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

REESTRUCTURACIÓN Y REFACTORIZACIÓN DE UNIT TESTS CON TESTSURGEON

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

PABLO IGNACIO ESTEFO CARRASCO

PROFESOR GUÍA:
ALEXANDRE BERGEL

MIEMBROS DE LA COMISIÓN:
ROMAIN ROBBES
JOSÉ PINO URTUBIA

Este trabajo ha sido parcialmente financiado por el programa Google Summer of Code 2012 y Object Profile.

SANTIAGO DE CHILE
NOVIEMBRE 2013

Resumen

Actualmente la actividad de Testing es fundamental dentro del ciclo de desarrollo de cualquier proyecto de software serio. Es más, las metodologías ágiles elevan su relevancia dentro de la construcción del software a tal nivel que está prohibido añadir una nueva funcionalidad sin que se haya escrito previamente un test que la valide.

A medida que el software crece en funcionalidades y cambian los requerimientos se vuelve más complejo. Es por eso que existen varias técnicas para reestructurar el código haciéndolo más flexible a los cambios y permitiendo que crezca.

Sin embargo, los test también crecen en número y en complejidad. Por lo que no son raros los casos de test redundantes tanto desde el punto de vista de su código fuente (duplicación de test) como de su ejecución. Pero a diferencia con el código "funcional", poco esfuerzo se ha realizado por parte de la industria por promover técnicas y crear herramientas que faciliten la tarea de mantener su estructura y diseño limpio.

Una de las consecuencias importantes de este problema, es el gran tiempo que toma ejecutar todos los tests. Al haber redundancia, la ejecución tarda más tiempo del necesario lo hace que los desarrolladores los corran con menos frecuencia e inclusive invierten menos tiempo en escribir nuevos test lo cual minimiza la cobertura. Esto último atenta críticamente en la confiabilidad del código base y por ende de la aplicación.

En este trabajo se propone una herramienta para detectar problemas de diseño de los tests. *TestSurgeon* aborda este problema desde dos perspectivas de análisis principales: su código fuente y su ejecución. A través de una intuitiva interfaz, el desarrollador puede navegar sobre las pruebas unitarias y realizar comparaciones entre tests guiado por métricas dedicadas que facilitan la detección de casos interesantes. Además provee una completa visualización que condensa dos métricas que describen y diferencian la ejecución de los test en comparación, permitiendo realizar un análisis eficaz. Finalmente, TestSurgeon permite detectar diferencias semánticas entre tests y encontrar redundancias entre estos para una posible refactorización.

Se presentan distintos escenarios de refactorización y reestructuración que son detectados por TestSurgeon. Estos son descritos con ejemplos reales en base a una experiencia de aplicación de *TestSurgeon* sobre los tests de *Roassal*, un motor de visualización ágil.

TestSurgeon ganó el primer lugar en la competencia internacional *ACM Student Research Competition* (categoría pregrado) durante la conferencia ICSE (principal en Ingeniería de Software) el año 2012.

A mi madre, que me ha apoyado siempre.

Agradecimientos

Primero quiero que agradecer a mis padres: especialmente a mi mamá por su apoyo, su aliento para superarme y pensar en grande. A mi papá por sus charlas motivacionales en los momentos complicados. Al Tomás por ser un gilcu.

A Pilar por ser mi gran compañera en este viaje en la universidad y todo lo que eso significa. A mis amigos que he conocido en la U: al grupo de la sección 3 (y los que se fueron agregando) y a los que fui conociendo en computa con quien he compartido todo este tiempo. A la Vane y a Juampi por su amistad estos últimos años.

No puedo olvidarme de aquellos profesores que me formaron y motivaron en el colegio: tía Paty, al profesor Jesús Castañeda y al profe Conejeros. Finalmente agradecer al profesor Alexandre Bergel, quien me planteó importantes desafíos los cuales me han abierto muchas puertas y oportunidades que ni soñaba.

A todos ellos: ¡Muchas gracias!

Tabla de contenido

Índice de tablas

Índice de ilustraciones

Capítulo 1

Introducción

El Testing es una actividad importante en el desarrollo de los proyectos de software en nuestro días. Los tests automatizados ayudan al desarrollador a asegurar calidad en su código y a detectar posibles *bugs* o defectos en la aplicación. Más aún, los tests pueden también ser considerados como una "documentación ejecutable", evitan tiempo de depuración, y pueden ser utilizados también como una primera aproximación a entender código ajeno.

Dada la naturaleza de las pruebas de software, su estudio e investigación se ha focalizado en mejorar su efectividad para hacer software cada vez más confiables y robustos. Al igual que el software funcional, los tests son parte del código fuente de la aplicación. La comunidad de ingeniería de software ha desarrollado diversas técnicas y metodologías para mantener limpio el diseño del código base (o funcional). Sin embargo, poco trabajo se ha realizado para que el código de los tests mantenga una estructura y diseño limpios. Situación que pasa a ser cada vez más problemática a medida que el software crece en tamaño y en complejidad.

1.1. Terminología

Para la lectura adecuada del documento, se presenta la siguiente terminología con los conceptos principales que se utilizarán a lo largo del documento:

test : Corresponde a un trozo de código que tiene como finalidad simular un comportamiento particular de alguna componente o componentes del sistema para realizar verificaciones que validen el comportamiento esperado de estas. También se le nombra: *método de test*, *test method* o *prueba de software*.

unit test : Corresponde a un grupo de tests que, en conjunto, verifican una funcionalidad en común. También se le nombra: *prueba unitaria* o *grupo de tests*.

smell : Se refiere a una deficiencia en el diseño del código que da pie a problemas en su mantenibilidad y/o extensibilidad. Cuando existen *smells* sobre el código se pruebas se nombra como un *test smell*.

1.2. Motivación

Actualmente el *testing* es una actividad clave dentro del desarrollo de cualquier proyecto de software serio. De hecho, las metodologías ágiles de desarrollo consideran la creación de tests como el punto de partida de la iteración o ciclo de desarrollo. Ejemplos de estas son: Test-Driven Development (TDD) y Behavior-Driven Development (BDD). El escribir tests que describen la funcionalidad esperada de una unidad de software (TDD) y/o el comportamiento esperado de un conjunto de unidades (BDD), son formas de producir código confiable, puesto que no se agrega otra funcionalidad sino hasta que existe un test que lo simula y describe su funcionamiento esperado.

A medida que el software crece en complejidad y tamaño, de igual manera el código se va volviendo más complejo, lo cual atenta contra el crecimiento del proyecto. Para esto, la industria y la academia han desarrollado diversas formas de reestructurar y/o refactorizar el código base para enfrentar el *cambio* de la forma menos costosa. Sin embargo esto no ha sido igual para el código que corresponde a los tests. A medida que el sistema evoluciona los tests necesitan hacerlo también para mantenerse actualizados con el sistema [25]. Pero muchas veces esto no sucede, los tests carecen del mismo cuidado que el código base por lo cual se hace necesario reescribir los tests. Es más, no es raro que se deje de escribir tests por el costo en tiempo que esto implica.

Por otro lado, la aplicación estricta de estas metodologías ágiles incrementa rápidamente la cantidad de tests. Y estos pueden clasificarse como: de bajo o alto nivel. Los tests de bajo nivel prueban las unidades fundamentales del sistema, mientras que los de alto nivel testean la interacción entre estas. Esto significa que al ejecutar tests de alto nivel, implícitamente se está siendo redundante con respecto a los de bajo nivel ya que está testeando una pieza de código que fue previamente ejecutada por uno o más tests de bajo nivel. Y esto puede repetirse en varios tests de alto nivel.

Esta redundancia representa un costo en tiempo que en algunos casos puede ser considerable, teniendo en cuenta que los tests se ejecutan frecuentemente cada vez que se agrega o modifica el código.

Bajo este escenario, se hace necesaria una forma de reducir esta brecha entre el código base y el código de test, y facilitar la detección de *test smells* que significan un gran costo tanto para los desarrolladores como el cliente.

1.3. Objetivos

El objetivo general de este trabajo es desarrollar una herramienta que permita a los programadores refactorizar y reestructurar sus tests de una manera más fácil, segura y que considere la versatilidad de su uso.

Objetivos Específicos

1. Identificación de las métricas relevantes para caracterización de tests methods desde el punto de vista de un análisis dinámico
2. Desarrollar una visualización que permita detectar redundancia y solapamiento entre grupos de test methods (o unit tests)
3. Investigar refactorizaciones automáticas y semi-automáticas para unit tests y sus implicancias el rendimiento y cobertura
4. Desarrollar una interfaz gráfica efectiva y usable para el uso cotidiano dentro del desarrollo de software

1.4. Estructura del documento

En el Capítulo 2 se presenta el problema de la mantenibilidad de tests en detalle, su contexto y relevancia. En el capítulo siguiente, un marco teórico entrega los conceptos técnicos necesarios (Sección 3.1), y se presenta una serie de antecedentes que muestra el trabajo realizado tanto por la academia como por la industria sobre el problema de diseño del código de tests (Sección 3.2). A continuación se presentan dos estudios (Sección 4.1 y Sección 4.2) realizados durante el desarrollo de esta memoria que revelan aspectos no estudiados del problema. El aprendizaje obtenido y las conclusiones de éstos determinaron varias decisiones de diseño de la solución.

Luego, con la base conceptual y contextual del problema presente, se describe en detalle la solución propuesta: *TestSurgeon*. Primero en forma conceptual en el Capítulo 5 y después en forma técnica, su implementación, en el Capítulo 6. Junto con esto, en el Capítulo 7 se muestra una aplicación de la herramienta sobre las pruebas unitarias de un software no-trivial.

Finalmente en el Capítulo 8 se revisa el cumplimiento de los objetivos planteados, las conclusiones del trabajo, así como también algunas directrices para continuarlo.

Se adjuntan además algunos apéndices que complementan el documento y entregan datos que por su extensión no pudieron ser incluidos directamente en los capítulos.

Capítulo 2

Especificación del Problema

2.1. Contexto: Test como "conductores" del diseño

Durante el desarrollo de un software una de las actividades principales es el testeo de los requerimientos. Existen variadas técnicas, metodologías y artefactos relacionados a esta actividad. Las pruebas automatizadas consisten en trozos de código donde interactúan distintas entidades del sistema para luego verificar su estados y, de esta forma, asegurarse de que el software funciona como se espera. La creación de pruebas automatizadas es una práctica cotidiana en cualquier proyecto de software y comprende parte importante de los artefactos generados.

En particular, los proyectos de software ágiles dan gran importancia a estos artefactos, ya que cada historia de usuario (requerimiento) debe tener pruebas de aceptación (tests de alto nivel) que validen el comportamiento del sistema que el cliente espera. Estos se escriben y detallan en conjunto con el cliente para asegurar que el código que se creará cumple lo que éste pide: ni menos, ni más.

Estas historias de usuario están escritas en lenguaje natural por lo que se deben *transcribir* a lenguaje técnico: código fuente. Entonces, antes de escribir una funcionalidad, los desarrolladores crean los tests: código fuente que garantiza que una característica se comporta como es requerida. Este proceso se denomina *Test-Driven Development* (o TDD) [2] o Desarrollo Dirigido por Pruebas y consiste en los siguientes pasos:

1. Escribir el test
2. Ejecutarlo. Falla ya que no se ha escrito el código que suple la nueva característica
3. Se escriben las clases y métodos que hacen que el test funcione
4. Se vuelve a ejecutar el test. Pasa.
5. Refactorizar el código recién escrito para mantener un diseño limpio y mantenible en el tiempo.

Esta práctica está dentro del núcleo de la filosofía que enmarca la agilidad a tal nivel que

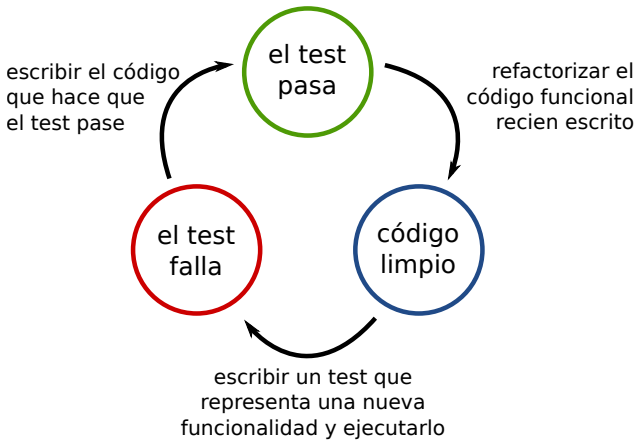


Figura 2.1: Esquema del ciclo de desarrollo aplicando Test-Driven Development

está prohibido agregar una nueva funcionalidad sin, previamente, haber escrito un test que lo valide [2]. Esta queda documentado en el libro de Robert Cecil Martin (más conocido como "Uncle Bob") [21] uno de los escritores del manifiesto ágil [9] 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."*

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.

Finalmente, de esta manera, los tests van empujando y a la vez conduciendo el desarrollo del código base o funcional.

2.2. 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. Sin embargo, en cuanto al código de las pruebas de software no sucede en igual proporción. Es más, éstos son raramente modificados y carecen del cuidado que se le da al código base, sin pensar su modularidad ni extensibilidad.

Esto sucede, en parte, por la importancia primaria de los tests: asegurar calidad. Cualquier modificación a los tests podría, eventualmente, mermar la confiabilidad en el software. Es decir, debilitar los tests.

Por tanto es fácil encontrar deficiencias como **solapamientos** entre test methods o más general, entre test cases. Estos solapamientos pueden corresponder a duplicación de código, o bien a reduncancia en ejecución: es decir que dos test methods tienen ejecuciones similares y por consiguiente representan una ineficiencia de tiempo.

Estas deficiencias en el diseño y calidad del código de las pruebas unitarias tienen consecuencias importantes en la calidad del código testeado y en el mismo proceso de desarrollo:

- **Rendimiento** Debido a los solapamientos previamente mencionados, muchas veces existe **ejecución redundante** que va en contra de las características deseables de una suite de tests. Este es un incentivo a dejar de ejecutar la batería de pruebas con la frecuencia necesaria.
- **Depuración**[1] Otra deficiencia conocida es que al correr los tests en presencia de un *bug*, éste queda en evidencia por muchos tests, lo cual dificulta la identificación de su causa y su corrección. Coloquialmente se hace más difícil responder la pregunta: *¿Cuál test miro primero?*
- **Documentación** Los tests son considerados también documentación ejecutable, por lo cual, cualquier cambio o reestructuración altera su calidad como documentación (o "readability" en inglés).

Lo anterior ilustra la versatilidad de las pruebas automatizadas en comparación a los demás artefactos de software. Sus múltiples usos y funciones hacen del problema de refactorización y reestructuración una labor compleja. Sin embargo, la característica prioritaria en cualquier alternativa de solución es la inocuidad: la confiabilidad del código fuente no puede disminuir por mejoras en el diseño u optimizaciones en los tests.

Un caso muy claro de la influencia del mal diseño y poco cuidado en los tests ocurre cuando el ciclo de desarrollo utiliza mecanismos de **Integración Contínua** (o *Continuous Integration* [8]). Esta práctica intenta integrar a producción tan pronto como sea posible. Entonces, cada nueva característica (o *feature*) se divide en partes más pequeñas que puedan ser desarrolladas rápidamente para que queden en producción. Entonces se crean los tests, se implementa la funcionalidad y se realiza la integración: compilación (de ser necesario) y pruebas de regresión (o *regression testing*). De esta manera se verifica que el código nuevo funciona y pasa los tests escritos anteriormente. 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.

Este proceso requiere ejecutar muchas veces las pruebas (al menos una vez por cada integración), por tanto, la redundancia en ejecución implica redundancia en tiempo. A modo de ejemplo, la empresa MediaGeniX[2] realiza la integración continua desde hace algunos años. En ella, treinta 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, en promedio, 3 integraciones diarias, por lo cual recurren a técnicas de paralelización de ejecución de tests para lograrlo. Independientemente de la estrategia, el costo en tiempo es alto. Esto se podría optimizar detectando redundancia en la ejecución.

[1] o *Debugging*.

[2] MediaGeniX [en línea] <`http://www.mediagenix.tv`>[Consulta: 03/11/2013]

Capítulo 3

Antecedentes

3.1. Marco Teórico

Esta sección pretende exponer, definir y desarrollar algunos conceptos necesarios para una adecuada lectura de este trabajo. Primero se presenta la técnica de *profiling* que permite analizar el comportamiento y rendimiento de un software. Luego, se muestra en detalle el framework de tests más utilizado en lenguajes de programación orientados a objetos: *SUnit*. Se detallan sus componentes principales y su arquitectura, conceptos que se utilizarán posteriormente para la descripción del problema y la solución propuesta.

3.1.1. Profiling

El *Perfil de Ejecución* (o *Execution Profiling*) es una forma de análisis dinámico que consiste en monitorear un software durante su ejecución para posteriormente analizar los datos obtenidos. Entonces los *Profilers* son herramientas para ayudar a los ingenieros de software a recolectar datos durante la ejecución y analizarlos, y así poder determinar de mejor manera en qué parte de un sistema de software se gasta el tiempo de ejecución.

Para la obtención de datos, existen varias técnicas de profiling. Los principales son profiling por instrumentación (o *Instrumentation-based profiling*) y a través de muestras (o *Sampling-based profiling*). En la primera, se inserta código de instrumentación antes (instrumentación estática) o durante la ejecución (instrumentación dinámica). En la primera se agrega el código justo antes de ejecutar y por tanto la sobrecarga (o overhead) es menor. Sin embargo, cualquier código generado dinámicamente no será instrumentado. En tanto, la técnica basada en muestras aproxima el tiempo que se gastó en un método de la aplicación deteniendo el programa y guardando la colección de métodos que fueron ejecutados.

Usualmente los *profilers* son utilizados con motivo de optimizar un sistema de software en cierto aspecto. Los datos a obtener a través de la técnica de profiling dependen completamente del problema a optimizar. Es por esto que el profiler a utilizar debe permitir la recolección de

los datos necesarios con una mínima sobrecarga (o *overhead*) para conseguir datos fidedignos y a la vez representativos.

La herramienta de profiling escogida para este trabajo es el framework *Spy* [4], que utiliza la técnica de profiling basado en instrumentación del tipo dinámico. Spy está diseñado para analizar softwares en el contexto de orientación a objetos ya que se permite obtener métricas contextuales como: el número de ejecuciones de un método, cantidad de objetos instanciados de una clase, etc. Para mayor información sobre este y su implementación en *TestSurgeon* ver la Sección 6.2.

Algunas de las aplicaciones exitosas basadas en Spy son *Hapao*[1]: una herramienta de cobertura de test desarrollada por Vanessa Peña y Alexandre Bergel [5], que permite gráficamente ver la cobertura de las clases y métodos contrastando la cobertura de los últimos con el grado de complejidad de éstos; y *Kai*[2]: un profiler que permite optimizar la ejecución de un software indicando, por ejemplo, dónde conviene incrustar una estructura de *caching* para evitar redundancia de cálculo. En la Figura 3.1 y Figura 3.2 se muestran capturas de pantalla de Hapao y Kai respectivamente.

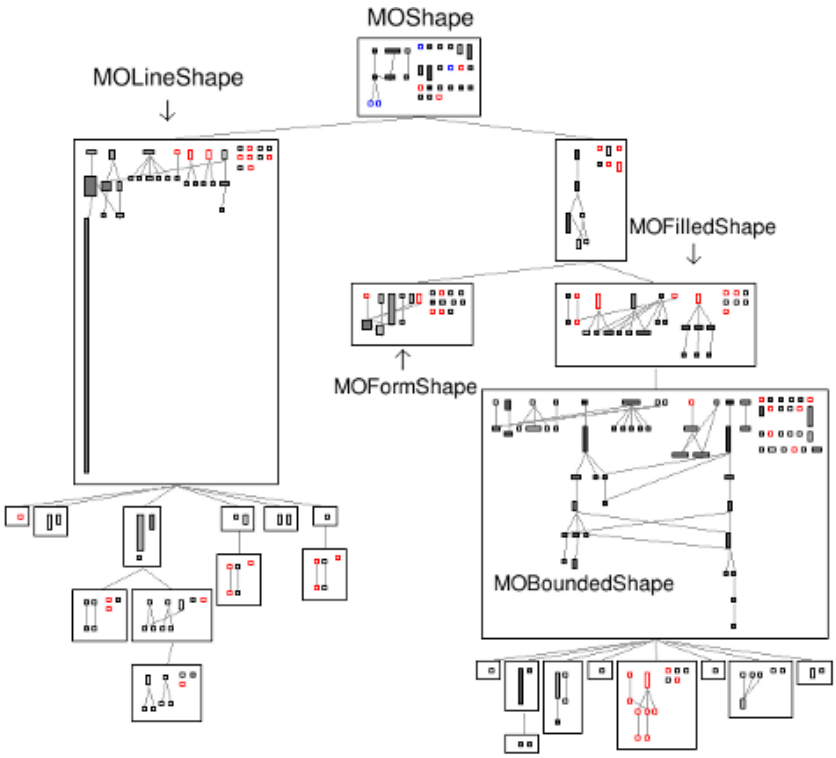


Figura 3.1: Ejemplo de la visualización de la cobertura de test en Hapao

[1]Hapao [en línea] <`http://hapao.dcc.uchile.cl`>[Consulta: 03/11/2013]

[2]Object Profile. Kai Profiler [en línea] <`http://www.objectprofile.com/#/pages/products/kai/overview.html`>[Consulta: 03/11/2013]

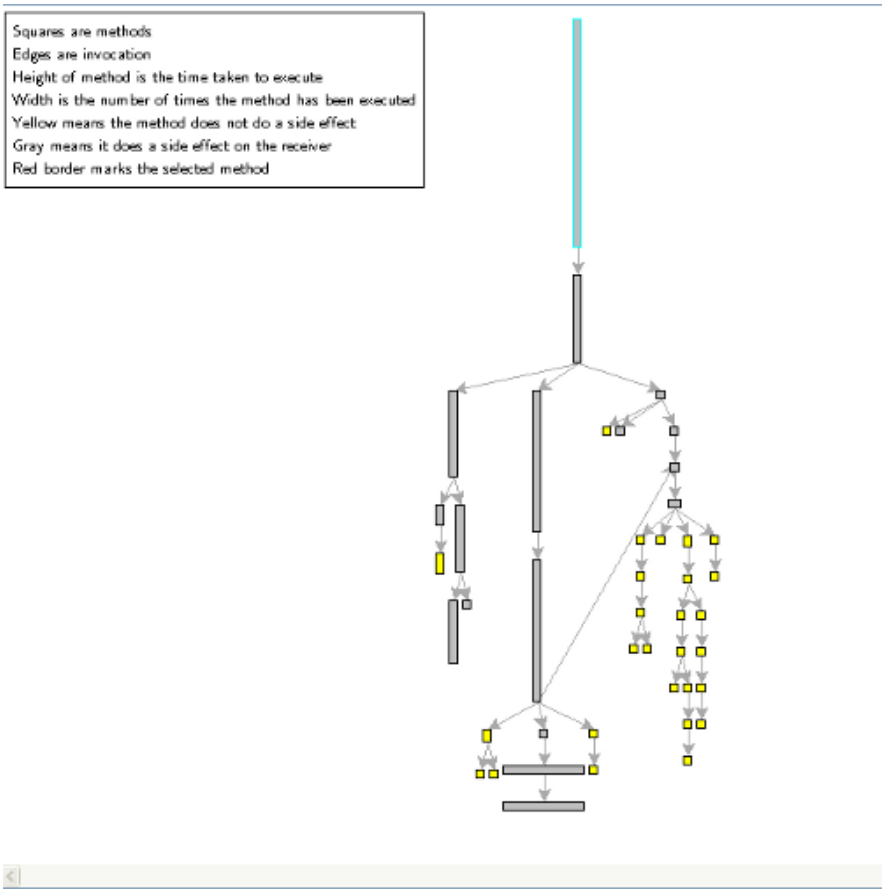


Figura 3.2: Diagnóstico de ejecución entregado por Kai

3.1.2. SUnit framework

SUnit [1] es un framework de testing que apoya la creación de tests automatizados. SUnit es la implementación en Smalltalk del framework xUnit[3] que de hecho históricamente es la primera. Las cualidades que caracterizan xUnit son:

- Los tests son repetibles: Cada test se puede ejecutar cuantas veces se quiera.
- Los tests se ejecutan sin necesidad intervención humana: Se puede dejar los tests "corriendo" durante la noche sin necesidad de intervenir en su ejecución.
- Los tests cuentan una historia: un test cubre al menos una funcionalidad del sistema. Un test entonces es un escenario que cualquier desarrollador puede leer para entender dicha funcionalidad.

En SUnit los tests están contenidos en clases llamadas unit tests. Los unit tests son clases que extienden o heredan de `TestCase`, una de las clases arquitectónicas del framework. Los unit tests suelen nombrarse con el sufijo `Test`, por ejemplo `MarioBrosTest`. Los tests o test methods, por su parte se nombran con `test` como prefijo, ejemplo `testMarioFallDownAndLoseOneLife`.

3.1.2.1. Arquitectura de *SUnit*

Estructuralmente, SUnit se compone de cuatro clases: `TestSuite`, `TestCase`, `TestResource` y `TestResult` como se muestra en Figura 3.3.

`TestCase` representa un unit test o una familia de test que comparten el mismo contexto. Esta clase está pensada para ser extendida por clases concretas. El contexto se especifica en el método `setUp` donde las variables de instancia de cada subclase de `TestCase` se inicializan y definen el contexto en el cual se ejecutará cada test. También se define el método `tearDown` que es responsable de limpiar los objetos que hayan sido inicializados y modificados durante la ejecución del test para que pueden ser reinicializados por el método `setUp`. Tanto `setUp` como `tearDown` son métodos abstractos. Entonces cada test se ejecuta de la siguiente manera:

$$\texttt{setUp} \rightarrow \texttt{testMarioFallDownAndLoseOneLife} \rightarrow \texttt{tearDown}$$

`TestSuite` contiene una colección de unit tests. Una instancia de `TestSuite` está compuesto por instancias de subclases de `TestCase` y `TestSuite`. Las clases `TestSuite` y `TestCase` forman el patrón composite.

`TestResult` representa los resultados de una ejecución de `TestSuite`. Contiene el número de tests que pasaron, el número de tests que fallaron y el número de errores.

`TestResource` representa un recurso que es usado por un test o un conjunto de tests. Un recurso se asocia con una clase hija de `TestCase` y se ejecuta automáticamente antes

[3] la **x** se reemplaza por el lenguaje de programación, por ejemplo: jUnit para Java.

de la ejecución de todos los tests y una sola vez.

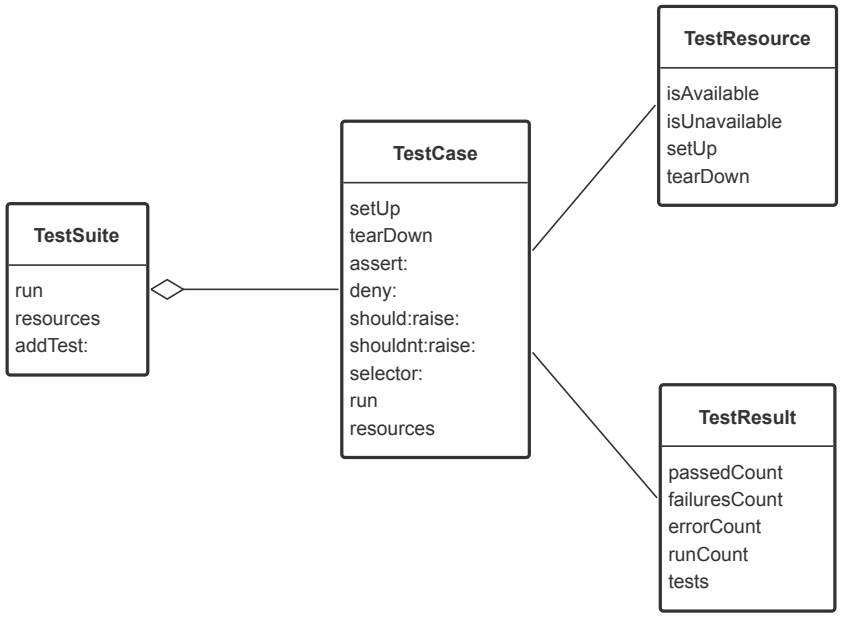


Figura 3.3: Las clases que representan el núcleo de *SUnit*

3.1.2.2. Estructura de un test

La estructura de cada test puede dividirse en dos partes: *fixture* (o escenario) y *assertions* (o verificaciones). En el fixture del test se crean e inicializan los objetos del escenario y se les hace interactuar a través de mensajes. No se considera parte del fixture todos los objetos inicializados y las operaciones realizadas en la ejecución del método `setup`, aunque sí afectan la interacción en el fixture. Posteriormente, en el test hay una serie de validaciones o verificaciones, donde se comprueba el estado de los objetos a través de métodos verificadores heredados desde `TestCase` tales como: `assert:` para verificar una afirmación, `deny:` para negarla, `should:raise:` para verificar que una excepción en particular se lanzó, y `shouldnt:raise:` para asegurar que no se lanzó cierta excepción.

A modo de ejemplo se muestra un código que bien podría ser un test básico del popular juego *Super Mario Bros.*, donde un plomero de traje rojo va pasando por escenarios llenos de trampas y dificultades para poder llegar al castillo donde supuestamente está su princesa y rescatarla. Una de las trampas más comunes y sencillas del juego son hoyos de profundidad infinita donde Mario puede caer y perder una vida. En el ejemplo, inicializamos la variable local `mario` con una instancia de la clase `MBMario` que por defecto tiene 3 vidas[4]. Luego lo insertamos al juego junto con un escenario en el cual, luego de dar tres pasos debiera caer a un hoyo y perder una vida. En el juego no se acaba sino hasta que a Mario se le acaban las vidas y vemos el mensaje de "Game Over".

[4]En la plataforma NES, Mario parte con 3 vidas.

```
MarioBrosTest>>testMarioFallDownAndLoseOneLife
    " *--* Fixture *--* "
    | game mario scene |
    game := MBGame new.
    mario := MBMario new. " todos saben que Mario parte con 3 vidas"
    scene := #(#floor #floor #hole #floor). " un escenario con un hoyo en la tercera posicion"

    game setScene: scene.
    game setPlayer: mario.

    " hagamos a mario avanzar en el escenario"
    mario moveOneStep.
    mario moveOneStep.
    mario moveOneStep.

    " *--* Assertions *--* "

    self assert: (game currentPosition = #hole) "mario cayo al hoyo"
    self assert: mario isDead. "deberia haber muerto puesto que cayo al hoyo"
    self assert: (mario lives = 2). "mario ahora tiene 2 vidas"
    self deny: game isGameOver.
```

3.2. Trabajo Relacionado

A continuación se presenta una revisión de la literatura sobre el problema de mantenibilidad de las pruebas de software, y una revisión también de las herramientas de cobertura de test que abordan tangencialmente el problema.

3.2.1. Revisión de la literatura

Luego de revisar la bibliografía existente, entre los problemas con mayor investigación está la cobertura de los tests: cómo medir y mejorar la cobertura del software testeado para aumentar la confiabilidad en éste; y el testeo por mutación: como mejorar la efectividad de los tests realizando variaciones en el código testeado que representan bugs y comprobar cuáles tests definidos atrapan o detectan esas anomalías. Con respecto al problema de mantenibilidad, las estrategias encontradas en la literatura corresponden a priorización de tests y comparación de tests, pero en volumen mucho menor a los tópicos mencionados anteriormente.

La priorización de tests y el *clustering* son acercamientos populares para manejar el gran costo de correr una cantidad considerable de tests. Shin Yoo *et al.* [31] han presentado un mecanismo para reducir las comparaciones 1-a-1 agruparlos y de esa manera facilitar la tarea humana de priorizar tests. Ellos crearon clusters de tests agrupándolos por similitud de ejecución. Cada ejecución fue representada por una cadena binaria donde cada dígito (1 o 0) representa si cierto método fue ejecutado durante ésta. Entonces, la similitud de cobertura es medida a través de una distancia binaria.

Vengala *et al.* [30] crearon un mecanismo para usar análisis estático y dinámico para optimizar las actividades de testing tales como *selección de test*, *eliminación de redundancia* y *priorización de test*. Aunque sus métricas siguen el mismo espíritu que *TestSurgeon*, están definidas para un ambiente de programación procedural lo cual es incompatible en un ambiente orientado a objetos.

Greiler *et al.* [15, 14] aborda el problema de entendimiento de tests suites proponiendo un framework sobre correlación entre similitud de tests (Test Similarity Correlator). Este último produce una traza de ejecución que caracteriza la ejecución de los tests a través de cuatro tipos de eventos: ejecución de un test method, evento de set-up y tear-down, ejecución de un método y lanzamiento de excepción. El entendimiento de test suites se realiza comparando las trazas de ejecución.

El código de test, desde el punto de vista de un artefacto de software, tiene múltiples usos. Uno de ellos es documentación [18]. Especialmente para desarrolladores nuevos que tienen que enfrentarse con sistemas legados, los tests son altamente relevantes como ejemplos de código. Siguiendo esta perspectiva, Lienhard *et al.* [20] proponen una herramienta que facilita la tarea de escribir nuevos tests para esos sistemas a través de una visualización que llaman *Test Blueprint*, como una guía para el entendimiento de los tests y el mantenimiento del código base.

Reichhart *et al.* [25] presenta *TestLint*, una herramienta para determinar la calidad de los tests. En su trabajo presenta una aplicación de esta sobre una gran cantidad de unit tests obtenidos desde distintos repositorios open source. Además entrega una gran lista de *test smells* de los cuales uno es analizado por *TestSurgeon* y corresponde al escenario descrito en la Sección 7.4.2.

De los autores encontrados, el que aborda con mayor profundidad el problema de mantenibilidad en los tests es Markus Gaelli. Dentro de su investigación [11, 13] presenta un estudio de clasificación de unit tests [12] representado por un árbol binario cuyas ramas indican cumplimiento de un criterio de clasificación. En este árbol, las hojas son los distintos tipos de test: Test donde se llama un solo método exactamente una sola vez(50 % del total de tests estudiados), Test donde se llama un solo método en varios escenarios distintos (15 %), etc. Luego, utilizando esta taxonomía crea un metamodelo para escribir tests a través de ejemplos (o tests de alto nivel) a través de composiciones de tests más pequeños y de bajo nivel. Este modelo facilita la detección de errores ya que al componer tests, un bug en el código base sólo rompería uno de los tests de los que conforman el test ejecutado. Gaelli en su trabajo pretende promover la reutilización de tests de bajo nivel para crear ejemplos de funcionamiento, o tests de alto nivel. De esta manera, generar ejemplos que sirven tanto como pruebas de software y documentación ejecutable [10], acercando la brecha entre las unidades de software y las pruebas unitarias.

3.2.2. Revisión de herramientas de cobertura de test existentes

La gran cantidad de herramientas disponibles ilustra la importancia de la cobertura de tests para los profesionales de la industria. Se realizó una revisión de muchos de ellos incluyendo Emma, EclEmma, JCover, GroboCodeCoverage, JCoverage, Parasoft JTest, Purity Plus, Semantic Designs, TCAT, Quilt, NoUnit, InsectJ, Hansel, Gretel, Jester, JVMDI Code Coverage, JBlanket, Coverlipse, Koalog, Cobertura, Mojo y Thucydides. En el Apéndice A se presenta la lista con las URL donde se puede obtener cada una de ellas. Esta recolección de las herramientas más populares se realizó a través de motores de búsqueda en la web, directorios de proyectos open source, descripciones de entornos de desarrollo y publicaciones científicas entre los que están: maven[5], sourceforge[6], github[7], bitbucket[8] y el Eclipse marketplace[9].

Dentro de las 22 herramientas de cobertura de test estudiadas, solo 3 soportan un análisis comparativo de cobertura de test.

JCover indica las variaciones de cobertura mostrando un gráfico tipo *scatterplot* puntos representando los pares *(version del software, % cobertura)*. Al parecer TCAT compara tests, sin embargo no hay suficiente información publicada, de hecho, su sitio web no ha sido actualizado desde el año 2008. Jester realiza mutación a los tests para verificar cuán capaz es el unit test de capturar variaciones en el código base.

[5]Apache Maven Project [en línea] <`http://maven.apache.org/`>[Consulta: 03/11/2013]
[6]Sourceforge [en línea] <`http://sourceforge.net/`>[Consulta: 03/11/2013]
[7]GitHub [en línea] <`http://github.com/`>[Consulta: 03/11/2013]
[8]BitBucket [en línea] <`http://bitbucket.org`>[Consulta: 03/11/2013]
[9]Eclipse Marketplace [en línea] <`http://marketplace.eclipse.org/`>[Consulta: 03/11/2013]

Capítulo 4

Investigación Previa

En esta sección se presentan dos investigaciones cuyo propósito es entender mejor el problema para una estrategia y diseño apropiado de la solución. Luego de la revisión de la literatura y el estado del arte de la industria, no se encontraron estudios afines ni herramientas que consideraran esta problemática.

4.1. Experimento: ¿Dos códigos de test parecidos prueban lo mismo?

Uno de los primeras actividades al enfrentar el problema de modularidad de los tests es mirar el código fuente. Luego de una breve y ligera inspección del código fuente podemos encontrar varios tests similares que difieren en pocas líneas lo cual motiva distintas estrategias de refactorización [26, 24, 17]. Sin embargo, surge la siguiente interrogante, ¿En qué afecta esto en la cobertura de estos tests? Es más, la pregunta motiva ir un paso atrás: si estos dos tests son similares en su código, ¿cubren los mismos métodos?.

De esta pregunta se desprenden dos enfoques para analizar similitud de test: estático y dinámico. En el análisis estático determinando similitud en el código fuente, y en el dinámico observando qué tan similar es su cobertura [16] lo cual solo se puede determinar ejecutándolo. Para responder esta pregunta se realizó un experimento que compara la correlación entre dos métricas que apuntan a comparar las pruebas de software desde estos dos aspectos.

4.1.1. Metodología

El experimento consta entonces de verificar si existe o no correlación entre la similitud de código (o *estática*) y la similitud de ejecución (o *dinámica*). Para ello se escogió dos métricas de similitud, una por cada acercamiento al problema, y se realizó comparaciones entre todos los tests de todos los unit tests de un software. Este experimento se realizó sobre los unit tests de 2 softwares con propósitos muy distintos. Estos son:

1. *Roassal*[1]: Motor de visualización ágil.
2. *Fuel*[2]: Framework de serialización de objetos desarrollado en Pharo.

El experimento fue implementado y ejecutado en la plataforma Pharo. Además las aplicaciones previamente descritas fueron seleccionadas pues ya han sido utilizadas en trabajos previos [3].

4.1.1.1. Métricas de similitud

Las métricas de similitud escogidas fueron normalizadas para obtener valores entre $[0, 1]$. Un valor muy cercano a 0 significa que los elementos son muy diferentes, y por el contrario, un valor muy cercano a 1 indica que los elementos son muy parecidos.

La *métrica de similitud estática*, $f(t_a, t_b)$, compara el código de los test según el número de líneas que tienen en común contra el total de líneas entre ambos (sin contar dos veces las líneas en común). Entre las líneas de un método de test no se contó la primera línea (firma del método) ni sus líneas en blanco. Formalmente, sea L_a el conjunto de líneas que componen el código del test t_a y L_b de t_b respectivamente, f se define como:

$$f(t_a, t_b) = \frac{|L_a \cap L_b|}{|L_a \cup L_b|}, \quad f(t_a, t_b) \in [0, 1]$$

La *métrica de similitud dinámica*, $g(t_a, t_b)$, por su parte, compara la cobertura de cada test según los métodos del código base que fueron testeados. Es decir, los métodos ejecutados en común contra el total de métodos llamados durante la ejecución de ambos métodos. Formalmente, sea T_a el conjunto de métodos llamados durante la ejecución del test t_a y T_b el correspondiente a t_b, g se define como:

$$g(t_a, t_b) = \frac{|T_a \cap T_b|}{|T_a \cup T_b|}, \quad g(t_a, t_b) \in [0, 1]$$

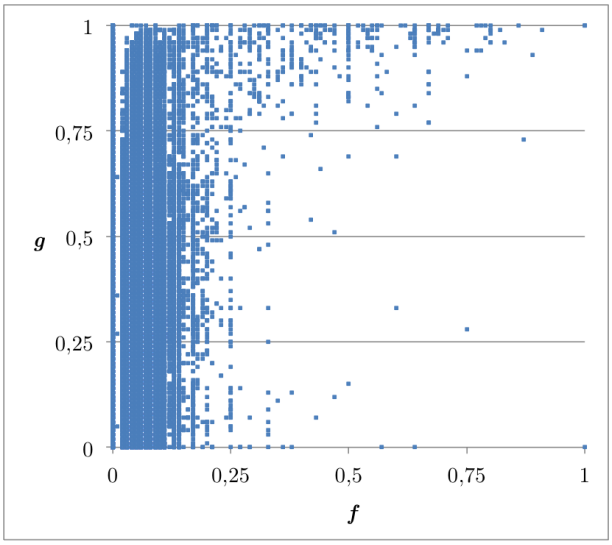
4.1.2. Resultados y Conclusiones

Luego de realizar el experimentos en los dos softwares mencionados, los resultados se presentan en los gráficos de puntos (o scatterplot) de la Figura 4.1, donde cada punto en el gráfico corresponde a un par $(f(t_a, t_b), g(t_a, t_b))$.

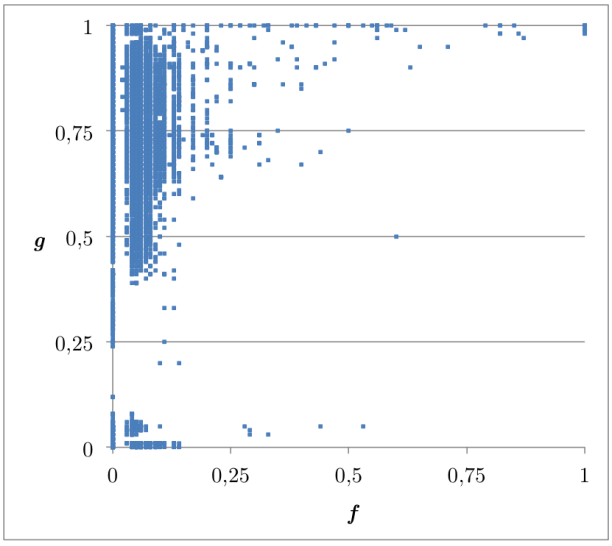
Los gráficos muestran que no existe correlación entre las dos métricas ya que en ninguno de los gráficos se ve alguna forma linear que lo indique. Esto se confirma con los valores del

[1]Para mayor información, ver Sección 7.1

[2]RMoD . Fuel [en línea] <`http://rmod.lille.inria.fr/web/pier/software/Fuel`>[Consulta: 03/11/2013]



(a) Roassal - $\rho_{f,g} = 0,1572$



(b) Fuel - $\rho_{f,g} = 0,1176$

Figura 4.1: Resultados comparación de métricas f y g

coeficiente de correlación de Pearson ($\rho_{f,g}$) de cada caso los cuales son muy cercanos a cero. Esto confirma que no existe correlación lineal entre las dos métricas.

Como consecuencia se puede deducir que dos tests con una gran porción de código duplicado no necesariamente tienen cubren los mismos métodos, o sea, no necesariamente testean la misma porción de código.

Se concluye que la duplicación de código entre tests no puede ser usado para inferir que son semánticamente similares. Por tanto, cualquier análisis comparativo entre tests debe hacerse considerando el código fuente y características de análisis estático, como también datos obtenidos durante su ejecución entre otros aspectos del análisis dinámico.

4.2. ¿Por qué la cobertura no es suficiente para comparar dos tests?

La cobertura de tests [16] intenta determinar qué proporción de una parte de un programa es ejecutada durante el ciclo de pruebas. La cobertura de test se reporta comúnmente como la proporción de paquetes, clases, métodos y líneas de código fuente que fueron ejecutados por los tests. Revisar el código cubierto por los tests es una técnica efectiva para identificar la porción de un software está asegurado por los tests. La medida tradicional de cobertura [22, 23] de test recae en marcar elementos estructurales del software tales como sentencias (o statements), métodos o clases que son cubiertos durante la ejecución.

Como sea la forma de marcar aquellos elementos, es igualmente útil para identificar y medir cual porción del código está siendo cubierta o no por los tests. Sin embargo, para conocer realmente si dos test están intentando testear lo mismo, diferenciarlos por lo su cobertura parece no ser suficiente [29, 14].

Para ilustrar lo anterior, se presenta el código de los tests: `testAddingColoredElement` y `testVisualizingBigClasses` :

```
1  testAddingColoredElement
2    canvas := ROView new.
3    el1 := ROBox green element.
4    el2 := ROBox red element.
5    canvas add: el1; add: el2.
6    self shouldnt: [ canvas open ] raise: Error
7
8  testVisualizingBigClasses
9    canvas := ROView new.
10   Collection withAllSubclasses do: [ :cls |
11     cls numberOfMethods > 10
12       ifTrue: [ el := ROBox red element ]
13       ifFalse: [ el := ROBox green element ].
14     canvas add: el ].
15   self shouldnt: [ canvas open ] raise: Error
```

El código de `testAddingColoredElement` crea un *canvas* (lienzo donde se dibujan los elementos gráficos) para posicionar sobre ésta una caja verde y otra roja. Por su parte, en `testVisualizingBigClasses` se crea una visualización que representa algunas clases coloreadas según una condición particular. Cada subclase de `Collection` está asociada a una caja. Si la clase tiene más de 10 métodos, la caja se pinta en rojo, de lo contrario en verde.

Los dos tests tienen un 95 % de métodos testeados en común, el 5 % que los diferencia se debe a algunos cachés que se activan cuando el número de cajas excede cierto número límite (este comportamiento no se muestra en el código presentado arriba).

Aunque exista un solapamiento considerable desde el punto de vista de los métodos que cubren, estos dos tests no deberían ser considerados redundantes pues el escenario mostrado en cada uno tiene su propia relevancia y no son comparables.

Por lo tanto, dos tests que son similares por su cobertura pueden ser semánticamente muy distintos. Esta diferencia semántica no es expresada por la cobertura y para revelarla se hace necesaria mayor información como por ejemplo más datos característicos de su ejecución e información experta por parte de los desarrolladores.

Capítulo 5

Descripción de la Solución

5.1. Comparando la ejecución de tests con *TestSurgeon*

A la luz de los resultados del estudio comparativo entre el acercamiento al problema desde la perspectiva del análisis estático y análisis dinámico de tests presentado en el Sección 4.1, se desprende que los tests deben ser analizados en base a lo que realmente ejecutan. Es por esto que se hace necesario que cualquier solución que apunte a abordar este problema, considere fuertemente un enfoque dinámico para el diagnóstico de la situación. Y también contrastarlo con el código fuente y otros aprontes de análisis estático para tener una visión más completa tanto del problema como de las alternativas de solución.

En respuesta a este escenario, se presenta *TestSurgeon*: un profiler para test unitarios. TestSurgeon realiza un monitoreo de la ejecución de test unitarios y recolecta datos sobre *qué* está siendo testeado (cobertura) y *cómo* (contexto de ejecución). TestSurgeon aproxima la similitud entre métodos de tests a través de una representación visual que relaciona el código testeado y métricas representativas que caracterizan (y diferencian) a los dos tests en comparación.

El objetivo de TestSurgeon es apoyar a los ingenieros de software a reestructurar y refactorizar sus tests unitarios a través de un análisis sobre las diferencias de ejecución de los tests.

5.1.1. Caso de uso

A modo de resumen, para usar TestSurgeon se siguen las siguientes etapas: primero se lanza TestSurgeon sobre conjunto de paquetes, inmediatamente después se realiza el profiling sobre los paquetes seleccionados. Una vez finalizada esta etapa, se abre una ventana con el browser (o navegador) donde se escogen dos unit tests (no necesariamente distintos), posteriormente se seleccionan dos métodos de test pertenecientes a los unit tests escogidos. Finalmente se realiza la comparación entre los test methods seleccionados a través de la visualización y la

zona de comparación de código. En TestSurgeon se realizan comparaciones de métodos de test uno-a-uno por lo cual, para diferenciarlos, se denominan *test rojo* y *test azul* en concordancia a los colores en la visualización.

Para ilustrar el uso de TestSurgeon se muestra a continuación un ejemplo sobre el software: *Roassal*.

Un análisis con *TestSurgeon* comienza lanzando el software desde una terminal de Pharo o *Workspace* con el siguiente comando[1]: `TSBrowser openOn: 'Roassal*'`. Este comando abre el browser y hace que se instrumenten todos los paquetes cuyo nombre comience con `Roassal` (ej: `Roassal-Core`, `Roassal-Layout` o `Roassal-Normalizers`, entre otros).

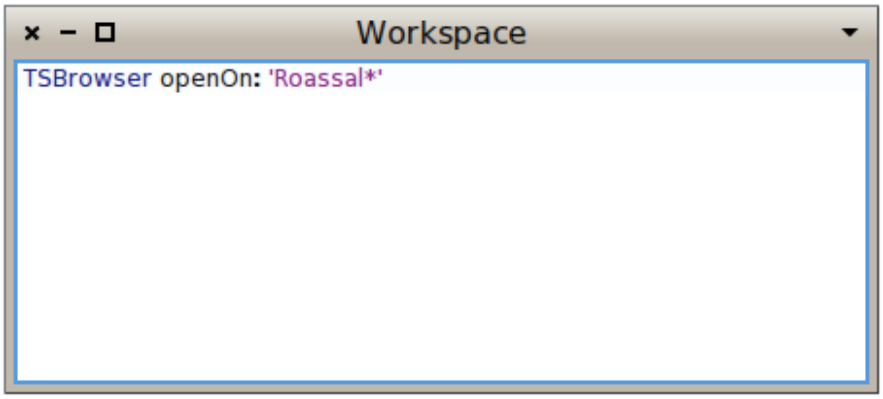


Figura 5.1: Lanzando el browser.

Luego, TestSurgeon realiza un profiling sobre la ejecución de los tests encontrados en todos los paquetes de Roassal. Esto consiste en: ejecutarlos, recolectar datos y generar los objetos según el modelo de objetos de Spy(ver Sección 6.2). Este proceso puede tardar desde unos segundos a algunos pocos minutos según la cantidad de tests encontrados y las operaciones propias dentro de cada test.

Una vez concluida la etapa de profiling, se despliega una ventana con cuatro zonas principales(ver Figura 5.2):

- **Test Unitarios**: En la zona superior-izquierda aparecen dos listas. Se puede seleccionar solo un unit test por cada lista.
- **Métodos de Test**: Justo abajo de los test unitarios aparecen dos listas con los métodos de test de los unit tests seleccionados. En cada lista se pueden seleccionar un método. El método seleccionado de la izquierda se denomina *test rojo* y el de la derecha *test azul*.
- **Visualización**: Luego de seleccionados los tests rojo y azul se despliega una visualización que muestra la diferencia en la ejecución de ambos tests.
- **Código Fuente**: Junto con la visualización se muestra el código fuente de cada test.

Primero se seleccionan los unit tests, uno por lista (Paso 1 y 2). De los unit tests escogidos, se debe seleccionar el test rojo en lista de la izquierda (Paso 3). Una vez seleccionado, la lista

[1]Para más detalles sobre este comando y la implementación del browser ver la Sección 6.4

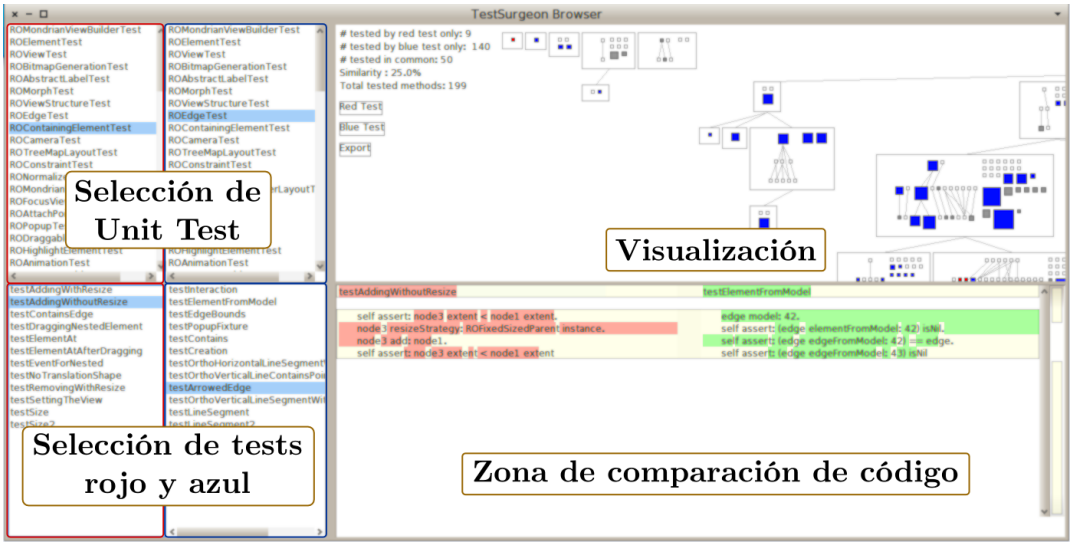


Figura 5.2: Zonas del browser de TestSurgeon.

de métodos de test de la derecha se ordenan verticalmente según similitud de cobertura con respecto al test rojo (más similar arriba, más diferente abajo). La métrica de similitud por cobertura es la misma que se utilizó en el experimento de similitud estática contra similitud dinámica definida en la Sección 4.1.1.1. Esta facilita la búsqueda de casos interesantes a ser comparados.

La cobertura, como métrica, no expresa el cómo los actores (objetos) se comportan (a través de métodos) mientras el test está siendo ejecutado. Por tanto se necesitan más datos [29, 14]. Entonces, una vez seleccionado el test azul (Paso 4) se puede realizar una mejor comparación observando una visualización (Paso 5) que justamente resalta las diferencias en la ejecución de ambos tests. Y finalmente contrastar (Paso 6) lo observado con el código fuente de ambos.

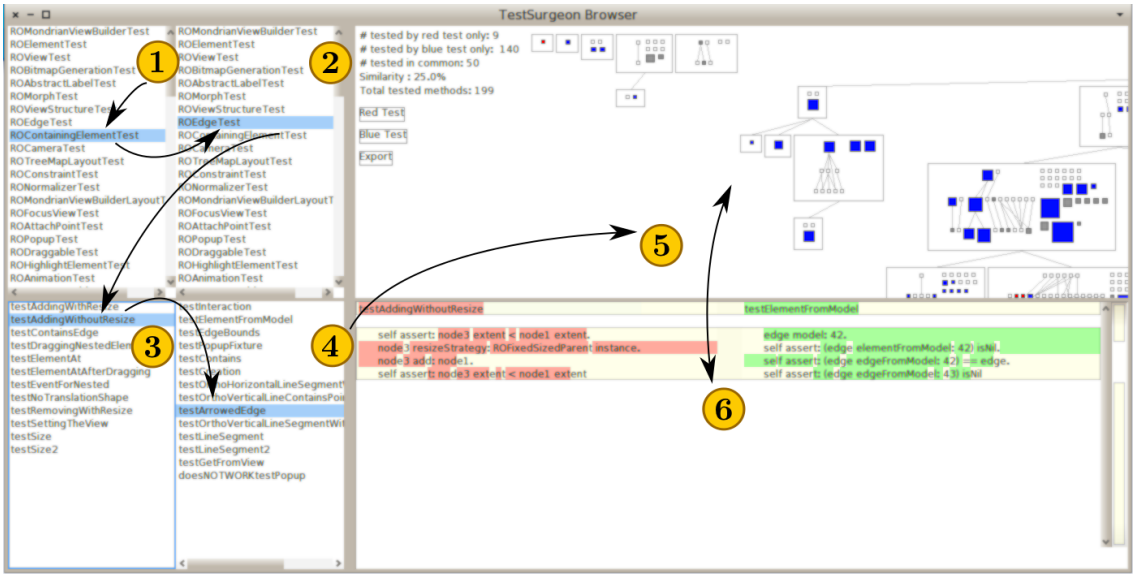


Figura 5.3: Modo de uso del browser.

5.2. Visualización: *Test Difference Blueprint*

El objetivo de la visualización es comparar y encontrar las diferencias de ejecución que existen entre dos métodos de test. Para esto se utilizó intensivamente la gráfica de *vistas polimétricas* [19], una técnica de visualización que permite identificar visualmente patrones en una gran masa de datos multidimensionales. Primero que todo, se debe hacer una comparación de la cobertura, es decir cuáles son los métodos y las clases testeadas por los métodos en comparación. Los elementos a visualizar, entonces, son las clases y sus métodos.

5.2.1. Clases y su jerarquía

En la visualización se muestran todas las clases del código base que estén cubiertas por al menos uno de los dos tests en comparación. Las clases se presentan visualmente como grandes rectángulos con fondo blanco y borde negro. Estos rectángulos están unidos por líneas que hacen gráfica la relación de herencia entre estas. Como Pharo es un lenguaje orientado a objetos con herencia simple (cada clase puede heredar de una sola clase padre) desde una clase hija llega solo una línea, pero desde una clase padre pueden salir varias líneas a las clases que heredan directamente de ésta. TestSurgeon muestra entonces la jerarquía de clases a través de una distribución espacial de las clases, en forma de árbol (o *tree layout*). De esta manera, la clase raíz (o la hereda de una clase que no está presente en los paquetes instrumentados) está en el tope superior de la imagen y en el siguiente nivel del árbol están las clases que heredan directamente de ésta, y así sucesivamente, formando un árbol.

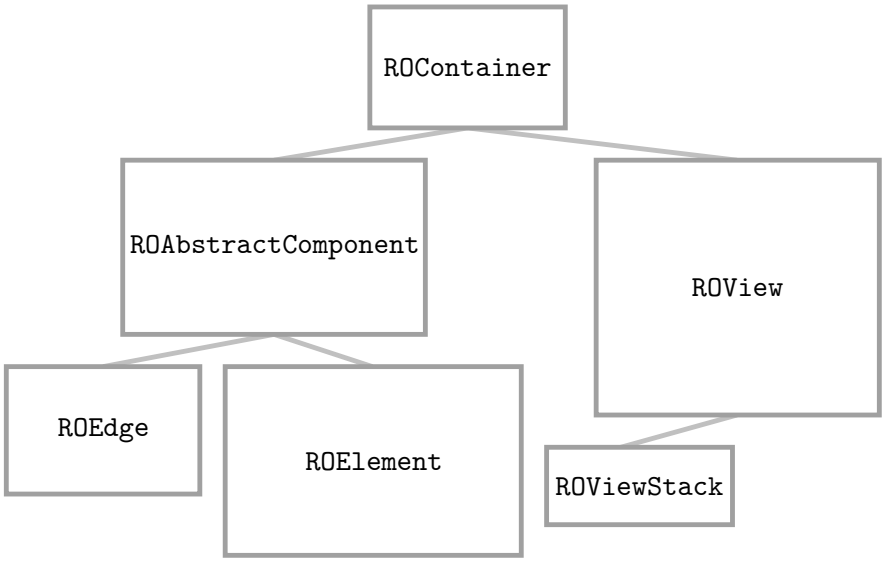


Figura 5.4: Jerarquía de la clase `ROContainer`

5.2.2. Coloreando métodos para visualizar cobertura

Los elementos principales en el análisis, según el enfoque de TestSurgeon, son los métodos de las clases del código base. Los métodos están representados como rectángulos (o cuadrados) de menor tamaño posicionados dentro de la clase que lo define. Los métodos heredados por la clase no están considerados en la visualización de dicha clase, sino que aparecen en la clase que lo define dentro de la jerarquía.

Cada método tiene un color asignado:

- **Rojo**: el método fue llamado únicamente durante la ejecución del test rojo.
- **Azul**: el método fue llamado únicamente durante la ejecución del test azul.
- **Gris**: el método fue ejecutado durante la ejecución de ambos tests.
- **Blanco**: el método no fue ejecutado.

Un aspecto importante dentro del análisis de cobertura es conocer los caminos de ejecución según las llamadas a métodos de una clase desde otro método definido en la misma clase. Estas llamadas son invocaciones a `self` (o `this` en Java) que permiten diferenciar si un método fue llamado directamente o a través de una invocación `self` desde otro método. Para graficar esto se utilizó una distribución gráfica de Sugiyama [28] (o Sugiyama Layout). En el tope superior están los métodos que realizan mayores invocaciones a otro métodos definidos en la clase y que además no son invocados por otros métodos de la clase. En el tope inferior los métodos que no tienen invocaciones a otros métodos definidos en dicha clase pero sí son invocados por uno o más métodos definidos en la clase. Las invocaciones entre métodos están graficados por líneas rectas entre el método invocador y el invocado.

En una distribución de grilla y posicionados a la derecha dentro del rectángulo de la clase, están los que no tienen invocaciones a otros métodos definidos en dicha clase ni son invocados por métodos definidos en esa clase.

En la Figura 5.5, los métodos `b` y `c` son invocados en el método `a`, es decir, en el cuerpo del método `a` están las líneas `self b` y `self c`. Se puede apreciar que el test rojo ejecuta `a`, `b` y `c`, y el test azul `b` solamente.

5.2.3. Métricas para caracterizar la ejecución de los métodos

Hasta ahora, la visualización sólo grafica lo que ha sido ejecutado por uno, otro u ambos tests, es decir, la cobertura. Sin embargo este acercamiento no parece ser tan preciso ni suficiente[2] para tener una idea real de cómo se ejecutaron dichos métodos. Allí es donde emerge la característica más distintiva e innovadora de la visualización de TestSurgeon: incorporar métricas que dan un contexto en el cual un método fue llamado durante la ejecución de los tests en comparación. Así no tan solo se visualiza cuáles fueron los métodos ejecutados por el test rojo o azul (o ambos) y cuáles no, sino que también muestra de qué manera tal método

[2]En el Sección 4.2 se explica a través de un ejemplo que efectivamente es necesaria más información que la cobertura para realizar una comparación efectiva.

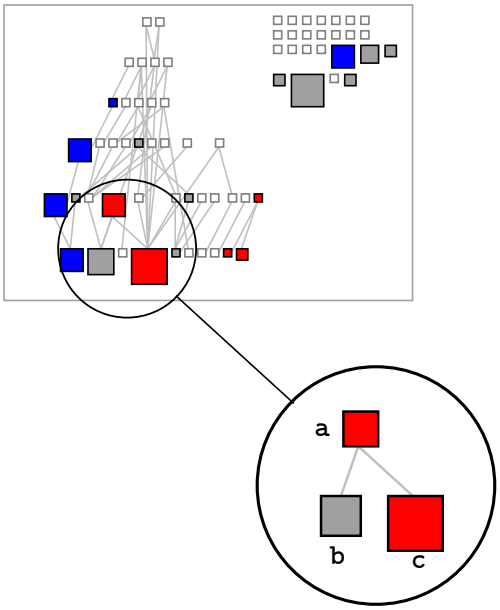


Figura 5.5: Ejemplo de Sugiyama Layout

fue ejecutado y cuán distinta fue su ejecución entre ambos tests.

El *Número de Llamadas* a un método (`NOC` - Number of Calls, en inglés) es una de las dos métricas escogidas para esto. Tal como su nombre lo indica, representa la cantidad de veces que un método fue llamado para su ejecución. De esta manera, entre más veces fue llamado, más relevante es para ese test.

Por su parte, el *Número de distintos objetos recibidores* de un método (`NODR` - Number of Different Receivers, en inglés) indica cuántos objetos distintos ejecutaron dicho método. Esta métrica es particularmente útil cuando su `NOC` es parecido ya que su `NODR` puede ser distinto y así diferencia las ejecuciones del test rojo y azul ofreciendo una perspectiva ortogonal en el análisis.

A modo de ejemplo, supongamos que un cierto método m_a fue llamado la misma cantidad de veces durante la ejecución del test rojo y el azul, entonces: $\texttt{NOC}(m_a, t_r) = \texttt{NOC}(m_a, t_b) = k$. Y consideremos que $\texttt{NODR}(m_a, t_r) = 1$ y $\texttt{NODR}(m_a, t_b) = k$. Durante la ejecución del test rojo, hubo solo una instancia de la clase que define m_a y recibió k veces dicho mensaje, un escenario muy distinto al del test azul, donde hubo k instancias por lo cual, cada una ejecutó m_a sólo una vez.

Dado que para un método tenemos dos métricas por cada test, necesitamos 4 dimensiones para graficarlo: $\texttt{NOC}(m_a, t_r)$, $\texttt{NOC}(m_a, t_b)$, $\texttt{NODR}(m_a, t_r)$ y $\texttt{NODR}(m_a, t_b)$. Ya que los métodos están gráficamente representados por rectángulos, se decidió que la medida de cada par de lados opuestos será la diferencia entre la métrica para cada test. Entonces, la altura de un rectángulo es la diferencia entre el número de llamados de dicho método durante la ejecución del test rojo y azul. Y su ancho es la diferencia entre el número de distintos objetos recibidores del método durante la ejecución de los tests en comparación.

Formalmente:

$$\text{altura}(m_a) = |\texttt{NOC}(m_a, t_r) - \texttt{NOC}(m_a, t_b)|$$

$$\text{ancho}(m_a) = |\texttt{NODR}(m_a, t_r) - \texttt{NODR}(m_a, t_b)|$$

Durante la implementación se vio que para ejecuciones muy distintas, los valores de las métricas de diferencia podrían ser muy grandes, generando enormes cuadrados que ocupan demasiado espacio en la gráfica. Para corregir esto se optó por ajustar las medidas utilizando una escala logarítmica. Esta decisión si bien afecta la precisión de los valores, no lo hace en forma significativa ni cambia la información a presentada, sin embargo mejora considerablemente la lectura facilitando la interpretación de la gráfica.

En caso que se necesite aún más detalle y precisión en las métricas de un método, es posible observar los valores exactos de estas al posicionar el cursor sobre un método. Al cabo de unos segundos aparece una ventana emergente (también conocidas como pop-up) con los valores respectivos de cada métrica, en rojo los correspondientes a ese método y el test rojo, y en azul las métricas correspondientes al test azul y el método seleccionado.

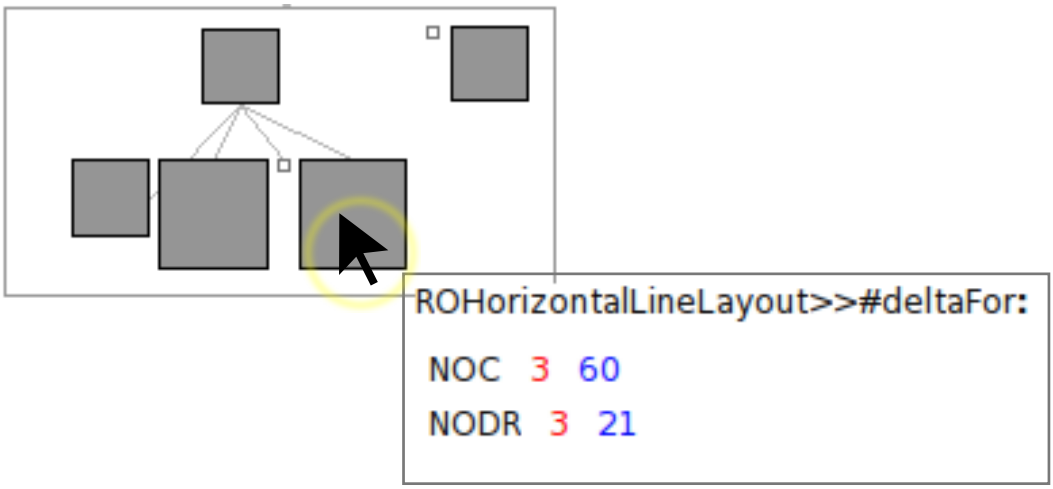


Figura 5.6: Ventana emergente con detalles de las métricas para el método `deltaFor:`.

5.3. Resumen: Interpretando la visualización

La *Test difference blueprint* facilita la comprensión de la diferencia de ejecuciones entre dos test methods diferenciándolas sobre lo que efectivamente testearon y contrastando la forma en que lo hicieron. Los colores ayudan a percibir características de cobertura, por lo que una alta porción de métodos grises significa que su cobertura es similar, es decir, en su ejecución hay varios métodos en común que se llamaron, lo cual los hace más similares.

La forma y tamaño de los métodos en la visualización habla de cómo fueron ejecutados los métodos y las métricas que describen esto son: cuantas veces fue llamado (altura) y cuantos objetos distintos lo ejecutaron (ancho). Dado que la visualización se enfoca en diferencias y

se están comparando dos tests, el ancho y alto de un método es la diferencia entre dichas métricas para cada ejecución de test. Entonces, si un método gris se ve pequeño en tamaño quiere decir que no hubo mucha diferencia entre cuantas veces fue ejecutado por el test rojo y azul, y la cantidad de instancias que ejecutaron ese método en ambos test fue similar.

En caso contrario, si un método gris es grande, quiere decir que ese método es particularmente relevante para uno de los dos tests ya que se ejecuta muchas más veces en uno de los dos casos o es ejecutado por una cantidad mucho mayor de instancias, o bien ambos casos.

Ahora, considerando los métodos de color rojo o azul, estos se interpretan en una forma ligeramente diferente. Primero, es importante considerar su proporción con respecto al total de métodos ejecutados. Si existe gran presencia de métodos rojos y/o azules estamos en un caso de dos tests con cobertura bastante distinta. Por lo tanto, su ejecución se enfoca en características distintas del software. Ahora bien, para casos en que la cobertura es similar (pocos métodos rojos y/o azules), el siguiente aspecto a observar es su forma. Si estos métodos son grandes, quiere decir que son muy relevantes para el test correspondiente porque fue ejecutado muchas veces o bien porque varios objetos de los creados en la ejecución, recibieron ese mensaje.

Recapitulando, y a grandes rasgos, se puede decir que la ejecución de dos test es similar si la visualización presenta una gran cantidad de métodos grises pequeños y algunos pocos métodos rojos y/o azules. Por el contrario, dos tests habrán tenido una ejecución muy diferente si se aprecian métodos grandes y con gran presencia de azules y rojos.

A continuación se presenta un resumen de la visualización:

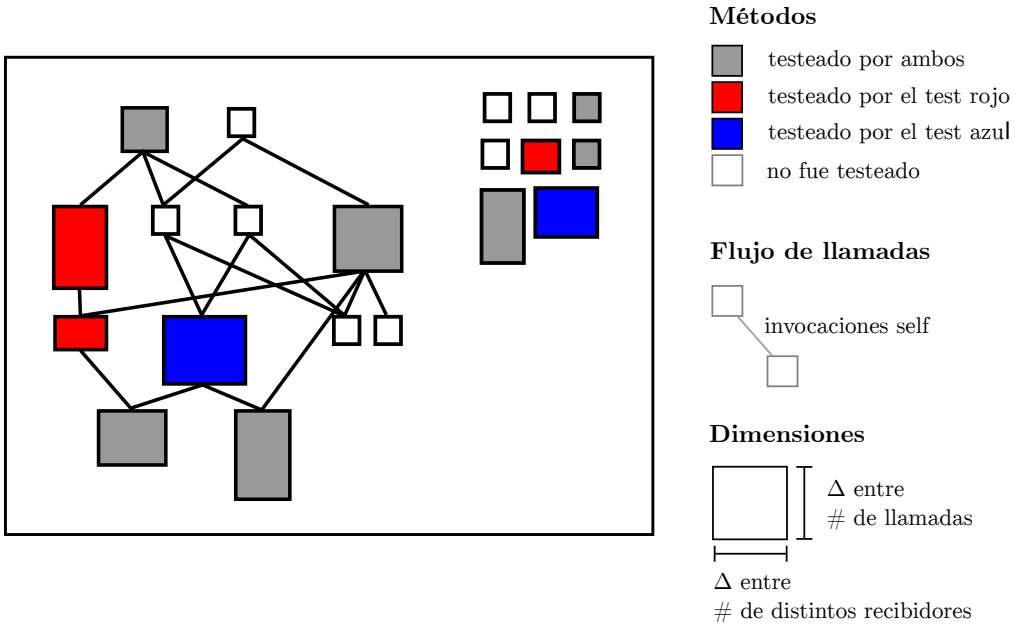


Figura 5.7: Test Difference Blueprint

Capítulo 6

Implementación

6.1. Arquitectura

El proyecto TestSurgeon fue completamente implementado en el lenguaje Pharo Smalltalk[1] ya que es en lenguaje en el que está escrito el framework de profiling utilizado (componente principal del proyecto). Pharo es un lenguaje de programación que implementa fielmente el paradigma de orientación a objetos. Sus características principales son, primero: entorno de desarrollo robusto totalmente integrado al lenguaje, y segundo: el sistema sigue una orientación a objetos pura, es decir, todo en Pharo es un objeto: Strings, Números, Clases, Métodos, Colecciones, Paquetes, etc.

Para ejecutar Pharo, es necesario utilizar una máquina virtual la cual es específica de cada sistema operativo y actúa como interfaz de este último. La imagen de Pharo es una "fotografía" (o *snapshot*) del sistema Pharo congelado en el tiempo, específicamente de la última vez que se ejecutó. Contiene todas las instancias creadas y en el estado en que estaban justo antes de que la máquina virtual se cerrara.

Sobre Pharo están: *Spy*, *Roassal* y *Spec*, las principales dependencias de TestSurgeon. *Spy* es un framework de profiling [4] usado en TestSurgeon, se comentará en detalle sobre éste en la Sección 6.2. *Roassal* es un software que permite desarrollar visualizaciones interactivas en forma fácil y rápida. Es usado por TestSurgeon para implementar la *Test Difference blueprint* (ver Sección 5.2), en la Sección 6.3 se hablará de esta. Finalmente *Spec* es un framework para construir interfaces gráficas y fue usado para la implementación del navegador de tests, ver Sección 6.4.

[1] Pharo Project [en línea] <`http://www.pharoproject.org`>[Consulta: 03/11/2013]

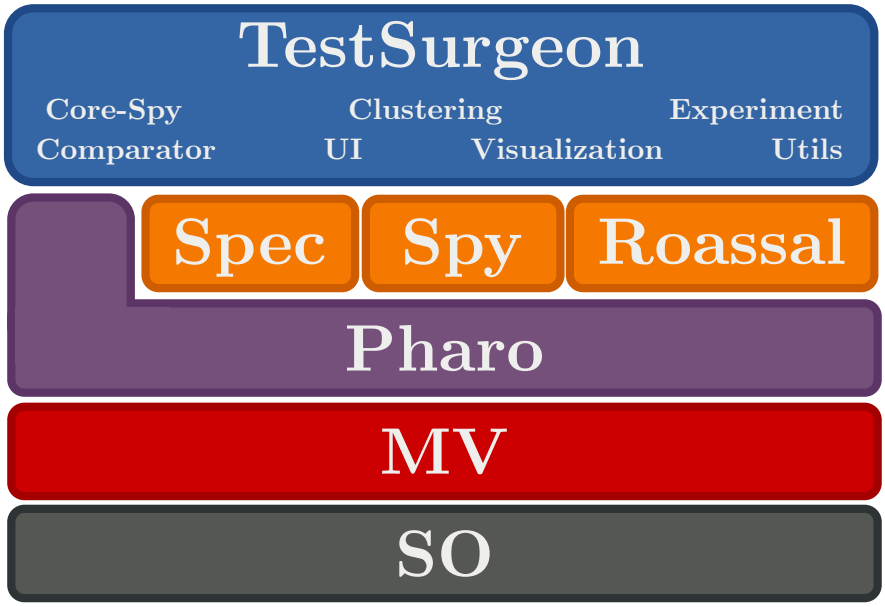


Figura 6.1: Arquitectura de *TestSurgeon*

6.2. Profiling

La componente principal de TestSurgeon es la que realiza el profiling de la ejecución de los unit tests. 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.

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 abstractos y deben concretarse en las aplicaciones que decidan usar Spy extendiendo sus clases.

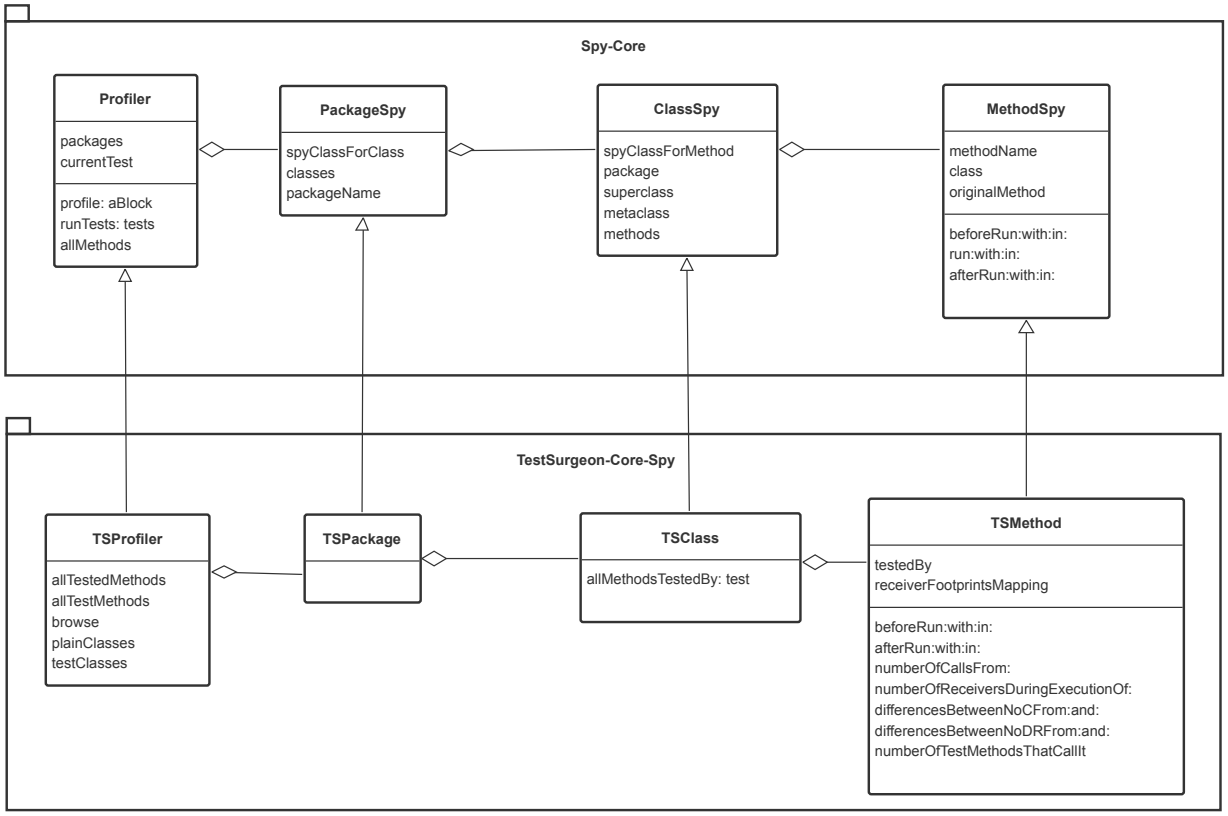


Figura 6.2: Diagrama de clases de `Spy-Core` y `TestSurgeon-Core-Spy`

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 esto que para representar un método de test se tiene también la clase `TSTestMethod` que encapsula una instancia de `TSMethod` correspondiente a un test method.

En el resto de la sección se describe en detalle las clases que componen al paquete `TestSurgeon-Core-Spy`.

6.2.1. 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 obtenida 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:

Selector	Resultado
`allTestMethods`	retorna una colección de todos los tests ejecutados (instancias de `TSMethod`
`allTestedMethods`	retorna una colección de todos los métodos testeados (instancias de `TSMethod`
`compiledToSpyMethod:`	encapsula un método compilado (instancia de `CompiledMethod`) retornándolo como instancia de `TSMethod` .
`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.

Tabla 6.1: API pública de las instancias de `TSProfiler`

6.2.2. Las clases `TSPackage` y `TSClass`

Estas dos clases, 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 la implementación de `TSPackage`.

`TSClass` representa tanto a las clases testeadas como a los 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.

6.2.3. La 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. Cada instancia de `TSMethod` representa un método ejecutado, es decir, tanto el test que comienza la ejecución como todos los métodos llamados durante esta. Los datos son almacenados en variables de instancia, y existe solo una instancia por método definido en el software a analizar, entonces, un método ejecutado tendrá la información sobre su ejecución en distintos tests donde haya sido llamado. Las variables de instancia se definen a continuación:

Selector	Resultado
`testedBy`	Diccionario que asocia cada test con las veces que el método fue llamado por éste durante la ejecución del primero
`originalMethod`	instancia de `CompiledMethod` correspondiente al método encapsulado.
`receiverFootprintsMapping`	Diccionario que asocia un test con todos los objetos *receivers* del método actual durante su ejecución.

Tabla 6.2: Atributos principales de `TSMethod`

Su comportamiento está dividido en dos partes principalmente: los métodos de profiling donde se registran los datos de ejecución, y sus métodos accesores: métodos que entregan la información obtenida. Los métodos de profiling son dos: `beforeRun: with: in:` y `aftefRun: with: in:`, a continuación se muestra el código del primero.

```
1  beforeRun: methodName with: listOfArguments in: receiver
2    | testMethod |
3    " Test en ejecucion "
4    testMethod := self profiler class currentTestMethod.
5    currentTestMethodSpy :=
6          spyMethodMapping at: testMethod ifAbsentPut:
7            [ self profiler >> testMethod class name >> testMethod selector ].
8
9    " Guardar en el diccionario de tests"
10   self addTestMethod: currentTestMethodSpy.
11
12   " Guardar receiver "
13   self updateReceiversWith: receiver for: currentTestMethodSpy.
```

En el código de `beforeRun:with:in:`, una instancia de `TSMethod` lo recibe. Lo primero que realiza es obtener el test que originó la ejecución. Para esto, realiza un llamado al profiler para solicitar el test que se está ejecutando en ese momento. El test se agrega al diccionario `spyMethodMapping` que representa una biyección entre los tests como `CompiledMethod` y su método Spy que lo encapsula.

Posteriormente, el test encapsulado se agrega como llave al diccionario `testedBy` (diccionario de tests que cada método del código base posee y contabiliza el número de llamados durante cada test) y se incrementa en 1 el contador. Además, es necesario identificar el objeto recibidor del mensaje (objeto que ejecuta dicho método). Para esto cada método tiene un diccionario llamado `receiverFootprintsMapping` donde se asocia el test con una colección de objetos que recibieron el mensaje durante su ejecución. Para esto, cada objeto va anotado por su valor de hash que lo representa en forma única e individual.

Las instancias de `TSMethod` son entonces las principales fuentes de datos para la creación de métricas y visualizaciones. Para esto cuenta una API pública que se detalla a continuación:

Selector	Resultado
`numberOfCallsFrom:`	Retorna el valor de la métrica *Number of Calls* (o `NOC`, ver Sección 5.2.3).
`numberOfReceiversDuringExecutionOf:`	Retorna el valor de la métrica *Number of Different Receivers* (o `NODR`, ver Sección 5.2.3).
`differencesBetweenNoCFrom:and:`	Retorna la diferencia entre la métrica `NOC` entre dos tests.
`differencesBetweenNoDRFrom:and:`	Retorna la diferencia entre la métrica `NODR` entre dos tests.
`numberOfTestMethodsThatCallIt`	Retorna el número de tests en los cuales es ejecutado.
`unitTestThatTestsIt`	Retorna el número total de unit tests donde algún test haya lo ejecutado.

Tabla 6.3: API pública de las instancias de `TSMethod`

6.2.4. La clase `TSTestMethod`

Como se explicó anteriormente, las instancias de `TSMethod` pueden encapsular un método compilado del código base o bien de los tests que inician la ejecución de éstos. La API pública presentada anteriormente está enfocada a consultas para métodos del código base. Para el análisis de los tests es necesario incluirlos también como entidades del modelo de clases. Para eso está `TSTestMethod` que básicamente es un *wrapper* de una instancia `TSMethod` que encapsula un test method.

A continuación se presenta su API pública:

Selector	Resultado
`numberOfTestedMethods`	Retorna el número de métodos testeados.
`testedClasses`	Retorna una colección con las clases (`TSClass`) que contengan métodos testeados.
`testedMethods`	Retorna una colección con los métodos (`TSMethod`) testeados.
`visualizeDifferencesWith:`	Retorna una instancia `TSTestDifferenceBlueprint`, la visualización comparativa con el test entregado como argumento.
`compareWith:`	Retorna una instancia `TSDynCoverageOtoOComp`, un comparador 1-a-1 de test que provee métricas de similitud dinámica.
`staticCompareWith:`	Retorna una instancia `TSStaticOtoOComp`, un comparador 1-a-1 de test que provee métricas de similitud estática.

Tabla 6.4: API pública de las instancias de `TSTestMethod`

6.3. Visualización

La implementación de *Test Difference Blueprint* es la clase `TSTestDifferenceBlueprint`. Para crear la visualización necesita de los datos de los tests a comparar, por lo cual dentro de sus atributos están `redTest` y `blueTest`, instancias de `TSTestMethod`. Además, posee un comparador de similitud dinámica (instancia de `TSDynCoverageOtoOComp`) el cual está almacenado en el atributo `comparator`. Para la información general sobre toda la instrumentación y ejecución es que se necesita una instancia de `TSProfiler` la cual está referenciada por el atributo `profiler`.

La visualización como tal corresponde instancia de `ROView`, que es, en palabras simples, un contenedor de los elementos gráficos de la visualización y es el responsable de mostrarlos apropiadamente. Por otro lado, enviando el mensaje `exportAs:`, el objeto crea un archivo de imagen vectorial (formato `SVG`) con el `String` entregado como parámetro como nombre. Técnicamente no hay más aspectos destacables que no se puedan apreciar en forma directa desde el código fuente. Más información sobre la visualización se puede encontrar en la Sección 5.2.

6.4. Browser

El browser o navegador permite acceder rápidamente a los métodos a través de menús que permiten seleccionar unit tests y los tests para realizar comparaciones rápidamente. Esta interfaz está implementada completamente usando el framework Spec. Gráficamente consiste en cuatro espacios: Selector de Unit Tests, Selector de Test methods, Zona de Visualización y Zona de Código. La disposición de estas zonas es como se muestra en la Figura 5.2. Para lanzar el browser se envía el mensaje `openOn:` a `TSBrowser` donde el argumento corresponde a una expresión regular que calza con el nombre de los paquetes a analizar. Por ejemplo, para analizar los tests de Spec, se debe ejecutar el comando `TSBrowser openOn: 'Spec-*'` porque los paquetes de Spec siguen la regla de nombrarse con `Spec-` como prefijo, y los paquetes instrumentados entonces serán: `Spec-Core`, `Spec-Examples`, `Spec-Layout`, `Spec-Tools`, `Spec-Tools-Editor`, `Spec-Widgets`, entre otros.

Como las comparaciones entre ejecuciones de tests son 1-a-1, necesitamos primero escoger dos unit tests, uno para los candidatos a test rojo y otro para los candidatos azules. Cada uno de estos menús de selección única son una instancia `IconListModel` y muestran una lista de los unit tests ordenados por tamaño (cantidad de tests) de mayor a menor. Una vez seleccionados ambos unit tests, se despliegan los menús inferiores, los selectores de test methods. Estos, también son objetos `IconListModel` pero con algunas diferencias: el menú para escoger test rojo ordena los tests en orden alfabético; por su parte el menú para test azul inicialmente está en orden alfabético pero una vez se haya seleccionado un test azul se ordena según similitud dinámica descendente con respecto al test rojo.

Cuando hay un test rojo y azul seleccionados, se refresca la visualización para mostrar la comparación de los elementos seleccionados y la zona de comparación de código. Sobre la

última, para ahorrar tiempo, se reutilizó esta componente (`DiffMorph`) desde el browser de administración y descarga de paquetes(*Monticello Browser*[2]). El comparador de código, éste compara línea a línea el código y marca de color (rojo y verde, no sigue la convención de TestSurgeon) las diferencias entre estos.

6.5. Clustering y Comparación de tests

En la Sección 7.3 se presenta el problema de la gran cantidad de posibles comparaciones que el desarrollador debe realizar usando el browser para detectar posibles *test smells*. Es por eso que se realizó un experimento agrupando los tests del unit test `ROMondrianViewBuilderTest` según los métodos que cubren sus test methods. Así crear clusters de tests similares y ahorrar una gran cantidad de comparaciones innecesarias.

6.5.1. Adaptación del Algoritmo Agrupamiento Aglomerativo Jerárquico

Para esto, se decidió utilizar el *Algoritmo de Agrupamiento Aglomerativo Jerárquico* [6, 27] (o *Hierarchical Agglomerative Clustering*). Este algoritmo pretende crear una jerarquía de clusters a través de fusionar clusters más similares (o cercanos) en cada paso. Al comienzo, existen N clusters de tamaño 1, es decir. Luego de calcular todas las distancias entre estos se fusionan los clusters con menor distancia, y ahora existen $N-1$ clusters: $N-2$ de tamaño 1 y el recién fusionado con 2 elementos. El algoritmo sigue hasta que se tiene 1 solo cluster de tamaño N. Las implementaciones comunes del algoritmo reciben de entrada un número k que corresponde al número de clusters deseado, entonces, el algoritmo cuenta con una condición de término que corresponde a finalizar la ejecución cuando la última fusión realizada haya generado en total: k clusters.

Las restricciones del problema a resolver es obtener una cantidad "razonable" de clusters (ver Sección C.1) que reduzcan la cantidad de comparaciones. Sin embargo, de poco sirve generar clusters muy heterogéneos ya que así se incluirían más casos de comparaciones innecesarias. Entonces no hay forma de determinar cuál k es más apropiado para este caso. Para esto se adaptó el algoritmo modificando la condición de término transformándolo en el siguiente invariante: el algoritmo debe en cada iteración generar clusters con una similitud interna mayor o igual a s_{min}. Es decir, una vez que los clusters fusionados formen uno con similitud interna s' tal que $s' < s_{min}$, dicho cluster debe deshacerse, y el algoritmo retorna la última colección de clusters que respetó el invariante. La similitud interna de un cluster se midió como el promedio de las similitudes de cobertura entre todos sus elementos, formalmente:

[2]Dynamic Web Development with Seaside . Saving your Package to Monticello [en línea] <`http://book.seaside.st/book/getting-started/pharo/monticello` >[Consulta: 03/11/2013]

$$\text{similitud\_interna}(C_k) = \frac{1}{|C_k| \cdot (|C_k| - 1)} \sum_{\text{i}=0}^{|C_k|} \sum_{j=0}^{|C_k|-1} g(C_{k_\text{i}}, C_{k_j})$$

Donde $g(C_{k_\text{i}}, C_{k_j})$ es la medida de similitud por cobertura entre el test $t_\text{i}, t_j \in C_k$, definida en la Sección 4.1.1.1. La experiencia en el uso del browser indica que dos tests con similitud sobre 90 % son relevantes y vale la pena analizarlos. Entonces para descubrir la similitud interna mínima más apropiada se realizó un experimento ejecutando el algoritmo para distintos valores de esta métrica: 95 %, 90 %, 85 %, 80 % y 75 %.

Otro aspecto relevante fue la elección de la métrica de similitud entre dos clusters. Entre las alternativas se decidió implementar tres distancias y evaluarlas en un experimento, estas son:

Single Link La distancia entre dos clusters corresponde a la distancia entre los dos elementos (uno de cada cluster) más cercanos. En términos de la métrica de similitud por cobertura g, la distancia se define como: dados C_p y C_q clusters,

$$D(C_p, C_q) = \max_{x \in C_p,\, y \in C_q} g(x, y)$$

Complete Link La distancia entre dos clusters corresponde a la distancia entre los dos elementos (uno de cada cluster) más lejanos. En términos de la métrica de similitud por cobertura g, la distancia se define como: dados C_p y C_q clusters,

$$D(C_p, C_q) = \min_{x \in C_p,\, y \in C_q} g(x, y)$$

Average Link La distancia entre dos clusters corresponde al promedio total de distancias entre todos los elementos de ambos clusters. En términos de la métrica de similitud por cobertura g, la distancia se define como: dados C_p y C_q clusters,

$$D(C_p, C_q) = \frac{1}{|C_p| \cdot |C_q|} \sum_{x \in C_p} \sum_{y \in C_q} g(x, y)$$

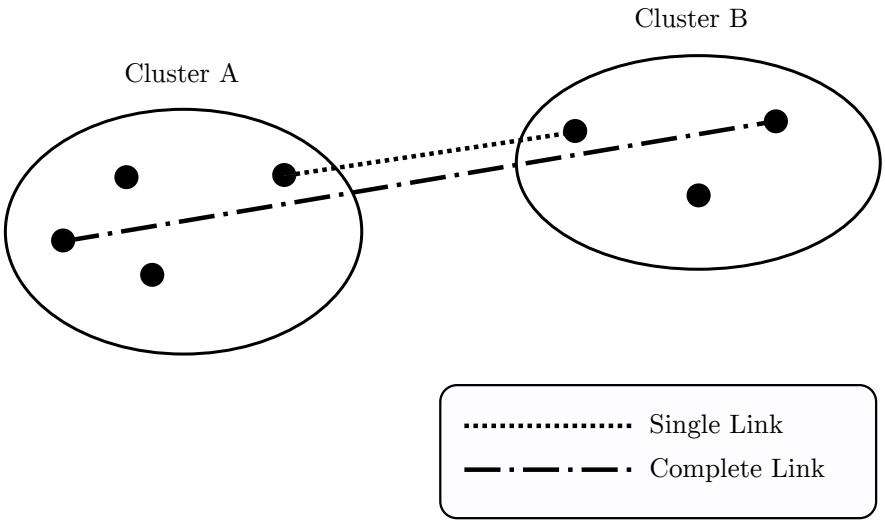


Figura 6.3: Distancias mínima (Single Link) y máxima (Complete Link) entre dos clusters

Finalmente el experimento de agrupamiento consistió en ejecutar el algoritmo de agrupación aglomerativo jerárquico para cada distancia y cada similitud interna mínima, con lo cual se obtuvo 15 particiones distintas de `ROMondrianViewBuilderTest`. Luego de evaluados los resultados (ver Apéndice C) se obtuvo que la mejor configuración corresponde a imponer una **similitud interna mínima** de **85 %** y usando la métrica de distancia **Complete Link**.

6.5.2. Diagrama de Clases

La implementación del algoritmo y el experimento descritos anteriormente consiste en 3 paquetes principales: `TestSurgeon-Clustering`, `TestSurgeon-Comparator` y `TestSurgeon-Experiments`.

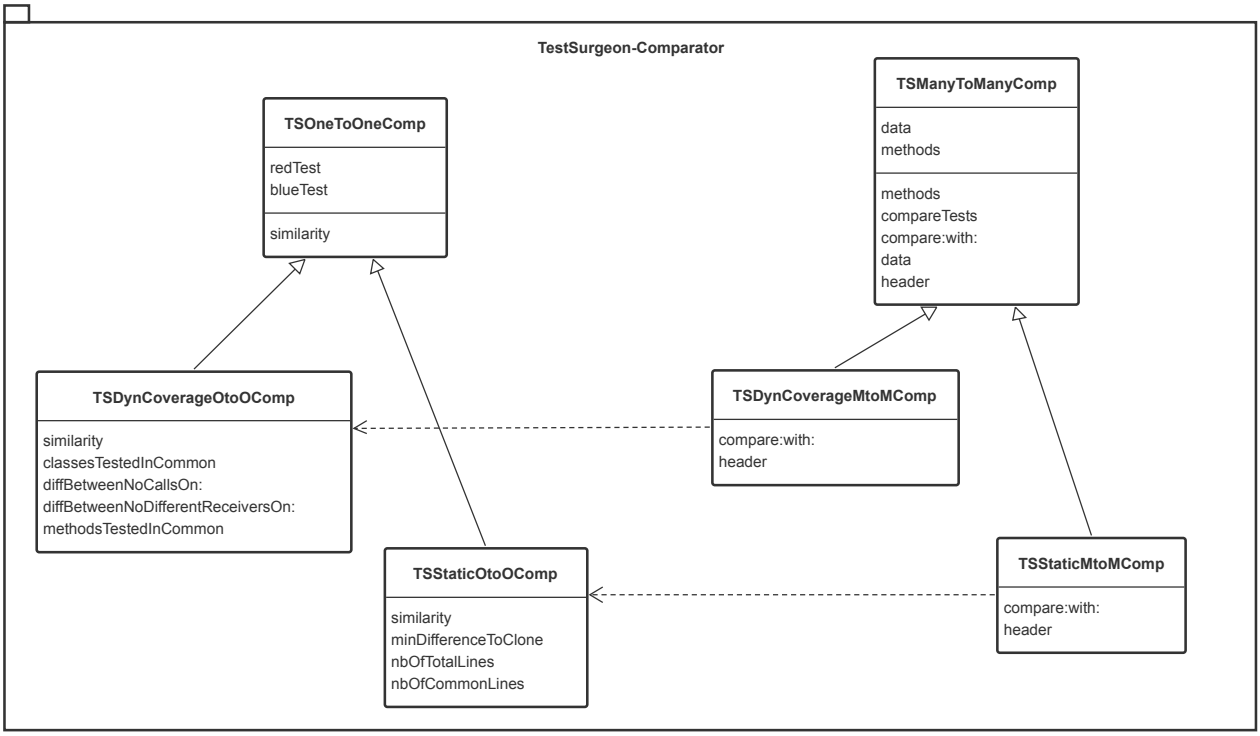


Figura 6.4: Diagrama de clases del paquete `TestSurgeon-Comparator`

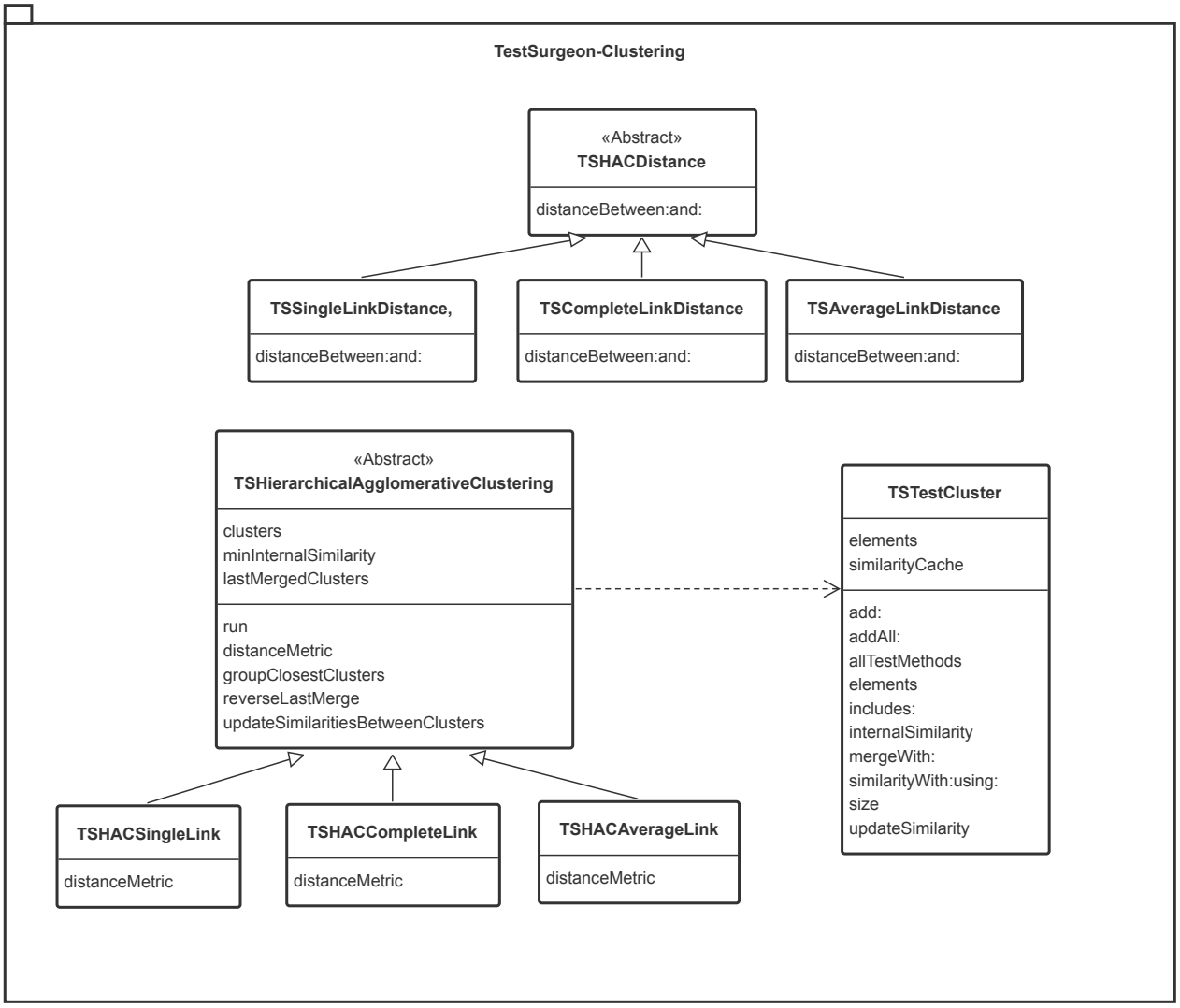


Figura 6.5: Diagrama de clases del paquete `TestSurgeon-Clustering`

El paquete `TestSurgeon-Clustering` contiene las clases necesarias para agrupar tests. La clase `TSHACDistance` define solamente el método abstracto `distanceBetween:and:` que recibe como argumentos dos instancias de la clase `TSTestCluster`, el cual debe ser sobrescrito por sus subclases. Estas son `TSSingleLinkDistance`, `TSCompleteLinkDistance` y `TSAverageLinkDistance` los cuales implementan dicho método según la definición de distancia correspondiente. Por su parte la clase `TSHierarchicalAgglomerativeClustering` contiene todo el comportamiento necesario para ejecutar el algoritmo salvo la métrica de distancia, que está representada por el método `distanceMetric` y es sobrescrito por sus tres subclases: `TSHACSingleLink`, `TSHACCompleteLink` y `TSHACAverageLink` los cuales retornan la clase que implementa dicha métrica. A continuación se muestra el código del método principal que ejecuta el algoritmo.

```
TSHierarchicalAgglomerativeClustering>>run
run
  " Al comienzo hay N clusters con un solo elemento "
  clusters := elements collect: [:el | TSTestCluster new add: el. ].

  [
    "Invariante"
    (clusters collect: #internalSimilarity) min >= minInternalSimilarity.
  ]
    whileTrue: [
      "computar las similitudes entre clusters y ordenarlas en orden descendente "
      self updateSimilaritiesBetweenClusters.
      self groupClosestClusters
  ].

  " el ciclo termino porque el ultimo merge produjo un cluster no deseado, por lo cual se va a separar"
  self reverseLastMerge.

  " etiquetar clusters y retornarlos "
  self tagClusters.

  ^ clusters
```

La clase `TSTestCluster` representa un grupo de test el cual responde a las siguientes mensajes:

Selector	**Resultado**
`internalSimilarity`	Retorna el valor de su similitud interna.
`mergeWith: anotherCluster`	Retorna una nueva instancia de `TSTestCluster` con la unión entre sus elementos y los de `anotherCluster` que representa la fusión de ambos.
`similarityWith: anotherCluster using: aTSHACDistanceClass`	Retorna la similitud con `anotherCluster` usando la métrica de distancia entregada como argumento (`aTSHACDistanceClass`).
`includes: element`	Retorna verdadero si `element` pertenece al cluster.
`size:`	Retorna la cantidad de elementos contenidos.

Tabla 6.5: API pública de las instancias de `TSTestCluster`

El paquete `TestSurgeon-Comparator` contiene a comparadores de test methods que ofrecen diversas métricas según el tipo de comparación que se realice. Las comparaciones pueden ser entre dos métodos de test (`TSOneToOneComp`), entre un método de test y un grupo de tests (`TSOneToManyComp`) y entre dos grupos de test (`TSManyToManyComp`). Los dos últimos comparadores reutilizan el comparador uno-a-uno aplicándolo a colecciones de test, por lo cual detallaremos el primero. La clase `TSOneToOneComp` es una clase abstracta que contiene dos atributos: `redTest` y `blueTest`, los tests a comparar. El comportamiento mínimo que

cualquier extensión de esta clase debe implementar es de retornar el valor de similitud entre el test rojo y azul a través del método `similarity`. Las dos clases que lo extienden son: `TSDynCoverageOtoOComp` y `TSStaticOtoOComp`. El primero implementa la métrica g presentada anteriormente, y la segunda implementa la métrica f (ver Sección 4.1.1.1) que mide la similitud de test según su código fuente. A continuación presentamos la API pública de cada una de estas clases.

Selector	**Resultado**
`similarity`	Retorna el valor de la métrica g entre los tests rojo y azul.
`diffBetweenNoCallsOn: aTSMethod`	Retorna la diferencia entre la cantidad de llamados recibidos por `aTSMethod` durante la ejecución del test rojo y la del test azul.
`diffBetweenNoDifferent-ReceiversOn: aTSMethod`	Retorna la diferencia entre la cantidad de objetos distintos que recibieron el mensaje `aTSMethod` durante la ejecución del test rojo y la del test azul.
`numberOfMethods-TestedInCommon`	Retorna la cantidad de métodos cubiertos en común.
`numberOfAllMethodsTested`	Retorna la cantidad de métodos cubiertos en total por ambos tests.

Tabla 6.6: API pública de las instancias de `TSDynCoverageOtoOComp`

Selector	**Resultado**
`similarity`	Retorna el valor de la métrica f entre los tests rojo y azul.
`nbOfCommonLines`	Retorna la cantidad de líneas de código en común entre ambos tests.
`nbOfTotalLines`	Retorna el total de líneas de código en ambos tests (contados sin repetición e ignorando espacios)
`minDifferenceToClone`	Retorna el menor valor comparando la diferencia entre el número de lineas en común y el numero de líneas de código definido por cada test. Si el valor es cero, indica que el código de un test está completamente dentro del otro test.

Tabla 6.7: API pública de las instancias de `TSStaticOtoOComp`

Finalmente, en el paquete `TestSurgeon-Experiments`, la clase `TSClusteringExp` realiza el experimento de correr el algoritmo de clustering bajo distintas tolerancias de similitud interna y distintas distancias de clustering cuyos resultados se pueden consultar en el Apéndice C. El método principal de esta clase se presenta a continuación.

```
TSClusteringExp>>run
  | clusteringAlgorithm minIntSimilerities tests |
  " Obtener los tests a agrupar "
  tests := (profiler classAt: #ROMondrianViewBuilderTest) allTestMethodsAsTSTestMethods.
  " Definir las similitudes minimas a evaluar"
  minIntSimilerities := #(0.95 0.9 0.85 0.8 0.75).

  " Correr el algoritmo con distintas distancias para cada similitud minima"
  TSHierarchicalAgglomerativeClustering subclasses do: [:HACAlgorithm |
    minIntSimilerities do: [:minIntSimilarity |
      |algo|
      " Inicializar el algoritmo y correrlo "
      algo := (HACAlgorithm
                elements: tests
                minimumInternalSimilarity: minIntSimilarity).
      algo run.

      " Adjuntar los resultados obtenidos "
      results addAll: algo results.
      clusteringAlgorithms add: algo.
    ]
  ].

  " Exportar los resultados a un archivo CSV "
  self exportResults.
```

6.6. Acceso al código

Todo el código detallado en las secciones anteriores está almacenado en el sitio *Smalltalk-Hub*[3], en el repositorio ubicado en la siguiente url:

`http://smalltalkhub.com/#!/~PabloEstefo/TestSurgeon/`

[3]Smalltalk Hub [en línea] <`http://www.smalltalkhub.com`>[Consulta: 03/11/2013]

Capítulo 7

Caso de estudio: Roassal

En el siguiente capítulo se presentará una aplicación de *TestSurgeon* analizando los tests de *Roassal*.

7.1. Roassal: motor de visualización

Roassal es un motor de visualización desarrollado en la plataforma Pharo. Roassal está hecho para visualizar e interactuar con datos arbitrarios definidos en términos de objetos y sus relaciones. Roassal es comúnmente utilizado para producir visualizaciones interactivas y su rango de aplicaciones es diverso. Sin embargo, es particularmente utilizado en el análisis de software (ver Figura 7.1), de hecho es una de las componentes principales de TestSurgeon.

A modo de ejemplo se adjunta el código que permite visualizar una jerarquía de clases y algunas propiedades de cada una de sus componentes usando Roassal. La clase visualizada es `Collection`: clase abstracta de la cual heredan distintos tipos de clases que representan colecciones de objetos, como por ejemplo: `Set`, `Array`, `OrderedCollection`, `String` y `Dictionary`, entre otros. La visualización presentada en la Figura 7.1 muestra a cada clase con una altura y ancho los cuales dependen de de la cantidad de métodos definidos y número de atributos respectivamente. El código y la imagen resultante se adjuntan a continuación.

```
view := ROView new.
classElements := ROElement forCollection:
  Collection withAllSubclasses.
" Crear elementos graficos a partir de la clase Collection y sus clases hijas "
classElements
  do: [:c |
    c width: c model instVarNames size.
    c height: c model methods size.
    c + ROBorder.
    c @ RODraggable ].
view addAll: classElements.
```

```
" Enlazar clases para formar la jerarquia "
associations := classElements collect: [:c |
   (c model superclass = Object)
      ifFalse: [ (view elementFromModel: c
      model superclass) -> c]
   ] thenSelect: [:assoc | assoc isNil not ].
edges := ROEdge linesFor: associations.
view addAll: edges.

" Aplicar una distribucion grafica en forma de arbol para visualizar mejor la jerarquia "
ROTreeLayout new on: view elements.
view open
```

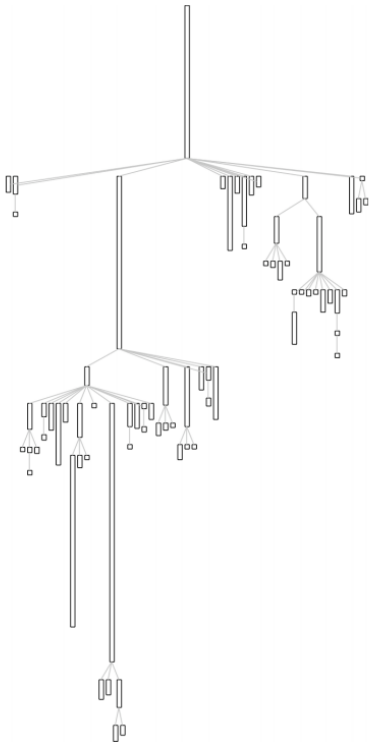


Figura 7.1: Visualización hecha en Roassal: Jerarquía de la clase `Collection` y sus clases hijas. El alto representa el número de métodos y el ancho el número de atributos

7.2. Análisis del estado de los tests

Como primera aplicación de *TestSurgeon* se consideró refactorizar el código de test de Roassal. La versión analizada es la número 441 que puede ser descargada desde su repositorio en SmalltalkHub[1]. Esta consta de 30 paquetes los cuales contienen 301 clases que, a su vez, definen 4.137 métodos. Estos últimos en total suman 29.720 líneas de código.

Por su parte, para su correcto funcionamiento se definieron 80 unit test que contienen,

[1]SmalltalkHub . Object Profile / Roassal [en línea] <`http://smalltalkhub.com/#!/~ObjectProfile/Roassal/versions/Roassal-VanessaPena.441`>[Consulta 03/11/2013]

todos juntos, 556 test methods. La cobertura de los tests alcanza un 72.37 %. En Roassal cada unit test está relacionado a una característica particular de Roassal como: manejo eventos, interacción, formas de las figuras(shapes), rendering, entre otros.

Sin embargo uno de esos unit tests parece contener una cantidad significativamente mayor de tests en comparación a los demás unit tests. El unit test en cuestión es `ROMondrianViewBuilderTest` el cual contiene 96 métodos de test que corresponde a un 17 % del total de test methods definidos en Roassal. Este unit test cubre las funcionalidades de la clase `ROMondrianViewBuilder`, una clase que proporciona una API de alto nivel para crear visualizaciones. Esta API proporciona la mayoría de las funcionalidades de Roassal, por lo cual `ROMondrianViewBuilderTest` prácticamente realiza llamados sobre gran parte del código de Roassal (además del código de la API).

Revisando el código se detectaron casos de duplicación de código y redundancia de ejecución entre tests de `ROMondrianViewBuilderTest` y con otros unit tests. Como son tests de alto nivel, cubren un funcionamiento más complejo (orquestando distintas unidades de Roassal) por lo cual la ejecución de éstos es más compleja y haciendo la tarea de depuración más engorrosa.

Además se verificó que la gran mayoría de los unit tests aporta una cantidad relativamente pequeña de tests. Esta situación se puede observar en la Figura 7.2 donde se muestra la distribución de los test methods. Los unit tests se muestran agrupados según la cantidad de tests que contienen. Observando la figura se aprecia que la mayoría de los unit tests en Roassal contiene una cantidad pequeña de tests. De hecho, el 55 % de los unit tests contiene a lo más 5 tests (ver Tabla B.1) y contando como máximo 10 métodos de test definidos, se alcanza un 83 % del total de los unit tests.

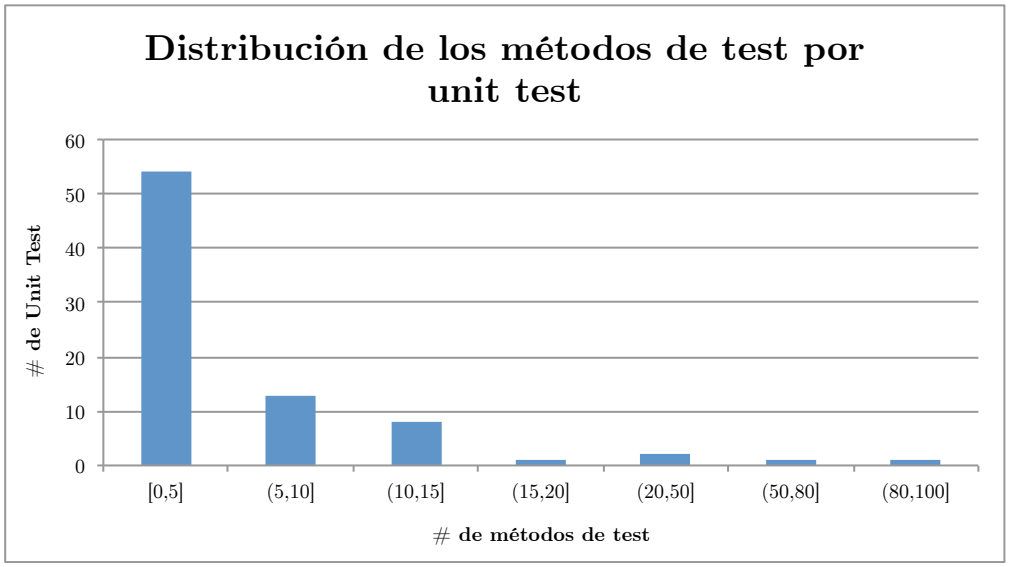


Figura 7.2: Distribución de los tests methods por unit test

Esta situación impacta en la calidad de documentación de Roassal, ya que los tests son la principal fuente de documentación actualizada. Comparativamente `ROMondrianViewBuilderTest` es mucho más grande que los otros unit tests y además es de gran tamaño. Esto complica al desarrollador novato en el proyecto ya que le será difícil encontrar aquel test que detalla

tal o cual funcionalidad particular. Esto hace que el proyecto sea más difícil de aprender y entender para desarrolladores externos.

7.3. Clustering por cobertura

Para analizar los tests de Roassal con TestSurgeon, es necesario realizar comparaciones 1-a-1 entre test methods. Los datos anteriormente expuestos muestran que las pruebas unitarias de Roassal contienen en total 556 test methods. Esto significa que potencialmente se realizarán $\frac{556 \cdot (556-1)}{2} = 154{,}290$ comparaciones. Las comparaciones 1-a-1 constan en interpretar la visualización de diferencias de ejecución y contrastarla con el código fuente para ver si es posible realizar una refactorización. Este trabajo requiere del criterio del desarrollador, por lo cual esa cantidad de comparaciones es inmanejable. Ahora, considerando que se analizará en particular el unit test `ROMondrianViewBuilderTest`, la cantidad de comparaciones son $\frac{96 \cdot (96-1)}{2} = 4{,}560$, que sigue siendo una cantidad muy grande.

Durante la realización de este trabajo se experimentó con tests de distintos softwares lo cuales fueron analizados con el browser de TestSurgeon. Una de las lecciones aprendidas es que la métrica de similitud por cobertura es un buen indicador para encontrar casos de comparación relevantes. Si bien no es una métrica precisa para detectar casos de refactorización, es una buena aproximación para saber en cuales parejas de tests enfocarse y cuales descartar ya que por su cobertura no son comparables. Más aun, la experiencia indica que pares de tests con menos de 80 % de similitud en cobertura son difícilmente comparables. Esta cota inferior es más estricta cuando la cantidad de métodos cubiertos en común es menor, y recíprocamente, se puede relajar la cota cuando la cantidad de métodos testeados en común es mayor.

Entonces, para reducir la cantidad de comparaciones y facilitar la detección de *test smells*, una buena alternativa es agrupar los tests usando la similitud de cobertura. De esta manera estamos agrupando los casos interesantes para después inspeccionarlos con el browser. Para agruparlos se decidió utilizar el algoritmo de agrupamiento aglomerativo jerárquico (o Hierarchical Agglomerative Clustering). Este algoritmo usa una métrica de similitud entre elementos que en este caso es la métrica de similitud por cobertura, y otra métrica que determina la distancia entre dos clusters (o grupos de elementos) para definir si están lo suficientemente cerca y fusionarlos o no. Luego de un experimento, se concluyó que la distancia Complete Link es la más representativa para esta situación[2].

Luego de aplicar el algoritmo sobre los tests definidos en `ROMondrianViewBuilderTest`, se obtuvo 22 clusters de los cuales sólo 6 son unitarios (*i.e.,* aquellos que contienen solo un elemento). Esta partición de los tests es particularmente buena ya que cada uno de los clusters formados son de un tamaño razonable y entre la diferencia de tamaños entre estos no es tan marcada (ver Figura 7.3). Además los elementos de cada grupo son bastante homogéneos, es decir, muy similares que justamente son los casos de interés del desarrollador. Como dato anecdótico, se encontró un cluster de dos tests que cubren exactamente los mismos métodos.

[2]El experimento de evaluación del algoritmo se presenta en detalle en el Apéndice C, y sus detalles técnicos en la Sección 6.5

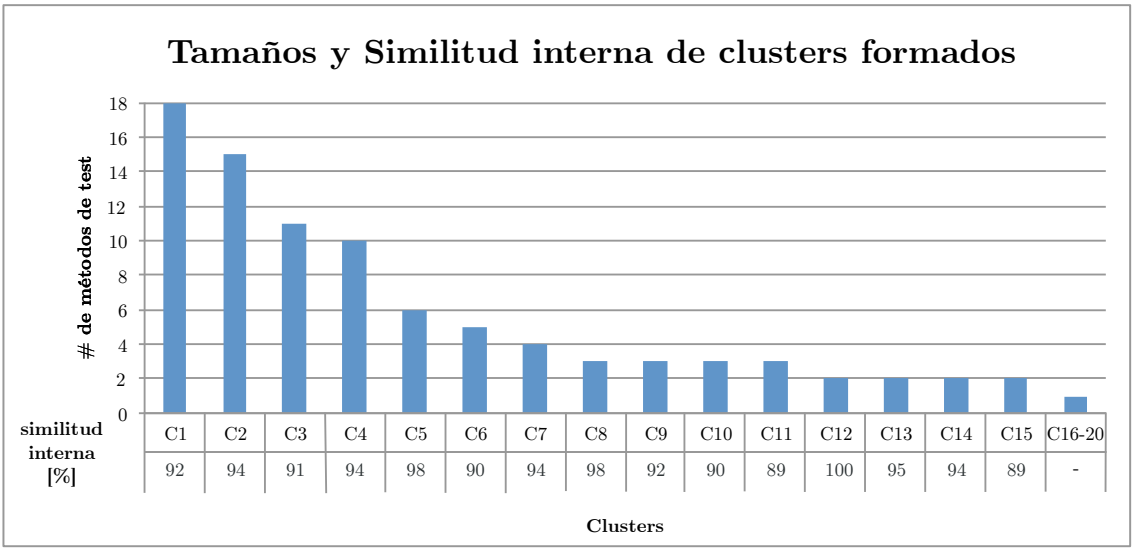


Figura 7.3: Los tests fueron agrupados en 22 grupos

Finalmente, el número de comparaciones se redujo a 405 comparaciones. Esto significa que la cantidad de comparaciones se redujo aproximadamente a un 9 % de la situación inicial, lo que permite al desarrollador enfocarse solamente en los mejores casos de comparación que son candidatos a refactorizaciones.

7.4. Escenarios de refactorización y restructuración de los tests

Una vez agrupados los tests según su similitud de cobertura, la probabilidad de encontrar casos de refactorización es mayor ya que la revisión se realiza sobre casos similares. Y en estos, las ejecuciones de los tests comprenden una importante porción de métodos y clases comunes lo cual apunta a las mismas características del software (ej: rendering, manejo de eventos, etc).

Luego de una exhaustiva inspección a través de comparaciones 1-a-1 entre tests del mismo grupo y de grupos distintos se detectaron tres escenarios de refactorizaciones posibles. En esta labor la *Test Difference Blueprint* juega un rol fundamental para rápidamente ver si es un caso interesante o pasar a otro. Luego de detectado un caso de interés se procede a observar la zona de comparación de código fuente para ver si procede alguna refactorización.

Estos casos fueron analizados y discutidos en conjunto con los principales desarrolladores de Roassal. A continuación se detallan los escenarios principales encontrados durante la experiencia.

7.4.1. Escenario #1: Identificando diferencias semánticas

La visualización actúa como un medio eficiente para, en forma rápida y fácil comparar tests: los colores y formas de los métodos ayudan a identificar cuales tests son semánticamente parecidos o diferentes. La Figura 7.4 presenta una representación que obtuvimos mientras navegamos a través de los métodos contenidos en `ROMondrianViewBuilderTest`.

El uso de colores marcados y distintos como: rojo, azul y gris, informan al desarrollador rápidamente sobre qué tan similar semánticamente son los tests en comparación. Los métodos grises, como ya se detalló, indican aquellos métodos que son llamados por ambos tests. Por su parte, métodos de colores rojo o azul son testeados solo por un test. De esta manera se aprecia en forma expedita el grado de similitud entre tests. El tamaño de los métodos indica la diferencia entre los dos tests para ese método. La Figura 7.4 ilustra una gran diferencia entre dos métodos de test.

En el browser de *TestSurgeon*, el código fuente del método está a un clic de distancia de la visualización. Los menús contextuales muestran varias opciones para navegar e inspeccionar cualquier elemento estructural de la visualización si fuera requerido para mayor detalle.

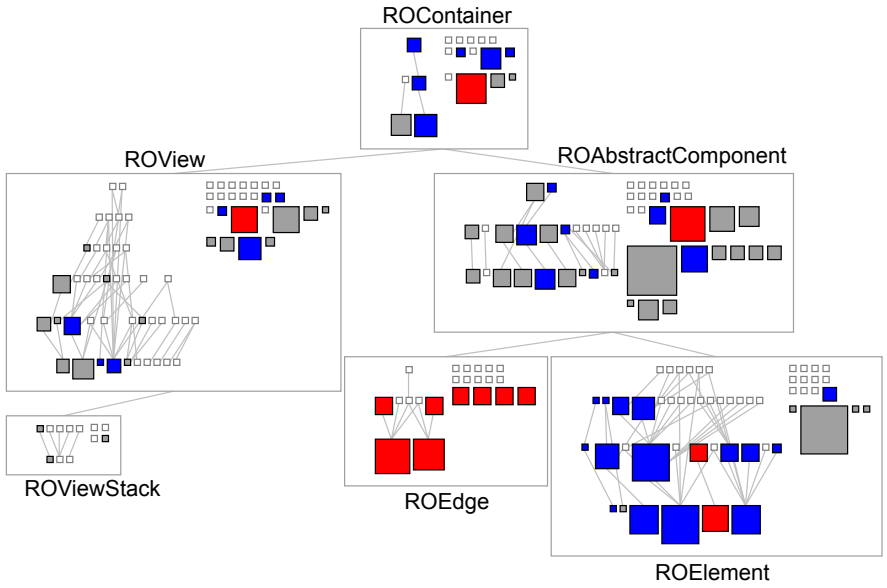


Figura 7.4: La presencia de muchos métodos rojos y azules indica el grado de diferencia entre los tests

En conclusión, TestSurgeon a través de su visualización permite rápidamente ver si la pareja de tests que se está comparando corresponde a un caso interesante o no.

7.4.2. Escenario #2: Definiendo inicialización del fixture

Es frecuente que aparezca que cada test define su propia inicialización antes de realizar alguna verificación (*assertion* en inglés) de invariantes usando la palabra clave `assert:`. La inicialización del test corresponde a la porción situada antes de la primera verificación (*i.e.*, antes de la primera llamada a `assert:`).

En nuestro caso de estudio, identificamos una cantidad significativa de tests que usan un escenario (o *fixture*) de ejecución similar. Considere los siguientes dos métodos de test definidos en `ROMondrianViewBuilderTest`:

```
1  testAbsolutePosition
2    | inner2 outter inner |
3    outter := view node: 'outter' forIt: [
4      inner := view node: 'inner' forIt: [
5        inner2 := view node: 'inner2'
6      ]
7    ].
8    view applyLayout.
9    self assert: outter absolutePosition = outter position.
10   ...
11
12
13 testPositionRelativeTo
14   | outterNode1 innerNode1 innerNode2 |
15   outterNode1 := view node: 'outter1' forIt:
16     [ innerNode1 := view node: 1 forIt: [
17     innerNode2 := view node: 2 ] ].
18   view applyLayout.
19   self assert: innerNode2 bounds = ((5@5) corner: (10@10)).
20   ...
```

Los dos test inicializan un escenario muy similar que consiste en nodos (elementos gráficos) anidados (uno dentro del otro) que se nombran con identificadores de variables locales del método. Posteriormente, se aplica el layout (ver método `applyLayout` en líneas 8 y 18) y finalmente se verifican distintas propiedades de los nodos como son: posición y tamaño.

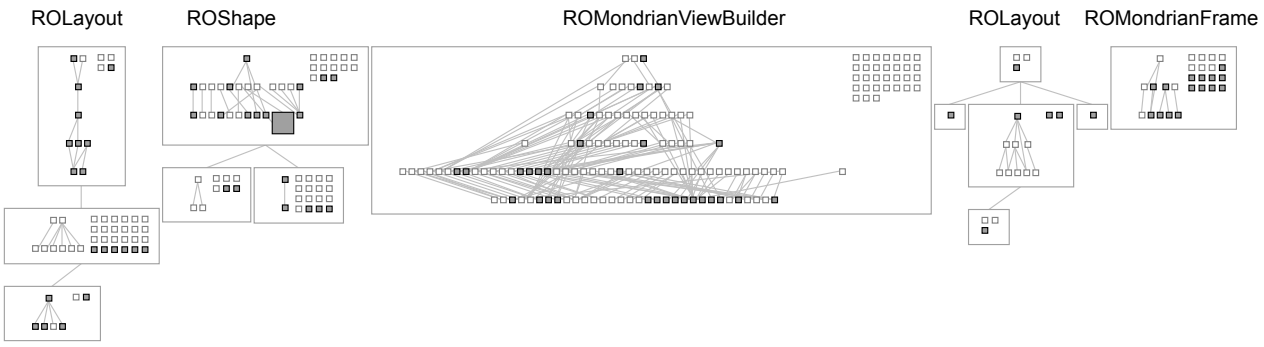


Figura 7.5: Dos tests con un escenario de ejecución común (métodos grises son los métodos comunes para ambos tests)

La Figura 7.5 muestra la vista panorámica de ejecución de ambos tests. En la figura se aprecia que 13 clases están involucradas en la ejecución de éstos. La presencia de cuadrados pequeños grises indica que los métodos correspondientes fueron testeados por los dos tests: `testAbsolutePosition` y `testPositionRelativeTo` y en una forma muy parecida. Esto significa que la cobertura de los tests es casi idéntica: exactamente la misma cantidad de métodos son ejecutados. La clase `ROShape` contiene un método grande, lo que indica que ese método es ejecutado "muchas más veces" y en "muchos más objetos distintos" en uno de los dos tests. Una ventana de popup se activa cuando se posiciona el cursor sobre un método. Allí se presentan en detalle las métricas de ejecución, permitiendo saber para cuál de los dos tests, ese método en particular, es más relevante.

Entonces, aunque el código fuente de dichos tests es distinto, ellos involucran un escenario de ejecución similar. Se refactorizó los dos tests en un nuevo unit test llamado `ROPositionTest`, donde el código de fixture que tenían en común se dejó en el método `setUp`. Los dos tests se migraron a este unit test y se acortaron de la manera que se presenta a continuación:

```
ROPositionTest>>setUp
  outter := view node: 'outter' forIt: [
    inner := view node: 'inner' forIt: [
      inner2 := view node: 'inner2'
    ]
  ].
  view applyLayout.

ROPositionTest>>testAbsolutePosition
  self assert: outter absolutePosition = outter position.
  ...

ROPositionTest>>testPositionRelativeTo
  self assert: inner2 bounds = ((5@5) corner: (10@10)).
  ...
```

El test case `ROMondrianViewBuilderTest` fue recortado y refactorizado en tests dedicados.

Este escenario no afecta el rendimiento de los tests ya que el código inicialización se ejecutará una vez por cada tests igual que antes de la refactorización. Sin embargo, mejora la calidad del test como documentación y le da flexibilidad al diseño del código. En cuanto a la depuración, favorece la identificación de errores o bugs ya que separa la inicialización general de los tests particulares.

7.4.3. Escenario #3: Mezclando o Removiendo métodos de test

Se encontraron casos donde dos tests están muy relacionados semánticamente. En ese caso, los dos test pueden ser fusionados en un solo test. La otra alternativa es mover uno de ellos, aquel cuya cobertura fuera menor, dentro de un test suite que no se ejecute con tanta frecuencia para así evitar redundancia.

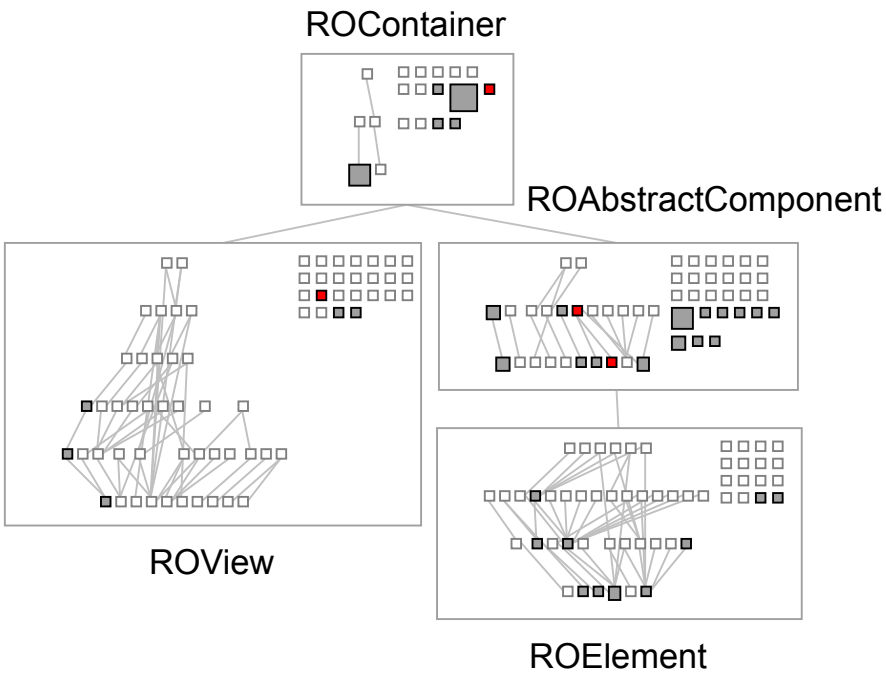


Figura 7.6: No hay mucha diferencia entre los dos tests

Considere la siguiente situación donde dos tests tienen distinto código fuente pero cubren el mismo código base. La Figura 7.6 muestra una jerarquía de clases compuesta por cuatro clases que son testeadas por dos tests similares. Todos los métodos cubiertos son ejecutados en forma similar por los dos tests, lo cual queda evidenciado en por la presencia de pequeños cuadrados grises. Inclusive, los métodos rojos también son pequeños, y pocos, lo que indica que la diferencia de cobertura es no significativa.

```
1 ROAddNameTest>>testAddTwice
2   ROAddName toElement: element.
3   ROAddName toElement: element.
4   self assert: view numberOfElements = 2.
5
6 ROAddNameTest>>testAddition
7   self assert: element view numberOfElements = 1.
8   ROAddName toElement: element.
9   self assert: element view numberOfElements = 2.
```

Estos dos tests pueden fusionarse en un solo test:

```
ROAddNameTest>>testAdditionAndAddTwice
  self assert: element view numberOfElements = 1.
  ROAddName toElement: element.
  self assert: element view numberOfElements = 2.

  ROAddName toElement: element.
  self assert: view numberOfElements = 2.
```

Finalmente los dos tests fueron refactorizados en un test único.

Este escenario puede disminuir o mantener el tiempo de ejecución, ya que al fusionar dos tests eventualmente se eliminan llamadas entre objetos.

7.5. Aprendizajes y Conclusiones

La experiencia de utilizar TestSurgeon para la refactorización de los tests de Roassal generó distintas lecciones y aprendizajes sobre esta actividad y sobre el complejo artefacto que son las pruebas de software.

La redundancia dinámica entre los tests existe y en gran proporción. Sin embargo, muchas veces es difícil de manejar u optimizar puesto que se encuentran en clases y paquetes con finalidades distintas. Entonces, en estos casos la refactorización requiere una reestructuración mayor de los tests involucrados. Lo cual impacta su utilización como documentación. Este tipo de redundancia se hace prácticamente imposible de eliminar, o bien, invita a reescribir los tests.

Los tres escenarios de uso de TestSurgeon respaldan su utilidad y son un buen apronte para seguir desarrollando la herramienta. Si bien, los casos de refactorización como los descritos en el escenario #2 son posibles de detectar a través de herramientas ya existentes de detección de duplicación, la visualización se puede saber si procede o no una refactorización. Ya que es información que sólo se puede conocer habiendo ejecutado los tests.

Por otro lado, la métrica g (similitud por cobertura) facilita en gran medida la detección de casos que no pueden ser encontrados por simple inspección del código.Así fue como se encontraron varios casos del escenario #3. Pares de tests con similar cobertura se contrastaban en la visualización y luego se revisaba el código buscando una posible refactorización. En caso de ser similares, se fusionaban.

Finalmente, la variedad de usos e intereses detrás de los tests hacen de éste un artefacto muy completo pero a la vez muy complejo de manejar y modificar. Considerando el estudio presentado en este capítulo, se puede concluir que TestSurgeon es una alternativa efectiva para la refactorización de unit tests.

Capítulo 8

Conclusión y Trabajo Futuro

8.1. Conclusión

En este trabajo se presentó en detalle el proceso de investigación, desarrollo e implementación de una solución para abordar problema de mantenibilidad del código en las pruebas de software o tests: *TestSurgeon*. Se mostró la importancia de un enfoque que combinara las características estáticas (código fuente) y dinámicas (ejecución) del artefacto correspondiente a los tests. Para diagnosticar el estado de los tests y detectar posibles *smells* es necesario conocer su ejecución. Luego de detectado una situación como esta, el diagnóstico debe contrastarse con código fuente involucrado ya que cualquier refactorización se realiza sobre éste.

TestSurgeon es una herramienta que provee un navegador de tests en el cual se pueden realizar comparaciones 1-a-1 entre test methods. Para la realización de ésta se dispone de una visualización llamada *Test Difference Blueprint* que resume varias métricas de la ejecución de los tests sobre el código base. Esta gráfica permite diferenciar rápidamente los tests considerando tanto la cobertura de los tests como otras métricas relevantes para contextualizar sus ejecuciones. Junto con esto, se provee de un comparador de código que además de mostrar el código fuente de cada test colorea las diferencias entre estos. En conjunto funcionan como una herramienta eficaz para la detección y posible corrección de *test smells*. Si bien TestSurgeon funciona solamente en la plataforma Pharo Smalltalk, es posible portarla a cualquier otro lenguaje orientado a objetos donde se pueda implementar el profiler Spy.

Ahora bien, contrastando lo anterior sobre los objetivos planteados al inicio del proyecto, se considera que todos estos fueron alcanzados en distintos grados. Se logró identificar las métricas clave para la caracterización de la ejecución de los tests: `NOC` y `NODR` (ver Sección 5.2.3) permiten contextualizar la ejecución de un método otorgando una valiosa información que favorece su comparación. Se encontraron además otras métricas como: `NOI` - Número de instancias creadas por clase o `NOMR` - Número de mensajes recibidos por todas las instancias de una clase, entre otras. Si bien estas métricas son un aporte y enriquecen la caracterización de la ejecución de un test en igual medida aumentan su complejidad, manejo y dificultan la elaboración de una visualización efectiva, intuitiva y simple.

Con respecto a lo último, justamente el asunto de la simplicidad y efectividad de la visualización fue un objetivo completamente logrado. La visualización representa de buena forma la metáfora de jerarquía de clases propia de un lenguaje de programación que sigue el paradigma de orientación a objetos como lo es Smalltalk (y su dialecto Pharo). Además, que los métodos, protagonistas de la gráfica, sean elementos estructurales de ésta refuerza la idea. Con lo cual se obtiene una visión contextual que permite una efectiva diferenciación semántica entre dos tests y favorece la rápida detección de *test smells* contando con información tanto de su performance como de su código.

En el Capítulo 7 se presenta un experimento de clustering de test utilizando métricas de cobertura en búsqueda de una solución semi-automática al problema de reestructurar test unitarios usando el navegador de TestSurgeon. Como resultado se logró crear 16 clusters con casos potencialmente interesantes para comparar, reduciendo el 91 % del total de casos que son irrelevantes. De esta manera se facilita la labor del desarrollador, la cual es, al menos desde el acercamiento de TestSurgeon irremplazable por lo crítico que resulta evaluar la situación y generar una reestructuración con el menor impacto en la confiabilidad de los tests posible.

Las refactorizaciones encontradas fueron de carácter conservadora y su realización dependía no tan solo de la información provista por visualización y métricas, sino que también de un análisis de factibilidad contrastando dicha situación con el código fuente. Sin embargo no se realizó un análisis del posible debilitamiento de los tests ni de una eventual disminución de cobertura posterior a las refactorizaciones. Por lo cual no se puede estar seguro de la inocuidad de esta técnica. Aunque esto puede ser verificado a través del uso de herramientas dedicadas. En conclusión, el tercer objetivo se considera logrado pero no en forma completa.

Finalmente, el último objetivo que corresponde al diseño e implementación de una interfaz de usuario sencilla y usable para el uso cotidiano de reestructuración de test por parte de un desarrollador se considera parcialmente alcanzado. Esto dado que, si bien no se realizó pruebas de usuario formales sobre esta herramienta, la experiencia presentando *TestSurgeon* en las conferencias de ESUG (European Smalltalk User Group) del año 2012 y en forma personal con colegas desarrolladores muestra resultados muy favorables. La aplicación y su interfaz es muy sencilla, de rápido aprendizaje y fácil de usar.

El trabajo realizado es de carácter innovador en su campo ya que provee una visualización expresiva e intuitiva para entender diferencias entre ejecución de tests, lo cual es clave para la detección de los *smells* y su refactorización y/o reestructuración. Lo anterior es avalado por la destacada participación en conferencias internacionales como por ejemplo la Conferencia Internacional en Ingeniería de Software [7] (ICSE12, principal conferencia en el área, L0) donde el trabajo obtuvo el primer lugar en la ACM Student Research Competition en la categoría de pregrado[1].

El refactoring o reestructuración es un proceso complejo con consecuencias en múltiples aspectos. Para lo cual se requiere de vasta información sobre las entidades de software a modificar. Realizarlo entonces con artefactos tales como las pruebas de software lo hace par-

[1]ACM Student Research Competition .Past Years' Winners [en línea] <`http://src.acm.org/previouswinners.html`>[Consulta: 03/11/2013]

ticularmente más riesgoso y complejo. Sin embargo el problema de mantenibilidad de los tests y el poco cuidado que se tiene de su diseño es real y debe ser enfrentado por su importancia en el resguardo de la calidad del software base. Si bien la academia ha realizado cierta investigación en este problema, esta es insuficiente y actualmente no existe un software que permita analizar este problema. *TestSurgeon* es una solución que ofrece una panorámica muy rica en información que incluye los factores clave para caracterizar las pruebas de software, detectar *smells* y evaluar alternativas de reestructuración. Los resultados obtenidos en las primeras aplicaciones muestran una efectividad razonable para ser un primer acercamiento al problema, lo que habla de una estrategia de solución promisoria para, a futuro, desarrollar y profundizar.

8.2. Trabajo Futuro

El presente informe presenta un caso de estudio donde se aplica la herramienta para un software en particular: Roassal. Durante la experiencia se logran detectar 3 casos de reestructuración los cuales fueron presentados en detalle en el Capítulo 7. Si bien estos escenarios no dependen en ningún caso del dominio específico del software analizado sería muy bueno ratificar la presencia de esos casos a través de un análisis similar a otros softwares. Junto con esto se podrían eventualmente encontrar otros escenarios de *test smells*.

En este trabajo se logró desarrollar una visualización intuitiva y concisa para el problema definido. Sin embargo, una mejora que se tuvo presente pero no alcanzó a implementarse fue la de diferenciar en la visualización la ejecución correspondiente al *fixture* de un test y las *assertions* (o verificaciones). En otras palabras, mostrar la visualización para comparar el escenario que ejecuta el test donde se crean los objetos y se los hace "actuar", y comparar por separado la parte de la ejecución donde se verifican las propiedades de los objetos creados. Esto facilitaría la detección de solapamientos y ejecución redundante, haciendo más rápida la detección de escenarios del tipo # 2 (ver Sección 7.4.2).

Durante la investigación de este trabajo se identificaron varias métricas que caracterizan la ejecución de un test desde distintos enfoques. Sin embargo el tiempo no alcanzó para implementar y estudiar todas ellas. Estas métricas son:

`NSCH` - **Número de cambios de estado** : Esta métrica permite conocer cuáles métodos realizaron cambios en alguno de los atributos del objeto donde fueron ejecutados. Esto agrega otra dimensión al análisis de un método en la ejecución.

`NODA` - **Número de distintos argumentos recibidos por un método** : Otra dimensión que caracteriza un método y su comportamiento durante la ejecución de un test es la cantidad de argumentos distintos que recibió un método las veces que fue llamado.

`NOI` - **Número de instancias creadas por clase** : Una métrica muy útil para detectar las clases relevantes para ciertos tests es el número de instancias de una clase creadas durante su ejecución.

`NOMR` - **Número de mensajes recibidos por todas las instancias de una clase** : Es in-

teresante saber cuantas veces las instancias de cierta clase ejecutó algún método. De esta manera se pueden diferenciar a las clases cuyas instancias fueron más activas durante la ejecución del test y por ende más relevantes.

Las métricas anteriores enriquecen la comprensión y el análisis de las entidades estudiadas (clases y métodos), pero a la vez aumentan la complejidad del problema dado el aumento de dimensiones. Para incluir estos datos y complementar la información que ya se entrega habrá que explorar con nuevas visualizaciones que permitan condensar y ponderar de manera correcta, veraz e intuitiva toda la información extra que se obtendría implementando las métricas anteriormente expuestas.

Bibliografía

[1] Oscar Nierstrasz Andrew P. Black, Stéphane Ducasse. Pharo by example: Chapter 7 - sunit, May 2011. `http://pharo.gforge.inria.fr/PBE1/PBE1ch8.html`.

[2] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman, 2002.

[3] Alexandre Bergel. Counting messages as a proxy for average execution time in pharo. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11)*, LNCS, pages 533–557. Springer-Verlag, July 2011.

[4] Alexandre Bergel, Felipe Ba nados, Romain Robbes, and David Röthlisberger. Spy: A flexible code profiling framework. *Journal of Computer Languages, Systems and Structures*, 38(1), December 2011.

[5] Alexandre Bergel and Vanessa Peña. Increasing test coverage with hapao. *Science of Computer Programming*, 2012.

[6] Daniel Defays. An efficient algorithm for a complete link method. *The Computer Journal*, 20(4):364–366, 1977.

[7] Pablo Estefo. Restructuring unit tests with testsurgeon. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1632–1634, 2012.

[8] Martin Fowler. Continuous integration, May 2006. `http://www.martinfowler.com/articles/continuousIntegration.html`.

[9] Martin Fowler and Jim Highsmith. The Agile manifesto. *Software Development Magazine*, 9(8):29–30, August 2001.

[10] Markus Gaelli. *Modeling Examples to Test and Understand Software*. PhD thesis, Citeseer, 2006.

[11] Markus Gaelli, Orla Greevy, and Oscar Nierstrasz. Composing unit tests. In *2nd International Workshop on Software Product Line Testing*, pages 16–22. Citeseer, 2006.

[12] Markus Gaelli, Michele Lanza, and Oscar Nierstrasz. Towards a taxonomy of unit tests. In *Proceedings of the 13th International European Smalltalk Conference, Brussels, Belgium*. Citeseer, 2005.

[13] Markus Gaelli, Rafael Wampfler, and Oscar Nierstrasz. Composing tests from examples. *Journal of Object Technology*, 6(9):71–86, 2007.

[14] Michaela Greiler, Arie van Deursen, and M Storey. Test confessions: a study of testing practices for plug-in systems. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 244–254. IEEE, 2012.

[15] Michaela Greiler, Arie van Deursen, and Andy Zaidman. Measuring test case similarity to support test suite understanding. In *Proceedings of the 50th international conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 91–107, Berlin, Heidelberg, 2012. Springer-Verlag.

[16] Susan Horwitz. Tool support for improving test coverage. In *Proceedings of the 11th European Symposium on Programming Languages and Systems*, ESOP '02, pages 162–177, London, UK, UK, 2002. Springer-Verlag.

[17] Rainer Koschke. Large-scale inter-system clone detection using suffix trees and hashing. *Journal of Software: Evolution and Process*, 2013. accepted for publication.

[18] Adrian Kuhn, Bart Van Rompaey, Lea Hänsenberger, Oscar Nierstrasz, Serge Demeyer, Markus Gaelli, and Koenraad Van Leemput. JExample: Exploiting dependencies between tests to improve defect localization. In P. Abrahamsson, editor, *Extreme Programming and Agile Processes in Software Engineering, 9th International Conference, XP 2008*, Lecture Notes in Computer Science, pages 73–82. Springer, 2008.

[19] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[20] Adrian Lienhard, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. Test blueprints — exposing side effects in execution traces to support writing unit tests. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08)*, pages 83–92. IEEE Computer Society Press, 2008.

[21] Robert Cecil Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[22] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 291–301, Washington, DC, USA, 2009. IEEE Computer Society.

[23] Paul Piwowarski, Mitsuru Ohba, and Joe Caruso. Coverage measurement experience during function test. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, pages 287–301, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[24] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 2013.

[25] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based assessment of test quality. *Journal of Object Technology*, 6(9):231–251, 2007.

[26] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. Technical report, Citeseer, 2007.

[27] Robin Sibson. Slink: an optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973.

[28] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *Systems, Man and Cybernetics, IEEE Transactions on*, 11(2):109–125, 1981.

[29] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. *Refactoring test code*. CWI, 2001.

[30] Vipindeep Vangala, Jacek Czerwonka, and Phani Talluri. Test case comparison and clustering using program profiles and static execution. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 293–294, New York, NY, USA, 2009. ACM.

[31] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 201–212. ACM, 2009.

Apéndice A

Herramientas de Cobertura revisadas

A continuación se listan las herramientas de cobertura de test analizadas y comparadas con *TestSurgeon*, con sus respectivas URL[1] donde se pueden obtener.

Emma [en línea] <http://emma.sourceforge.net/>

EclEmma [en línea] <http://www.eclemma.org>

JCover [en línea] <http://www.mmsindia.com/JCoverSampleScreen.html>

GroboCodeCoverage [en línea] <http://groboutils.sourceforge.net/codecoverage/index.html>

JCoverage [en línea] <http://maven.apache.org/maven-1.x/plugins/jcoverage/> Desafortunadamente, ya no sigue siendo desarrollado

Parasoft JTest [en línea] <http://www.parasoft.com/jsp/products/jtest.jsp?itemId=14>

Purity Plus [en línea] <http://www-03.ibm.com/software/products/us/en/purifyplus>

Semantic Designs [en línea] <http://www.semdesigns.com/Products/TestCoverage/>

TCAT [en línea] <http://www.testworks.com/Products/TCAT-Java/tcat.java.html>

[1]Todas las URL fueron consultadas por última vez el día 03/11/2013

Quilt [en línea] <http://quilt.sourceforge.net/quilt-0.4/>

NoUnit [en línea] <http://nounit.sourceforge.net/>

InsectJ [en línea] <http://insectj.sourceforge.net>

Hansel [en línea] <http://hansel.sourceforge.net/>

Gretel [en línea] <http://www.cs.uoregon.edu/research/perpetual/dasada/Software/Gretel/>

Jester [en línea] <http://jester.sourceforge.net/>

JVMDI Code Coverage [en línea] <http://jvmdicover.sourceforge.net/>

JBlanket [en línea] <http://csdl.ics.hawaii.edu/research/jblanket/>

Coverlipse [en línea] <http://coverlipse.sourceforge.net>

Koalog [en línea] <http://freecode.com/projects/kcc>

Cobertura [en línea] <http://cobertura.sourceforge.net>

Mojo [en línea] <http://mojo.codehaus.org/cobertura-maven-plugin/>

Thucydides [en línea] <http://docs.codehaus.org/display/SONAR/Thucydides+Plugin>

Apéndice B

Distribución de Tests en Roassal

A continuación se muestra la información sobre la distribución de los tests en los unit test de Roassal (versión 441).

# test	# unit test	%	% acumulado
$[0, 5]$	54	68	68
$(5, 10]$	13	16	84
$(10, 15]$	8	10	94
$(15, 20]$	1	1	95
$(20, 50]$	2	3	98
$(50, 80]$	1	1	99
$(80, 100]$	1	1	100
TOTAL	80	100	100

Tabla B.1: Distribución de los tests methods por unit test en Roassal

Unit Test	#	Unit Test	#
ROMondrianViewBuilderTest	96	ROGridLayoutTest	3
ROElementTest	63	RONormalizerSpecificTest	3
ROViewTest	42	ROViewStackTest	3
ROBitmapGenerationTest	25	ROAbstractLayout	2
ROAbstractLabelTest	17	ROAnnouncerTest	2
ROMorphTest	14	ROArrowTest	2
ROViewStructureTest	14	ROBorderTest	2
ROEdgeTest	13	ROCircleLayoutTest	2
ROCameraTest	12	ROFlowLayoutTest	2
ROContainingElementTest	12	ROForceBasedLayoutTest	2
ROTreeMapLayoutTest	12	ROLineTest	2
ROConstraintTest	11	ROMondrianCacheTest	2
RONormalizerTest	11	RONativeExampleUtilityTest	2
ROAttachPointTest	9	ROPlatformTest	2
ROFocusViewTest	9	ROTracingCanvasTest	2
ROHighlightElementTest	9	ROAbsorbLayoutTranslatorTest	1
ROPopupTest	9	ROAbstractCanvasTest	1
RODraggableTest	8	ROCenteredLabelTest	1
ROAnimationTest	7	ROCircleTest	1
ROMondrianViewBuilderLayoutTest	7	ROCollectionTest	1
ROPopupMondrianTest	7	ROCountingVisitorTest	1
ROPopupViewTest	7	RODirectLayoutTranslatorTest	1
RoassalExporterHTMLTest	6	ROHorizontalTreeLayoutTest	1
ROEaselTest	6	ROImageTest	1
ROViewMiniMapTest	6	ROLabelTest	1
ROZOrderingTest	6	ROLayoutSteppingTest	1
ROAddNameTest	5	ROLayoutTest	1
RoassalSerializerExporterTest	5	ROLayoutTranslatorTest	1
ROBoxTest	5	ROMotionMoveTest	1
ROEventTest	5	ROOrderedCollection	1
ROMenuElementTest	5	ROParentElementResizeStrategyTest	1
ROScrollbableTest	5	ROPharoCanvasTest	1
ROShapeTest	5	RORemoveNodeTest	1
RoassalExporterSVGTest	4	ROSmoothLayoutTranslatorTest	1
ROColorTest	4	ROSugiyamaLayoutTest	1
ROExpandChildrenOnClickTest	4	ROTranslationTest	1
ROMondrianScatterLayoutTest	4	ROTreeLayoutTest	1
RORemoveEdgeTest	4	ROVisitorTest	1
RODecoratorTest	3	ROZoomOnClickTest	1
ROGraphTransformationTest	3	ROTest	0

Total unit tests: 96	**Total test methods**: 556

Tabla B.2: Número de tests definidos en cada Unit Test de Roassal

Apéndice C

Experimento: Agrupando Métodos de Test

Un problema del *approach* de *TestSurgeon* es la gran cantidad comparaciones 1-a-1 entre tests, en el caso de `ROMondrianViewBuilderTest` al contener 96 tests, se deben realizar potencialmente $\frac{1}{2} \cdot 96 \cdot (96 - 1) = 4{,}560$ comparaciones. Muchas de estas son innecesarias debido a que los tests se enfocan en partes del sistema muy distintas o sin ninguna relación. Una manera de reducir estas comparaciones es agrupar tests similares y así realizar comparaciones entre elementos de estos grupos solamente. La característica de los tests a comparar es su cobertura, ya que entre mayor sea ésta, más chances hay de encontrar un caso de refactorización/reestructuración.

Para esto se decidió utilizar el algoritmo de **Agrupación Aglomerativo Jerárquico** (o **Hierarchical Agglomerative Clustering** , **HAC**). El comienza con N clusters que contienen 1 (o clusters *unitarios*) solo elemento y en cada paso fusiona los dos clusters más cercanos repitiendo esto hasta que quede 1 cluster con N elementos formando una jerarquía. Entonces, al terminar su ejecución, el usuario puede solicitar la partición con k clusters.

Sin embargo no se sabe cuántos clusters se necesitan. Pero sí se sabe que deben tener una similitud interna suficiente para que las comparaciones sean fructíferas y no una pérdida de tiempo. La experiencia indica que para que esto suceda la similitud entre un par de tests debe ser de al menos un 80 %, por lo cual la similitud interna de cada cluster (promedio entre las similitudes de todos los pares de tests en el cluster) debe ser al menos de un 75 %.

Por otro lado, tampoco se sabe qué medida de distancia entre clusters es la más apropiada para este problema: *Single Link*, *Complete Link* o *Average Link* [1].

[1] La definición formal de cada una de la métricas de distancia se presenta en la Sección 6.5.1

C.1. Metodología

Para determinar entonces los mejores parámetros para correr el algoritmo HAC se realizó un experimento en el cual se ejecutó el algoritmo variando tanto la distancia utilizada como la similitud interna mínima exigida. Las tolerancias de similitud interna (s_{min}) variaron de la siguiente forma: 95 %, 90 %, 85 %, 80 % y 75 %.

Para la evaluación posterior de los resultados se consideraron las siguientes condiciones de una "buena agrupación":

Condición #1 El cluster no-unitario (contiene más de 1 elemento) con menor s deben tener buena similitud interna (ojalá varios puntos porcentuales más que s_{min}).

Condición #2 Los clusters no-unitarios no deben ser tan distintos en su tamaño. Es decir se debe evitar que haya un cluster gigante y muchos pequeños que es justamente el problema inicial.

Condición #3 Debe tener pocos clusters unitarios. Y los que existan deben ser suficientemente distintos a los demás clusters.

El otro criterio complementario a los anteriores es el número total de comparaciones con la nueva agrupación de test. Esta métrica asume que no se harán comparaciones de elementos entre clusters o serán mínimas. Para el caso de k clusters donde $\texttt{size}(C_\text{i})$ corresponde al número de elementos que contiene el i-ésimo cluster , el total de comparaciones se calcula como:

$$\sum_{\text{i}=1}^{k} \frac{1}{2} \, \texttt{size}(C_\text{i}) \times (\texttt{size}(C_\text{i}) - 1)$$

C.2. Resultados

Como resultado de este experimento se obtuvo 15 configuraciones representadas en pares (*distancia* , s). Cada una de éstas está graficada en forma de anillo en la Figura C.1, clasificadas verticalmente según la distancia utilizada y horizontalmente según el valor de s_{min}. En la gráfica de anillo, cada punto (o vértice del polígono) representa un cluster, y su distancia del centro del anillo representa su valor de similitud interna. Así los puntos que están en el borde externo del anillo corresponden a clusters con $s = 100\,\%$ y que generalmente corresponden a clusters que nunca se fusionaron y contienen un solo elemento.

Como era de esperar, a mayor s_{min} se obtuvieron mayor cantidad de clusters sin importar la distancia escogida. A medida que la s_{min} disminuía una cantidad menor de clusters pero más heterogéneos.

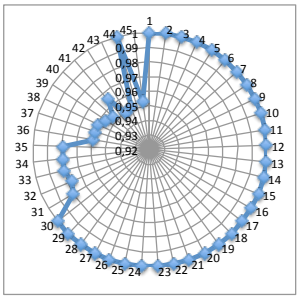
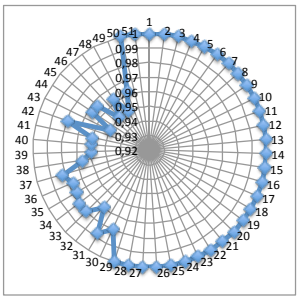
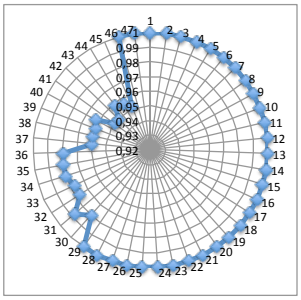
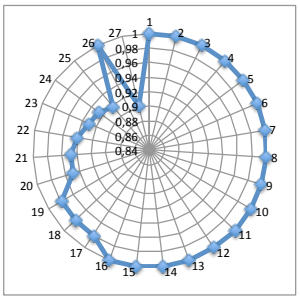
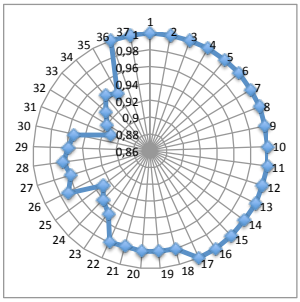
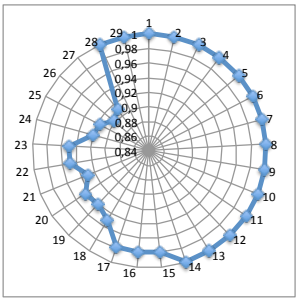
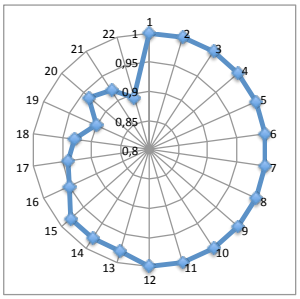
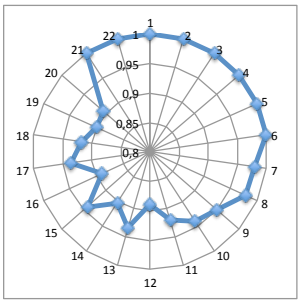
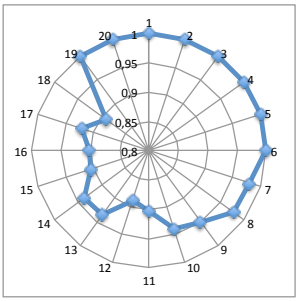
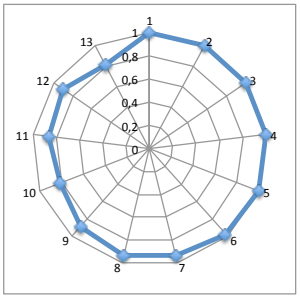
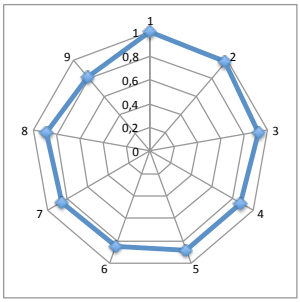
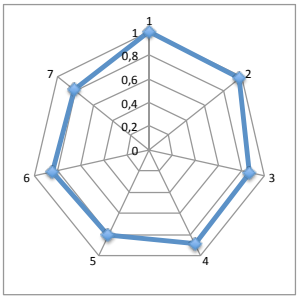
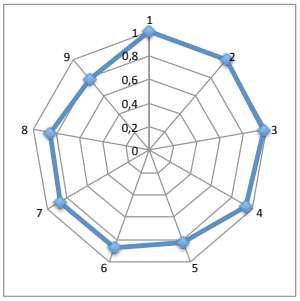
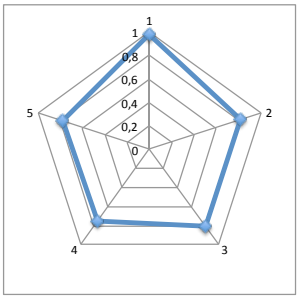
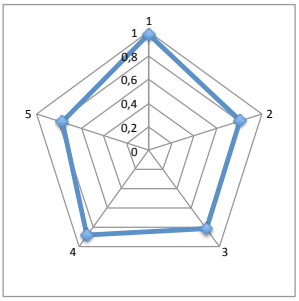
	Single Link	Complete Link	Average Link
95%			
	(b) 45 Clusters	(c) 51 Clusters	(d) 47 Clusters
90%			
	(f) 27 Clusters	(g) 37 Clusters	(h) 29 Clusters
85%			
	(j) 22 Clusters	(k) 22 Clusters	(l) 20 Clusters
80%			
	(n) 13 Clusters	(ñ) 9 Clusters	(o) 7 Clusters
75%			
	(q) 9 Clusters	(r) 5 Clusters	(s) 5 Clusters

Figura C.1: Resultados experimento de Clustering de Tests por Similitud en Cobertura

Luego de una inspección entre los distintos resultados, el parámetro de similitud interna mínima escogido fue de 85 %, ya que los tres resultados generan una cantidad de clusters razonable y con muy buena similitud interna. Entre las tres opciones, se descartó la opción (*Single Link*, 85 %) ya que la mitad de sus clusters son unitarios (condición 3), y tampoco respeta la condición 2, ya que tiene un cluster con más de 30 tests, el siguiente en tamaño 18 (casi la mitad del primero) y el siguiente 10 (casi la mitad del tercero y un tercio del primero) y luego muchos clusters pequeños.

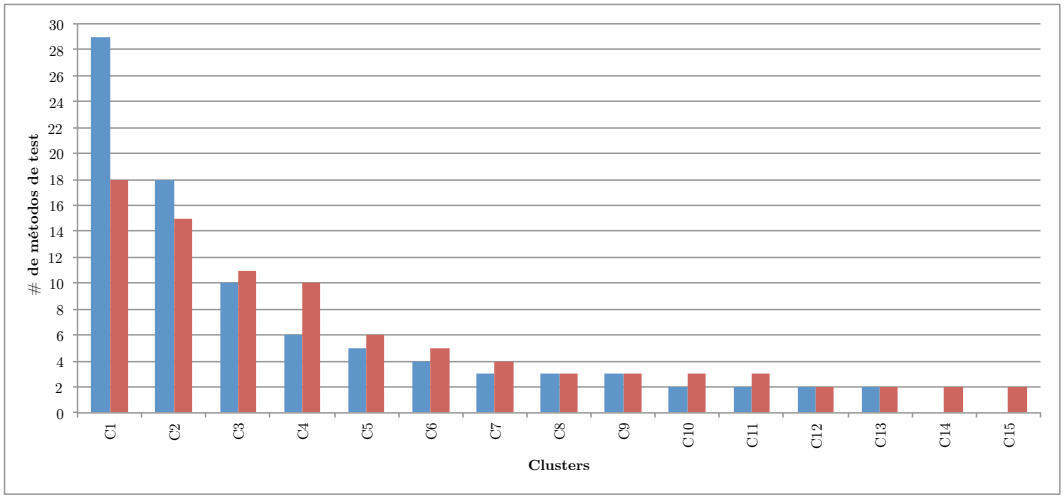
Entre las dos alternativas restantes se realizó una comparación mas minuciosa. Por un lado, la alternativa (*Complete Link*, 85 %) genera dos clusters más que (*Average Link*, 85 %) (ver Figura C.1 (k) y (l)), los dos casos presentan igual número de clusters unitarios. La Figura C.2 muestra los gráficos comparativos según tamaño y similitud interna. En ambos gráficos los clusters están ordenados de izquierda a derecha según su tamaño (número de elementos) y en orden decreciente.

La Figura C.2(a) permite revisar la condición 2. Allí se aprecia que la mejor distribución de los tests se da en (*Complete Link*, 85 %) ya que los clusters no difieren tanto en tamaño. Además en el caso de (*Average Link*, 85 %) se observa un escenario similar al de (*Single Link*, 85 %) con un cluster de gran tamaño, el siguiente la mitad del primero, y así sucesivamente. Inicialmente se pensó en el caso en que el cluster $C_{1,AL}$ del caso *Average Link* fueran los clusters $C_{1,CL}$ y $C_{2,CL}$ de *Complete Link* fusionados, pero luego de una inspección con TestSurgeon determinó que esto no era así y que de hecho los dos clusters $C_{1,CL}$ y $C_{2,CL}$ eran bastante distintos ya que la similitud entre sus elementos en general era menor al 80 % (menor que la s_{min}).

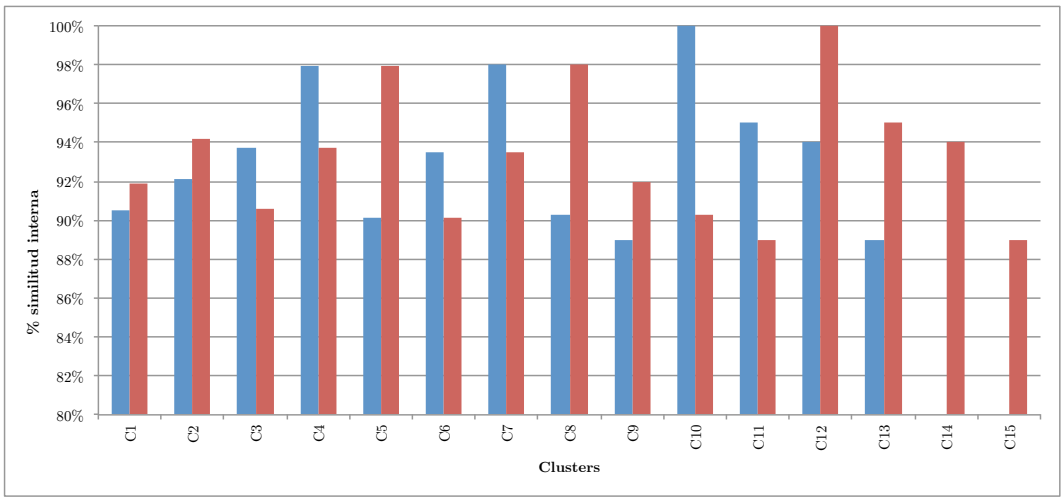
Esta mejor distribución en el caso de *Complete Link* se explica porque la métrica es más conservadora. Ya que los clusters fusionados en cada paso del algoritmo son los más cercanos considerando como distancia entre clusters los elementos más lejanos de cada cluster. Es decir los tests fusionados son realmente muy cercanos ya que la cota mínima de similitud interna es bastante exigente. En contraparte, *Average Link*, al considerar los promedios de similitudes no toma en cuenta la varianza de similitudes que existen entre las parejas de tests de los clusters.

La Figura C.2(b) permite revisar la condición 1. En este caso el contraste es aun mayor ya que en general los clusters de (*Complete Link*, 85 %) son más homogéneos, es decir, sus elementos son más similares. Y el cluster más heterogéneo ($C_{11,CL}$) tiene una similitud interna de 89 % y es el único caso. Además el 60 % de los clusters formados tiene s sobre el 92 % lo cual habla de clusters equilibrados en tamaño y muy similares entre ellos. Es decir en (*Complete Link*, 85 %) además de reducir el número total de comparaciones, los pares de tests a comparar son entre sí muy similares (en cada cluster).

Finalmente, traduciendo lo anterior a número de comparaciones se tiene: (*Average Link*, 85 %) = 648 comparaciones, y (*Complete Link*, 85 %) con **405 comparaciones**. Por lo cual la configuración escogida es **(*Complete Link*, 85 %)**.



(a) Comparación de los tamaños de cada cluster



(b) Comparación de la similitud interna de cada cluster

Figura C.2: Análisis comparativo entre los dos mejores agrupamientos
Azul: *Average Link*, 85 %
Rojo: *Complete Link*, 85 %

C.3. Conclusión

En conclusión se obtuvo una reducción de comparaciones que inicialmente eran 4.560 a 405. Luego del agrupamiento se descartó el 91 % de las comparaciones iniciales las cuales eran innecesarias puesto que la cobertura entre los elementos no daba lugar a un caso de interés. Esto significa una inmensa reducción de tiempo para el desarrollador y le permite enfocarse en los casos realmente interesantes donde existe la posibilidad de reestructuración o refactorización.