Cátedra "Ingeniería de software" 4to Año – 2016
Prof. Adjunta: Ing. Mónica Colombo
Prof. JTP: Lic.Graciela M. Lastra
Ayudantes: Ing. Alejandro Zerbetto
Ing. Matías Martínez

Trabajo Práctico

"Test unitario"

Objetivo:

- Conocer conceptos relacionados a las pruebas unitarias.
- Comprender el uso de las pruebas unitarias.
- Tomar conciencia del proceso destinado a asegurar la validez del producto, diseñando un software partiendo de pruebas unitarias.
- Investigar y comparar herramientas utilizadas para codificación de pruebas unitarias.

Modalidad de trabajo

- Se debe trabajar sobre un repositorio en github, para lo cual cada integrante deberá crearse una cuenta y trabajar desde la misma. De esta forma quedará un registro completo de las tareas realizadas por cada uno.
- Se debe presentar un informe escrito (puede ser formato PDF enviado por email) con los puntos 3, 4, 5, 7 y 8.

Tareas a Realizar:

1. Leer el material introductorio.
2. Leer el enunciado propuesto.
3. Escribir 5 pruebas unitarias en lenguaje coloquial por cada integrante del equipo.
4. Investigar y seleccionar una herramienta para escribir las pruebas unitarias, justificar la elección de la misma.
5. Cada integrante debe seleccionar 2 pruebas unitarias (no necesariamente de las escritas por él) y haciendo uso de la herramienta seleccionada debe codificar la prueba unitaria.
6. Haciendo uso de las pruebas unitarias escritas se debe codificar el programa.
7. Ejecutar los test y mostrar los resultados de los mismos.
8. Elaborar una conclusión sobre la práctica realizada.

MATERIAL INTRODUCTORIO

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.

¿Qué es una prueba unitaria?

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

Por eso es que las pruebas unitarias deben acompañar al desarrollo del software y no esperar a que esté listo para incorporar dichas pruebas con fines de detectar errores.

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.

Estructura de una prueba unitaria

Arrange (Preparar): Es la parte del unit test en donde se configura todo el código para ejecutar la prueba unitaria.

Act (Actuar): Esta es la fase del unit test en donde se ejecuta el código a probar.

Assert (Afirmar): Es la sección de la prueba unitaria en donde se prueba el resultado del mismo.

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.
- **Mejoran 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.

Consideraciones

Como en la adopción de cualquier otra disciplina, la incorporación 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.

Tips para escribir buenas pruebas unitarias

- **Cada test debe ser independiente al resto**. Cualquier comportamiento dado debe corresponderse con un único test. Esto es así para que, en caso de modificar una funcionalidad, solo un test se vea afectado. Esta regla posee algunos matices:
    - **No debemos realizar afirmaciones innecesarias**. No es productivo realizar afirmaciones para algo que ya ha sido comprobado en otro test: con esto no ampliamos la cobertura de nuestras pruebas, sino que únicamente añadimos la frecuencia con la que nos será advertido un determinado error. En esta línea, existe una máxima (algo radical) que defiende una única afirmación lógica por cada test unitario.
    - **Comprueba solo una unidad de código cada vez**. Los tests, una vez más, deben ser independientes: deben centrarse en una única unidad de código. Si éstos se solapan, un cambio puede resultar en la necesidad de actualizar toda una batería en cascada.
    - **Solo si la arquitectura lo exige, recurriremos a los mocks**. Cuando necesariamente el resultado de un test depende de la funcionalidad de otra unidad de código y esto es inalterable, usaremos un mock u objeto dummy. Este punto es delicado y muchos puristas defenderán que si una suite precisa de mocks es porque no está bien planteada. Este punto asume que no existe alternativa y, por tanto, el uso de estos objetos de sustitución está justificado.
    - **Evitar o reducir al máximo las condiciones previas a la ejecución de una suite**. En la misma línea que los anteriores puntos, una batería de tests que precisa de pre-configurar un escenario con muchos parámetros dificulta la claridad del código. Además, alteran la naturaleza de los tests ya que, finalmente, no estamos trabajando sobre unidades de código sino sobre sistemas dependientes.
- **Nombra los tests de un modo claro, consistente y unívoco**. Puede resultar obvio, pero en la práctica es frecuente perder la consistencia en cómo nombramos los tests. Debemos tener en cuenta que son la documentación más fiable de la que disponemos: como desarrolladores, no debemos fiarnos de especificaciones escritas, diagramas o wireframes; la única verdad sobre el funcionamiento de un software es la que podemos rastrear a través de su propio código. Los tests aportan una capa de información (además de una estructura lógica) valiosísima a dicha documentación. Un test unitario no debería presentar comentarios: su propio nombre debería ser lo suficientemente explícito como para que no quepan dudas sobre su área de influencia.

ENUNCIADO PROPUESTO

Quiero lanzar al mercado un software educativo para enseñar matemáticas a niños. Necesito que puedan jugar o practicar a través de una página web.

El juego servirá para que los niños practiquen diferentes temas dentro de las matemáticas y el sistema debe recordar a cada niño, que tendrá un nombre de usuario y una clave de acceso. El sistema registrará todos los ejercicios que han sido completados y la puntuación obtenida para permitirles subir de nivel si progresan.

Existirá un usuario tutor que se registra a la vez que el niño y que tiene la posibilidad de acceder al sistema y ver estadísticas de juego del niño. El tema más importante ahora mismo es la aritmética básica con números enteros.

Es el primero que necesito tener listo para ofrecer a los profesores de enseñanza primaria un refuerzo para sus alumnos en el próximo comienzo de curso. El módulo de aritmética base incluye las cuatro operaciones básicas (sumar, restar, multiplicar y dividir) con números enteros.

Los alumnos no solo tendrán que resolver los cálculos más elementales sino también resolver expresiones con paréntesis y/o con varias operaciones encadenadas. Así aprenderán la precedencia de los operadores y el trabajo con paréntesis: las propiedades distributiva, asociativa y conmutativa.

Los ejercicios estarán creados por el usuario profesor que introducirá las expresiones matemáticas en el sistema para que su resultado sea calculado automáticamente y almacenado. El profesor decide en qué nivel va cada expresión matemática. En otros ejercicios se le pedirá al niño que se invente las expresiones matemáticas y les ponga un resultado.

El programa dispondrá de una calculadora que sólo será accesible para los profesores y los jugadores de niveles avanzados. La calculadora evaluará y resolverá las mismas expresiones del sistema de juego. Cuando el jugador consigue un cierto número de puntos puede pasar de nivel, en cuyo caso un email es enviado al tutor para que sea informado de los logros del tutelado. El número mínimo de puntos para pasar de nivel debe ser configurable.