Resumen algoritmos y programación 3.

Preguntas de parcial 18/5/17.

Unit testing guideliness dice que hay que reproducir fallas para reproducir bugs. Para que? Por que es importante?

Se crea un test donde ese error quede evidenciado y a partir de eso se comienza a trabajar para solucionarlo y que el test pase.

Es importante ya que nos permite ver con claridad que es lo que estamos tratando de solucionar y no estar trabajando 'a ciegas ' y podemos reproducir ese escenario las veces que sea necesarias. Además que estaríamos sumando una prueba en la que estamos documentando un caso importante.

Explique que es una refactorización y que se busca en la práctica del refactoring.

Es un cambio en nuestro código que nos permite mantener la calidad de nuestro código sin que ocurra un cambio en su funcionalidad, se mantienen las pruebas como reaseguro.

(Definición posta: es una técnica que buscar mejorar la calidad del código con vistas a facilitar su mantenimiento, mejorar su legibilidad y facilidad de comprensión pero sin alterar el comportamiento observable del mismo).

- Si tengo código repetido en una misma clase, se podría extraer ese código repetido en un método privado dentro de esa clase.
- Entre dos clases hermanas, "Subiria ese código" a una clase padre y que lo hereden estas clases hermanas.(Podria ser protegido si no lo invocamos desde afuera)
- Entre dos clases no vinculadas: habriamos podido extraer el código en un método y colocarlo en una clase generada especialmente, a la que las clases anteriores accedan por delegación.

Se dice que es una refactorización, cuando al volver a ejecutarse las pruebas de regresión, estas pasan y el comportamiento continua siendo el mismo.

Parcial 9/10/17.

En el marco de TDD, apenas escrita una prueba, hay que chequear que la misma falle. Por que le parece importante esta recomendación?.

Estaba debe fallar, ya que primero se debe crear la prueba y luego implementar el código para que la prueba pase.

Primero estamos chequeando que se ha cargado correctamente el framework de pruebas con el que vamos a trabajar

Una vez que el código falla podemos comenzar a implementar lo que estamos probando, sin ningún tipo de condicionamiento previo. De manera incremental vamos a ir implementando la

solución hasta que la prueba no falle. De esta manera vamos escribiendo de a poco código a la vez y ejecutando la prueba. Evitamos código no necesario y lo mantenemos simple y funcional.

Esto nos da el beneficio que las pruebas se convierten en especificaciones de lo que se espera como respuesta del objeto. Al ir definiendo pruebas a cada vez tenemos desarrollo incremental muy genuino.

Resumen de lecturas.

Whats a model for? Martin Fowler.

El valor del modelado esta enteramente ligado a su impacto en el software que estas produciendo, no es algo que sea fundamental para la importancia del cliente.

Si el modelado mejora la calidad o reduce el costo entonces el modelo tiene valor. Pero el modelo no tiene valor por si mismo.

Puede brindar un mayor entendimiento del software o del dominio que el software respalda.

Recordar que las necesidades individuales difieren y que necesitas encontrar cual es la combinación mas efectiva para la gente con la que estas trabajando.

Al tener un modelo bien detallado debes preguntarte cuanto estas ganando. Mucho detalle pierde claridad en el flujo de control.

Es difícil entender un sistema complejo al sumergirse en todos sus detalles.

La cuestión están primero poder entender las estructuras claves que hacen que el software funcione y luego se podrá investigar los detalles.

El modelo esquelético es mejor que un modelo de cuerpo entero. Primero uno debe darse cuenta por donde empezar y luego hondar en los detalles.

Además es mas fácil de mantener que no de cuerpo entero, ya que los detalles importantes cambian con menor frecuencia. Si la gente quiere saber mas detalles puede explorar el código una vez que el esqueleto les haya provisto una visión general.

El principal objetivo de trabajar con objetos es poder ver las interfaces, no la estructura de datos interna.

GetterEradicator – Martin Fowler.

Se recomienda no escribir getters si no una vez que realmente los vamos a necesitar.

El punto principal del encapsulamiento no es acerca de esconder la información si no de esconder las decisiones de diseño. Especialmente en áreas donde ese diseño puede que cambie.

Cuando pensamos en encapsulamiento es mejor preguntarnos que parte de variabilidad (¿?) estoy escondiendo y porque en vez de que estoy exponiendo información.

Es común ver procedimientos obteniendo información de un objeto y utilizando para hacer algo. Cuando es el objeto quien debería realizar esa acción. Es una violación del principio “Tell, dont ask”.

Entonces al decir a la gente que no utilic e getters ayuda a alejarlos de esta situación.

Otra señal de peligro son las clases contenedoras de datos con getters y setters, sin comportamiento alguno. En esta situación hay que sospechar y verificar quien usa estos datos y fijarse si este comportamiento puede ser movido al objeto.

Un regla a tener en cuenta es que las cosas que cambian juntas deben estar juntas. Datos y comportamiento que se usan juntos, cambian juntos.

The Art of Enbugging – Andy Hunt and Dave Thomas

Nosotros mismos provocamos errores en nuestro código.

Una de las mejoes maneras de mantener los errores lejos es. Separar las ‘preocupaciones’. Diseñar el código en clases y modulos bien definidos, cada uno con sus propias responsabilidades y una semántica bien auto documentada.

Nuestro objetivo principal es escribir ‘Shy code’. Un código que no revela mucho de si mismo a nadie y no habla con los demás mas que lo necesario.

Shy code nunca muestra lo privado a sus amigos.

“Tell, dont ask”

La idea de código OO es pedirle un comando a una entidad para que realice cierta acción. Por ejemplo en SmallTalk se invocan métodos vistos como mensajes entre objetos y no llamadas a funciones.

POO le dice a los objetos que hagan cosas.

En el modelo que queremos seguir, nosotros enviamos comandos a objetos diciendoles que queremos que hagan. Definitivamente no queremos consultar a un objeto acerca de su estado y en base a eso, tomar una decisión y luego decirle al objeto que hacer. No es algo que shy code haría.

Estando en el rol de ‘caller’ no debemos tomar decisiones en base al estado de un objeto y luego cambiar el estado del mismo. Eso es la responsabilidad del objeto y no la nuestra. Si somos nosotros quienes tomamos decisiones por afuera de ese objeto estamos violando la encapsulación del mismo, y eso traerá bugs.

Una consulta nos da información del estado de un objeto y no modifica el mismo externamente.

Comandos y consultas separados mantienen nuestro código ‘shy’.

Cada método debe ser un comando que realiza una acción o una consulta que devuelve datos al caller. Pero no ambos.

Al caller no le importa como estos comandos están implementados. Solo sabe de su exitencia y su post condición. Esto nos permite que si ocurre un cambio interno en la implementación el caller no tiene porque saberlo ni preocuparse.

Una buena idea sugiere que un objeto solo debe llamarse a :

-Si mismo

- Algún parámetro que fue pasado en el método

- Cualquier objeto que el creo.

- objetos componentes del mismo.

8 Principles of better unit testing – Dror Helper.

Las pruebas unitarias son cortas, rápidas y automatizadas. Se prueba que una parte especifica del programa funcione. Prueban funcionalidades especificas de un método o una clase. Tiene una condición clara, pasa o no pasa. Con estos unit test los programadores pueden saber si su código funciona o no previamente a ser enviado a QA(calidad).

El test solo va a fallar cuando se introduce un nuevo bug al sistema.

Cuando un test falla es fácil de entender el por que.

<u>1 Saber que estamos testeando.</u>

Si no sabemos bien que es lo que queremos testear terminamos haciendo un test largo y seguramente este probando varias cosas

No hay que caer en la tentación que meter varios test en un solo método ya que este seria muy frágil y complejo.

Probar una sola cosa nos da por resultado un test mas leíble y cuando este falle es mas fácil saber que es lo que lo causo y solucionarlo.

<u>2 Unit test deben ser ‘self sufficient’</u>

Un buen unit test debe ser aislado. Evitar dependencias de configuraciones del enterno, valores o databases.

No se debe depender de que se ejecuten otros test antes que el mismo y o del orden en que los test fueron ejecutados.

3 Deben ser determinísticos (Misma entrada produce la misma salida)

Los tests deben pasar siempre salvo en caso que haya un error y este test pase.

No se debe depender de la maquina en donde se este probando, de la velocidad de la conexión a internet.

Una practica que se debe evitar es utilizar random inputs ya que 'randomized data' ya que en caso de que en un caso particular el test falle no podríamos repetirlo, ya que los valores cambian en cada ejecución.

4 Convención de nombres

Cuando un test con un nombre descriptivo falla es mucho mas fácil de entender que fue testeado y por que esto fallo.

5 Do repeat yourself

En producción se debe evitar el código duplicado porque causa problemas de mantenibilidad. En las pruebas unitarias es aceptable esta duplicación ya que esto nos facilita la lectura.

Es preferible tener 4-5 test con código similar que no entender un test que falle utilizando algún método para no tener código repetido.

6 Probar resultados no implementaciones

Para tener pruebas exitosas se deben probar casos en las que se falle si ocurrió un error o si el requerimiento/comportamiento cambio. Los tests no deben fallar si ocurrió algún cambio interno ya que esto no habría por que probarlo. (De por si no deberíamos tener acceso a la implementación interna)

Las implementaciones tienen a cambiar, asi que hay que testear el comportamiento y no como este fue implementado.

En el caso de los métodos privados solo testearlos si tenemos alguna buena razón para hacerlo. Caso contrario evitarlo. (Podria romperse en una refactorización).

7 Evitar overespecification

No armar un escenario perfecto para nuestros tests.

Por ejemplo. Evitar que en el test un cierto método deba ser llamado 3 veces para que el test pase. Esto trae consigo mucha fragilidad.

8- Usar un framework aislado.

Hay casos en que los tests dependen de dependencias externas. Esto nos complica a la hora de crear los mismos.

Nos conviene usar un framework que nos evite crear objetos falsos a mano si no utilizando unas pocas llamadas a una api

Unit testing Guidelines

- Mantener los test pequeños y rapidos :
  Esto nos permite podes correr los mimos antes de cada code check in.
- Deben ser automatizados y no interactivos.
- Deben ser simples de correr ya sea por comando o click
- Realizar análisis de cobertura. Investigar que partes del código están siendo recorridas y cuales no.
- Apenas ocurre un error repararlo. Dejar lo que se este haciendo hasta solucionarlo.
- Mantener las pruebas a nivel unitario. UT es acerca de testear clases y comportamientos de manera aislada. Evitar la tentación de probar un work-flow completo.
- Empezar con tests simples. Nos permite saber si el framework esta bien cargado y el build environment es correcto.
- Mantener los test independientes, que su ejecución no dependa de que otros tests hayan sido ejecutados antes.
- Mantener los test cercanos a la clase que se quiere testear(Directorio y package).
- Nombras los tests de manera correcta y descriptiva.
- Se debe probar la public API, no hay porque testear métodos privados. SI hay métodos privados que necesitan ser testeados, considerar refactorizarlos en métodos públicos de una clase de utilidades.
- Pensar de manera ‘black box’
- Pensar ‘white box’ -> Preferimos usar debugging.
- Probar también casos triviales.
- Saber distinguir entre execution coverage y actual test coverage. Buscar casos limites a pesar de que ese código ya haya sido recorrido en algún test.
- Probar con casos borde.
- Cuando los casos borde fueron probados se pueden generar algunos parámetros random. (Se contradice con 8 Principles of Better Unit Testing)
- Probar cada feature de a una vez, evitar juntarlas en una sola prueba.
- Usar asserts explicitos. Usar assertEquals instead assertTrue por ejemplo.
- Probar casos negativos como por ejemplo cuando se debe lanzar una excepción.
- Diseñar código con los test en mente.

- No conectarse a fuentes externas predefinidas. Los tests se tienen que escribir independientemente del contexto en donde van a ser ejecutadas. Se deben poder correr en cualquier momento y en cualquier lugar.
- Conocer el costo de realizar pruebas. Hacerlas cuesta, pero no hacerlas también.
- Priorizar las pruebas unitarias ya que son la base de la pirámide del testing.
- Preparar el test para fallas de código sin que se tenga que romper el test suite.
- Escribir un test para reproducir un bug(Desarrollado en la primer pregunta de parcial de este apunte).
- Saber las limitaciones. Que un test falle indica que el código tiene errores. Pero que uno pase no prueba nada. La aplicación mas poderesa de UT es verificar y docuementar los requerimientos a bajo nivel y pruebas de regresión para corroborar que el invariante no cambio en una refactorización.