



Test-First Programming

¿Por qué hacer pruebas?





¿Los programadores reales no hacen prueba?

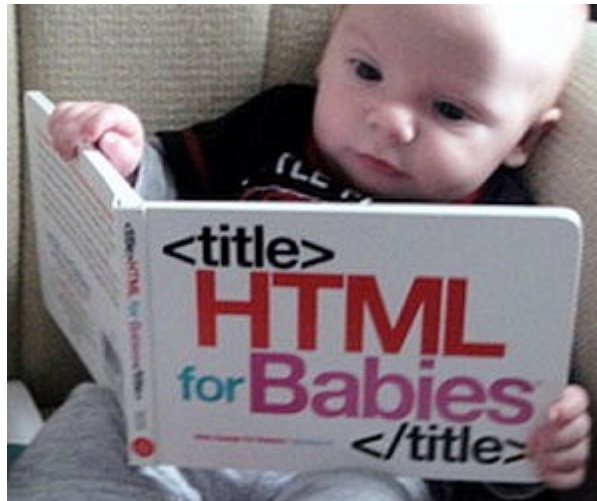
Aquí están las 5 principales razones por las que un estudiante no quiere probar su código:

- 5) Quiero terminar esto rápido, las pruebas me van a retrasar.

¿Los programadores reales no hacen prueba?

Aquí están las 5 principales razones por las que un estudiante no quiere probar su código:

- 4) Empecé a programar cuando tenía 2 años. ¡No me insulte probando mi código perfecto!



¿Los programadores reales no hacen prueba?

Aquí están las 5 principales razones por las que un estudiante no quiere probar su código:

- 3) Las pruebas son para programadores incompetentes que no pueden hackear.





¿Los programadores reales no hacen prueba?

Aquí están las 5 principales razones por las que un estudiante no quiere probar su código:

- 2) “Somos estudiantes de la UNAM, nuestro código debe funcionar, si no condena enérgica”

Los programadores reales no hacen prueba?

1) "La mayoría de los métodos en Graph.java, tal como están implementadas, son uno o dos métodos de línea que dependen únicamente de los métodos en HashMap o HashSet.

Asumo que estos métodos funcionan a la perfección y, por lo tanto, no hay necesidad de probarlas."

- Un extracto de un correo electrónico de un estudiante





Cambio de enfoque

- El mayor problema para estos estudiantes es el “**optimismo**”.
- Hay que cambiar esa perspectiva: busca cosas que pueden salir mal, en lugar de asumir que todo irá bien, es decir, sean “**pesimistas**”.



Ejemplo concreto

- Las pruebas no son ciencia de cohetes.
- Excepto cuando lo es: un caso famoso fue el vehículo de lanzamiento Ariane 5, diseñado y construido para la Agencia Espacial Europea en la década de 1990.

Ejemplo concreto

Se autodestruyó 37 segundos después de su primer lanzamiento. El motivo fue un error del software de control que no se detectó. El software de guía de Ariane 5 se reutilizó desde el Ariane 4, que era un cohete más lento. Como resultado, cuando el cálculo de la velocidad se convirtió de un número de punto flotante de 64 bits (un doble en la terminología de Java, aunque este software no estaba escrito en Java) a un entero de 16 bits con signo (un corto), desbordó el pequeño entero e hizo que se lanzara una excepción.



Ejemplo concreto

El manejador de excepciones se había desactivado por razones de eficiencia, por lo que el software de orientación colapsó ... y sin orientación, el cohete también lo hizo. El costo de la falla fue de \$ 1 mil millones ... un costo que podría haberse evitado con mejores pruebas.



Principios básicos

Las pruebas son solo una **parte** de un **proceso** más general **llamado validación**. El propósito de la validación es **descubrir problemas** en un programa y, por lo tanto, aumentar su confianza en la correctes del programa.

La validación incluye:



Principios básicos

- **Razonamiento formal** sobre un programa, generalmente llamado verificación. La verificación construye una prueba formal de que un programa es correcto.



Principios básicos

La verificación es tediosa de realizar a mano, y el **soporte automatizado** de herramientas para la verificación sigue siendo un área activa de investigación. Sin embargo, se pueden verificar formalmente **piezas pequeñas y cruciales** de un programa, el intérprete de códigos de bytes en una máquina virtual, o el sistema de archivos en un sistema operativo.




Principios básicos


- **Razonamiento informal o también llamado Revisión de código.** Hacer que **otra persona lea** cuidadosamente **su código**, y razone de manera informal al respecto, puede ser una buena manera de descubrir errores. Es como tener a alguien más revisando un ensayo que tú escribiste.



Principios básicos

- **Prueba** Ejecutar el programa con entradas cuidadosamente seleccionadas y verificar los resultados.

- 
- Incluso con la mejor validación, es muy difícil lograr una calidad perfecta en el software.
 - Estas son algunas de las **tasas típicas de defectos residuales** (errores que quedan después de que el software sea utilizado) por kloc (**mil líneas** de código fuente):
 - 1 - 10 defectos / kloc: software industrial típico.
 - 0.1 - 1 defectos / kloc: validación de alta calidad. Las bibliotecas de Java pueden alcanzar este nivel de correctes.
 - 0.01 - 0.1 defectos / kloc: la mejor validación de seguridad crítica. La NASA y compañías como Google pueden alcanzar este nivel.



Esto puede ser desalentador para sistemas grandes. Por ejemplo, si ha enviado un millón de líneas del código fuente de la industria típica (1 defecto / kloc), ¡significa que tuvo 1000 errores!

¿Como programadores que queremos?

- saber cuándo el producto es lo suficientemente **estable** como para **lanzar**





¿Como programadores que queremos?

- **entregar el producto con una tasa** de falla conocida (preferiblemente **bajo**)
- **ofrecer garantía**

Por qué las pruebas son difíciles

- **Las pruebas exhaustivas no son factibles.** El espacio de posibles casos de prueba es generalmente demasiado grande para cubrir exhaustivamente. Imagine probar exhaustivamente una operación de multiplicación de flotantes de 32 bits, $a * b$. ¡Hay 2^{64} casos de prueba!



Por qué las pruebas son difíciles

- **Las pruebas de riesgo** ("simplemente inténtalo y ve si funciona") es menos probable que encuentren errores, a menos que el programa tenga tantos errores que una entrada elegida arbitrariamente es más probable que falle en vez de tener éxito. Tampoco aumenta nuestra confianza en la correctes del programa.

Haciendo una calculadora

$$2+2=4$$

$$2*2 =4 \text{ pero si hubiera probado}$$

$$3+3=9$$

$$3*3 =9$$



Por qué las pruebas son difíciles

- **Las pruebas aleatorias o estadística, otras disciplinas** de ingeniería pueden probar pequeñas muestras aleatorias (por ejemplo, el 1% de los discos duros fabricados) e inferir la **tasa de defectos** para todo el lote de producción. Los sistemas físicos pueden usar muchos trucos para acelerar el tiempo, como abrir un refrigerador 1000 veces en 24 horas en lugar de 10 años. Estos trucos dan índices de fallas conocidas (por ejemplo, duración media de un disco duro), pero suponen continuidad o uniformidad en el espacio de defectos. Esto es cierto para los artefactos físicos.

Por qué las pruebas son difíciles

- Las pruebas aleatorias o estadísticas no funcionan bien para el software.
- El **comportamiento del software varía** de forma **discontinua y discreta en el espacio de posibles entradas**. El sistema parece funcionar bien en una **amplia gama de entradas**, y luego **falla abruptamente** en un único punto límite. La famosa falla de division Pentium afectó aproximadamente a 1 en 9 mil millones de divisiones. <http://www.willamette.edu/~mjaneba/pentprob.html>
- Los desbordamientos de pila, los errores de memoria y los errores de desbordamiento numérico tienden a suceder abruptamente, y siempre de la misma manera, no con variación probabilística.

errores de desbordamiento (como Ariane 5) suceden abruptamente

Ponerse la camiseta de prueba

- Las pruebas requieren tener la **actitud correcta**. Cuando está codificando, su **objetivo es hacer que el programa funcione, pero como tester, que realmente desea hacerlo fallar.**



Ponerse la camiseta de prueba

- Esa es una diferencia sutil pero importante. Es demasiado **tentador tratar** el código que acaba de escribir **como algo** precioso, una cáscara de huevo **frágil**, y probarlo muy ligeramente solo para ver si funciona.



Ponerse la camiseta de prueba

- En cambio, debes **ser brutal**. Un buen tester maneja un mazo y **supera al programa donde sea que sea vulnerable, para que esas vulnerabilidades puedan ser eliminadas.**



Pruebas antes de programar

- **Prueba temprano y con frecuencia. No deje las pruebas hasta el final, cuando tenga una gran cantidad de código no validado. Dejar las pruebas hasta el final solo hace que la depuración sea más larga y más dolorosa, porque los errores pueden estar en cualquier parte de tu código. Es mucho más agradable probar tu código a medida que lo desarrollas.**



Pruebas antes de programar

En test-first-programming,
escribes **pruebas antes** incluso de
escribir cualquier código.

Pruebas antes de programar

El desarrollo de una función única procede en este orden:

- **Escribe una especificación para la función o método.**
- **Escriba pruebas que ejerzan la especificación.**
- **Escribe el código real. Una vez que su código aprueba las pruebas que escribió, listo.**

Pruebas antes de programar

- La **especificación** describe el **comportamiento de entrada y salida** de la función. Proporciona los tipos de parámetros y cualquier restricción adicional sobre ellos (por ejemplo sqrt , el parámetro no debe ser negativo). También proporciona el tipo de valor de retorno y cómo el valor de retorno se relaciona con las entradas. **En el código, la especificación consiste en la firma del método y el comentario anterior que describe lo que hace.**

Pruebas antes de programar

- Escribir pruebas primero es una buena forma de entender la especificación. La **especificación también puede tener errores: incorrecta, incompleta, ambigua**, falta de casos en las esquinas. **Intentar escribir pruebas puede descubrir estos problemas temprano, antes** de que haya perdido el tiempo **escribiendo una implementación de una especificación con errores.**



Elegir Casos de Prueba

Queremos **elegir un conjunto de casos de prueba** lo suficientemente **pequeños para ejecutarse rápidamente**, pero lo **suficientemente grandes** como para validar el programa.

Elegir Casos de Prueba

- Para hacer esto, **dividimos el espacio de entrada en subdominios**, cada uno formado por un conjunto de entradas. **Los subdominios cubren por completo el espacio de entrada**, de modo que cada entrada se encuentra en al menos un subdominio.
- Luego **elegimos un caso de prueba de cada subdominio**.

Ejemplo: max()

- Veamos un ejemplo de la biblioteca de Java: max() función entera , que se encuentra en la clase Math

```
/**  
 * @param a   an argument  
 * @param b   another argument  
 * @return the larger of a and b.  
 */  
public static int max(int a, int b)
```



Matemáticamente, este método es una función del siguiente tipo:

$\text{max} : \text{int} \times \text{int} \rightarrow \text{int}$

Desde la especificación, tiene sentido particionar esta función como:

$a < b$

$a = b$

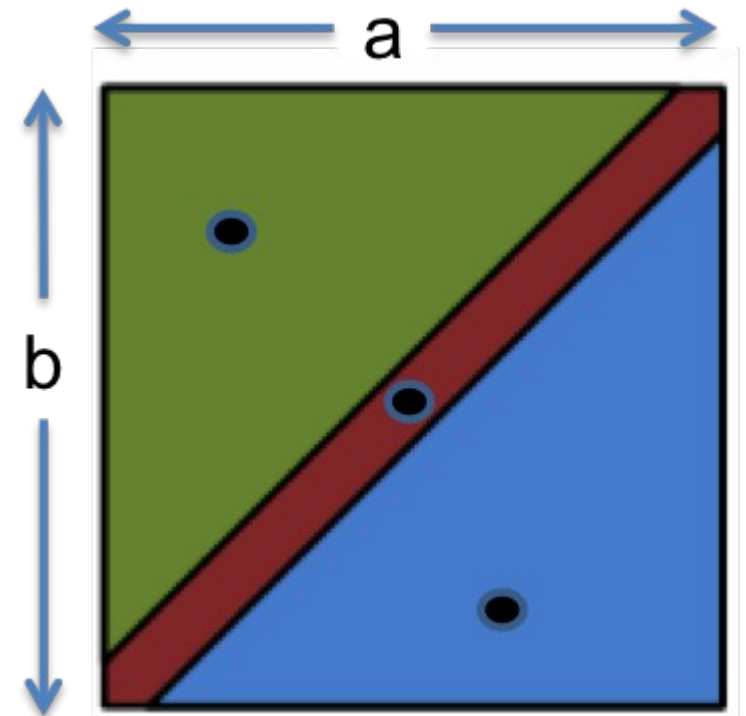
$a > b$

Nuestro conjunto de pruebas podría ser:

$(a, b) = (1, 2)$ para cubrir $a < b$

$(a, b) = (9, 9)$ para cubrir $a = b$

$(a, b) = (-5, -6)$ para cubrir $a > b$



- ¿Con eso cubrimos todos los posibles subdominios?



Incluir límites en la partición

Los errores suelen ocurrir en los límites entre subdominios. Algunos ejemplos:

- 0 es un límite entre números positivos y números negativos
- Los valores máximo y mínimo de los tipos numéricos, como `int` y `double`
- vacío (la cadena vacía, la lista vacía, la matriz vacía) para los tipos de colección
- El primer y último elemento de una colección.

Incluir límites en la partición

¿Por qué los errores ocurren a menudo en los límites? Una razón es que los programadores a menudo cometen errores **off-by-one** (como escribir en \leq lugar de $<$, o inicializar un contador a 0 en lugar de 1).

- ¿Cuales son los limites en una lista?

Incluir límites en la partición

- Algunos límites deben manejarse como casos especiales en el código. Otra es que los límites pueden ser lugares de discontinuidad en el comportamiento del código. Cuando una variable `int` crece más allá de su valor positivo máximo, por ejemplo, de repente se convierte en un número negativo.

Incluir límites en la partición

Es importante incluir límites como subdominios en su partición, de modo que esté eligiendo una entrada del límite.

Vamos a rehacer $\text{max} : \text{int} \times \text{int} \rightarrow \text{int}$



Incluir límites en la partición

Partición en:
relación entre a y b

$$a < b$$

$$a = b$$

$$a > b$$

Incluir límites en la partición

Partición en:
valor de a

$$a = 0$$

$$a < 0$$

$$a > 0$$

$a =$ entero mínimo

$a =$ entero máximo

Incluir límites en la partición

Partición en:
valor de b

$$b = 0$$

$$b < 0$$

$$b > 0$$

$b =$ entero mínimo

$b =$ entero máximo

Incluir límites en la partición

Ahora escojamos valores de prueba que cubran todas estas clases:

- (1, 2) cubre $a < b$, $a > 0$, $b > 0$
- (-1, -3) cubre $a > b$, $a < 0$, $b < 0$
- (0, 0) cubre $a = b$, $a = 0$, $b = 0$
- (Integer.MIN_VALUE, Integer.MAX_VALUE)
cubre $a < b$, $a = \text{minint}$, $b = \text{maxint}$
- (Integer.MAX_VALUE, Integer.MIN_VALUE)
cubre $a > b$, $a = \text{maxint}$, $b = \text{minint}$



Cubra cada parte

Cada parte de cada dimensión está cubierta por al menos un caso de prueba, pero no necesariamente cada combinación. Con este enfoque, el conjunto de pruebas max puede ser tan pequeño como **5 casos de prueba** si se **elige cuidadosamente**. Ese es el enfoque que adoptamos anteriormente, nos permitió elegir 5 casos de prueba, aunque si no se hubieran considerado cuidadosamente podrían haber sido más.



Mas tipos de Pruebas



Blackbox Testing (pruebas de Caja negra)

- Recuerda que **la especificación** es la descripción del comportamiento de la función: los tipos de parámetros, el tipo de valor de retorno y las restricciones y relaciones entre ellos.



Blackbox Testing (pruebas de Caja negra)

La prueba de Blackbox significa **elegir casos de prueba solo a partir de la especificación**, no la implementación de la función. Eso es lo que hicimos con nuestros ejemplo anterior. Particionamos y buscamos límites en el máximo sin mirar el código real para estas funciones.



Whitebox Testing or glass box Testing (pruebas de caja blanca o caja de vidrio)

Elegir **datos de prueba con conocimiento de implementación**

- **si la implementación selecciona diferentes algoritmos dependiendo de la entrada**, debe elegir las entradas que cubran todos los algoritmos

Debe tener cuidado de que las pruebas no dependan de los detalles de implementación

- las buenas pruebas deben ser modulares, dependiendo solo de la especificación, no de la implementación

Documentando su estrategia de prueba

```
/**
 * Reverses the end of a string.
 *
 * For example:
 *   reverseEnd("Hello, world", 5)
 *   returns "Hellodlrow ,"
 *
 * With start == 0, reverses the entire te
xt.
 * With start == text.length(), reverses n
othing.
 *
 * @param text      non-null String that wil
l have
 *
 *                  its end reversed
 * @param start     the index at which the
 *                  remainder of the input i
s
 *
 *                  reversed, requires 0 <=
 *                  start <= text.length()
 * @return input text with the substring f
rom
 *
 *                  start to the end of the s
tring
 *
 *                  reversed
 */
static String reverseEnd(String text, int
start)
```

Documentar la estrategia en la parte superior de la clase de prueba:

```
/*
 * Testing strategy
 *
 * Partition the inputs as follows:
 * text.length(): 0, 1, > 1
 * start:         0, 1, 1 < start < text.length(),
 *                  text.length() - 1, text.length()
 * text.length()-start: 0, 1, even > 1, odd > 1
 *
 * Include even- and odd-length reversals because
 * only odd has a middle element that doesn't move.
 *
 * Exhaustive Cartesian coverage of partitions.
 */
```

Documente cómo se eligió cada caso de prueba, incluidas las pruebas de caja blanca:

```
// covers test.length() = 0,
//      start = 0 = text.length(),
//      text.length()-start = 0
@Test public void testEmpty() {
    assertEquals("", reverseEnd("", 0));
}

// ... other test cases ...
```



Documentando su estrategia de prueba

- Para la función de ejemplo de la anterior diapositiva es cómo podemos documentar la estrategia de prueba en la que trabajamos en el ejercicio de partición anterior. La estrategia también aborda algunos valores límite que no consideramos antes.

Coverage (Cobertura)

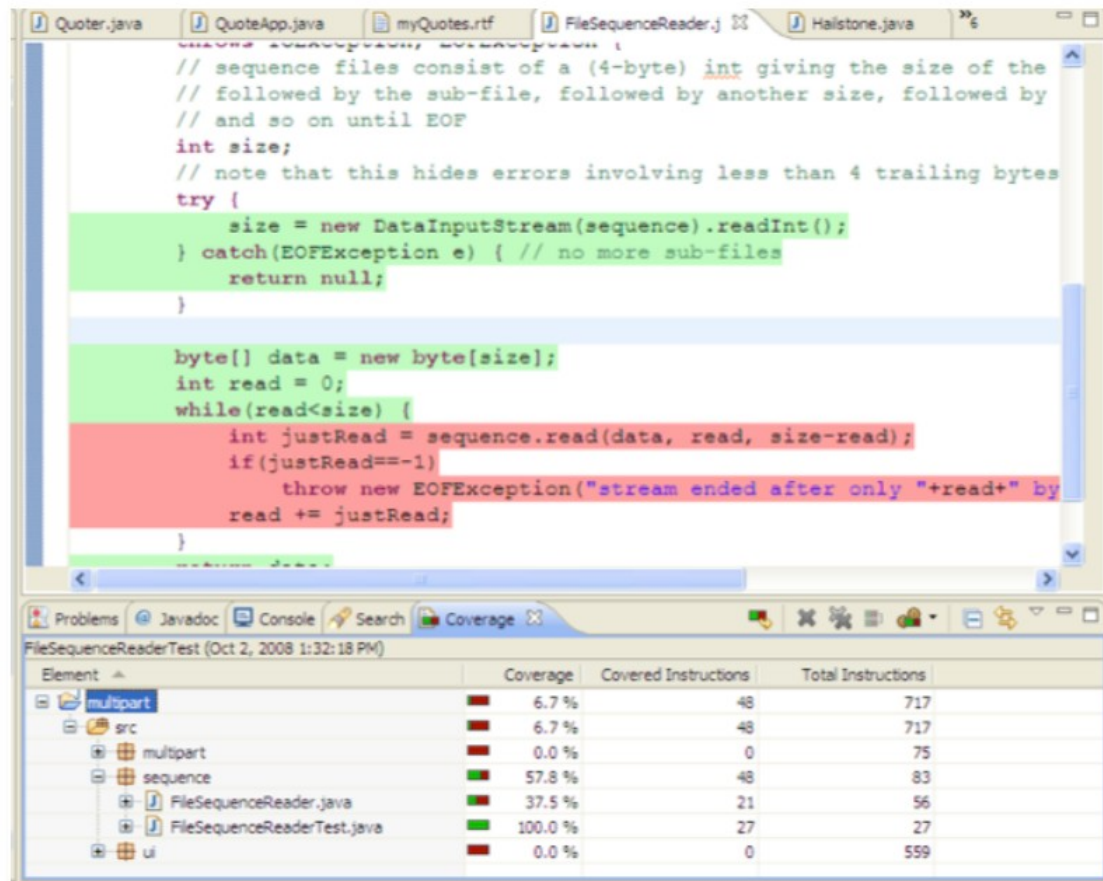
Es una técnica para asegurarse de que sus pruebas realmente estén probando su código o la cantidad de código que cubre al ejecutar la prueba.

- **Cobertura de estado** : cada instrucción accesible en el programa sea ejecutada por al menos un caso de prueba.
- **Cobertura de ramas** : por cada if o while declaración en el programa, son a la vez la verdadera y falsa la dirección tomada por algunos casos de prueba?
- **Cobertura de ruta** : ¿se toman todas las combinaciones posibles de ramas, todas las rutas a través del programa, en algún caso de prueba?

Herramientas de Cobertura

- Una buena herramienta de cobertura de código para Eclipse es Eclemma.

<https://www.eclemma.org/>



The screenshot shows the Eclipse IDE with a Java file named `FileSequenceReader.java` open. The code is a Java class that reads a sequence of files. It includes comments and a `try` block for reading a 4-byte integer size, followed by a `while` loop that reads data from the sequence. The code is color-coded with syntax highlighting.

Below the code editor, the **Coverage** tab is active, displaying a table of coverage data for the `FileSequenceReaderTest` (Oct 2, 2008 1:32:18 PM). The table has four columns: **Element**, **Coverage**, **Covered Instructions**, and **Total Instructions**.

Element	Coverage	Covered Instructions	Total Instructions
multipart	6.7 %	48	717
src	6.7 %	48	717
multipart	0.0 %	0	75
sequence	57.8 %	48	83
FileSequenceReader.java	37.5 %	21	56
FileSequenceReaderTest.java	100.0 %	27	27
ui	0.0 %	0	559



Herramientas de Cobertura

Las líneas que han sido ejecutadas por el conjunto de pruebas son de color verde y las que aún no están cubiertas son rojas. Si ves este resultado en la herramienta de cobertura, tú próximo paso sería crear un caso de prueba que haga que se ejecute el bucle while, y agregarlo a su conjunto de pruebas para que las líneas rojas se vuelvan verdes.



Unit Testing (Pruebas unitarias)

Un programa bien probado tendrá **pruebas** para **cada módulo individual** (donde un módulo es un **método o una clase**) que contiene. Una prueba que prueba un módulo individual, de **manera aislada** si es posible, se denomina **prueba unitaria**.

Unit Testing (Pruebas unitarias)

La prueba de módulos en aislamiento conduce a una depuración mucho más fácil. Cuando falla una prueba unitaria de un módulo, puede estar más seguro de que el error se encuentra en ese módulo, en lugar de en cualquier parte del programa.



Integration test (Pruebas de integración)

- Lo contrario de una prueba unitaria es una prueba de integración, que prueba una **combinación de módulos**, o incluso **todo el programa**. Si todo lo que tienes son pruebas de integración, cuando falla una prueba, tienes que buscar el error. Puede ser en cualquier parte del programa.

"Cada quien haga su parte y mañana lo juntamos todo"





Integration test (Pruebas de integración)

Las pruebas de integración siguen siendo importantes, porque un programa puede **fallar en las conexiones** entre módulos. Por ejemplo, un módulo puede esperar entradas diferentes de las que realmente recibe de otro módulo. Pero si tiene un conjunto completo de pruebas de unidades que le dan confianza en la exactitud de los módulos individuales, entonces tendrá que buscar mucho menos para encontrar el error.



Automated Testing and Regression Testing (Pruebas automatizadas y pruebas de regresión)

Nada hace que las pruebas sean más fáciles de ejecutar, y más probable que se ejecuten, que la automatización completa.

Las pruebas automatizadas significan ejecutar las pruebas y verificar sus resultados automáticamente.

Automated Testing and Regression Testing (Pruebas automatizadas y pruebas de regresión)

Un controlador de prueba no debe ser un programa interactivo que le solicite entradas e imprima resultados para que lo compruebe manualmente. En cambio, un controlador de **prueba debe invocar el módulo en casos de prueba fijos y verificar automáticamente que los resultados sean correctos.** El resultado del controlador de prueba debe ser "todas las pruebas son correctas" o "estas pruebas fallaron: ..." Un buen marco de prueba, como **JUnit**, lo ayuda a crear conjuntos de pruebas automatizadas.



Automated Testing and Regression Testing (Pruebas automatizadas y pruebas de regresión)

Una vez que tenga automatización de prueba, **es muy importante volver a ejecutar sus pruebas cuando modifique su código.**

Esto evita que su programa regrese - **introduciendo otros errores cuando arregla nuevos errores o agrega nuevas características.** Ejecutar todas sus pruebas después de cada cambio se llama **prueba de regresión.**

Automated Testing and Regression Testing (Pruebas automatizadas y pruebas de regresión)

¿Como se realizan las pruebas? ¿De donde las saco?

Cada vez que encuentre y solucione un error, tome la información que provocó **el error y agréguelo a su suite de pruebas automatizada** como caso de prueba. Este tipo de caso de prueba se llama **prueba de regresión**. Esto ayuda a poblar su suite de pruebas con **buenos casos de prueba**. Recuerde que una prueba es buena si provoca un error, ¡y **cada prueba de regresión lo hizo en una versión de su código!** El ahorro de las pruebas de regresión también protege contra las reversiones que reintroducen el error. El error puede ser un error fácil de realizar, ya que sucedió una vez.



Automated Testing and Regression Testing (Pruebas automatizadas y pruebas de regresión)

Las pruebas de regresión solo son prácticas si las pruebas se pueden ejecutar a menudo, automáticamente. A la inversa, si ya tiene pruebas automatizadas para su proyecto, entonces también puede usarlo para evitar regresiones. Por lo tanto, las **pruebas de regresión automatizadas** son una de las mejores prácticas de la ingeniería de software moderna.



Resumen

- Test-first programming. Escribe pruebas antes de escribir código.
- Partición y límites para la elección sistemática de casos de prueba.
- Pruebas de caja negra nos ayudan a ver malas especificaciones.



Resumen

- Prueba de caja blanca y cobertura de estado de cuenta para completar un conjunto de pruebas.
- Realización de pruebas unitarias de cada módulo, en el mayor aislamiento posible.
- Pruebas de regresión automatizadas para evitar que los errores vuelvan.



Resumen

Y esto ¿como para que me servirá?

- **A salvo de bugs.** La prueba consiste en encontrar errores en su código, y test-first programming consiste en encontrarlos lo antes posible, inmediatamente después de que los haya introducido.

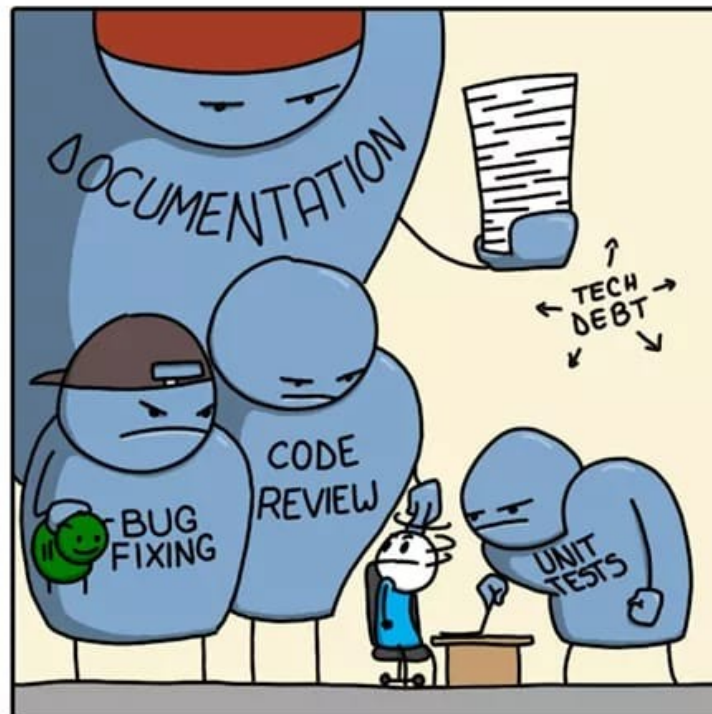
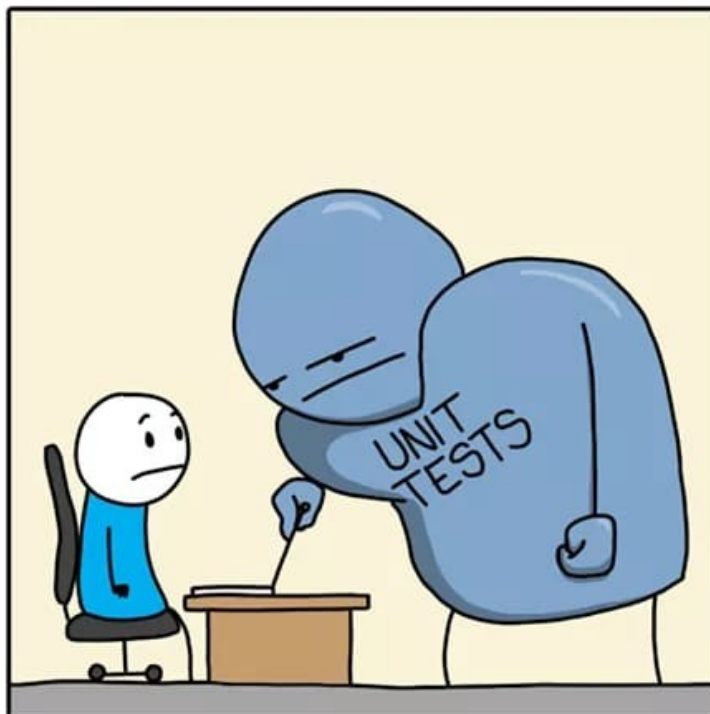
```
// Querido colega programador:  
//  
// Cuando escribí este código, sólo Dios y yo  
// sabíamos cómo funcionaba.  
// Ahora, ¡sólo Dios lo sabe!  
//  
// Así que si está tratando de 'optimizarlo'  
// y fracasa (seguramente), por favor,  
// incremente el contador a continuación  
// como una advertencia para su siguiente colega:  
//  
// total_horas_perdidas_aquí = 189
```



Resumen

Y esto ¿como para que me servirá?

- **Fácil de comprender.**
- **Listo para el cambio.** La preparación para el cambio se consideró al escribir pruebas que solo dependen del comportamiento de la especificación.





¿Son todos los tipos de pruebas que existen?

- No, hay todavía más, pero estas se aplican en un entorno lo más parecido al de producción.



Pruebas de rendimiento

- Pruebas de carga
- Prueba de estrés
- Prueba de estabilidad (soak testing)
- Pruebas de picos (spike testing)



Pruebas de carga

- Una prueba de carga se realiza generalmente para **observar el comportamiento de una aplicación bajo una cantidad de peticiones esperada**. Esta carga puede ser el número esperado de usuarios concurrentes utilizando la aplicación y que realizan un número específico de transacciones durante el tiempo que dura la carga.



Prueba de estrés

Se va doblando el número de usuarios que se agregan a la aplicación y se ejecuta una prueba de carga hasta que se rompe. Este tipo de prueba se realiza para determinar la solidez de la aplicación en los momentos de carga extrema y ayuda a los administradores para determinar si la aplicación rendirá lo suficiente en caso de que la carga real supere a la carga esperada.

Prueba de estabilidad (soak testing)

Esta prueba normalmente se hace para determinar si la aplicación puede aguantar una carga esperada continuada.

Implican probar un sistema con una carga de producción típica, durante un período de disponibilidad continuo, para validar el comportamiento del sistema bajo el uso de producción.

Generalmente esta prueba se realiza para determinar si hay alguna fuga de memoria en la aplicación.



Pruebas de picos (spike testing)

- La prueba de picos, como el nombre sugiere, trata de observar el comportamiento del sistema variando el número de usuarios, tanto cuando bajan, como cuando tiene cambios drásticos en su carga. Esta prueba se recomienda que sea realizada con un software automatizado que permita realizar cambios en el número de usuarios mientras que los administradores llevan un registro de los valores a ser monitorizados.

¿Tienen alguna pregunta?

