**Pruebas unitarias**



**Alumno: Edinson Manuel sotil tixe**

**Profesor: Eric Gustavo Coronel Castillo**

**Curso: Herramientas para el Desarrollo de Software**

**Carrera: computación informática**

**Resumen**

En programación, una **prueba unitaria** es una forma de comprobar el correcto funcionamiento de una unidad de código. Por ejempen diseño estructurado o en diseño funcional una función o un procedimiento, en diseño orientado a objetos una clase. Esto sirve para asegurar que cada unidad funcione correctamente y eficientemente por separado. Además de verificar que el código hace lo que tiene que hacer, verificamos que sea correcto el nombre, los nombres y tipos de los parámetros, el tipo de lo que se devuelve, que si el estado inicial es válido entonces el estado final es válido

La idea es escribir casos de prueba para cada función no trivial o método en el módulo, de forma que cada caso sea independiente del resto. Luego, con las Pruebas de Integración, se podrá asegurar el correcto funcionamiento del sistema o subsistema en cuestión.

**Introducción**

Todos los programadores saben que deben realizar pruebas a su código, pocos lo hacen de manera proactiva, la respuesta generalizada al “¿por qué no?” es “no tengo tiempo”.

Este apuro se convierte en un círculo vicioso. Cuanta más presión se siente, menos pruebas se realizan. Cuantas menos pruebas se realizan, menos productivo se es y el código se vuelve menos estable. Cuanto menos productivo y preciso se es, más presión se siente.

**Desarrollo**

Una prueba unitaria, o “unit test”, es un método que prueba una unidad estructural de código.

Contrariamente a lo que piensan muchos desarrolladores –que el desarrollo de pruebas unitarias resta tiempo a tareas más importante– las pruebas unitarias por lo general son simples y rápidas de codificar, el desarrollo de una prueba unitaria no debería tomar más de cinco minutos.

Debido a la diversidad de definiciones, convendremos que una “buena” prueba unitaria tiene las siguientes características:

* **Unitaria**, prueba solamente pequeñas cantidades de código.
* **Independiente**, no debe depender ni afectar a otras pruebas unitarias.
* **Prueba métodos públicos**, de otra forma la prueba sería frágil a cambios en la implementación y no se podría utilizar en pruebas de regresión.
* **Automatizable**, la prueba no debería requerir intervención manual.
* **Repetible y predecible**, no debe incidir el orden y las veces que se repita la prueba, el resultado siempre debe ser el mismo.
* **Profesionales**, las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.
* Respecto al último punto y contrariamente a lo que piensan muchos desarrolladores –que el desarrollo de pruebas unitarias resta tiempo a tareas más importante– las pruebas unitarias por lo general son simples y rápidas de codificar, el desarrollo de una prueba unitaria no debería tomar más de cinco minutos.

#### Ventajas

* Las pruebas unitarias buscan aislar cada parte del programa y mostrar que las partes individuales son correctas, proporcionando cinco ventajas básicas:
* **Fomentan el cambio**, las pruebas unitarias facilitan la reestructuración del código (refactorización), puesto que permiten hacer pruebas sobre los cambios y verificar que las modificaciones no han introducido errores (regresión).
* **Simplifican la integración**, permiten llegar a la fase de integración asegurando que las partes individuales funcionan correctamente. De esta manera se facilitan las pruebas de integración.
* **Documentan el código**, las propias pruebas pueden considerarse documentación, ya que las mismas son una implementación de referencia de como utilizar el código.
* **Separación de la interfaz y la implementación**, la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro (ver pruebas mock).
* **Menos errores y más fáciles de localizar**, las pruebas unitarias reducen la cantidad de errores y el tiempo en localizarlos.
* **Pueden mejorar el diseño**, la utilización de prácticas de diseño y desarrollo dirigida por las pruebas (Test Driven Development o TDD) permite definir el comportamiento esperado en un paso previo a la codificación.
* **Puede ser la forma más simple de verificar el funcionamiento**, en situaciones como el desarrollo de una API o un componente que brinda servicios del cual no se cuenta aún con un cliente para consumirlos.
* Adoptar el uso de pruebas unitarias como una disciplina generalizada puede ser un trabajo costoso, pero no debe ser considerado una desventaja, debe entenderse que esta actividad nos servirá para ahorrar tiempo en el futuro al disminuir la ocurrencia de errores.
* La utilización de pruebas unitarias permitirá mejorar progresivamente la calidad del código en la medida que los desarrolladores aumenten la calidad de las pruebas y la cobertura del mismo, por ejemplo, **"en**
* **MicroGestion durante el año 2011 se dedicaron 1160 horas a la resolución de errores en la fase de desarrollo y en el período de garantía, y en 2012 se registraron 710 horas**

Vamos a utilizar JUnit en netbeans . Para ello deberemos crear una serie de clases en las que implementaremos las pruebas diseñadas. Esta implementación consistirá básicamente en invocar el método que está siendo probado pasándole los parámetros de entrada establecidos para cada caso de prueba, y comprobar si la salida real coincide con la salida esperada. Esto en principio lo podríamos hacer sin necesidad de utilizar JUnit, pero el utilizar esta herramienta nos va a ser de gran utilidad ya que nos proporciona un *framework* que nos obligará a implementar las pruebas en un formato estándar que podrá ser reutilizable y entendible por cualquiera que conozca la librería. El aplicar este *framework* también nos ayudará a tener una batería de pruebas ordenada, que pueda ser ejecutada fácilmente y que nos muestre los resultados de forma clara mediante una interfaz gráfica que proporciona la herramienta. Esto nos ayudará a realizar pruebas de regresión, es decir, ejecutar la misma batería de pruebas en varios momentos del desarollo, para así asegurarnos de que lo que nos había funcionado antes siga funcionando bien.

Para implementar las pruebas en JUnit utilizaremos dos elementos básicos:

* Por un lado, marcaremos con la anotación @Test los métodos que queramos que JUnit ejecute. Estos serán los métodos en los que implementemos nuestras pruebas. En estos métodos llamaremos al método probado y comprobaremos si el resultado obtenido es igual al esperado.
* Para comprobar si el resultado obtenido coincide con el esperado utilizaremos los métodos assert de la librería JUnit. Estos son una serie de métodos estáticos de la clase Assert (para simplificar el código podríamos hacer un *import* estático de dicha clase), todos ellos con el prefijo assert-. Existen multitud de variantes de estos métodos, según el tipo de datos que estemos comprobando (assertTrue, assertFalse, assertEquals, assertNull, etc). Las llamadas a estos métodos servirán para que JUnit sepa qué pruebas han tenido éxito y cuáles no.
* Cuando ejecutemos nuestras pruebas con JUnit, se nos mostrará un informe con el número de pruebas éxitosas y fallidas, y un detalle desglosado por casos de prueba. Para los casos de prueba que hayan fallado, nos indicará además el valor que se ha obtenido y el que se esperaba.
* Además de estos elementos básicos anteriores, a la hora de implementar las pruebas con JUnit deberemos seguir una serie de buenas prácticas que se detallan a continuación:
* La clase de pruebas se llamará igual que la clase a probar, pero con el sufijo -Test. Por ejemplo, si queremos probar la clase MiClase, la clase de pruebas se llamará MiClaseTest.
* La clase de pruebas se ubicará en el mismo paquete en el que estaba la clase probada. Si MiClase está en el paquete es.ua.jtech.lja, MiClaseTest pertenecerá e ese mismo paquete. De esta forma nos aseguramos tener acceso a todos los miembros de tipo protegido y paquete de la clase a probar.
* Mezclar clases reales de la aplicación con clases que sólo nos servirán para realizar las pruebas durante el desarrollo no es nada recomendable, pero no queremos renunciar a poner la clase de pruebas en el mismo paquete que la clase probada. Para solucionar este problema lo que se hará es crear las clases de prueba en un directorio de fuentes diferente. Si los fuentes de la aplicación se encuentran normalmente en un directorio llamado src, los fuentes de pruebas irían en un directorio llamado test.
* Los métodos de prueba (los que están anotados con @Test), tendrán como nombre el mismo nombre que el del método probado, pero con prefijo test-. Por ejemplo, para probar miMetodotendríamos un método de prueba llamado testMiMetodo.
* Aunque dentro de un método de prueba podemos poner tantos assert como queramos, es recomendable crear un método de prueba diferente por cada caso de prueba que tengamos. Por ejemplo, si para miMetodo hemos diseñado tres casos de prueba, podríamos tener tres métodos de prueba distintos: testMiMetodo1, testMiMetodo2, y testMiMetodo3. De esta forma, cuando se presenten los resultados de las pruebas podremos ver exactamente qué caso de prueba es el que ha fallado.

### Pruebas utilizando objetos simulados (mock)

Las pruebas Mock son pruebas unitarias que utilizan objetos simulados ("mock") que sustituyen a los objetos reales utilizados por la clase o fragmento de código a probar.

Por ejemplo, se tiene una clase Calculator que necesita de un DAO (Objeto de Acceso a Datos) para obtener información de la base de datos, este DAO es lo que llamamos el “objeto real”. Si quisiéramos realizar una prueba de la clase Calculator, deberíamos pasarle al objeto una conexión válida a la base de datos y asegurarnos de que los datos que necesita existan.

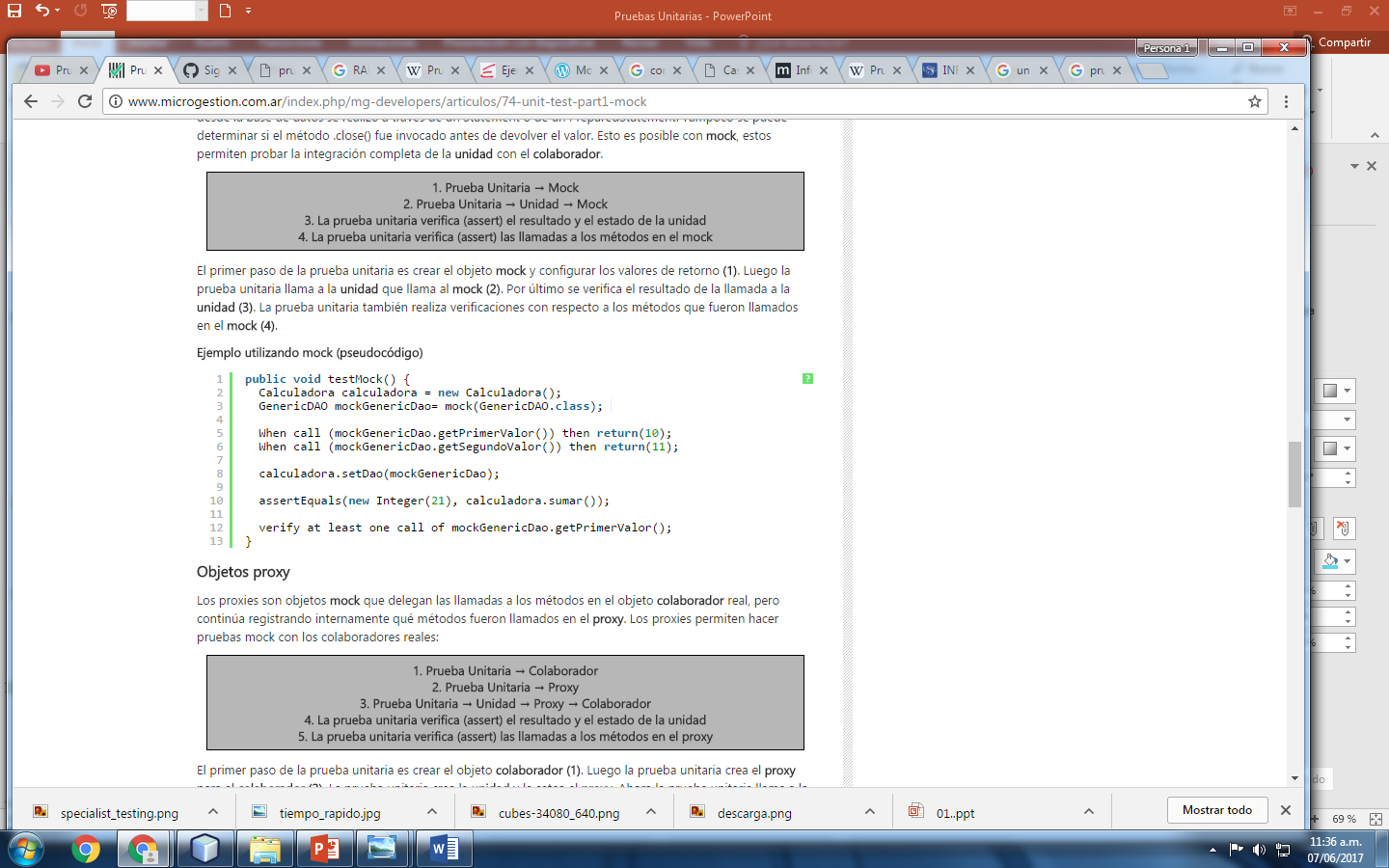
Este proceso de determinar los datos que el objeto necesita e insertarlos en la base requiere de mucho trabajo.

En lugar de esto, se puede proveer una instancia **falsa** del DAO que sólo devuelva lo que necesitamos para la prueba. Esta clase no va a tomar la información de la base de datos, sino que va a ser un **mock**.

Reemplazar el objeto real hace que probar la clase Calculator sea mucho más simple. Estrictamente hablando, dicho objeto que reemplaza al DAO real no sería un **mock**, sino un **stub**. Luego hablaremos de esta diferencia.

Este ejemplo puntual de la clase Calculator se puede generalizar diciendo que es una unidad (Calculator) utilizando un **colaborador** (DAO).

**Ejemplo**



#### **Consideraciones**

* Como en la adopción de cualquier otra disciplina, la incoporación de pruebas unitarias no está exenta de problemas o limitaciones, a continuación se enumeran algunas consideraciones a tener en cuenta
* Por estar orientada a la prueba de fragmentos de código aislados, la pruebas unitarias no descubrirán errores de integración, problemas de rendimiento y otros problemas que afectan a todo el sistema en su conjunto.
* En alguno casos será complicado anticipar inicialmente cuales son los valores de entradas adecuados para las pruebas, en esos casos las pruebas deberán evolucionar e ir incorporando valores de entrada representativos.
* Si la utilización de pruebas unitarias no se incorpora como parte de la metodología de trabajo, probablemente, el código quedará fuera de sincronismo con los casos de prueba.
* Otro desafío es el desarrollo de casos de prueba realistas y útiles. Es necesario crear condiciones iniciales para que la porción de aplicación que está siendo probada funcione como parte completa del sistema al que pertenece.
* Escribir código para un caso de pruebas unitario tiene tantas probabilidades de estar libre de errores como el mismo código que se está probando.

**Conclusiones**

En este primer artículo de la serie dedicada a las Pruebas Unitarias hemos realizado una introducción al tema, realizamos una descripción de las ventajas que supone su adopción, e intentamos incorporar información de utilidad verificada durante la implementación de la disciplina dentro de MicroGestion.

La introducción nos sirvió como base para describir de manera teórica la utilización de objetos mock en los casos que es necesario simular los objetos utilizados por las clases cuyo funcionamiento necesitamos verificar

**Fuentes**

[**https://es.wikipedia.org/wiki/Prueba\_unitaria**](https://es.wikipedia.org/wiki/Prueba_unitaria)

[**http://www.jtech.ua.es/j2ee/publico/lja-2012-13/sesion04-apuntes.html**](http://www.jtech.ua.es/j2ee/publico/lja-2012-13/sesion04-apuntes.html)

**https://www.youtube.com/watch?v=PmnuBzZ6cjE**