A picture containing text

Description automatically generated

Especificación y diseño de software

Prof. Daniel Eduardo Madriz Huertas.

Semestre I

2022

Investigación #1: Unit Testing

José Alejandro Chavarría Madriz 2019067306

Sebastián Mora Godínez 2019??????

Índice

[Pruebas unitarias 3](#_Toc97999330)

[Definición 3](#_Toc97999331)

[Beneficios y limitaciones 3](#_Toc97999332)

[Framework de pruebas unitarias en Java: JUnit 6](#_Toc97999333)

[Framework de pruebas unitarias en JavaScript 8](#_Toc97999334)

[Ejemplo aplicado en Java 9](#_Toc97999335)

[Ejemplo aplicado en ¿? 14](#_Toc97999336)

[Conclusiones 14](#_Toc97999337)

[Bibliografía 15](#_Toc97999338)

# Pruebas unitarias

## Definición

El concepto prueba unitaria se puede definir en términos generales, como la forma de validar que pequeñas piezas de código llamadas unidad funcionen tal y como se espera, pero si se va analiza el térmico con más profundidad, se pueden encontrar con diversas definiciones que tratan de enfocar o definir este concepto de diferentes maneras. Sin embargo, todas las definiciones coinciden que todas las pruebas unitarias deben de:

* + - * Verificar pequeñas piezas de código llamadas unidad.
      * Realizar esta verificación de manera rápida.
      * Hacerlo de manera aislada.

## Beneficios y limitaciones

Las pruebas unitarias, así como muchos otros métodos de pruebas automatizadas, han tomado muchísima importancia en la escena del desarrollo de software. En muchas organizaciones las pruebas automatizadas son completamente obligatorias, el código y sus pruebas son uno solo. [lab] Y por supuesto, las pruebas unitarias son parte principal del proceso de testing.

Esta importancia que reciben las pruebas unitarias está completamente fundamentada en los beneficios que traen consigo. Dentro de los principales beneficios se puede citar [lab] [kralj]:

* Mejora la calidad del código: La simple necesidad que escribir código que debe ser testeado más tarde obliga al programador de ser consciente de la manera en que escribe su código. Debe asegurarse de que su cogido tenga alta cohesión y poco acoplamiento, es decir, que las dependencias de su código sean mínimas y accesibles, así como mantener las clases pequeñas y específicas.
* Encontrar bugs tempranamente: Gracias a la retroalimentación que ofrecen las pruebas unitarias muchos errores pueden ser encontrados en durante las pruebas y no en producción. Los programadores, asumiendo que existe la cantidad necesaria de pruebas unitarias, son capaces de detectar un error básicamente en el momento en que es programado.
* Facilita el cambio: Con un soporte robusto de pruebas unitarias cambios grandes al código pueden ser realizados con mucha más facilidad y tranquilidad, ya que un error ocasionado por el cambio puede ser rápidamente detectado, es decir, el cambio puede ser probado muy rápidamente y asegurar que el código se mantiene funcional.
* Provee documentación: la existencia de las pruebas ofrece una plataforma a otros programadores para entender el funcionamiento individual de cada módulo. La serie de casos analizados en el código de la prueba unitaria ofrece un vistazo a la manera en que se espera que el código se comporte.
* Agilidad: La naturaleza de las pruebas automatizadas es aumentar la productividad, ahorrar tiempo de pruebas manuales y permitir que el código “se pruebe solo”. Las pruebas unitarias abren paso al “Test Driven development” y en general a un desarrollo más rápido de integración continua. Fragmentos de código ya probados son generados más rápida y frecuentemente.

Como se menciona anteriormente beneficios son suficientes para convertir a las pruebas en un procedimiento estándar en muchas ocasiones. Pero como cualquier otra inversión, las pruebas unitarias poseen un costo de oportunidad, o desventajas, que se deben de tener en cuenta, por ejemplo [kralj]:

* Costo en recursos: La creación de pruebas unitarias puede resultar un proceso costo, tanto en tiempo como en personal. La naturaleza de las pruebas puede ser compleja y cada una debe ser programada manualmente. Y, lógicamente, se requiere de personal con las habilidades y conocimiento suficientes para crear las pruebas unitarias.
* Difícil de implementar en código de legado: La mejor forma de programar pruebas unitarias es durante del desarrollo del código mismo. Un tercero al que se le entrega un código desconocido sin ninguna prueba, e inclusive con pobre documentación (o ninguna), va a tener un reto enorme intentando crear pruebas unitarias para este.
* Poca aplicabilidad en GUI: El código de las interfaces graficas no apropiado para un set de pruebas unitarias. Puede ser sumamente complicado escribir código de pruebas unitarias para validar la funcionalidad de la interfaz. Es mejor considerar otro tipo de pruebas más adecuados.
* No es infalible: A pesar del gran impacto que pueden tener las pruebas unitarias a la hora de encontrar errores, estas están limitadas a la destreza del programador y la cantidad de recursos que se vayan a destinar a la creación de las pruebas. Para sistemas grandes las pruebas unitarias ayudan a mantener muchos más escenarios de uso bajo control, pero es prácticamente imposible asegurar funcionalidad perfecta en todos los escenarios.

# Framework de pruebas unitarias en Java: JUnit

JUnit es un open-source framework de pruebas unitarias para Java. Es de los mas utilizados, de hecho, datos de GitHub muestran que JUnit está en primer lugar con un 62% de presencia en proyectos en Java. [overops] Fue originalmente desarrollado por Kent Beck y Erich Gamma bajo la filosofía de “primero testear, luego codificar”, por lo que JUnit es sumamente sencillo, esto con el fin de aumentar la velocidad y la calidad con la que se escribe código.

JUnit es capaz de ejecutar pruebas automáticamente y comparar los resultados del código inmediatamente con los resultados esperados, dando feedback en el momento. Cuenta con con *annotations* o anotaciones, que permiten identificar métodos específicos de testing, y con *assertions* o afirmaciones, que verifican los resultados. Entre las anotaciones existen:



Estas se escriben justo antes del método y determinan en qué momento del ciclo de pruebas serán ejecutados. Por otra parte, las afirmaciones permiten validar los resultados esperados, algunas son:



Estos deben ser escritos dentro de un método con alguna de las anotaciones anteriores.

Para ejecutar las pruebas Junit es compatible con muchos IDEs del mercado [junit], incluyendo Intellij IDEA, Eclipse y Visual Studio Code. Además, posee una implementación por consola de comandos. Junit permite correr cada prueba de manera individual, correr toda una clase de pruebas e inclusive paquetes enteros con múltiples clases de pruebas. Si son exitosas el IDE marcará la prueba con verde, de lo contrario mostrará el error.

# Framework de pruebas unitarias en JavaScript

Jest es un framework de pruebas unitarias creador por Facebook para testing en JavaScript. Es altamente usado en React pero es utilizado en todo tipo de bibliotecas o framework. Jest es conocido por enfocarse en la simplicidad de las pruebas y al tratarse de un framework no necesita de dependencias adicionales para funcionar como es el cado de Mocha.

Jsest ofrece métodos llamados *matchers* que permiten verificar diferentes valores. El método *toBe* de estos *matchers* que ofrece Jest, permite verificar que dos valores son iguales.

test('two plus two is four', () => {  
 expect(2 + 2).toBe(4);  
 });

En caso que se desee verificar si dos objetos son iguales se debe utilizar el método *toEqual.*

# Ejemplo aplicado en Java con JUnit

Para este ejemplo se ha creado una aplicación sencilla que contiene una clase llamada *BestMovieOscar*. Su constructor recibe número entero de 2010 a 2020 que representa el año, y a partir del mismo asigna un valor a sus atributos *year*, *name*, *director*, *genre*. El constructor de la clase se ve así:



Los métodos *setName*, *setDirector* y *setGenre*, son una serie de cláusulas if que dependiendo del año asignan el string indicado, por ejemplo:



Adicionalmente existe un método que retorna un array con todos estos datos:



Conociendo la naturaleza de la clase, podemos analizar algunos posibles casos de pruebas implementados en JUnit, todos escritos en una clase aparte a la cual se le denominó *BestMovieOscarTest.*

El primer test que se realiza es para el año 2010, tal que:



Aquí se esta asegurando que al crear la instancia de la clase los atributos sean los correctos mediante assertEquals, además de probar indirectamente que los métodos *get* sean efectivos. (Nota: Aquí podemos vislumbrar una de las limitaciones de las pruebas unitarias, si se deseará probar cada posible resultado habría que hacer un test igual al anterior para cada año disponible, lo cual consume tiempo).

Si la prueba es exitosa se mostrará un checkmark (en VSCode) al lado de la línea del método de pruebas.

Text

Description automatically generated

Otra prueba que se puede realizar y que tiene un comportamiento un tanto diferrent es aquella que utiliza el *assertThrows* pues para el correcto funcionamiento se ha de utilizar como segundo arguemnto una función lamba de contenga el código que dispara el arguemto. Para ello se escribe una prueba unitaria para los casos extremos de los años 2009 y 2021.



Si la ejecución del código dentro del segundo argumento en forma de funcion lamda no genera la clase de argumento en la primera entrada de la función la prueba falla, de lo contrario el check aparecerá.

El conjunto de pruebas puede verse al lado izquierdo en el editor de código, cada una con el nombre del método que se ha escogido:

Graphical user interface, text

Description automatically generated

Esto permite verificar rápidamente el resultado de todas las pruebas.

Ahora, imaginemos que por error para el año 2010 el if dentro del *setName* es cambiado a 2011, de forma que el código tendría un error donde para ese año no se asiganria un dato a este atributo:



Ni el compilador, ni el IDE son capaces de determinar que existe un bug, pues el código es completamente funcional. Sin embargo, al ejecutar el archivo de pruebas unitarias nos encontraremos con que no todas las pruebas son exitosas:

A picture containing graphical user interface

Description automatically generated

Y al acceder al error veremos un mensaje que nos describe exactamente lo que se esperaba y lo que se recibió en realidad:

A screenshot of a computer

Description automatically generated with medium confidence

Así la prueba unitaria a detectado efectivamente un problema que debe ser solucionado. (Nota: Para esta implementación esto es un golpe de suerte, dejando más clara la no infalibilidad de las pruebas unitarias, pues si dicho error hubiera ocurrido para un año sin una prueba unitaria podría haber pasado desapercibido).

# Ejemplo aplicado en ¿?

# Conclusiones

* …
* …
* Las pruebas unitarias pueden mejorar la calidad del código.
* Las pruebas unitarias pueden ayudar a encontrar bugs de manera temprana.
* Las pruebas unitarias facilitan los cambios en el código.
* Las pruebas unitarias son una fuente indirecta de documentación
* Las pruebas unitarias son capaces de aumentar la agilidad de un proyecto.
* Las pruebas unitarias tienen un costo significativo en recursos.
* Las pruebas unitarias no son fáciles de implementar en código de legado.
* Las pruebas unitarias son complicadas de generar para código de GUI.
* La efectividad de las pruebas unitarias depende en gran medida de la calidad y la cantidad de código que evalúa.
* La efectividad de las pruebas unitarias depende en gran medida de la calidad y la cantidad de código con las que son programadas.
* JUnit es uno de los frameworks de pruebas unitarias más utilizados para Java.
* JUnit es facil de utilizar gracias su simpleza e integración con IDEs de uso común.
* …
* …

# Bibliografía

[?] PerformanceLab, “Why Is Unit Testing Important in Software Development?,” 2021. [Online]. Available: <https://performancelabus.com/unit-testing-importance/#1>

[?] JUnit, “JUnit 5,” 2022. [Online]. Available: <https://junit.org/junit5/>

[?] K. Kralj, “What Are Advantages and Disadvantages of Unit Testing?,” 2022. [Online]. Available: <https://methodpoet.com/unit-testing-advantages-and-disadvantages/>

[?] A. Zhitnitsky, “JUnit vs TestNG: Which Testing Framework Should You Choose?,” 2016. [Online]. Available: <https://www.overops.com/blog/junit-vs-testng-which-testing-framework-should-you-choose>