

Pruebas unitarias

Marzo 2006

@autor: Jorge Rodriguez

Probar código nunca tuvo tanta importancia en el ciclo de desarrollo de una aplicación hasta hace algunos años, donde se ha desatado una revolución en los procesos de desarrollo, apareciendo nuevas y ágiles forma de construcción, donde ejecutar pruebas (o al menos pruebas unitarias) pasó de ser un proceso tedioso (como las antiguas que se ejecutaban para cumplir estándares como el ISO 9001) a ser un forma de trabajo integrada y productiva en los nuevos procesos de desarrollo.

A pesar de existir diferentes tipos y técnicas de pruebas (unitarias, de integración, aceptación, carga y stress), en este artículo hablaremos del uso de **Pruebas Unitarias** (Unit Test), pues de todo el conjunto de pruebas, es considerada la más importante para garantizar un proyecto exitoso, por tanto debe ser introducida como una actividad más en el desarrollo.

Que son las pruebas unitarias

Son pruebas dirigidas a probar clases java aisladamente y están relacionadas con el código y la responsabilidad de cada clase y sus fragmentos de código más criticos.

Por ejemplo una clase que envíe e-mails, debe ser capaz de probarse aislada chequeando que sea capaz de enviar e-mails, y asegurandonos de probar todas las posibilidades de fallo y éxito. En un modelo basado en pruebas, se deben definir casos de prueba para cada clase de forma aislada, una prueba no debe depender de otras clases.

Porque realizar pruebas unitarias

- **Asegura calidad del código entregado.** Es la mejor forma de detectar errores tempranamente en el desarrollo. No obstante, esto no asegura detectar todos los errores, por tanto prueba de integración y aceptación siguen siendo necesarias.
- **Ayuda a definir los requerimientos y responsabilidades de cada método en cada clase probada.**
- **Constituye una buena forma de ejecutar pruebas de concepto.** Cuando es necesario hacer pruebas de conceptos sin integrar usar pruebas unitarias se convierte en un método efectivo.
- **Permite hacer refactoring tempranamente en el código.** No es necesario todo un ciclo de integración para hacer refactoring en la aplicación, basta con ver como se comporta un caso de prueba para hacer refactoring unitario sobre la clase que estamos probando en cuestión.
- **Permite incluso hacer pruebas de stress tempranamente en el código.** Por ejemplo un método que realice una consulta SQL que exceda los tiempos de aceptación es posible optimizarla antes de integrar con la aplicación.
- **Permite encontrar errores o bugs tempranamente en el desarrollo.** Y está demostrado que mientras más temprano se corrijan los errores, menos costará corregirlos.

Técnicas para hacer nuestro código testeable.

- **Codificar sobre interfaces más que sobre clases.** Mientras más usemos interfaces más plugabilidad de soluciones tendremos en run-time o test-time. De esta forma estamos probando el “**que**” y no el “**como**” y esto es precisamente en lo que debe enfocarse un prueba unitaria
- **Usar la Ley de Demeter**¹. Esta ley dice que un objeto solo debe depender de objetos que esten muy cercanos a el, o sea no debería usar objetos globales.

```
public class ClaseAProbar implements ServicioAProbar {
    ....
    protected Helper helper;
    ...
    public void metodoAProbar() {
        ...
        helper.metodo(); // BIEN según ley de demeter. La clase solo depende de Helper

        helper.getSegundoHelper().metodo(); // MAL según ley de demeter pues aca nuestro
        // objeto depende tanto de helper com de SegundoHelper.
    }
    ...
}
```

- **Minimizar dependencia sobre APIs especificas de ambientes.** Imaginen que seamos capaces de probar un EJB, sin tener que deployarlo sobre un EJB Container. Esto más que una técnica es una buena práctica que hemos reiterado a lo largo del documento.
- **Asegurarnos que cada objeto tiene un conjunto definido de responsabilidades.** Objetos con mucha responsabilidad y dependencias son dificiles de probar.
- **Ocultar detalles de implementación.** Un caso de prueba no debe tener conocimiento de como fue implementado el método que se esta probando, de esta forma no tendria influencia en ello.

1 <http://c2.com/cgi/wiki?lawOfDemeter>

// A CONTINUACIÓN UN ERROR , estamos asumiendo que se usa la API java-mail de Sun para enviar el e-mail. Esto presume dependencia de API.

```
public void testEnviaMail() {
    javax.mail.internet.MimeMessage email = new javax.mail.internet.MimeMessage();
    email.setFrom("Jorge Rodriguez <jrodriguez@interplanet.cl>");
    ....
    email.setText("ADIOS mundo cruel !!!");
    claseAProbar.enviaMail(email);
    // TODO, revisar si nos llego el mail
}
```

// AHORA CORREGIDO, no sabemos como la clase hace para enviar el mail, solo sabemos que lo envia.

```
public void testEnviaMail() {
    String from = "Jorge Rodriguez <jrodriguez@interplanet.cl>";
    String to = "Jorge Rodriguez <jrodriguez@interplanet.cl>";
    String texto = "ADIOS mundo cruel !!!";
    claseAProbar.enviaMail(from, to, texto);
    // TODO, revisar si nos llego el mail
}
```

JUnit

Existen mejores métodos de probar código java que el tradicional **main()**. JUnit es una herramienta simple, Open Source que se ha convertido en el estándar para probar clases java de forma unitaria. Es fácil de usar y configurar por lo que practicamente no existe curva de aprendizaje, salvo el cambio de mentalidad que debe producir en las mentes de los programadores al usar técnicas que faciliten el uso de unidades de prueba.

JUnit esta diseñado para reportar fallos o exitos sobre código sin necesidad de interpretar el reporte, pues es bien sencillo. JUnit ejecuta casos de prueba (Test Cases) contra un conjunto de objetos y provee formas de inicializar y configurar cada Caso.

El uso de JUnit normalmente involucra los siguientes pasos:

- **Crear una subclase de junit.framework.TestCase.**
- **Opcionalmente sobre-escribir el método setUp() que será invocado en la inicialización de objetos y variables usados por todos los casos de prueba.** No todos los casos de uso necesitan esto. Note que setUp() es invocado antes de cada caso particular.
- **Opcionalmente sobre-escribir el método tearDown().** Método que será invocado al final de cada caso de prueba y que nos sirve para liberar recursos usados en la prueba o incluso para volver atrás lo probado, por ejemplo cuando el caso de prueba involucre la actualización de datos en una base de datos relacional.
- **Adicionar métodos de prueba a la clase.** Note que no necesitamos implementar ninguna interface, pues JUnit usa el paquete reflection del java para detectar automaticamente métodos test. Dichos métodos son reconocidos por su asignatura, la cual debe tener la forma public void test<Descripcion>(), además pueden lanzar cualquier exception. A continuación

un ejemplo real del uso de JUnit².

² Pertenece al set de pruebas de una librería autilitaria de Interplanet S.A

```

public class PAdminTemplateTests extends TestCase {

    /** servicio de operaciones sobre TAM */
    private TAMOperations tam;

    /** (non-Javadoc)
     * @see junit.framework.TestCase#setUp()
     */
    protected void setUp() throws Exception {
        super.setUp();
        ApplicationContext app =
            new ClassPathXmlApplicationContext("applicationContext.xml");
        this.tam = (TAMOperations) app.getBean("tamOperations");
    }

    /** (non-Javadoc)
     * @see junit.framework.TestCase#tearDown()
     */
    protected void tearDown() throws Exception {
        super.tearDown();
    }

    /**
     * Test method for 'cl.interplanet.dao.tam.support.PAdminTemplate.createUser(...)'
     */
    public void testCreateUser() {
        this.tam.createUser("TATA",
            new PDRgyUserName("cn=TATA,cn=Users,dc=achs,dc=cl", "Bravo", "Bravisimo"),
            "ESTE ES EL TATA", "000001", null, true, false, true);
    }

    /**
     * Test method for 'cl.interplanet.dao.tam.support.PAdminTemplate.deleteUser(...)'
     */
    public void testDeleteUser() {
        this.tam.deleteUser("TATA", false);
    }

    /**
     * Test method for 'cl.interplanet.dao.tam.support.PAdminTemplate.importUser(...)'
     */
    public void testImportUser() {
        this.tam.importUser("21493497",
            new PDRgyUserName("CN=Cuenta CI. Informacion,CN=Users,DC=achs,DC=cl",
                null, null), true);
    }

    /**
     * Test method for 'cl.interplanet.dao.tam.support.PAdminTemplate.setAuthRuleText(...)'
     */
    public void testSetAuthRuleText() {
        String txt = "<xsl:choose><xsl:when " +
            "test=\\(azn_engine_target_resource=\"" +
            "/WebSEAL/marte-marte/webapp/010002)\\\"> " +
            "!FALSE!</xsl:when><xsl:otherwise> " +
            "!TRUE!</xsl:otherwise> " +
            "</xsl:choose>";
        this.tam.setAuthRuleText("app-mantencion-rule", txt);
    }

    ...
}

```

Es posible usar una serie de **test runners** provistos por JUnit para correr nuestros **test cases** que se ejecutan y muestran los resultados de formas diferentes. Los más usados son el Text runner y el Swing runner que lo muestra de forma gráfica. Pero en lo particular yo prefiero ejecutar JUnit desde el **IDE**³ o mejor aún, desde **ant**, desde donde es posible además automatizar la ejecución de los casos de prueba generando reportes en html de cada ejecución.

A continuación algunos pantallazos del uso de JUnit en **eclipse**

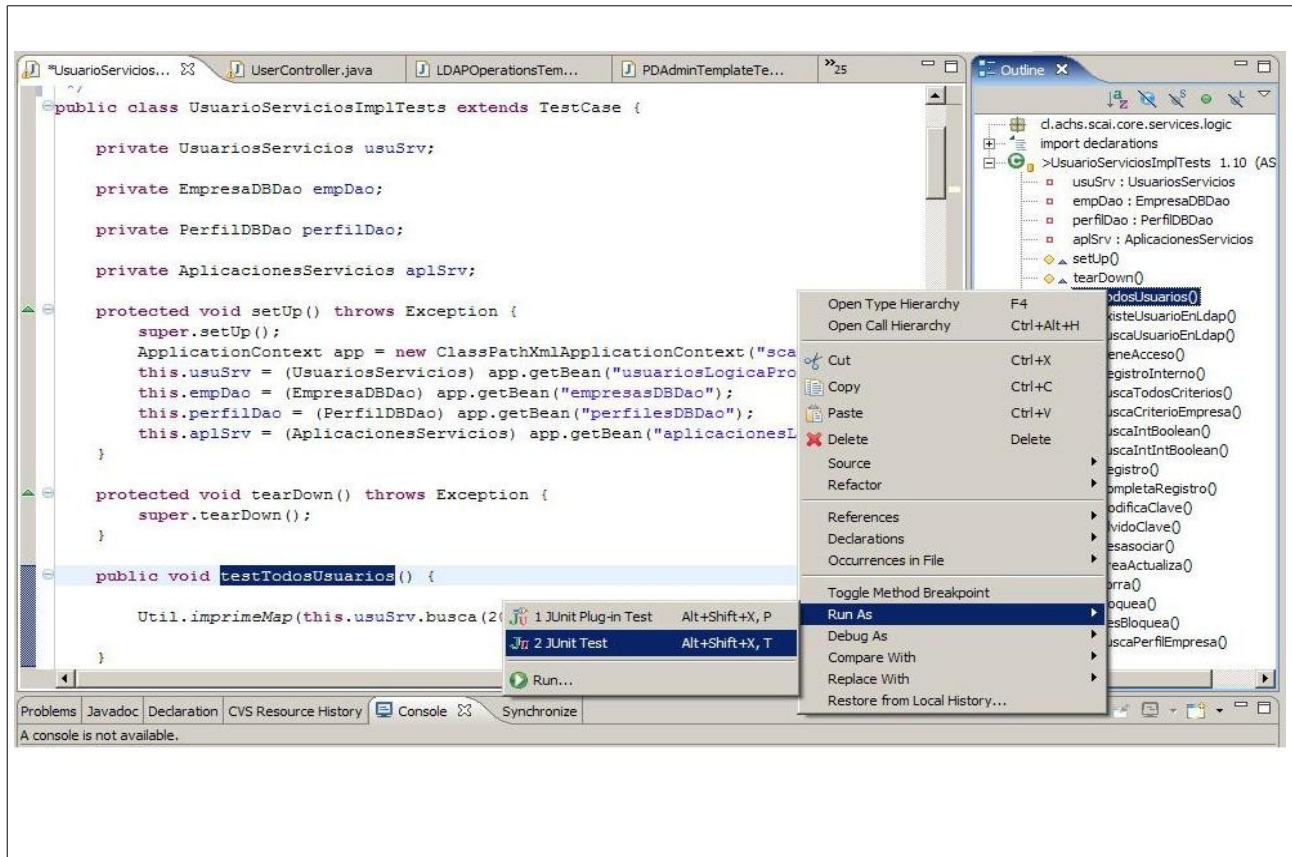


Fig 1 – Ejecución de un Caso de Prueba desde el IDE eclipse.

3 La mayoría de los IDEs en su últimas versiones soportan la ejecución de **TestCases**


```

<target name="run-tests" depends="compile-test" description="Ejecuta todos los casos de prueba y genera reportes">

    <property name="reports.dir" value="${target.junit.reports.dir}"/>
    <property name="summary.dir" value="${target.junit.summary.dir}"/>

    <mkdir dir="${reports.dir}"/>
    <mkdir dir="${summary.dir}"/>

    <junit printsummary="yes" haltonfailure="no" haltonerror="no">

        <jvmarg line="-Djava.awt.headless=true"/>

        <!-- Must go first to ensure any jndi.properties files etc take precedence -->
        <classpath location="${target.testclasses.dir}"/>
        <classpath location="${target.classes.dir}"/>

        <!-- Need files loaded as resources -->
        <classpath location="${test.dir}"/>

        <classpath refid="all-libs"/>

        <formatter type="plain" usefile="false"/>
        <formatter type="xml"/>

        <batchtest fork="yes" todir="${reports.dir}">
            <fileset dir="${target.testclasses.dir}" includes="${test.includes}" excludes="${test.excludes}"/>
        </batchtest>

    </junit>

    <junitreport todir="${reports.dir}">

        <fileset dir="${reports.dir}">
            <include name="TEST-*.xml"/>
        </fileset>
        <report todir="${summary.dir}"/>

    </junitreport>

</target>

```

Y ante la ejecución del comando `> ant run-tests` se genera el siguiente reporte

Unit Test Results				
				Designed for use with JUnit and Ant .
Summary				
Tests	Failures	Errors	Success rate	Time
89	0	89	0.00%	240.682
Note: <i>failures</i> are anticipated and checked for with assertions while <i>errors</i> are unanticipated.				
Packages				
Name	Tests	Errors	Failures	Time(s)
cl.achs.scai.core.services.dao.db.hibernate	43	43	0	173.607
cl.achs.scai.core.services.dao.ldap.ad	2	2	0	0.160
cl.achs.scai.core.services.logic	44	44	0	66.915

Resumen

Las pruebas deben usarse en todo el ciclo de desarrollo de un proyecto, sobre todo debe formar parte del día a día del programador, formulando filosofías como la TDD (Test Driven Development) donde el método es probar antes de implementar.

A un nivel de expertise superior, un arquitecto java o desarrollador avanzado debería ser capaz de escribir builds de automatización de ejecución de pruebas unitarias, que en conjunto con herramientas como *CruiseControl*⁵ fomenten el uso de la teoría de Integración Continua⁶ de Martin Fowler.

5 <http://cruisecontrol.sourceforge.net/>

6 <http://www.martinfowler.com/articles/continuousIntegration.html>