UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

REESTRUCTURACIÓN Y REFACTORIZACIÓN DE UNIT TESTS CON TESTSURGEON

MEMORIA PARA OPTAR AL TÍTULO DE TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

PABLO IGNACIO ESTEFO CARRASCO

PROFESOR GUÍA:
ALEXANDRE BERGEL

MIEMBROS DE LA COMISIÓN:
ROMAIN ROBBES
JOSÉ PINO URTUBIA

SANTIAGO DE CHILE
JULIO 2013

Shegó el resumeeeen

Por mi y por todos mis compañeros.

Agradecimientos

Vale cabros :D

Índice general

1. Introducción **1**

2. Especificación del Problema **2**
2.1. Merco Teórico 2
2.1.1. Profiling 2
2.1.2. SUnit framework 2
2.2. Contexto: Test como "conductores" del diseño 2
2.2.1. Problema: ¿Cómo mantener los tests?. 3

3. Trabajo relacionado **5**
3.1. Cobertura de Código 5
3.2. Comparación de Tests 5

4. Descripción de la Solución **6**
4.1. Solución 6
4.2. Implementación 6
4.2.1. Profiling 6
4.2.1.1. La clase `TSProfiler` 7
4.2.1.2. Las clases `TSPackage` y `TSClass` 8
4.2.1.3. Las clase `TSMethod` 8
4.2.1.4. Las clase `TSTestMethod` 9
4.2.2. Visualización 9
4.2.3. Clustering 9
4.2.4. Browser 9

5. Caso de estudio: Roassal **10**

6. Conclusión y Trabajo Futuro **11**
6.1. Conclusión 11
6.2. Trabajo Futuro 11

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

En el desarrollo de un software uno de los artefactos importantes son sus pruebas o *tests*. Estos corresponden a

Durante el desarrollo de un software uno de las actividades principales es la prueba de requerimientos. Existen variadas técnicas, metodologías y artefactos relacionados a esta actividad. En particular, la creación de pruebas automatizadas es una práctica cotidiana en cualquier proyecto de software e inclusive es actualmente considerado parte de los artefactos generados durante éste.

En particular las metodologías ágiles dan una importancia fundamental a los tests de tal manera que está prohibido agregar una nueva funcionalidad sin, previamente, haber escrito un test que lo valide[**??**]. Esta queda documentado en el libro de Robert Cecil Martin (más conocido como"Uncle Bob")[**??**] uno de los escritores del manifiesto ágil[**??**] quien declara:

> *"The iteration between writing test cases and code is very rapid [...]. As a result, a very complete body of test cases grows along with the code."*

Capítulo 2

Especificación del Problema

2.1. Merco Teórico

2.1.1. Profiling

La técnica de *profiling* consiste en medir ciertos aspectos de interés en la ejecución del software, como por ejemplo: Tiempo de ejecución de un método, número de llamados de un método, cantidad de memoria utilizada por las instancias de una clase, número de distintas instancias de una clase, entre otros.

El análisis apoyado por profilers se le denomina *análisis dinámico* ya que los datos son obtenidos en tiempo de ejecución. Es decir, se analiza lo que realmente sucede durante la ejecución del software, lo que otorga información valiosa para prever el comportamiento del sistema en producción.

Posteriormente el usuario de los datos puede generar reportes, gráficos y/o visualizaciones que presentan un diagnóstico veraz del sistema analizado.

Los productos de Object Profile entregan reportes visuales usando Roassal. Tanto Kai como Hapao entregan visualizaciones interactivas (ver Figuras **??** y **??**) que facilitan la comprensión del sistema y la detección de anomalías para una rápida intervención en el código.

2.1.2. SUnit framework

2.2. Contexto: Test como "conductores" del diseño

Durante el desarrollo de un software uno de las actividades principales es el testeo de los requerimientos. Existen variadas técnicas, metodologías y artefactos relacionados a esta actividad. En particular, la creación de pruebas automatizadas es una práctica cotidiana en

cualquier proyecto de software y comprende parte importante de los artefactos generados.

En particular las metodologías ágiles dan una importancia fundamental a estos artefactos ya que cada historia de usuario debe tener pruebas de aceptación (tests de alto nivel) que se escriben y detallan con el cliente para crear un modelo mental en común, evitando documentación extremadamente detallada de requerimientos que muy probablente cambiarán y evitar ambiguedades que pudieran provocar historias de usuario poco precisas.

Prueba de aceptación –¿TDD

y que eventualmente sobre el software a crear con a los tests de tal manera que está prohibido agregar una nueva funcionalidad sin, previamente, haber escrito un test que lo valide[**??**]. Esta queda documentado en el libro de Robert Cecil Martin (más conocido como"Uncle Bob")[**??**] uno de los escritores del manifiesto ágil[**??**] quien declara:

> *"The iteration between writing test cases and code is very rapid [...]. As a result, a very complete body of test cases grows along with the code."*

Uno de los ejemplos más conocidos de implementación de estos principios es la metodología de Desarrollo Dirigido por Pruebas o *Test-Driven Development*[**??**] el cual propone que la implementación de una funcionalidad del software debiera seguir los siguientes pasos:

1. Escribir un test que valide el funcionamiento esperado de la nueva característica
2. Escribir código base mínimo que haga pasar dicho test
3. Refactorizar el código

En la práctica, aplicando TDD se obtiene una gran cantidad de pruebas unitarias (unit tests) que representan los casos de prueba de la aplicación (Test Cases). Cada Unit Test contiene varios métodos de tests (test methods) que cubren los distintos aspectos a verificar en la funcionalidad que se está testeando.

2.2.1. Problema: ¿Cómo mantener los tests?.

La comunidad de ingeniería de software ha producido herramientas efectivas y buenas prácticas para lograr refactorizaciones en el código base que otorguen un buen diseño de código. Sin embargo, en cuanto al código de los tests unitarios no es así. De hecho, estos son raramente modificados y carecen del cuidado que se le da al código base y no se piensa en su modularidad ni extensibilidad.

Por lo cual no es difícil encontrar deficiencias como **solapamientos** entre test methods o más general, entre test cases. Estos solapamientos pueden ser de carácter estático: duplicación de código, o bien dinámicos, es decir que dos tests methods tienen ejecuciones similares y por consiguiente testean lo mismo.

Estas deficiencias en el diseño y calidad del código de las pruebas unitarias tiene conse-

cuencias importantes en la calidad del código testeado y en el mismo proceso de desarrollo:

- **Performance** Debido a los solapamientos previamente mencionados, muchas veces existe **ejecución redundante** que va en contra de las características deseables de una suite de tests[**??**]. Muchas veces los tests dejan de ejecutarse con la frecuencia deseada.
- **Debugging** Otra deficiencia conocida es cuando al correr los tests en presencia de un defecto, éste se queda en evidencia por muchos tests methods, lo cual dificulta la identificación de la causa del bug y su corrección. Coloquialmente se hace más difícil responder la pregunta: *¿Cuál test miro primero?*

De esta manera la confiabilidad en el producto y su calidad se ven impactadas negativamente.

Un caso muy claro del impacto del mal diseño y poco cuidado en los tests sucede en la práctica de **Integración Contínua** (Continuous Integration)[**??**]. En esta práctica cada *feature* se implementa tan pronto como es posible, se testea y se pasa a producción de inmediato. De esta manera el equipo de desarrollo realiza varias integraciones por día y el cliente obtiene rápidamente las nuevas funcionalidades a medida que las solicita.

A modo de ejemplo, la empresa MediaGeniX[1] realiza la integración continua desde hace un tiempo. Ahí, 30 desarrolladores trabajan sobre el mismo producto y cada uno de ellos construye varias versiones al día. Cada versión que se va a pasar a producción debe pasar su suite de pruebas que comprende alrededor de 30.000 tests. Ellos realizan al menos 3 integraciones diarias en promedio por lo cual recurren a técnicas de paralelización de ejecución de tests para lograrlo. Esto introduce un alto costo económico asociado al los recursos de hardware(servidores, clusters) y además un alto costo en tiempo.

Esto evidencia la relevancia de mejorar la performance mediante la refactorización y restructuración de los unit tests.

[1] MediaGeniX, `http://www.mediagenix.tv`

Capítulo 3

Trabajo relacionado

3.1. Cobertura de Código

La mayoría de la investigación y aplicaciones relacionados a los tests va por el lado del coverage, que es uno de los indicadores más importantes al momento de determinar la confiabilidad de un software.

En ese aspecto

3.2. Comparación de Tests

Capítulo 4

Descripción de la Solución

4.1. Solución

Como se detalló anteriormente se desea restructurar y refactorizar los tests porque X, Y y Z.

Considerando el estudio comparativo entre datos estáticos y dinámicos Profiling pq asi se ve lo que realmente se ejecuta Métrica

Clustering –¿reducir el esfuerzo de la inspección Visualizacion –¿inspección

4.2. Implementación

El proyecto TestSurgeon fue completamente implementado en el lenguaje Pharo Smalltalk*Pharo*[1] ya que es en lenguaje en el que está escrito el framework de profiling utilizado.

4.2.1. Profiling

La componente principal de TestSurgeon es la que realiza el profiling. Por eso se escogió una herramienta que fuera fácil de utilizar y que permitiera obtener la mayor cantidad de datos sobre la ejecución de un test.

El framework de profiling escogido fue *Spy* y está escrito en el lenguaje Pharo. Entre sus características está que es un profiler orientado a los métodos. Considerando que en Pharo todas las interacciones corresponden a objetos que se comunican a través de mensajes, este punto es muy importante. Si se quiere conocer el comportamiento de un test con gran detalle, el hacer profiling de los métodos que son llamados durante su ejecución es fundamental.

[1] Pharo Project - `http://www.pharoproject.org` - última visita 12-07-2013

Además, Spy posee una arquitectura muy simple y fácil de extender. Consta de cuatro clases principales: `Profiler`, `PackageSpy`, `ClassSpy` y `MethodSpy`. `Profiler` es la clase principal con las características necesarias para la instrumentación, ejecución y obtención de datos. Las demás clases contienen información sobre el profiling de los paquetes instrumentados y de sus clases y métodos respectivamente.

La clase `MethodSpy` técnicamente es un *wrapper* de un método en Pharo (una instancia de la clase `CompiledMethod`) que acumula datos sobre el contexto de su ejecución. Para esto, la clase provee los métodos `beforeRun: with: in:` y `aftefRun: with: in:` que son ejecutados justo antes e inmediatamente después de ejecutado el método instrumentado. Así se obtienen los datos del impacto de la ejecución de dicho método en el objeto y en los objetos con los cuales colabora. Estos métodos son abtractos y deben concretarse en las aplicaciones que decidan usar Spy extendiendo sus clases.

Para hacer uso de Spy se deben entonces extender las clases antes mencionadas y acondicionarlas al dominio específico del problema. Estas clases son: `TSProfiler`, `TSPackage`, `TSClass`, `TSMethod`. Para las aplicaciones que usan esta información, se necesitan obtener datos de ejecución tanto de los tests como del código testeado, es por eso, que para representar un método de test se tiene también la clase `TSTestMethod` que es un wrapper de la clase `TSMethod` donde se guardan métricas y datos procesados de ésta que corresponden a un test method y no a un método testeado, como por ejemplo: `testedMethods`, `testedClasses` o `visualizeDifferencesWith:`, entre otros.

A continuación se describen las clases que componen al paquete `TestSurgeon-Core-Spy`:

4.2.1.1. La clase `TSProfiler`

Es la clase principal, su interfaz componen los siguientes constructores `buildForClassCategory:` y `buildForPackagesMatching:`. En ambos, el argumento es un objeto String, en el primero especifica una categoría y en el segundo una expresión regular que representa un grupo de categorías o paquetes de software de Pharo. Luego de obtenidos la categoría o las categorías, las instrumenta creando las clases `TSPackage`, `TSClass` y `TSMethod`, posteriormente corre los tests y finalmente retorna la instancia del profiler.

La instancia de `TSProfiler` tiene los siguientes métodos:

`allTestMethods` retorna una colleción colección con instancias de la clase `TSMethod` que corresponde a test methods.

`allTestedMethods` retorna una colleción colección con instancias de la clase `TSMethod` que corresponde a los métodos testeados.

`compiledToSpyMethod:` se le pasa como argumento una instancia de la clase `CompiledMethod` y retorna la instancia `TSMethod` que lo encapsula.

`plainClasses` retorna una colección con todas las clases instrumentadas (instancias de `TSClass`) que no corresponden a unit tests.

`testClasses` retorna una colección con todas las clases instrumentadas (instancias de `TSClass`) que corresponden a un unit test.

4.2.1.2. Las clases `TSPackage` y `TSClass`

Estas dos clases como ya se dijo, representan a los paquetes y clases instrumentadas respectivamente. En general no se necesitó información particular de los paquetes por lo que no hay nada que comentar sobre su implementación.

Sobre `TSClass` esta representa tanto a las clases testeadas como a lo unit tests, y posee mayoritariamente métodos de ayuda o *helpers* para las distintas aplicaciones (visualización, clustering, etc) como por ejemplo: `allMethodsTestedBy:` donde se le entrega un `TSTestMethod` y se retorna todos los métodos definidos en su clase que son llamados durante la ejecución de ese test.

4.2.1.3. Las clase `TSMethod`

El profiling de Spy es orientado a los métodos, por tanto en esta clase es donde se registran todos los datos de la ejecución para caracterizar a los tests y poder diferenciarlos.

Su comportamiento está dividido en dos partes principalmente: los métodos de profiling donde se registran los datos de ejecución, y sus accesores, métodos que entregan la información obtenida. Los métodos de profiling son dos: `beforeRun: with: in:` y `aftefRun: with: in:` (ver código en 4.2.1.3 y en 4.2.1.3)

```
1  beforeRun: methodName with: listOfArguments in: receiver
2      | testMethod |
3      testMethod := self profiler class currentTestMethod.
4      currentTestMethodSpy :=
5          spyMethodMapping at: testMethod ifAbsentPut:
6              [ self profiler >> testMethod class name >> testMethod selector ].
7
8      "Local copy of instrumented test method"
9      self addTestMethod: currentTestMethodSpy.
10
11     "ADD RECEIVER"
12     self updateReceiversWith: receiver for: currentTestMethodSpy.
13
14     "SIDE EFFECTS"
15     receiverSnapshot := receiver snapshotAsInteger.
```

```
afterRun: methodName with: listOfArguments in: receiver

    "Count if it was a change in the receiver state"
    (receiver snapshotAsInteger ~= receiverSnapshot) ifTrue: [self plusOneIn: stateChanges at:
        currentTestMethodSpy ].

    currentTestMethodSpy := nil.
    receiverSnapshot := nil.
```

4.2.1.4. Las clase `TSTestMethod`

4.2.2. Visualización

4.2.3. Clustering

4.2.4. Browser

Todo el código detallado en las secciones anteriores está almacenado en el sitio *Smalltalk-Hub*[2] el repositorio ubicado en la siguiente dirección: `smalltalkhub.com/#!/~PabloEstefo/TestSurgeon/`.

[2]Smalltalk Hub - `http://www.smalltalkhub.com` - última visita 12-07-2013

Capítulo 5

Caso de estudio: Roassal

Capítulo 6

Conclusión y Trabajo Futuro

6.1. Conclusión

6.2. Trabajo Futuro

- Explorar otras visualizaciones - Probar con más softwares - Más métricas a considerar (ver repo GrandFinals)