UNIVERSIDAD CIENTIFICA DEL SUR



TÍTULO: Pruebas Unitarias

CURSO: Lenguaje de Programación

PROFESOR: Gustavo Coronel

INTENGRANTES:

* Fernando Cuesta
* Alonso Amado

ENTREGA: Febrero 2017

# Índice

[Índice ii](#_Toc475528579)

[Resumen 1](#_Toc475528580)

[Antecedentes 3](#_Toc475528581)

[Capítulo 1 Fundamento teórico. 4](#_Toc475528582)

[Capítulo 2 Caso Demostrativo. 8](#_Toc475528583)

[Capítulo 3 Discusión. 10](#_Toc475528584)

[Capítulo 4 Conclusiones. 12](#_Toc475528585)

[Capítulo 5 Recomendaciones. 13](#_Toc475528586)

[Bibliografía 14](#_Toc475528587)

# Resumen

Cuando se implementa software, resulta recomendable comprobar que el código que hemos escrito funciona correctamente. Para ello, implementamos pruebas que verifican que nuestro programa genera los resultados que esperamos.

La prueba unitaria es un método que prueba una unidad de código, se utiliza para comprobar que un método concreto del código de producción funciona correctamente, probar las regresiones o realizar pruebas relacionadas (buddy) o de humo. Al hablar de una unidad de código nos referimos a un requerimiento.

La prueba unitaria cuenta con las siguientes características:

* Prueba solamente pequeñas cantidades de código: Solamente prueba el código del requerimiento específico.
* Se aísla de otro código y de otros desarrolladores: La prueba unitaria prueba exclusivamente el código relacionado con el requerimiento y no interfiere con el trabajo hecho por otros desarrolladores.
* Solamente se prueban los endpoints públicos: Esto principalmente porque los disparadores de los métodos privados son métodos públicos por lo tanto se abarca el código de los métodos privados dentro de las pruebas.
* Los resultados son automatizados: Cuando ejecutamos las pruebas lo podemos hacer de forma individual o de forma grupal. Estas pruebas las hace el motor de prueba y los resultados de los mismos deben de ser precisos con respecto a cada prueba unitaria desarrollada
* Repetible y predecible: No importa el orden y las veces que se repita la prueba, el resultado siempre debe de ser el mismo.
* Son rápidos de desarrollar: Contrariamente a lo que piensan los desarrolladores, acerca de que el desarrollo de pruebas unitarias quita tiempo, por lo general deben de ser simples y rápidos de desarrollar. Difícilmente una prueba unitaria deba de tomar más de cinco minutos en su desarrollo.

Para el caso específico de la presente investigación, nos enfocamos en el uso de las prueba unitarias basadas en Java, por medio de la herramienta de software libre basada en Java, que lleva por nombre JUnit, por medio del IDE NetBeans.

# Antecedentes

JUnit es una herramienta para la realización de pruebas de unidad de software desarrollado con Java, que es una herramienta de software libre, por lo cual se puede extender.

JUnit fue desarrollada por Erich Gamma y Kent Beck, adoptado y apoyada por grupos partidarios de la programación extrema, la cual, entre otras cosas, sigue una política de primero probar y luego codificar. Esta y todas las herramientas descendientes de JUnit consisten de una serie de clases que auxilian en la preparación y codificación de casos de prueba y algunos mecanismos auxiliares, que en conjunto permite ejecutar y verificar el cumplimiento de los casos de prueba. Además provee una interfaz que permite automatizar la ejecución de grupos de casos de prueba.

Debido al éxito alcanzado por JUnit, es que este se ha incluido en varios IDE, siendo los dos de más importancia, NetBeans y Eclipse. De preferirse su uso con java desde el SDK (Standard Development Kit, por sus siglas en inglés), también es posible descargar JUnit y copiarlo en un directorio.

# Capítulo 1 Fundamento teórico.

En los últimos años las pruebas unitarias han tomado mucha importancia en el desarrollo de software. Estas pruebas buscan conocer el funcionamiento de las unidades de software de forma temprana. Uno de los motivos por el cual estas pruebas son ahora más conocidas y más usadas es la incorporación de metodologías ágiles como Scrum y XP.

Cuando probamos un programa, lo ejecutamos con unos datos de entrada (casos de prueba) para verificar que el funcionamiento cumple los requisitos esperados. Definimos prueba unitaria como la prueba de uno de los módulos que componen un programa.

En los últimos años se han desarrollado un conjunto de herramientas que facilitan la elaboración de pruebas unitarias en diferentes lenguajes. Dicho conjunto se denomina XUnit. De entre dicho conjunto, JUnit es la herramienta utilizada para realizar pruebas unitarias en Java.

El concepto fundamental en estas herramientas es el caso de prueba (test case), y la suite de prueba (test suite). Los casos de prueba son clases o módulos que disponen de métodos para probar los métodos de una clase o módulo concreta/o. Así, para cada clase que quisiéramos probar definiríamos su correspondiente clase de caso de prueba. Mediante las suites podemos organizar los casos de prueba, de forma que cada suite agrupa los casos de prueba de módulos que están funcionalmente relacionados.

Las pruebas que se van construyendo se estructuran así en forma de árbol, de modo que las hojas son los casos de prueba, y podemos ejecutar cualquier subárbol (suite).

De esta forma, construimos programas que sirven para probar nuestros módulos, y que podremos ejecutar de forma automática. A medida que la aplicación vaya avanzando, se dispondrá de un conjunto importante de casos de prueba, que servirá para hacer pruebas de regresión. Eso es importante, puesto que cuando cambiamos un módulo que ya ha sido probado, el cambio puede haber afectado a otros módulos, y sería necesario volver a ejecutar las pruebas para verificar que todo sigue funcionando.

Encontraremos una distribución de JUnit, en la que habrá un fichero JAR, junit.jar, que contendrá las clases que deberemos tener en el CLASSPATH a la hora de implementar y ejecutar los casos de prueba.

Al momento de incorporar pruebas unitarias al desarrollo de software varios aspectos deben ser considerados. Algunos son los siguientes:

* Qué es una unidad de software
* Quién diseña y ejecuta las pruebas, cómo lo hace y cuándo lo hace
* Cuándo se puede liberar la unidad (criterio de liberación)
* Qué medidas (métricas) se deben usar
* Qué herramientas usar

Sin considerar estos y otros aspectos la introducción de las pruebas unitarias en una organización puede ser un fracaso. Probablemente, el mayor riesgo que se corre con las pruebas unitarias es “caer” en un ciclo code-fix. Este tipo de ciclo ocurre normalmente cuando se depende únicamente de las pruebas unitarias para construir unidades de calidad. Lo único que se logra en esos casos son unidades remendadas de baja calidad. Si bien existen diversos enfoques y algunos procesos de desarrollo proponen variantes, normalmente las pruebas unitarias son realizadas por la misma persona que construyó la unidad.

El conocimiento de la unidad, por haberla construido, permite encontrar los defectos de forma rápida luego que alguna prueba falla. Sin embargo, existen diversas variantes. Por ejemplo, el diseño de las pruebas lo realiza una persona distinta a quien desarrolló la unidad; pero, la ejecución y corrección de defectos las realiza el desarrollador.

También al introducir las pruebas unitarias en una organización se debe considerar cómo se van a realizar las mismas. Esto implica definir la estrategia y las técnicas de pruebas unitarias a ser utilizadas. Un enfoque interesante es combinar técnicas de caja negra y técnicas de caja blanca.

Las técnicas de caja negra son pruebas que se desarrollan a partir de la especificación de la unidad a probar. En estas lo importante es probar la funcionalidad que la unidad debe cumplir. Las técnicas de caja blanca son pruebas que se basan en el código de la unidad. Lo fundamental de estas pruebas es lograr cubrir (ejecutar) ciertas sentencias y condiciones del código.

Existen diferentes técnicas de caja negra y caja blanca. La introducción de las pruebas unitarias en una organización debe hacerse con una clara definición de qué técnicas van a ser utilizadas. Sin esta definición distintas unidades construidas por distintos desarrolladores tendrán diferencias notorias en la calidad de la unidad liberada.

Una de las definiciones más importantes a realizar es cuándo se van a diseñar y ejecutar estas pruebas. Aquí es donde se define si la organización va a realizar code-fix o no. Una mala idea es depender de las pruebas unitarias; diversos estudios empíricos indican que la efectividad de la misma es de aproximadamente 50%. Esto quiere decir que si detectamos 4 defectos durante estas pruebas (y los corregimos), la unidad tendrá aproximadamente otros 4 defectos más al ser liberada. Por suerte la forma de no depender de estas pruebas, y sacar el mayor provecho de las mismas, es muy sencilla: revisar antes de probar. El momento de ejecutar las pruebas unitarias es siempre luego de estar seguros que lo que se ha construido es una unidad de calidad. Para esto es necesario diseñar la unidad, revisar el diseño, codificar la unidad y revisar el código de forma estática. Luego, las pruebas unitarias serán una validación de que lo que se ha construido es una unidad de calidad, detectando y corrigiendo muy pocos defectos durante las mismas. Lamentablemente, la dependencia en las pruebas unitarias es un error común en la industria de desarrollo de software.

Aplicando lo anterior a Java, JUnit es un conjunto de clases opensource que nos permiten probar nuestras aplicaciones Java. Podemos encontrar información actualizada de JUnit en http://www.junit.org

* 1. **Razones para 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 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 cómo se comporta un caso de prueba para hacer refactoring unitario sobre la clase que estamos probando.

•Permite incluso hacer pruebas de estress 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 corrigan los errores, menos costará corregirlos.

* 1. **Test drive development (TDD)**

Existe una metodología de desarrollo denominada desarrollo guiado por pruebas (Test Driven Development, TDD) que se basa precisamente en implementar primero las pruebas y a partir de ellas implementar las funcionalidades de nuestra aplicación.

Para asegurarnos que la aplicación funciona correctamente y sin ningún bug, la única manera es realizar todas las pruebas posibles en todos los escenarios. Como ya hemos comentado esto es un trabajo largo y tedioso, con lo que optamos por la realización de pruebas automáticas. Pero ¿cómo podemos asegurarnos de que hemos cubierto el 100% del código con pruebas? La manera más eficiente de lograrlo es escribir las pruebas antes que el código. De aquí nace la metodología TDD o Test drive development.

Test drive development o TDD a partir de ahora, es una metodología de diseño e implementación de software, esta metodología se centrada en 3 pilares básicos:

* La implementación de las funciones justas que el cliente necesita y no más.
* La minimización del número de defectos que llegan al software en fase de producción.
* La producción de software modular, altamente reutilizable y preparado para el cambio.

**TDD sigue un sencillo algoritmo de 3 pasos:**

1. Escribir la especificación del requisito (el ejemplo, el test).
2. Implementar el código según dicho ejemplo.
3. Refactorizar para eliminar duplicidad y hacer mejoras.

Una vez terminado los 3 pasos el algoritmo realizamos otra iteración completa, y así sucesivamente hasta que abarcamos el 100% de los requisitos.

**Ventajas**

* Los programadores que usan TDD para un proyecto desde cero son raras las veces que necesitan usar un debbuger.
* Ayuda a producir aplicaciones de más calidad y en menos tiempo.
* Además de validar el cumplimiento de los requisitos, también puede guiar el diseño de un programa.
* Cuando es bien utilizado, nos asegura que todo el código está cubierto por una prueba.

**Limitaciones**

El desarrollo guiado por pruebas requiere que las pruebas puedan automatizarse. Esto resulta complejo en los siguientes dominios:

* Interfaces Gráfica de usuario (GUIs).
* Objetos distribuidos, aunque los objetos simulados (MockObjects) pueden ayudar.
* Bases de datos. Hacer pruebas de código que trabaja con base de datos es complejo porque requiere poner en la base de datos unos datos conocidos antes de hacer las pruebas. Todo esto hace que la prueba sea costosa de codificar.
  1. **Mock objects**

Un Mock es solo un objeto de prueba. (simple o complejo de acuerdo a cada caso particular).

En casos de ejemplos muy sencillos, en los que el método a probar recibe todos los datos de entrada necesarios mediante parámetros, se tiene controlados los escenarios de pruebas. Sin embargo, en una aplicación real en la mayoría de los casos el comportamiento de los métodos no dependerá únicamente de los parámetros de entrada, sino que también dependerá de otros datos, como por ejemplo datos almacenados en una base de datos, o datos a los que se accede a través de la red. Estos datos también son una entrada del método de probar, pues el resultado del mismo depende de ellos, por lo que en nuestras pruebas de JUnit deberíamos ser capaces de fijarlos para poder predecir de forma determinista el resultado del método a probar.

Sin embargo, dado que muchas veces dependen de factores externos al código de nuestra aplicación, es imposible establecer su valor en el código de JUnit.

Si estamos trabajando en equipo, mucho de nuestro código impacta a otras personas y viceversa, de todo este "proyecto/s" y muchas veces nos codeamos con cosas que presumimos se van a desarrollar más adelante, debido a la complejidad de la aplicación o por su diseño. Para paliar un poco estas situaciones se suele utilizar una estrategia con “Mock objects” u objetos simulados.

Esta estrategia consiste en generar objetos que cumplan con la interfaz que necesitemos pero sin implementación. Para realizar este cometido existen numerosas librerías en java. Algunas de ellas son:

* jMock – <http://www.jmock.org/>
* Mockito – <https://code.google.com/p/mockito/>
* EasyMock – <http://easymock.org/>

# Capítulo 2 Caso Demostrativo.

Dentro de los elementos que tiene incorporado el IDE NetBeans, se encuentra JUnit. Para emplear JUnit se debe realizar los siguientes procedimientos:

1. Se compila la clase que desea probar y se deja abierta (su código aparece en la ventana superior derecha de la aplicación).
2. Clic derecho sobre la clase Pruebas, seleccionar nuevo, al final del menú seleccionar Otro; se abre una ventana donde se ofrecen opciones, seleccionar en categorías “Unit Test”, en tipos de archivos escoger “JUnit Test”. Por default se llamará Test <nombre de la clase a probar>. Dar OK.
3. En un directorio asociado al proyecto donde se utiliza la clase que se va a probar, se guarda la clase probadora. Por default crea métodos de prueba por cada método de la clase y traen llamadas a fail (método que siempre falla y envía un mensaje).
4. Se reemplazan los métodos que se desea usar y se agregan otros de ser necesario.
5. Se compila la clase probadora.
6. En el menú Run, opción Other, se selecciona Test <nombre de clase a probar> y en la ventana de abajo aparecerá la salida de las pruebas. A diferencia del método a pie, aparecerá una línea por caso de prueba ejecutado.

**Aserciones de JUnit**

Las aserciones se utilizan para evaluar si el valor retornado por el método es el esperado.

A continuación se presenta una tabla describiendo las aserciones de JUnit:

* assertEquals: Evalúa si dos objetos pasados por parámetro son iguales.
* assertFalse: Evalúa si la condición pasada es false.
* assertNotNull: Comprueba que la condición no sea nula.
* assertNotSame: Comprueba que dos objetos no sean la misma instancia.
* assertNull: Comprueba si un objeto es nulo.
* assertSame: Opuesto a assertNotSame.
* assertTrue: Opuesto a assertFalse.
* fail: Útil para detectar si estamos en un sitio del programa donde no deberíamos estar.

**Caso de aplicación:**

Se requiere un Software que permita calcular la nota final de un alumno e indicar si este aprueba o no el curso. Los datos a ingresar deben ser el Nombre del curso, Nombre del Alumno, Nota Práctica 1 y 2, Nota de Participación activa y Evaluación final. Cada nota obtenida tendrá un peso específico en relación al puntaje final, según se detalla a continuación:

Practica1: 20%

Practica1: 25%

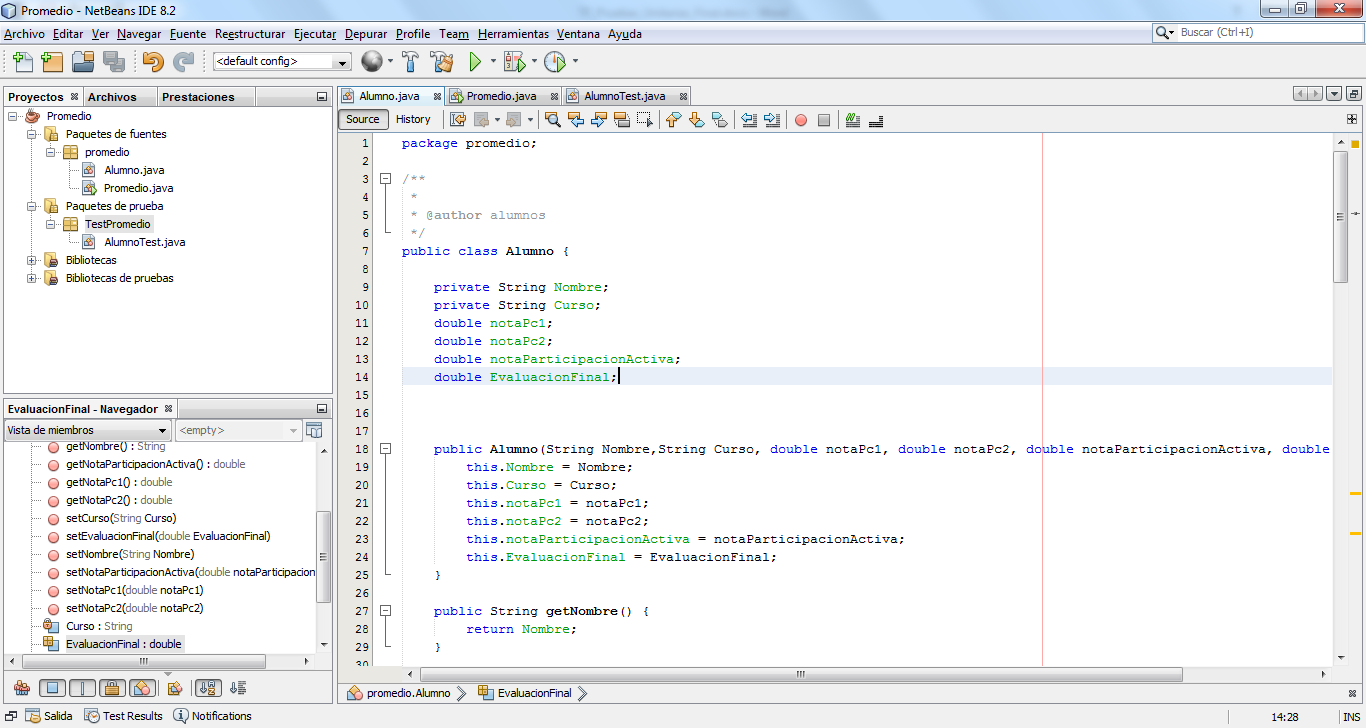
Participación activa: 25%

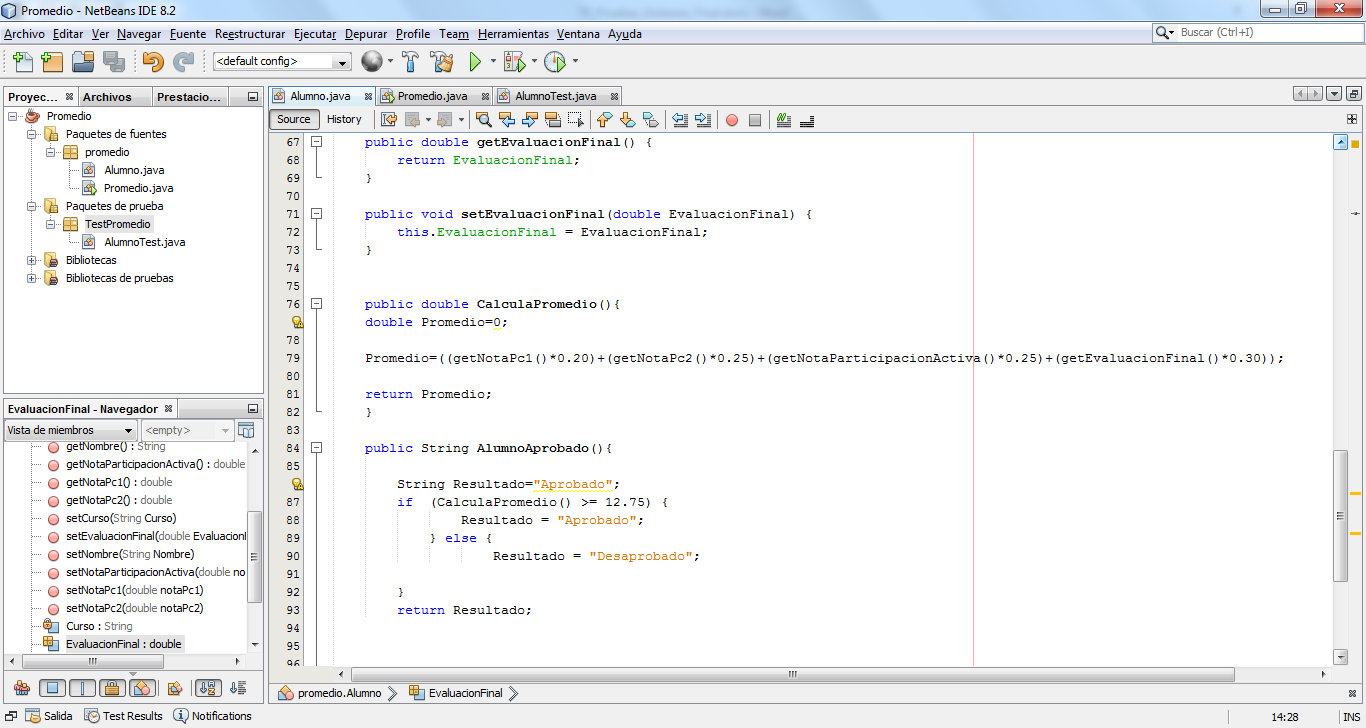
Evaluación final: 30%

**Paquete fuente: Promedio**

Clase Principal: Promedio

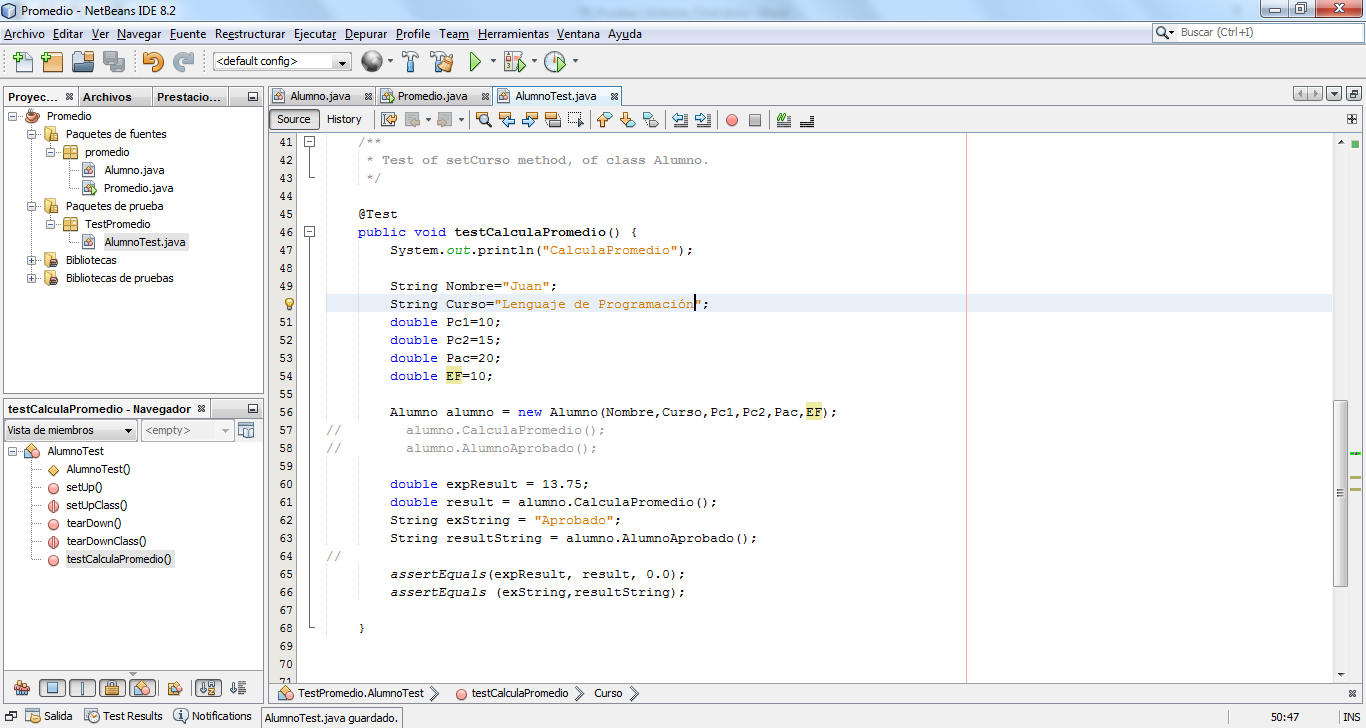
Clase Model y Service: Alumno





**Paquete prueba: TestPromedio**

Clase @test: AlumnoTest



# Capítulo 3 Discusión.

En el transcurso del curso y en los casos investigados, podemos evidenciar que cuando probamos un programa, lo ejecutamos con unos datos de entrada (casos de prueba) para verificar que el funcionamiento cumple los requisitos esperados. Definimos prueba unitaria como la prueba de uno de los módulos que componen un programa.

Esto nos lleva a reflexionar, que pueden existir errores tanto en la implementación como en la prueba. Pero es importante notar que sin pruebas confiamos ciegamente en la implementación. Si hay un error al correr las pruebas, el error puede estar en la implementación o en las pruebas. Si las pruebas corrieron exitosamente, entonces la implementación cumple con los casos presentados en las pruebas; en otras palabras, puede que tanto la implementación como las pruebas estén mal. Obviamente, es más difícil equivocarse en dos lugares (implementación y pruebas) que sólo en uno (implementación).

También es importante mencionar que las pruebas sólo garantizan que nuestro programa cumple con los escenarios mencionados, más no que está 100% libre de errores.

El objetivo no es crear tantas pruebas como sea posible. Más bien, es crear el mínimo número de pruebas que cubran el mayor número de casos o escenarios de nuestra implementación.

# Capítulo 4 Conclusiones.

Al desarrollar un nuevo software o sistema de información, la primera etapa a considerar es la etapa de pruebas unitarias. En la cual como mencionamos anteriormente, se encuentran presentes las pruebas de caja negra y de caja blanca en las cuales, en una se realiza un análisis de los datos de entrada y de salida y en otro se analiza el proceso interno del sistema para evaluar las inconsistencias que pueda estar presentando el sistema, para así llegar a una corrección de los mismos y proseguir con la nueva fase del proceso de desarrollo del sistema.

Podemos destacar que las pruebas unitarias son una forma de probar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.

El objetivo fundamental de las pruebas unitarias es asegurar el correcto funcionamiento de las interfaces, o flujo de datos entre componentes de manera tal que a la hora de realizar una unificación de los diferentes componentes que conforman el sistema en general, exista una congruencia que favorezca el desarrollo de la aplicación que se quiere realizar.

# Capítulo 5 Recomendaciones.

Este tipo de pruebas debe ser realizado por personal especializado en la aplicación de pruebas a nivel de software, el cual debe estar familiarizado en el uso de herramientas de depuración y pruebas, igualmente deben conocer el lenguaje de programación en el que se está desarrollando la aplicación.

Separar pruebas unitarias (comprueban una funcionalidad concreta) de las de integración (prueban varios elementos). Así será más fácil diagnosticar dónde se producen los errores.

Documentar en el javadoc de cada método de test los pasos que ejecuta y los resultados esperados. Así será más fácil arreglarlo cuando falle por un cambio en la aplicación.

Indicar los datos en ficheros de xml o properties en lugar de en el código Java. Normalmente se repiten los datos en múltiples tests y así sólo habrá que cambiarlos en un sitio.

Integración continua: lanzar todos los tests al desplegar una versión para comprobar que sigue funcionando todo lo de la versión anterior.

Crear una clase DefaultTestSet de la que extiendan las clases que contienen tests. En ella se crearían los objetos necesarios, se leerían los ficheros de configuración, etc.

Usar los métodos @before y @after para agrupar pasos que se repitan al principio y al final de todos los métodos de una clase. Así se evita repetir código, y cuando hay que hacer cambios sólo se modifica en un lugar.

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

**Técnicas para hacer nuestro código testeable.**

• Codificar sobre interfaces más que sobre clases. Mientras más usemos interfaces más plugabilidad de soluciones tendremos en run­time o test­time. De esta forma estamos probando el “que” y no el “como” y esto es precisamente en lo que debe enfocarse una prueba unitaria

Usar la Ley de Demeter. Esta ley dice que un objeto solo debe depender de objetos que estén muy cercanos a él, o sea no debería usar objetos globales.

# Bibliografía

Avila, A., Camilloni, L., Marotta, F. (S.F.). Pruebas Unitarias en Java JUnit y TestNG. Grupo de Ingeniería de Software, UdelaR.

Fernandez, J. (2005). Pruebas de unidad utilizando JUnit.

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

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