

UNIVERSIDAD DE BURGOS

ESCUELA POLITÉCNICA SUPERIOR



Grado en Ingeniería Informática

Desarrollo avanzado de sistemas software

PG01 - Medir para caracterizar entidades de productos y procesos software

Alumnos	David Atienza González Alejandro Revilla Gistaín
Tutor	Carlos López Nozal DEPARTAMENTO DE INGENIERÍA CIVIL Área de Lenguajes y Sistemas Informáticos

Burgos, 27 de febrero de 2014



Este documento está licenciado bajo [Creative Commons Attribution 3.0 License](https://creativecommons.org/licenses/by/3.0/)

Índice de contenido

1	Objetivos	3
2	Enunciado	3
	2.1 Descripción del producto.....	3
	2.2 Descripción del proceso.....	3
3	Desarrollo	4
4	Cuestiones	10

Índice de ilustraciones

Ilustración 1: Creación de la clase de pruebas.....	4
Ilustración 2: Cobertura para el método getInstance().....	5
Ilustración 3: Cobertura para el método acquireReusable().....	7
Ilustración 4: Cobertura para el método testReleaseReusable().....	10
Ilustración 5: Commits realizados en el repositorio.....	10

Índice de tablas

Tabla 3.1: Estructura base de la clase de pruebas.....	5
Tabla 3.2: Método testGetInstance().....	5
Tabla 3.3: Métodos testAcquireReusable() y setUp().....	6
Tabla 3.4: Método testReleaseReusable().....	9
Tabla 4.1: Modificación sugerida a releaseReusable().....	11

1 OBJETIVOS

- Comprender los objetivos de medición relacionados con la caracterización y la evaluación de productos, procesos y recursos software
- Comprender y aplicar técnicas de medición sobre entidades de productos software relacionados con conjuntos de pruebas de software
- Comprender, aplicar y analizar medidas relacionadas sobre entidades de proceso y recursos de prueba del software

2 ENUNCIADO

En la práctica se va a simular un pequeño desarrollo de un producto software para realizar mediciones sobre él. El objetivo es establecer un caso de estudio que sirva para caracterizar y evaluar tanto el producto desarrollado como el proceso seguido.

2.1 Descripción del producto

Dado un código de ejemplo del patrón diseño creacional `Object Pool`, se debe crear una batería de pruebas tal que las coberturas de sus clases sean del 100%. El código de las clases se puede obtener en el repositorio <https://github.com/clopezno/poolobject>. La batería de pruebas Junit debe estar contenida en la clase `ubu.gii.dass.test.c01.ReuseblePoolTest.java`.

2.2 Descripción del proceso

El proceso de desarrollo de la batería de pruebas se va a gestionar utilizando el control de versiones del sistema Git proporcionado por el repositorio de proyectos GitHub (<https://github.com>).

Los pasos para gestionar el procesos son los siguientes:

1. Cada miembro del equipo tiene que estar registrado en GitHub.
2. Uno de los miembros tiene que realizar un **fork del repositorio** donde se encuentra el código que se quiere probar <https://github.com/clopezno/poolobject>. El nuevo repositorio tiene que ser público.
3. Invitar al resto de miembros del equipo para que puedan participar en el desarrollo del conjunto de pruebas.
4. Cada nuevo test realizado ejecutar un `commit/push` al repositorio del grupo. El texto del `commit` tiene que describir el caso de prueba añadido

3 DESARROLLO

Nuestra clase de pruebas denominada `ReusablePoolTest.java` va a contener un método con pruebas por cada uno de los métodos de la clase a la que hace referencia (`ReusablePool.java`). Para ello seleccionamos todos los métodos en el momento de crear la clase de prueba (JUnit Test Case).

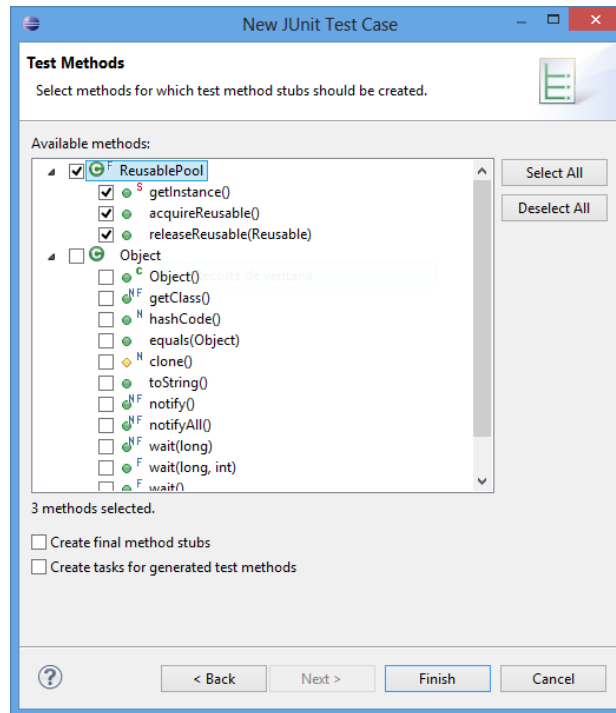


Ilustración 1: Creación de la clase de pruebas.

Obtenemos el esqueleto de nuestra clase.

```
package ubu.gii.dass.test.c01;
import static org.junit.Assert.*;
import org.junit.Test;

public class ReusablePoolTest {
    @Test
    public void testGetInstance() {
        fail("Not yet implemented");
    }
    @Test
    public void testAcquireReusable() {
        fail("Not yet implemented");
    }
}
```

```

@Test
public void testReleaseReusable() {
    fail("Not yet implemented");
}
}

```

Tabla 3.1: Estructura base de la clase de pruebas.

El primer método a implementar es `testGetInstance()`, el cual debe probar que la clase `ReusablePool` solo devuelve una única instancia de si misma, y que por lo tanto aplica de forma correcta el patrón Singleton.

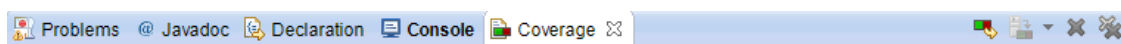
```

/**
 * ReusablePool utiliza el patron Singleton. Comprueba que en sucesivas
 * llamadas de getInstance, se devuelve la misma instancia.
 */
@Test
public void testGetInstance() {
    ReusablePool instance = ReusablePool.getInstance();
    assertEquals("No se cumple el patrón Singleton.", instance,
        ReusablePool.getInstance());
}

```

Tabla 3.2: Método `testGetInstance()`.

La compilación es correcta, y la cobertura proporcionada es del 100%, tal y como muestra la Ilustración 2.



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
PoolObject	37,9 %	50	82	132
src	37,9 %	50	82	132
ubu.gii.dass.c01	29,9 %	35	82	117
Client.java	0,0 %	0	37	37
ReusablePool.java	50,8 %	32	31	63
ReusablePool	50,8 %	32	31	63
acquireReusable()	0,0 %	0	20	20
releaseReusable(Reusable)	0,0 %	0	11	11
getInstance()	100,0 %	9	0	9
ReusablePool(int)	100,0 %	23	0	23

Ilustración 2: Cobertura para el método `getInstance()`.

El resto de métodos no tienen implementada todavía su prueba, por lo que su cobertura es del 0%.

El segundo método, `acquireReusable()`, devuelve instancias de `Reusable` hasta agotar todas las almacenadas.

La política concreta utilizada en este ejemplo es mantener dos instancias de la clase `Reusable`. En el caso de recibir una petición y no existir instancias disponibles lanza una excepción `NotFreeInstanceException`.^[1]

Como se puede ver en la Tabla 3.3, la prueba se encarga de adquirir todas las instancias de `Reusable` disponibles hasta agotar el almacén. En ese momento debe saltar la excepción indicando la falta de recursos, la cual es propagada y capturada por el parámetro `expected` de `@Test`, finalizando la prueba con éxito.

Para conseguir que tras la prueba el almacén de `Reusable` vuelva a su estado inicial, todas las instancias adquiridas son almacenadas en una lista, y mediante el método `setUp()` (el cual se ejecuta antes de cada test) son vueltas a insertar.

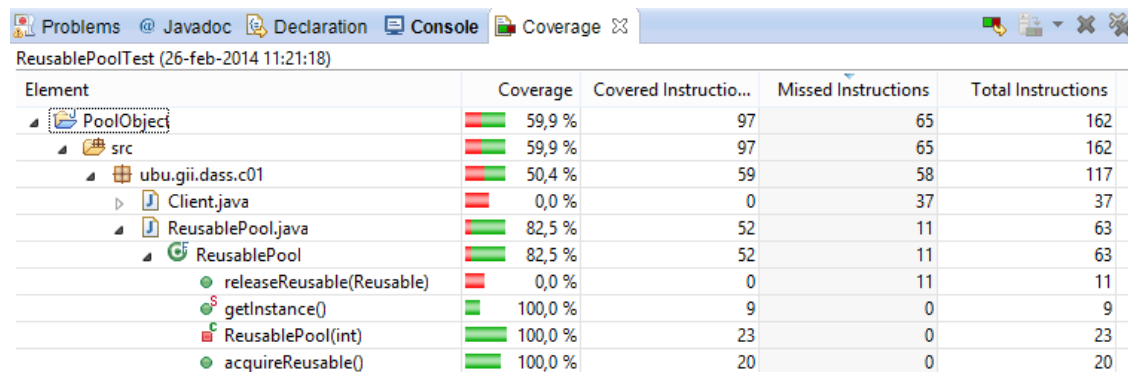
```
/**
 * Comprueba que el método lanza su excepción cuando no hay mas reusables.
 *
 * @throws NotFreeInstanceException
 *         No hay mas reusables.
 */
@Test(expected = NotFreeInstanceException.class)
public void testAcquireReusable() throws NotFreeInstanceException {
    ReusablePool instance = ReusablePool.getInstance();

    while (true) {
        Reusable reusable = instance.acquireReusable();
        pool.add(reusable);
    }
}

/**
 * Se ejecuta antes de cada test, para asegurar que ReusablePool mantiene el
 * numero inicial de Reusable.
 */
@Before
public void setUp() {
    for (Reusable r : pool) {
        ReusablePool.getInstance().releaseReusable(r);
    }
    pool.clear();
}
```

Tabla 3.3: Métodos `testAcquireReusable()` y `setUp()`.

La cobertura que se consigue con esta prueba es del 100%.



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
PoolObject	59,9 %	97	65	162
src	59,9 %	97	65	162
ubu.gii.dass.c01	50,4 %	59	58	117
Client.java	0,0 %	0	37	37
ReusablePool.java	82,5 %	52	11	63
ReusablePool	82,5 %	52	11	63
releaseReusable(Reusable)	0,0 %	0	11	11
getInstance()	100,0 %	9	0	9
ReusablePool(int)	100,0 %	23	0	23
acquireReusable()	100,0 %	20	0	20

Ilustración 3: Cobertura para el método `acquireReusable()`.

El último método que debemos probar es `releaseReusable()`, el cual se encarga de insertar las instancias de `Reusable` de nuevo en el `ReusablePool` para que puedan ser reutilizadas. Antes de realizar una inserción comprueba que la instancia pasada como argumento no se encuentre ya en el almacén.

Para probar este método se han tenido en cuenta tres escenarios:

- Comprobar la correcta inserción y adquisición de una instancia.
- Comprobar que no se permite la inserción de instancias duplicadas.
- Comprobar que no se permite la inserción de elementos nulos.

La tabla Tabla 3.4 muestra el código del método.

Como se puede ver, en primer lugar se vacía el almacén de toda instancia. A continuación se inserta una única instancia y se prueba su adquisición. Se prueba así el primer escenario.

Par el segundo escenario se insertan por duplicado las instancias en el almacén, para luego volver a adquirirlas. Se compara el tamaño (número de instancias) de la lista actual de instancias adquiridas con el tamaño original previo a toda modificación. Si ambos valores son idénticos significa que se han descartado las instancias duplicadas.

El tercer y último escenario supone comprobar la inserción de elementos nulos en el almacén, lo que generaría un efecto no deseado. Para ello se introduce un elemento nulo en el almacén (`null`) y se vuelven a recuperar todas las instancias. Si alguna de ellas hace referencia a un elemento nulo, la prueba falla.

La ejecución de la prueba falla en ese último punto, ya que la implementación del método `releaseReusable()` permite la inserción de elementos nulos.


```
/**
 * Valida el funcionamiento de releaseReusable. Se comprueban los siguientes
 * comportamientos:
 *
 * - releaseReusable devuelve un objeto al pool. - releaseReusable no
 * permite duplicar elementos en el pool. - releaseReusable permite
 * introducir elementos nulos.
 */
@Test
public void testReleaseReusable() {
    ReusablePool reusablePool = ReusablePool.getInstance();
    int sizePool;
    /**
     * Obtiene todos los reusables.
     */
    try {
        while (true) {
            Reusable reusable = reusablePool.acquireReusable();
            pool.add(reusable);
        }
    } catch (NotFreeInstanceException e) {
    }

    sizePool = pool.size();

    /**
     * Devuelve el ultimo reusable.
     */
    if (!pool.isEmpty()) {
        reusablePool.releaseReusable(pool.remove(pool.size() - 1));
    }

    try {
        Reusable reusable = reusablePool.acquireReusable();
        pool.add(reusable);
    } catch (NotFreeInstanceException e) {
        fail("Deberia haber un reusable disponible en el pool.");
    }

    /**
```

```
    * Comprueba que no se pueden introducir elementos repetidos.
    */
    for (int i = 0; i < sizePool; i++) {
        Reusable reusable = pool.remove(0);
        reusablePool.releaseReusable(reusable);
        reusablePool.releaseReusable(reusable);
    }

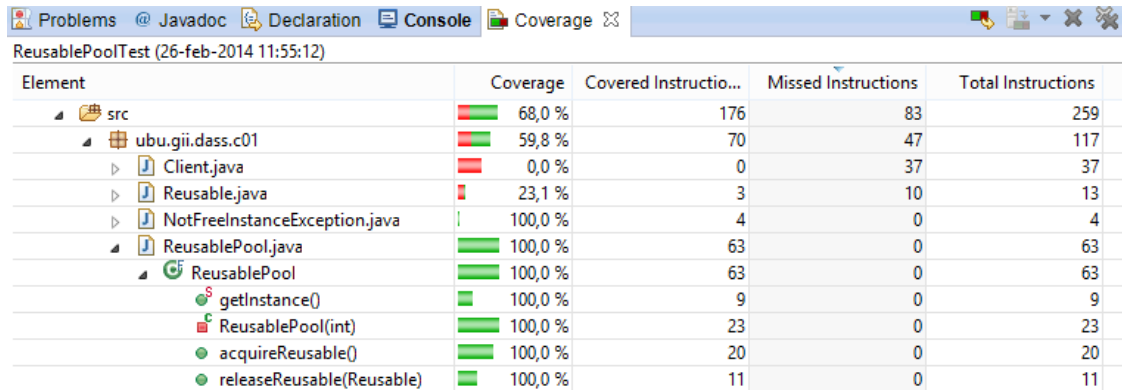
    try {
        while (true) {
            Reusable reusable = reusablePool.acquireReusable();
            pool.add(reusable);
        }
    } catch (NotFreeInstanceException e) {
    }

    if (pool.size() != sizePool) {
        fail("ReusablePool permite tener elementos repetidos.");
    }

    /**
     * Comprueba que no se pueden devolver elementos nulos al pool.
     */
    reusablePool.releaseReusable(null);
    try {
        Reusable reusable = reusablePool.acquireReusable();
        if (reusable == null) {
            fail("ReusablePool permite introducir elementos nulos");
        }
    } catch (NotFreeInstanceException e) {
    }
}
}
```

Tabla 3.4: Método `testReleaseReusable()`.

La cobertura final que se consigue de la clase `ReusablePool` a través de estas pruebas es del 100% para todos sus métodos.



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
src	68,0 %	176	83	259
ubu.giii.dass.c01	59,8 %	70	47	117
Client.java	0,0 %	0	37	37
Reusable.java	23,1 %	3	10	13
NotFreeInstanceException.java	100,0 %	4	0	4
ReusablePool.java	100,0 %	63	0	63
ReusablePool	100,0 %	63	0	63
getInstance()	100,0 %	9	0	9
ReusablePool(int)	100,0 %	23	0	23
acquireReusable()	100,0 %	20	0	20
releaseReusable(Reusable)	100,0 %	11	0	11

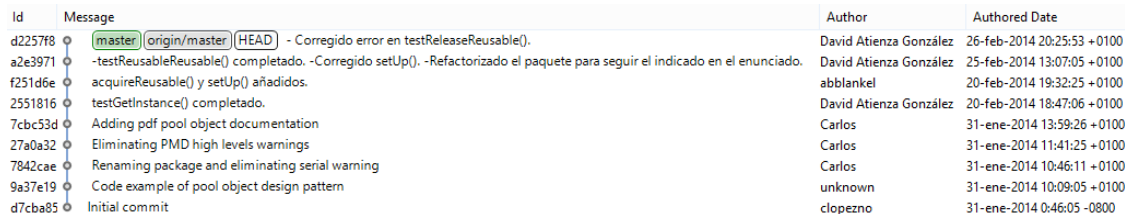
Ilustración 4: Cobertura para el método `testReleaseReusable()`.

4 CUESTIONES

¿Se ha realizado trabajo en equipo?

En la Ilustración 5 pueden verse los `commits` realizados en el repositorio de trabajo en Github.

Si no tenemos en cuenta los `commits` realizados en el proyecto original desde el que realizamos el `fork`, hemos realizado 4 `commits`.



Id	Message	Author	Authored Date
d2257f8	[master] origin/master HEAD - Corregido error en testReleaseReusable().	David Atienza González	26-feb-2014 20:25:53 +0100
a2e3971	-testReusableReusable() completado. -Corregido setUp(). -Refactorizado el paquete para seguir el indicado en el enunciado.	David Atienza González	25-feb-2014 13:07:05 +0100
f251d6e	acquireReusable() y setUp() añadidos.	abblankel	20-feb-2014 19:32:25 +0100
2551816	testGetInstance() completado.	David Atienza González	20-feb-2014 18:47:06 +0100
7cbc53d	Adding pdf pool object documentation	Carlos	31-ene-2014 13:59:26 +0100
27a0a32	Eliminating PMD high levels warnings	Carlos	31-ene-2014 11:41:25 +0100
7842cae	Renaming package and eliminating serial warning	Carlos	31-ene-2014 10:46:11 +0100
9a37e19	Code example of pool object design pattern	unknown	31-ene-2014 10:09:05 +0100
d7cba85	Initial commit	clopezno	31-ene-2014 0:46:05 -0800

Ilustración 5: Commits realizados en el repositorio.

Observando la descripción de los `commits`, podemos ver fácilmente la distribución del trabajo realizada.

David Atienza González ha realizado dos métodos: `testGetInstance()` y `testReleaseReusable()`.

Alejandro Revilla Gistain ha realizado los otros dos métodos: `testAcquireReusable()` y `setUp()`. Debido a una configuración previa del parámetro `user.name` de git, el commit realizado no aparece con el nombre del autor sino bajo el alias de `abblankel`.

¿Tiene calidad el conjunto de pruebas disponibles?

Consideramos que nuestro conjunto de pruebas tiene calidad. En primer lugar, la cobertura de nuestras pruebas es del 100%, lo cual es indicativo del esfuerzo realizado en conseguir un conjunto de pruebas fiables.

Además, hemos intentado realizar un conjunto de pruebas que sea lo más versátil posible. Una de las consecuencias de este enfoque, es que suponemos que la clase *ReusablePool*, podría contener un número de objetos *Reusable* diferente de 2. De este modo, si modificamos *ReusablePool* para que contenga, por ejemplo, 10 objetos *Reusable*, nuestro conjunto de pruebas seguiría siendo válido. Consideramos clave este tipo de desarrollo para evitar excesivas modificaciones de las pruebas ante el menor cambio del código.

Asimismo, los casos de prueba están adecuadamente comentados, documentados y reflejan de un modo preciso los errores encontrados durante la realización de las pruebas. Una adecuada documentación de las pruebas, es un parámetro a considerar al hablar de la calidad de las pruebas.

Por último, valoramos que los casos de prueba escogidos en las pruebas unitarias son los adecuados, tanto en número como en tipo.

¿Cuál es el esfuerzo invertido en realizar la actividad?

El esfuerzo invertido es de aproximadamente 5 horas. Unas 2 horas para el desarrollo del código fuente y 3 horas para la realización del presente documento.

¿Cuál es el número de fallos encontrados en el código?

A través de las pruebas realizadas (Tabla 3.4) se ha comprobado que el método `releaseReusable()` permite la inserción de elementos nulos.

Recomendamos la modificación del método `releaseReusable()`, por el siguiente código:

```
public void releaseReusable(Reusable r){
    if (r != null && reusables.contains(r)==false){
        reusables.add(r);
    }
}
```

Tabla 4.1: Modificación sugerida a `releaseReusable()`.

Referencias

- [1] Universidad de Burgos *PoolObject* [Documento en línea]
[Fecha de consulta: 26/02/2014]. Disponible en
<<https://github.com/clopezno/poolobject/blob/master/PoolObject.pdf>>