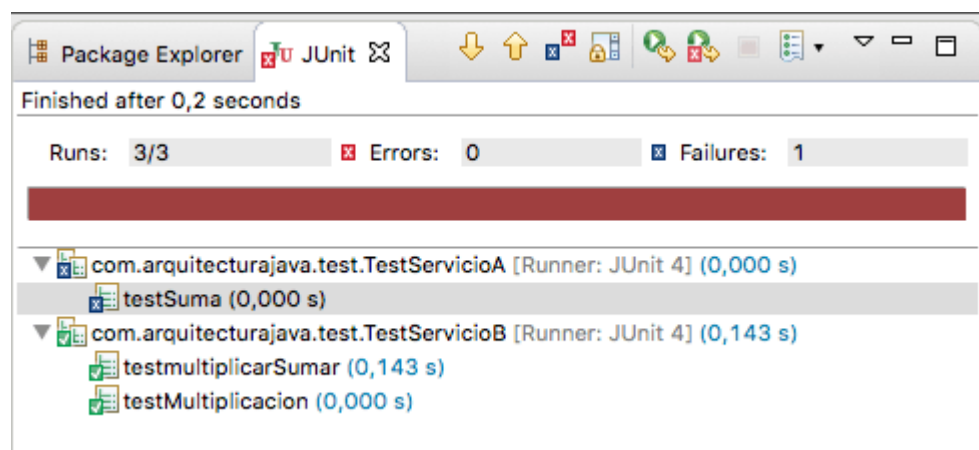
```
@Test
public void testmultiplicarSumar() {
```

```
ServicioA servicioA=mock(ServicioA.class);
when(servicioA.sumar(2,3)).thenReturn(5);

ServicioB servicioB= new ServicioBImpl();
servicioB.setServicioA(servicioA);
Assert.assertEquals(servicioB.multiplicarSumar(2, 3, 2),10);

}
```

Hemos creado un mock que simula ser el ServicioA para la operación de suma(2,3) y devuelve como resultado 5. Si volvemos a ejecutar los test:



Ahora solo falla un test , hemos conseguido aislar cada uno de ellos y los desarrolladores podrán encontrar los errores de forma fácil. Para ello hemos construido un objeto Mock utilizando Java Mockito.

Otros artículos relacionados:

[Java Proxy](#) ,

[Java Annotations](#)