Capitulo 3. “Test Driven Development”

3.1 Uso de JUnit como “framework” para realizar pruebas unitarias

Como ya se mencionó en el marco teórico “Test Driven Development” es una técnica de programación extrema en la cual primero se desarrollan las pruebas unitarias antes de escribir cualquier línea de código funcional.

Actualmente existen varios “frameworks” para poder usar esta técnica, su uso depende en su mayoría del lenguaje de programación sobre el cual se trabaja. Entre ellos cabe destacar los siguientes:

- PHPUnit
- JUnitPerf
- HttpUnit
- JWebUnit
- Cactus
- JFCUnit
- JXUnit
- Jester
- JUnit

Durante la implementación de ésta tesis se usará JUnit como marco de trabajo para adoptar el uso de “Test Driven Development”.

Kent Beck es el precursor de JUnit, en sus comienzos solo contaba con 3 clases y 12 métodos [Beck, 2004], además de estar desarrollado en “Smalltalk”. JUnit surgió bajo la necesidad de crear un entorno con el cual se pudieran realizar pruebas unitarias de una forma fácil, fue por ello que Kent Beck debido a un proyecto de consultaría en el que trabajaba se dio a la tarea de desarrollar toda una plataforma con la que pudiera realizar las pruebas unitarias.

Después de desarrollar y entregar una estructura de soporte definida al cliente con la que pudiera automatizar las pruebas unitarias, Kent Beck notó el valor dicha estructura había adquirido durante el proceso de desarrollo del software, es por ello que comenzó a distribuirlo de forma gratuita a la comunidad de “Smalltalk” en el año de 1994.

Para el año de 1997, durante un viaje a Atlanta a la Conferencia de Programación Orientada a Objetos, Kent Beck junto con Erich Gamma colaboran para adaptar al lenguaje Java el “framework” desarrollado por Beck en “Smalltalk”. Al final de las conferencias Martin Fowler comienza a distribuirlo. Posteriormente se convertiría en JUnit [Beck, 2004].

Después de dicho congreso y ya estando en Zurich, Beck y Gamma, André Weinand, desarrolla una interfaz gráfica “AWT(Abstract Window Toolkit)”, para JUnit, después de ello se realiza su primer distribución.

Actualmente el “framework” cuenta con 28 extensiones en diferentes lenguajes de programación. JUnit permite a los programadores de una forma fácil y entendible quitar la barrera entre la programación y las pruebas unitarias.

3.2 Noción Básica del Uso de “Test Driven Development” como “framework”

El uso de tecnologías como “Test Driven Development” no significa que se debe usar algún otro paradigma para realizar pruebas unitarias durante el proceso de desarrollo de software. Para ello se demostrará con un pequeño ejemplo el uso de esta técnica.

En este caso se quiere saber cuál es el tamaño de un arreglo de elementos, por lo tanto se espera que la prueba después de agregar ciertos elementos al arreglo muestre su tamaño actual.

La forma más sencilla de realizar dicha verificar el tamaño de dicho arreglo es el imprimir el tamaño del arreglo antes y después de que se dé añaden elementos al arreglo.

A continuación se presenta el código en Java que imprime el tamaño del arreglo mencionado.

```
Arraylist test = new Arraylist();

System.out.println(“Antes de agregar elementos al Array: ” +test.size()  );

test.add(“elemento 1”);

test.add(“elemento 2”);
```

System.out.println(“Tamaño despues de agregar elementos al Array: ” +test.size());

Para hacer un poco más eficiente la prueba anterior lo que se va a hacer es el agregar una condición en primer término para verificar si el arreglo comienza en cero y después para verificar si dicho arreglo cuenta con 2 elementos, dichas condiciones serán impresas y se verificará si se cumplen o no dichas condiciones, por lo tanto se imprimirá un “true” o un “false.”

Enseguida se presenta la variante al código:

Arraylist test = new Arraylist();

System.out.println(“El arreglo comienza en 0: ” + test.size() == 0);

test.add(“elemento 1”);

test.add(“elemento 2”);

System.out.println(“El arreglo tiene 2 elementos: ” +test.size() == 2);

Finalmente para que nuestro primer ejemplo quede listo se creará un método que lanzará una excepción en caso de recibir un “false” de la condición que se envía como parámetro, con este método en caso de que el valor esperado no sea correcto se lanzará una excepción indicando que la prueba unitaria no ha sido exitosa. Agregando el método mencionado la primera prueba unitaria usando “Test Driven Development” quedaría de la siguiente forma:

Arraylist test = new Arraylist();

```
assertTrue(test.size() == 0);

test.add("elemento 1");

test.add("elemento 2");

assertTrue(test.size() == 2);



//método assertTrue(boolean condicion)

void assertTrue(boolean condicion) throws Exception

{

    If(! condicion)

        throw new Exception("Prueba unitaria fallida");

}
```

Debido a que un programa considerable está compuesto de más de una sola prueba unitaria, es necesario crear toda una infraestructura que JUnit ya posee.

Tal como lo presenta Cabalar el proceso de creación de pruebas unitarias usando JUnit es el siguiente [Cabalar, 2008]:

1. Creación de un test, el cual debe fallar ya que aun no se cuenta con el código funcional.

2. Creación del código funcional

3. Corrección del test, y modificación para que el test sea exitoso.

4. Repetición de los 3 primeros pasos, durante la etapa de codificación de software.

JUnit posee su propia implementación, sin embargo actualmente se encuentra integrado en los siguientes IDE's de desarrollo de software:

- Eclipse
- BlueJ
- IntelliJ IDEA
- NetBeans

Para los propósitos de ésta tesis se uso la integración con el entorno de desarrollo NetBeans 5.5. La versión de JUnit usada es la versión 3.8.1.

Algunas reglas básicas para la implementación de JUnit para la realización de pruebas unitarias son las siguientes:

- Toda clase de prueba debe extender de la clase "TestCase":

  public class CSVImporterTest extends TestCase

- Todo método de prueba dentro de la clase de prueba debe comenzar con test:

  public void testLoadTableName() throws Exception

A continuación se detalla la utilización del JUnit en el desarrollo de la presente tesis.

Para cada uno de los puntos funcionales del sistema se creó una clase de pruebas en el cual se implementaron las pruebas unitarias necesarias. En la imagen 3.2.1 se muestra el paralelismo.

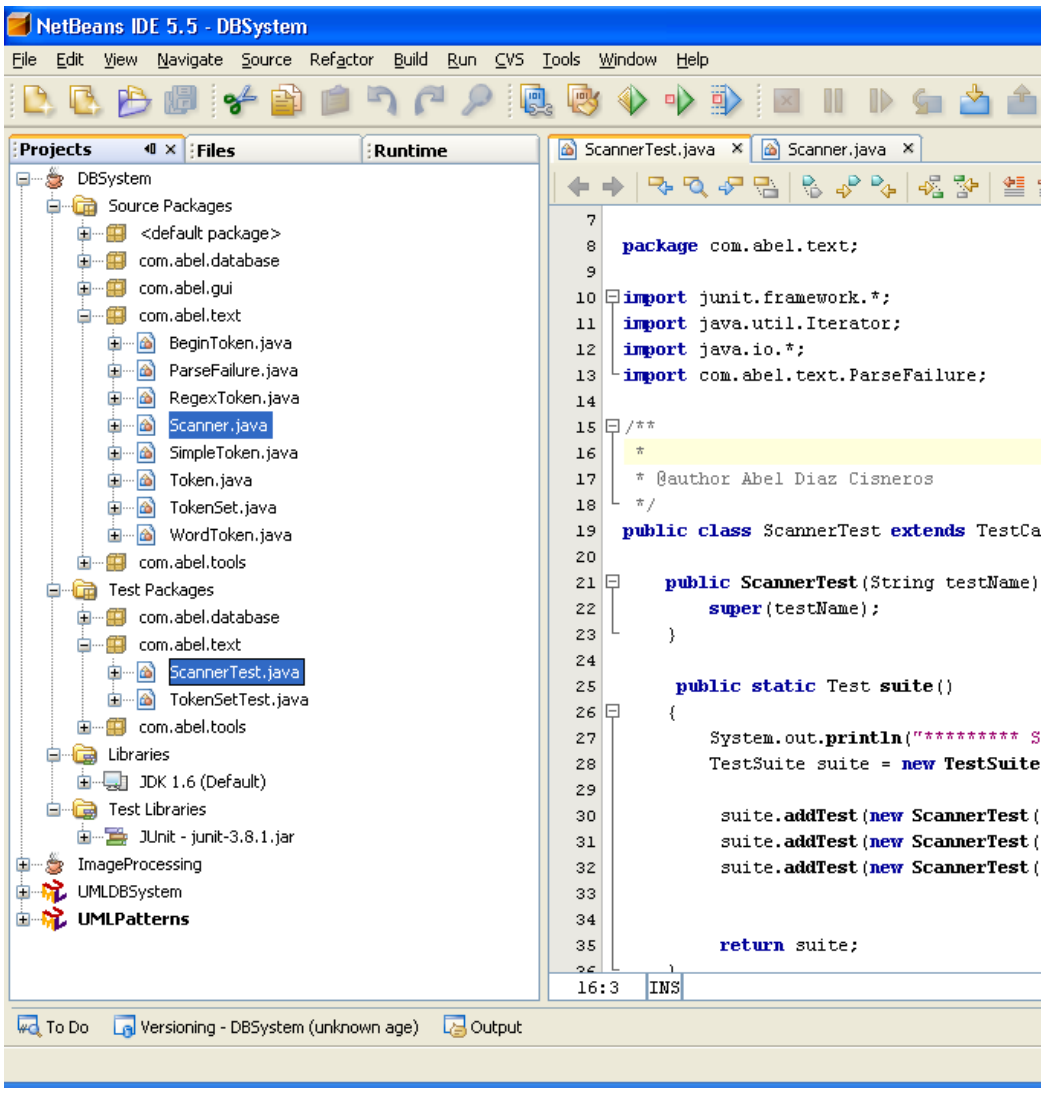

Imagen 3.2.1 Creación de clase de pruebas usando JUnit

Cada clase de prueba contiene los siguientes elementos:

- Extiende de la clase “TestCase”.

- Implementan el método “**suite()**”, el cual se encarga de ejecutar solo los métodos llamados desde éste método. En la imagen 3.2.2 se muestra el código del método suite de una clase de prueba.

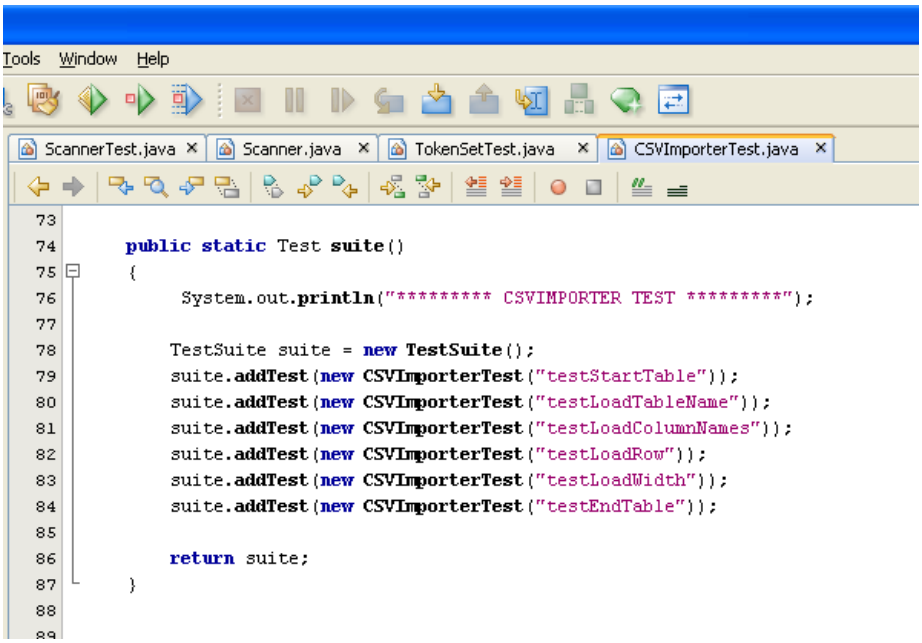

Figura 3.2.2 Método suite()

- Implementan el método “**setUp()”** que se encarga de inicializar variables y objetos usados durante ejecución de la clase de pruebas unitarias. En la figura 3.2.3 se muestra el uso del método “setUp()”.

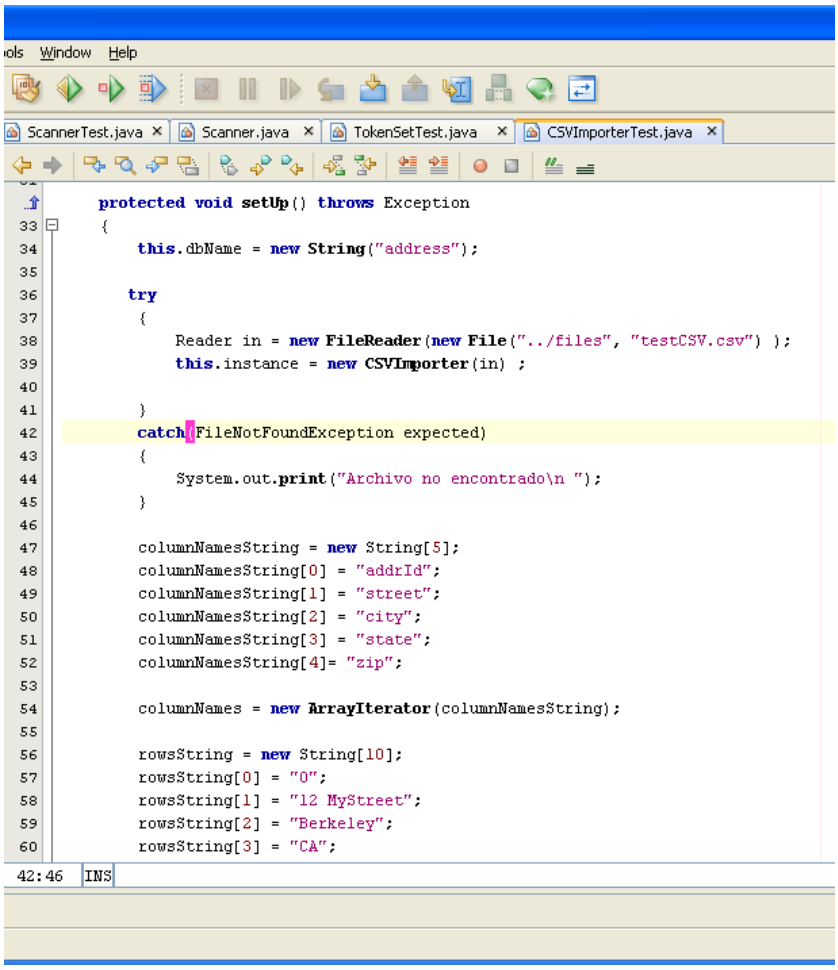

Figura 3.2.3 Método setUp()

- Implementan el método "**tearDown()**" que tiene la función de recolector de basura de los objetos y variables inicializadas durante la método "setUp()" y durante la ejecución de la clase de pruebas.

- Implementación de cada uno de los métodos que contienen las pruebas unitarias para el código funcional.

Cabe destacar que para que un método de prueba sea exitoso o no, son usados los métodos "**assert**", tanto para comparar los valores esperados contra los valores reales, o

directamente hacer que el método de prueba falle. En la tabla 3.2.4 se describen los métodos implementados del JUnit.

Método assert de JUnit	**Comprobación a realizar**
assertTrue(expresión)	Verifica que la expresión se evalúe en “true”, en caso contrario marca un fallo en el método de prueba.
assertFalse(expresión)	Verifica que la expresión se evalúe en “false”, de lo contrario el método de prueba falla.
assertEquals(valor esperado, valor real)	Verifica que el valor esperado sea igual al valor real, de lo contrario el método de prueba falla.
assertNull(objeto)	Verifica que el objeto sea nulo, de lo contrario el método de prueba falla.
assertNotNull(objeto)	Verifica que el objeto no sea nulo, de lo contrario el método de prueba falla.

assertSame(objeto esperado, objeto real)	Verifica que el objeto esperado sea igual al objeto real, de lo contrario el método de prueba falla.
assertNotSame(objeto esperado, objeto real)	Verifica que el objeto esperado sea diferente al objeto real, en caso contrario el método de prueba falla.
fail()	Forza que el método de prueba finalice con un fallo.

Tabla 3.2.4 Métodos assert()

En la figura 3.2.5 se muestra un método de prueba usado en ésta tesis.

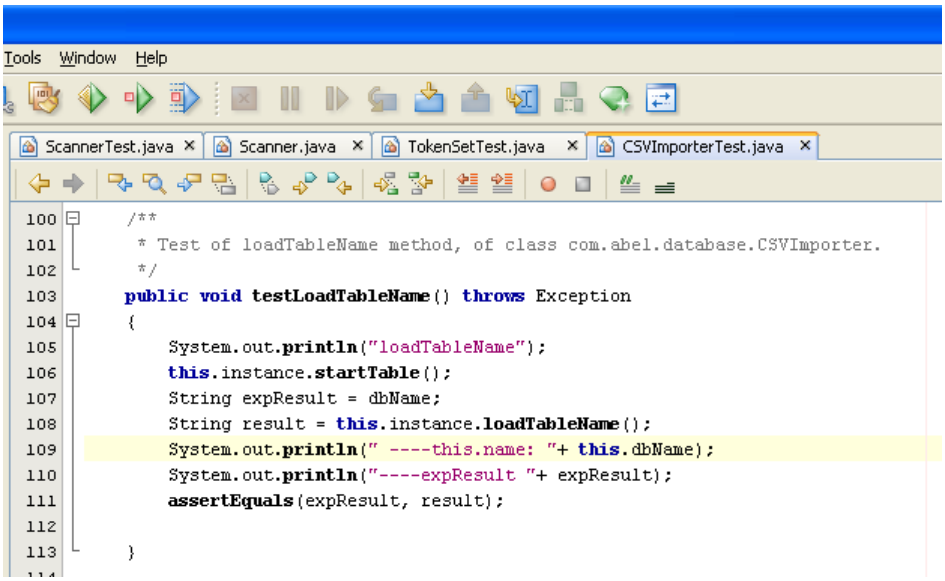

Figura 3.2.5 Metodo de prueba usando JUnit

3.3 Características de JUnit

Algunas de las características más importantes de JUnit para el desarrollo de pruebas unitarias son las siguientes:

- Ejecución de las pruebas de forma automática.
- Ejecución de varias pruebas de forma simultanea
- Provee un lugar para poder concentrar el resultado de todas las pruebas [Beck, 2004].
- Fácil manejo de la herramienta ya que proporciona los resultados esperados y los resultados actuales y ofrece un reporte de la ejecución de las pruebas.
- Con la realización de las pruebas unitarias al código funcional, se está asegurando los requerimientos del cliente sean los deseados.
- Provee soporte para los desarrolladores de cómo modificar el sistema sin alterar el comportamiento actual del sistema en cuestión [Storani, 2008] .
- Sirve como documentación sobre el funcionamiento del sistema para los desarrolladores [Storani, 2008].

El uso de este “framework” permite que los sistemas sean más adaptables al cambio debido a que es mucho más fácil detectar y aislar las pruebas unitarias realizadas a cada parte del sistema.

3.4 Limitaciones del “Framework”

Algunas limitaciones del uso de “Test Driven Development” son:

- El uso de TDD es difícil en situaciones en las que se requieren pruebas completas de funcionalidad del sistema, por ejemplo en el uso de GUI’s, programas que trabajan con Bases de Datos relacionales y algunos que trabajan en diferentes configuraciones de red [Storani, 2008].

- Si la gerencia de los proyectos no está convencida de que mediante el uso de TDD se mejorara el producto final, solo lo verán como una pérdida de tiempo [Storani, 2008].

- Las pruebas en el desarrollo de software tiene una menor importancia que el desarrollo o la arquitectura de software, un ejemplo de ello es el Visual Studio 2005 “Architect Edition”, el cual carecía de las facilidades ofrecidas en la “Testing Edition” [Storani, 2008].

- Las pruebas deben ser parte del completo mantenimiento de un proyecto, ya que las pruebas mal diseñadas y codificadas son propensas a convertirse en fallas que son difíciles de mantener, por lo tanto importante diseñar y codificar test de bajo y fácil mantenimiento [Storani, 2008].