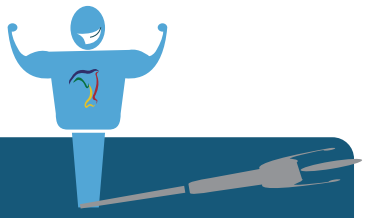
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

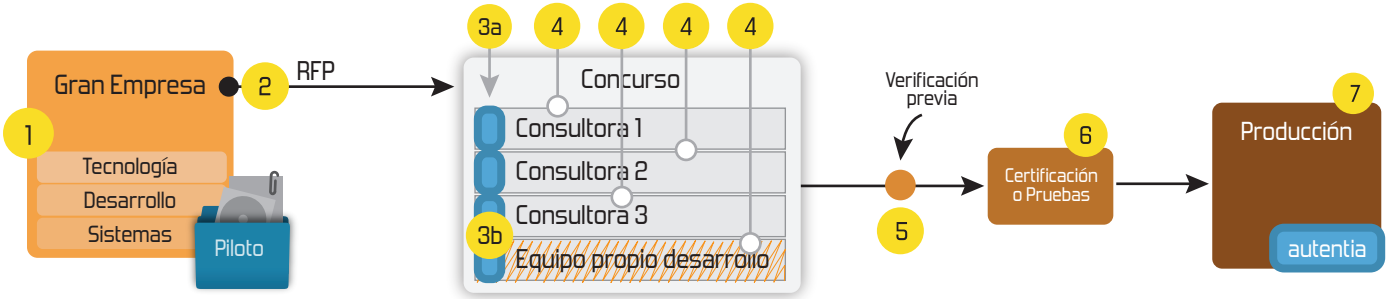
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

- Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery
- Gestor portales (Liferay)
- Gestor de contenidos (Alfresco)
- Aplicaciones híbridas
- Control de autenticación y acceso (Spring Security)
- UDDI
- Web Services
- Rest Services
- Social SSO
- SSO (Cas)
- JPA-Hibernate, MyBatis
- Motor de búsqueda empresarial (Solr)
- ETL (Talend)
- Dirección de Proyectos Informáticos.
- Metodologías ágiles
- Patrones de diseño
- TDD
- Tareas programadas (Quartz)
- Gestor documental (Alfresco)
- Inversión de control (Spring)
- BPM (jBPM o Bonita)
- Generación de informes (JasperReport)
- ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Jose Manuel Sánchez Suárez

Consultor tecnológico de desarrollo de proyectos informáticos.

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/J2EE

Ver todos los tutoriales del autor

Fecha de publicación del tutorial: 2013-12-20 Tutorial visitado 1 veces Descargar en PDF

¿Mockear métodos estáticos?, con el soporte de PowerMock.

0. Índice de contenidos.

1. Introducción

PowerMock es un framework que extiende tanto EasyMock como Mockito complementándolos y añadiendo la posibilidad de:

- mockear invocaciones a métodos estáticos,
- mockear clases y métodos marcados como finales,
- acceder a la verficación del estado de atributos privados,
- eliminación de inicializadores estáticos,
- ...

Para conseguirlo usa su propio classloader, instrumentalizando el código con el soporte de javassist y podemos ejecutarlo tanto con Junit como con TestNG.

Si ya estamos usando Mockito, empezar a usar PowerMock implica continuar con la misma filosofía pero una potencia adicional a la hora de ampliar nuestras posibilidades de mockear clases. No obstante lo anterior, no se recomienda su uso de forma generalizada, vamos que "puede tener más peligro que Golum en una joyería" sino lo usamos en su justa medida, ya sabéis aquello de, "la potencia sin control no sirve de nada".

En este tutorial vamos a ver qué necesitamos para hacer uso de PowerMock, cómo mockear una clase estática para solventar una dependencia heredada y, como consecuencia, hablaremos de inyección de dependencias y diseño orientado a objetos.

2. Entorno.

El tutorial, y el código que contiene, han sido escritos usando el siguiente entorno:

- Hardware: Portátil MacBook Pro 15' (2.4 GHz Intel Core i7, 8GB DDR3 SDRAM).
- Sistema Operativo: Mac OS X Lion 10.7.5
- PowerMock 1.5.2
- Mockito 1.9.5
- JUnit 4.11

3. Configuración.

Lo primero que tenemos que decidir es que implementación de PowerMock vamos a instalar, para Mockito o EasyMock, dependiendo de la librería que estemos usando. Nosotros vamos a hacer uso de la de Mockito.

Como siempre, presuponemos que hacemos uso de una herramienta de automatización de tareas y control del ciclo de vida de nuestro proyecto, de este modo, con el soporte de maven solo necesitaríamos añadir las siguientes dependencias a nuestro pom.xml:

```
<properties>
    <powermock.version>1.5.2</powermock.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.powermock</groupId>
        <artifactId>powermock-module-junit4</artifactId>
        <version>${powermock.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.powermock</groupId>
        <artifactId>powermock-api-mockito</artifactId>
        <version>${powermock.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>1.8.5</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Maven se encargará de incluir las dependencias al ejecutar los tests, así hemos configurado el scope, no solo las establecidas sino las dependencias indirectas.

Sin el soporte de Maven tendríamos que añadir las librerías manualmente a nuestro proyecto de recursos:

- junit-4.11.jar
- hamcrest-core-1.1.jar
- mockito-all-1.9.5.jar
- powermock-all-1.9.5.jar
- powermock-mockito-1.5.2-full.jar
- javassist-3.18.0-GA.jar

descargándolas de la zona de descargas de PowerMock.

4. Encapsulando llamadas a métodos estáticos.

Y ahora que lo tenemos configurado... no vamos a usarlo!.

Vamos a presuponer que tenemos la siguiente clase de contexto dentro de nuestro framework de desarrollo:

```
package com.autentia.tutoriales.powermock;

import java.net.MalformedURLException;
import java.net.URL;

public class LegacyContext {

    private static URL url;

    static{
        try {
            url = new URL("http://192.168.1.11/");
        } catch (MalformedURLException e) {
            throw new IllegalStateException(e);
        }
    }

    public static URL getUrl() {
        return url;
    }

}
```

Lo de menos es su lógica de negocio, la idea es mostrar que tenemos una dependencia con una clase heredada que expone cierta funcionalidad a través de un método estático y esta clase es de infraestructura con lo que no podemos modificarla.

La recomendación es encapsular la invocación a esa clase en un wrapper de aplicación de modo tal que evitemos tener que mockear una invocación a un método estático.

Imagina que necesitamos acceder al Contexto para componer URLs, podemos entonces preparar una interfaz:

```
package com.autentia.tutoriales.powermock;

public interface URLComposer {

    String getWSServicesURL();

    String getRestServicesURL();

}
```

Y una clase de implementación que encapsule las invocaciones a métodos estáticos

» Hello World en IOS sin StoryBoard

» Cómo integrar un Job de Talend a nuestro proyecto Java

Últimos Tutoriales del Autor

» Integración de la gestión de proyecto de Redmine en Eclipse con el soporte de Mylyn.

» Omnifaces: una librería de utilidades para JSF2

» JUnit test runners

» Ejecución de un análisis en sonar con el soporte de una tarea ant.

» Ejecución de tests de integración en aplicaciones OSGI con el soporte de Arquilian.

Últimas ofertas de empleo

2011-09-08
Comercial - Ventas - MADRID.

2011-09-03
Comercial - Ventas - VALENCIA.

2011-08-19
Comercial - Compras - ALICANTE.

2011-07-12
Otras Sin catalogar - MADRID.

2011-07-06
Otras Sin catalogar - LUGO.

```java
package com.autentia.tutoriales.powermock;

public class LegacyURLComposer implements URLComposer{

    private static LegacyURLComposer instance;

    public LegacyURLComposer(){

    }

    public static LegacyURLComposer getInstance(){
        if (instance == null){
            instance = new LegacyURLComposer();
        }
        return instance;
    }

    public String getWSServicesURL() {
        return LegacyContext.getUrl() + "ws/services/";
    }

    public String getRestServicesURL() {
        return LegacyContext.getUrl() + "rest/services/";
    }

}
```

De este modo, en cualquier clase o servicio de negocio podríamos disponer una dependencia del contrato, de la interfaz que encapsula la lógica de obtención de las URLs y, a su vez, la invocación a los métodos estáticos, bien asignándola a través de constructor o propiedad.

```java
package com.autentia.tutoriales.powermock;

public class OurBusinessService {

    private URLComposer urlComposer;

    public OurBusinessService(URLComposer urlComposer){
        this.urlComposer = urlComposer;
    }

    public String doSomeThingWithTheURL() {
        System.out.println(urlComposer.getWSServicesURL());
        return "ok";
    }

}
```

Así, en nuestros tests, haciendo uso de mockito y sin necesidad de "elevar" su soporte a PowerMock podríamos resolver la dependencia:

```java
package com.autentia.tutoriales.powermock;

import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

import org.junit.Before;
import org.junit.Test;

public class OurBusinessServiceTest {

    private OurBusinessService businessService;

    @Before
    public void setUp(){
        final URLComposer urlComposer = mock(URLComposer.class);
        when(urlComposer.getWSServicesURL()).thenReturn("http://localhost/ws/services/");
        businessService = new OurBusinessService(urlComposer);
    }
    @Test
    public void testThatBusinessServiceMethodDontThrowsAnError(){
        businessService.doSomeThingWithTheURL();
    }
}
```

Nuestro test resuelve la inyección de la dependencia de negocio, sustituyendo la estrategia de resolución de URLs en el entorno de tests por un mock. También estamos haciendo nuestro código extensible y cerrado a modificaciones.

Quién necesite hacer uso de nuestra clase de servicio tendrá la responsabilidad de crear una instancia de la clase de implementación de la interfaz URLComposer; si estamos trabajando con el soporte de un framework de inyección de dependencias (Spring, CDI, Google Guice, PicoContainer,...), dicha inyección la configuraremos en la estrategia del contexto de IoC.

Pero... este no era un tutorial de PowerMock... ;)

5. Mockeando un método estático.

Si, aún lo anterior, seguimos con la necesidad de mockear la llamada a un método estático podemos hacer uso de un runner de junit que PowerMock pone a nuestra disposición para correr un test con su soporte.

```java
@RunWith(PowerMockRunner.class)
@PrepareForTest( { LegacyContext.class})
public class URLComposerTest {
```

En la anotación @PrepareForTest debemos marcar las clases con métodos estáticos que queremos instrumentalizar o mockear.

Una vez marcados, en el método anotado con @Before poder usar los métodos estáticos de PowerMockito para crear un mock estático y mockear sus métodos estáticos; a continuación, un ejemplo:

```java
package com.autentia.tutoriales.powermock;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.when;

import java.net.MalformedURLException;
import java.net.URL;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.powermock.api.mockito.PowerMockito;
import org.powermock.core.classloader.annotations.PrepareForTest;
import org.powermock.modules.junit4.PowerMockRunner;

@RunWith(PowerMockRunner.class)
@PrepareForTest( { LegacyContext.class})
public class URLComposerTest {

    @Before
    public void setUp() throws MalformedURLException {
        PowerMockito.mockStatic(LegacyContext.class);
        when(LegacyContext.getUrl()).thenReturn(new URL("http://localhost/"));
    }

    @Test
    public void testMockStaticComposer() throws Exception {
        assertEquals("http://localhost/ws/services/", LegacyURLComposer.getInstance().getU
    }

}
```

Eso es todo y también podemos crear stubs, verificadores, usar los matchers de mockito,... y, con ello, darle rienda a nuestra imaginación, ¿no?

Nadie dice que tu diseño no pueda tener clases con métodos estáticos, como todo, según para qué. En general, las clases con métodos estáticos responden más a la necesidad de disponer de librerías de funciones con variables estáticas, típico de lenguajes estructurados, que a un diseño orientado a objetos.

Aún lo anterior, `java.lang.Math` es una clase con el 100% de métodos estáticos y un constructor privado; idem con `org.apache.commons.lang.StringUtils`.

Ahora, como puedo probarlo, ¿toda mi funcionalidad se va a basar en métodos estáticos?, no!!!, debes tener en cuenta que en una clase estática:

- no podemos usar polimorfismo,
- si marcamos la clase para implementar una interfaz los métodos estáticos no forman parte del contrato,
- no se puede clonar, ni serializar, y
- no es susceptible de ser inicializada de forma tardía o lazy, es estática y se crea con la carga de clases,

Un Helper o una clase de utilidades que no necesita guardar el estado de los objetos sería la justificación para tener una clase con métodos estáticos, para todo lo demás, patrones de diseño en orientación a objetos.

6. Referencias.

- http://code.google.com/p/powermock/
- http://stackoverflow.com/questions/519520/difference-between-static-class-and-singleton-pattern
- http://www.jramoyo.com/2013/03/static-methods-and-unit-testing.html

7. Conclusiones.

Si no hay otra, no hay otra y, por poder, podemos mockear lo que se nos ponga por delante ;)

Un saludo.

Jose

jmsanchez@autentia.com