FACULTAD DE INGENIERÍA
UNIVERSIDAD DE LA REPÚBLICA

Framework para Selección de Estrategias de Testing unitario

Proyecto de Grado

María Elisa Presto Fros
mpresto@fing.edu.uy

Tutor
Diego Vallespir
dvallesp@fing.edu.uy

Instituto de computación
Montevideo, Uruguay
Diciembre, 2010

"Las fallas más notorias en la historia del desarrollo del software fueron todas debidas a defectos en las unidades, defectos que hubiesen sido encontrados con un apropiado testing unitario"

Boris Beizer

Agradecimientos

A mi familia por ser mi sostén espiritual indispensable.
A mis amigas del alma por estar siempre.
A los Ingenieros Luis y Seba por su incondicionalidad.
A Diego por el apoyo y tiempo dedicado.
A mis compañeras Mariana y Mónica por el apoyo y la experiencia brindadas.
A todos los que me acompañaron a lo largo de este proyecto.

María

Índice general

Índice de figuras

Índice de cuadros

Resumen

Si bien las pruebas unitarias ya estan establecidas en la industria de software no todos los programadores cuentan con la formación necesaria para probar adecuadamente sus unidades de código. El propósito de este trabajo es brindar al desarrollador una herramienta que lo asista en sus pruebas unitarias, brindándole información de las unidades de código que éste construye. Para esto se estudiaron e investigaron métricas de código Java, técnicas de verificación unitaria, tipos de defectos y trabajos que relacionan métricas de código como predictoras de defectos y técnicas útiles para evidenciar ciertos tipos de defectos. Con toda esta información se creó un prototipo que permite al desarrollador obtener de forma ágil información sobre unidades de código que ha desarrollado y técnicas de verificación que pueden ser útiles para probar dichas unidades.

Palabras clave: Prueba unitaria, testing de software, Ingeniería de software, técnica de verificación, métrica de código.

Capítulo 1

Introducción

La calidad del software se ha convertido en un factor imprescindible para su competitividad. El área Testing de Software ha cobrado relevancia dentro de la Ingeniería de Software como proporcionador de información sobre su calidad. Hoy en día el software está en todos lados, y cada vez mayor cantidad de personas utilizan el software en su vida diaria; el software está aumentando en cantidad y en criticidad, se utiliza software para intervenciones quirúrgicas, para pilotear un automóvil, para manejar dinero y cientos de otras aplicaciones críticas. La calidad del software surge como forma de brindar seguridad, y confianza a los usuarios de estos sistemas.

Existen diferentes niveles de pruebas de software, una posible clasificación se basa en la etapa del desarrollo en que se realizan, de aquí podemos identificar las pruebas unitarias, las pruebas de integración y las pruebas del sistema. Las pruebas unitarias son pruebas de los componentes individuales, en las pruebas de integración se prueban componentes combinados para crear componentes más grandes, y las pruebas del sistema tienen como objetivo verificar que el sistema cumple con sus requerimientos, se prueba todo el sistema integrado. En este proyecto se hace énfasis en las pruebas unitarias, su importancia radica en que el costo relativo de encontrar un defecto en las primeras etapas del desarrollo del software es menor, y en que los defectos críticos por lo general se encuentran en las unidades de código.

Este trabajo se centra en la construcción de una herramienta de apoyo al desarrollador con el fin de asistirlo en la elección de técnicas para probar las unidades de software que construye. Esta herramienta se especializa en pruebas unitarias para código Java.

Luego se debe elegir una técnica para probar las unidades de código, esta elección no es una tarea trivial. Existen diversos factores que pueden influir en la elección de las técnicas a utilizar. Como por ejemplo el tipo de software, el tamaño, el lenguaje en que fue codificado, el equipo de desarrollo. De los múltiples factores que pueden afectar la elección de una técnica de verificación se utilizaron las métricas de código y datos históricos de defectos que ha cometido el desarrollador; a partir de estas y de datos históricos de técnicas que han sido efectivas para evidenciar tipos de defectos específicos, se seleccionan técnicas para probar una unidad de código.

1.1. Motivación y objetivos

Como se mencionó es menos costoso encontrar un defecto en la fase de construcción que encontrarlo en la fase de testing del software (diseño de casos de prueba,

ejecución de caso de prueba, re trabajo) y el costo de encontrar un defecto luego de liberado el software es aún mayor (no conformidad del cliente, re trabajo).

En la figura 1.1 se ilustra el costo relativo al momento en que se evidencia un defecto en el software.

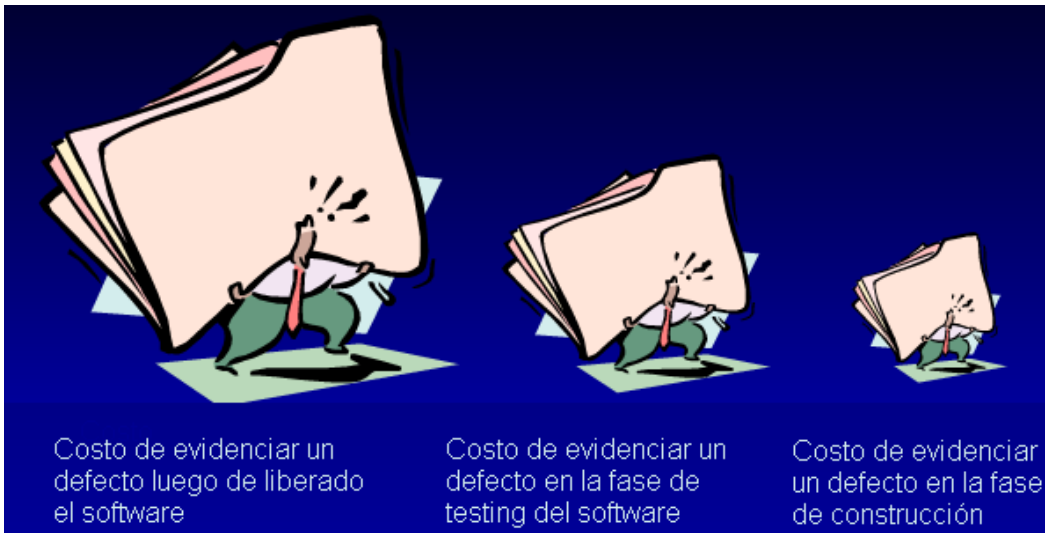

Figura 1.1: Costo relativo al momento en que se descubre un defecto

Por otro lado gran parte de los desarrolladores de software tienen una característica similar: el apego a considerar "obras primas" a sus creaciones. Con frecuencia se los escucha decir que lo que desarrollaron fue plenamente probado por ellos y no tiene defectos. El desarrollador tiende a pensar que su "obra" no contiene defectos, y muchas veces mediante un comportamiento consciente o inconsciente no encuentra sus propios errores. Se pretende ayudar al desarrollador a encontrar los defectos que contienen sus unidades de código, para suavizar esta tendencia se propone llevar un registro de errores comunes cometidos por el desarrollador. Este registro se utilizará para predecir los errores que pueda cometer el desarrollador y a partir de estos posibles errores seleccionar una técnica útil para evidenciarlos.

La elección de técnicas de verificación es un proceso complejo y que depende de multiples factores, con lo detallado en el párrafo anterior se pretende introducir características propias del desarrollador e introducir características del código, el tamaño y la estructura lógica del mismo.

Para seleccionar técnicas de verificación se utiliza la relación conocida entre medidas de código y tipos de defectos; y datos históricos de los defectos que ha generado el desarrollador, esta información debe proveerla el desarrollador.

1.2. Contexto y objetivos específicos

Para recomendar técnicas de verificación se utilizaron datos de la relación entre técnicas de verificación y tipos de defectos, métricas de código y tipos de defectos y datos históricos del programador, para esto se utilizaron 3 bases de conocimiento:

1. Una que contenga los tipos de defectos cometidos por el programador en otros desarrollos,

2. una que contenga una relación entre métricas de código y tipos de defectos,

3. y una tercer base de conocimiento conteniendo una relación entre tipos de defectos y técnicas de testing útiles para evidenciarlos.

Dados los defectos que ha cometido el programador, se obtiene de la base de conocimiento 3) posibles técnicas de testing unitario, dadas métricas del código a probar se obtiene de la base de conocimiento 2) tipos de defectos y con éstos se obtienen de la base de conocimiento 3) posibles técnicas de testing. Cabe señalar que la base de conocimiento 1) inicialmente para un programador está vacía, el mismo deberá registrar los defectos que generó para su posterior uso. En la figura 1.2 se ilustra el proceso descrito:

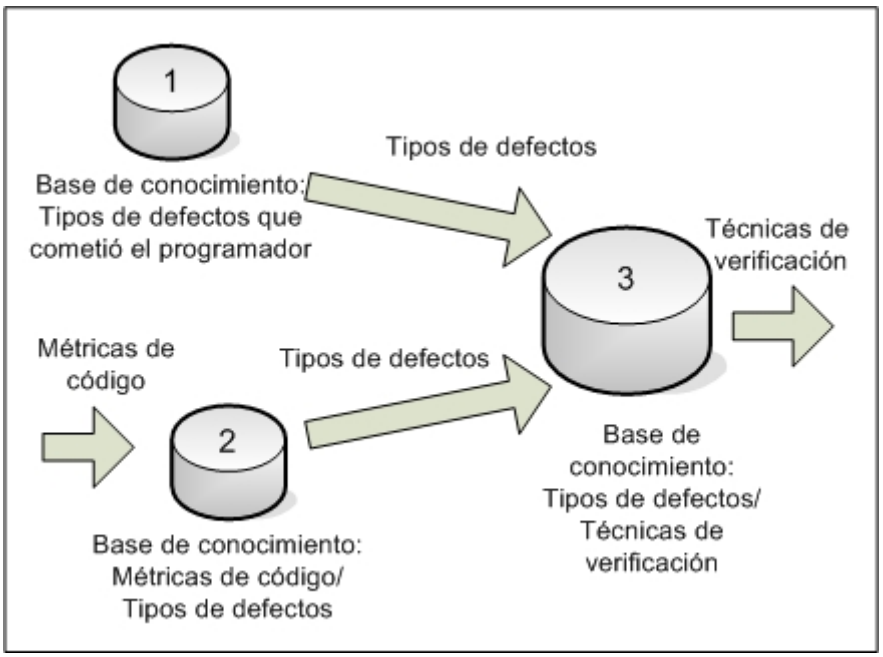

Figura 1.2: Cómo se seleccionan las técnicas de verificación

A continuación se presentan los objetivos de este trabajo:

1. Construir un framework que seleccione y sugiera técnicas de testing unitario para probar unidades de código

2. Construir un plugin para Eclipse que utilice el framework con la finalidad de brindar la funcionalidad integrada con la plataforma de desarrollo.

Al definir el alcance de la solución se determinó que el framework generará un archivo, que fue nombrado “resultados.txt” conteniendo:

1. métricas del código Java que se quiere probar,

2. datos simulados de técnicas que experimentalmente han sido útiles para probar clases con métricas similares,

3. datos simulados de técnicas útiles en evidenciar los defectos que históricamente ha contenido el código generado por el desarrollador de la unidad a probar.

Aclaración: Para los items 2 y 3 se utilizaron datos simulados debido a que actualmente no se cuenta con datos históricos y datos experimentales.

1.3. Solución propuesta

Con el fin de lograr los objetivos el trabajo se dividió en estudio e investigación, y construcción de un framework y un plugin de eclipse que utiliza dicho framework.

El estudio y la investigación se centró en métricas de código, en trabajos sobre Métricas como predictoras de defectos, en la búsqueda de herramientas libres para obtener métricas de código Java, y en técnicas de testing unitario.

Los prototipos: FSet (Framework para Selección de Estrategias de Testing unitario) y PlugSet (Plugin para la Selección de Estrategias de Testing unitario) fueron construidos en Java en la plataforma Eclipse teniendo como requerimiento no funcional relevante la flexibilidad para extensión, modificación y portabilidad.

Finalmente en conjunto con el tutor se co-dirigió una asignatura (módulo de taller) en la cuál el estudiante implementó un plug-in para Netbeans, como forma de complementar FSet. Esta implementación se ideó como modo de demostrar la portabilidad lograda para FSet.

1.4. Organización del documento

Este documento se estructura en 6 capítulos y 2 apéndices. A continuación se describe brevemente el contenido de éstos.

En el capítulo 1 se introduce al proyecto, detallando sus objetivos y qué lo motivó.

En los capítulos 2 y 3 se resumen los conceptos más importantes para comprender la solución al problema planteado, se estudian trabajos relacionados y se presentan las herramientas estudiadas para el cálculo de métricas de código.

Siguiendo en el capítulo 4, con los requerimientos del framework FSet, detalles del diseño y la implementación al problema planteado. Además se describen algunos de los problemas encontrados en el desarrollo del mismo.

El capítulo 5 consta de 3 partes, en una primera parte se brinda un marco teórico de plug-ins y plug-ins para eclipse. Luego se detallan los requerimientos, el diseño y aspectos de implementación elegidos para el desarrollo del plug-in para eclipse. Como tercera y última parte se describe el plugin desarrollado por un estudiante en forma paralela en una asignatura (Módulo de taller). Este trabajo se llevó a cabo bajo la supervisión Diego Vallespir y Mª Elisa Presto.

En el capítulo 6, se describen las conclusiones, aportes y trabajos futuros de este proyecto.

Para terminar se incluyen 2 apéndices, el apéndice A contiene el informe final del módulo de taller “Construcción de un plug-in para Netbeans” y el apéndice B contiene un tutorial de instalación de PlugSet.

Capítulo 2

Métricas de Código y Herramientas

En esta sección se describe qué son las métricas de código, algunas de ellas y cuál es su utilidad. Se describen también las herramientas que calculan métricas de código Java estudiadas para su posterior uso como predictoras de defectos.

2.1. Métricas de código

Las métricas de software son una parte importante en el proceso de desarrollo de software. Hay una gran variedad de métricas que miden diferentes atributos del software a través de todo el ciclo de vida del mismo. Una **métrica** es una forma de medir (método de medición, función de cálculo o modelo de análisis) y una escala, definida para medir [Vis].

Para gestionar un proyecto de software es de utilidad cierta información, por ejemplo, conocer la productividad de los programadores, cuáles son los recursos necesarios para satisfacer los objetivos, la complejidad del código de un sistema. Esta información puede ser de utilidad para la toma de decisiones. De aquí surge la inquietud de obtener indicadores medibles. En ingeniería de software se utilizan diversas métricas: del proceso, del producto, de recursos. Existen métricas para medir tamaño del código, la estructura de datos del código, la estructura lógica del código. En este proyecto las métricas serán utilizadas para predecir posibles tipos de defectos del código medido. Las métricas de código son dependientes del lenguaje y del estilo de programación. En este proyecto se buscó medir código Java.

Existen diversos tipos de métricas y diversas métricas. Se describen únicamente las métricas utilizadas en este trabajo.

Métricas de tamaño:

Una de las métricas más usadas es el número de líneas de código (LOCS). Aunque parece sencilla no lo es tanto, ya que existen diferentes alternativas: contar sólo las líneas de código ejecutable, añadirle las declaraciones de datos, contar las líneas de comentarios. Se puede medir la cantidad de LOCS de un método, de una clase, de todo un software. Otra medida que se suele tomar, principalmente al analizar una clase, es la cantidad de métodos que ésta define.

Estructura lógica:

Una de las métricas de estructura lógica más popular es la llamada complejidad ciclomática (NCC), ésta fue diseñada por McCabe en 1976, esta medida sirve para estimar la facilidad de testear y entender el código. Tiene su origen en la teoría de grafos, e indica el número de regiones del mismo [McC76]. Aplicada al software, es el número de caminos linealmente independientes que contiene un programa. Como tal, puede ser usada para indicar el esfuerzo necesario para testear un programa.

El grafo G se arma considerando las aristas entre nodos como las bifurcaciones en la secuencia, debido a sentencias condicionales, bucles, "saltos a". Un programa sin bifurcaciones tiene complejidad ciclomática de 1.

- NCC(G) = E - N + 2, dónde
- G es el grafo asociado al programa
- E es el número de arcos de G
- N es el número de nodos de G

Por ejemplo para el siguiente código:

```
public class HelloTesting {
  public static void main( String args[] ){

(1)   if (s.compareTo("Inglés")== 0)
      System.out.print("Hello Testing");
(2) else if (s.compareTo("Americano") == 0)
      System.out.print("Hi Testing");
     }
}
```

En el programa anterior existen saltos en cada sentencia if, en (1) y en (2). En estos 2 puntos el programa se bifurca generando caminos independientes. En la figura 2.1 se ilustra el grafo del flujo de control del programa, G contiene 5 vértices y 6 aristas. La complejidad ciclómatica NCC(G) = 6 - 5 + 2 = 3 que coincide con la cantidad de regiones generadas por G.

Cómo medir:

Es posible medir manualmente o en forma automática. A continuación se listan algunas ventajas/desventajas de ambos métodos.

- Revisión manual de código: Insume tiempo, dinero, el revisor puede cometer errores.
- Revisión automatizada de código: Insume tiempo de programación inicialmente.

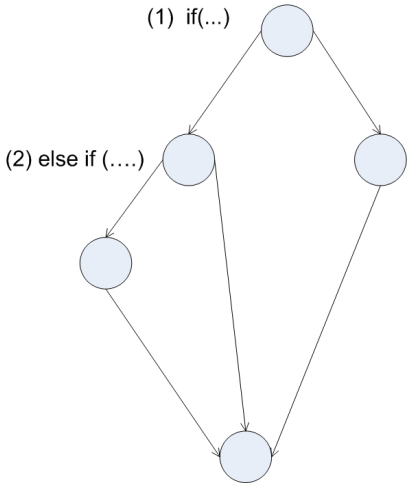

Figura 2.1: Grafo de flujo de control del programa HelloTesting

2.2. Objetivos de las mediciones

Las métricas se utilizan para diferentes propósitos, tales como predecir costos, gestionar actividades, obtener información sobre la calidad del código, medir productividad.

Se mide para comprender que sucede durante el desarrollo y el mantenimiento; Si se comprende que sucede es posible controlar y predecir posibles problemas. Todos estos esfuerzos ayudan a mejorar el proceso usado y el software.

El uso de métricas brinda al desarrollador y al equipo de desarrollo una idea de cómo es el código que han desarrollado. Utilizar la información brindada por las métricas permite obtener una mejor visión de posibles riesgos en áreas complejas del sistema y de código heredado (sí lo hubiera).

Por ejemplo, un equipo de desarrollo que identifica dónde están las partes más complejas del sistema podrá estimar tiempos, predecir y mitigar riesgos. Al detectar mediante métricas áreas “complejas” podría marcarlas para futuras revisiones, ya que normalmente son las que contienen mayor cantidad de defectos.

En otras palabras las métricas se utilizan para obtener información sobre un código fuente, que usadas de manera adecuada pueden contribuir a detectar tempranamente problemas y así reducir tiempos y costos de desarrollo.

2.3. Herramientas que calculan métricas de código de Java

En esta sección se describen las herramientas estudiadas y evaluadas. Todas son software libre, para medir código escrito en Java.

2.3.1. Evaluación y selección

En el cuadro 2.1 se muestran las herramientas estudiadas y las métricas de código que brindan. Sólo se muestran las métricas que brindan datos sobre clases, ya que este

Herramientas	NCC	Métodos por clase	locs clase	locs por método
Eclipse Metrics	X	X	X	-
JavaNCSS	X	X	X	X
Sonar	X	X	X	-
Proximity Alert	-	-	-	-
CyVis	X	X	-	X

Cuadro 2.1: Herramientas estudiadas y las métricas que brinda cada una

estudio se centra en clases y no en todo el software.

Para evaluar cada herramienta se creó un ambiente nuevo, esto implica que se instaló cada herramienta por separado y se las evaluó en forma independiente unas de otras. En una primer instancia se evaluó la factibilidad en aspectos de instalación y facilidad de uso. Luego se evaluó la correctitud de las métricas calculadas efectuando pruebas con programas sencillos.

Se describen las razones por las cuáles se seleccionó o no cada herramienta estudiada. De cada herramienta lo que interesa es:

- la facilidad de instalación,
- la posibilidad de ser invocada desde código Java,
- las características y mediciones que proporciona. En particular las métricas relacionadas con una clase.

La herramienta $EclipseMetrics$ es un plug-in que calcula diferentes métricas para código Java, permite conocer continuamente la "salud" del código. Brinda una exportación a HTML de los valores de las mediciones, incluye histogramas de los valores medidos y un espacio de trabajo para obtener un resumen visual rápido.

Luego de ser estudiada se considera que es fácil de instalar y que el cálculo de las métricas es correcto. Sin embargo fue descartada porque no es posible invocarla en el código, es utilizada como funcionalidad desde el entorno eclipse. Se estableció una comunicación vía correo electrónico con el equipo desarrollador de la herramienta para su eventual invocación desde el código. Luego de intercambiar posibles opciones, se concluyó que es necesario modificar gran parte del código de la herramienta y por esto fue descartada para este proyecto [Wal]. Es posible leer más sobre Eclipse Metrics en:

`http://eclipse-metrics.sourceforge.net`

La herramienta $ProximityAlert$ también fue descartada porqué no es posible invocarla desde el código. Se estableció una comunicación con sus desarrolladores para su posterior invocación desde el código de forma genérica. Se concluyó que no está programada con estos fines y debería modificarse su código, por lo que queda fuera del alcance de este proyecto [Cas].

Se evaluó la herramienta $Sonar$ pero no se consiguió invocarla en forma genérica desde el código. Se intentó mantener una comunicación con su desarrollador pero no fue posible.

Una de las herramientas estudiada y seleccionada es $JavaNCSS$, ésta se utilizó para obtener el número de LOCs.

JavaNCSS es una suite que calcula métricas de código Java versión 1.4 o superior. Ofrece una API que puede usarse desde la línea de comandos, o incluirse en un proyecto invocando sus funcionalidades desde el código. JavaNCSS opcionalmente puede

presentar su salida en una interfaz gráfica o en un archivo XML. Esta API es mantenida en la actualidad y su última versión se puede encontrar en la página web:

`http://www.kclee.de/clemens/java/javancss/`

Características y mediciones que proporciona

- JavaNCSS puede ser aplicada a todo el proyecto, a un método o a una clase
- Número de complejidad ciclomática (McCabe)
- Número de clases por paquete. El número de clases por paquete brinda una idea de su tamaño y responsabilidad
- Número de métodos por paquete. Es útil para analizar la responsabilidad del paquete
- Número de líneas de código sin líneas de comentarios. Esta métrica proporciona un valor (NCSS) de la cantidad de código que contiene cada clase, paquete y/o método
- Número de bloques de documentación Javadoc
- Número de líneas de comentarios por clase y método
- Cálculo de valores promedio
- Es software libre (GNU GPL).

Provee 4 librerías empaquetadas javancss.jar, ccl.jar, y jhbasic.jar y javacc.jar, para usarlas se deben añadir al ClassPath de Java.

De JavaNCSS se obtuvieron para este proyecto la cantidad de líneas de código totales de una clase y la cantidad de líneas de código por cada método que la clase declara. En este cálculo de LOCS se consideran las líneas de código ejecutable.

En un principio se iba a utilizar el número complejidad ciclomática brindado por JavaNCSS, finalmente esta medida no fue utilizada ya que al evaluar la herramienta se detectaron errores en el cálculo. Estos errores ocurren cuando el código incluye algunas sentencias como “return”. De todas formas, para evacuar algunas dudas se estableció una comunicación vía correo electrónico con Clemens Lee el contacto que proporciona la página WEB. Gracias a esta comunicación se obtuvo el código fuente y se logró comprender algunos detalles de la herramienta. Por más información es posible consultar la página WEB:

`http://iso25000.com/index.php/javancss.html`

Se continuó la búsqueda de herramientas para obtener el número de complejidad ciclomática, de esta búsqueda surgió $Cyvis$. Esta herramienta analiza el código de una clase a partir del archivo en byte code (“.class”), esto se consideró una ventaja ya que sin el correspondiente código fuente es capaz de analizar la clase. Se logró establecer una comunicación directa con el desarrollador de la herramienta quien facilitó la última versión y algunas sugerencias a tener en cuenta al momento de utilizarla. Se evaluó la herramienta, y finalmente se concluyó la correctitud y la posiblidad de obtener el número de complejidad ciclomática.

Características y mediciones que proporciona

- Mide el 100 % de las aplicaciones en Java, con soporte para Java 1.5
- Calcula las métricas a partir de archivos ".class", por lo tanto no es necesario tener el código fuente de la clase
- Calcula con múltiples hilos, lo que la hace muy eficiente
- Brinda métricas detalladas por proyecto, por paquete y por clase
- Proporciona una interfaz con tablas y gráficas, configurable
- Es posible utilizarla desde la línea de comandos
- Los resultados pueden ser exportados a HTML o a XML

2.4. Métricas como predictoras de defectos

Existen algunas investigaciones y experimentos sobre las métricas de software como predictoras de defectos. Aquí exponemos algunos de los resultados relevantes para este proyecto.

2.4.1. Estudio Jiang - Cukic - Menzies - Bartlow

Estos estudios llevados a cabo en el "Departamento de ciencias de la computación e ingeniería eléctrica de West Virginia University" tratan sobre la efectividad de modelos predictivos basados en métricas de requerimientos, de diseño y de código. Estos modelos se utilizan para predecir en qué módulos se encontrarán los defectos.

En este estudio se trabajó con trece conjuntos de datos del programa "NASA Metrics". Se concluyó que modelos predictivos basados en métricas de código resultan útiles para la verificación y validación [JCMB].

2.4.2. Estudio Nagappan - Ball - Murphy

Estudios realizados en Microsoft demostraron que los beneficios que obtiene una organización de software estimando la calidad del producto dependen de qué tan temprano en el ciclo de desarrollo se estime.

Para proporcionar estas estimaciones anticipadas:

- Se usó el caso de estudio empírico de dos grandes Sistemas Operativos comerciales: Windows XP y Windows Server 2003
- Se usaron datos históricos y métricas de código para predecir el éxito o fracaso de estos Sistemas Operativos

Este estudio demostró que datos históricos (en una línea de un producto de software) pueden ser útiles, incluso en escala como el Sistema Operativo (Windows) [NBM].

2.4.3. Estudio Fenton - Ohlsson

Miembros de "IEEE Computer Society" analizaron cuantitativamente errores y fracasos en un sistema de software complejo, y describieron una serie de resultados. Este es esencialmente un estudio cuantitativo de fallas de un software de un sistema comercial. En este análisis se centraron en la distribución de defectos, el uso de datos históricos para predecir futuros defectos y métricas para la predicción de defectos.

Se encontró una fuerte evidencia de que un pequeño número de módulos contienen la mayor cantidad de defectos. Sin embargo, en ninguno de los casos se encontró evidencia de que el tamaño o la complejidad de los módulos indique que éstos sean propensos o no a contener defectos [FO99].

2.4.4. Estudio Menzies - Greenwald - Fran

"Data Mining Static Code Attributes to Learn Defect Predictors" es un artículo publicado por la IEEE en 2007, que analiza datos de código estático utilizando inteligencia artificial y algoritmos de aprendizaje automático para predecir defectos en un código. En este estudio se plantea que en un proyecto de ingeniería de software, la extracción de tales datos puede ser útil para aprender a predecir factores de calidad del software a partir de (por ejemplo) los registros históricos de código defectuoso y no defectuoso; y plantea analizar potenciales puntos problemáticos, propensos a contener defectos, podrán entonces examinar con más detalle la verificación de modelos. Este artículo plantea la posibilidad de definir una línea de base utilizando conjuntos de datos de diferentes investigadores. Plantea que se compartan y utilicen estos datos para experimentar.

El conjunto de los mejores predictores es frágil a pequeñas variaciones en los datos, en otras palabras pequeñas variaciones en los datos pueden hacer que los mejores predictores no sean los mismos. Este resultado ofrece una nueva visión de porque los resultados previos fueron tan contradictorios, no se estaban considerando las variaciones de los datos.

Esto explica porque la cantidad de lineas de código es mejor predictora de defectos en algunos casos y que la complejidad ciclomática de McCabe en otros; debido a pequeñas variaciones en los datos.

En este artículo se concluye contrariamente al pesimismo que **obtener métricas de código estático es útil para predecir defectos.**

Las conclusiones respecto a las mejores métricas son muy frágiles, es decir, todavía no se pueden afirmar que sucede cuando se cambia el conjunto de datos. En términos más generales, la conclusión a alto nivel es que no es adecuado utilizar sólo un conjunto de datos y sólo un experimentador.

Este estudio plantea que en investigaciones futuras deberían trabajar numerosos investigadores en conjunto, y evaluar técnicas de predicción de defectos con métodos de aprendizaje automático superiores a los evaluados en este trabajo [MGF07].

2.4.5. Comentarios

Basándonos en las ideas y resultados expuestos en los artículos mencionados se decidió utilizar para la elección de técnicas de verificación unitaria:

- métricas de código e

- información histórica (La información histórica se incluye como “representante” del contexto.)

Capítulo 3

Técnicas de verificación

En este capítulo se describen conceptos básicos de las pruebas de software y se describen de algunas técnicas utilizadas para el testing unitario.

3.1. Conceptos de pruebas de software

En este capítulo se definen algunos aspectos y definiciones relacionadas con la Ingeniería del software y con las pruebas de software, útiles para el proyecto.

En la comunidad informática es común que los términos verificación y validación se utilicen indistintamente, en la Ingeniería de software se distinguen estos términos y son usados con un significado diferente. Para comprender conceptos que refieren a la calidad del software es útil tener clara la diferencia entre ellos. A continuación se exponen definiciones dadas en Sommerville [Som05]:

Verificación: Actividad que busca comprobar que el sistema cumple con los requerimientos especificados, Sommerville recomienda preguntarse: ¿El software está de acuerdo con su especificación? en cambio la **Validación:** es la actividad que busca comprobar que el software hace lo que el usuario espera, en este caso recomienda preguntarse: ¿El software cumple las expectativas del cliente?

3.1.1. Errores, defectos y fallas

Es común que en la jerga del software no se haga distinción entre estos términos. Pero existen diferencias en lo que a pruebas de software se refiere y en este contexto los tres términos son muy diferentes pero intrínsecamente relacionados.

Error: Un error es una equivocación cometida por una persona, en lo que al software se refiere podría ser un programador que introduce código defectuoso, un diseñador que comete un error diseñando.

Defecto o Falta (fault): un defecto es el resultado de un error cometido por un humano, el código contiene defectos, un documento de requerimientos, un plan de pruebas.

Falla (failure): una falla ocurre al ejecutar código defectuoso, es el resultado externo del error cometido por un humano, los defectos son el resultado interno del error cometido. Es un desvío respecto al comportamiento requerido del sistema.

En la figura 3.1 se ilustra como un error cometido por el programador introdujo un defecto que hace que el software de un aeropuerto falle y pueda implicar la caída de un

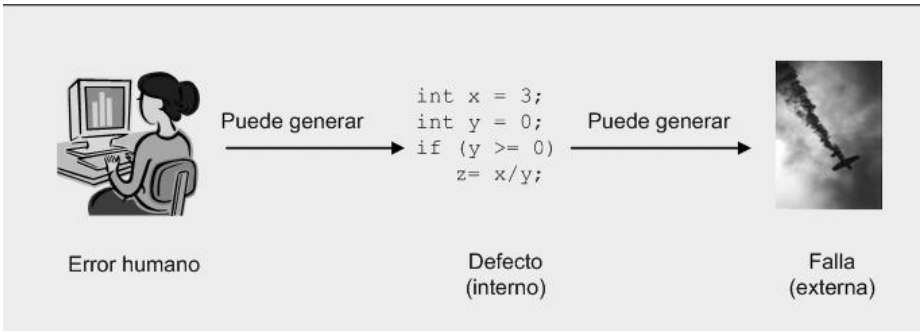

Figura 3.1: Errores, defectos y fallas

avión.

Bug El témino bug se usa para referirse a un defecto o una falla. Bug puede usarse para hacer referencia a un defecto, una falla o una limitación en el programa que le resta valor para el usuario.

Prueba o test es una actividad realizada para evaluar la calidad del producto y mejorarla, identificando defectos y problemas[Sab99].

Caso de prueba Un caso de prueba se compone de un conjunto de entradas, condiciones de ejecución y resultados esperados. Cada caso de prueba se diseña con un objetivo particular. Para determinar si la salida obtenida es correcta, debemos tener con qué comparar. Para esto usamos el **oráculo**, éste determina para cada entrada la salida esperada. El oráculo varía para cada proyecto de desarrollo de software, algunos ejemplos de oráculo pueden ser personas (con diferentes opiniones), modelos matématicos, modelos de software, diagramas (ej. UML), documentos de especificación de requerimientos, documentos de diseño, software similar, versiones anteriores del software, normas o estándares [TW].

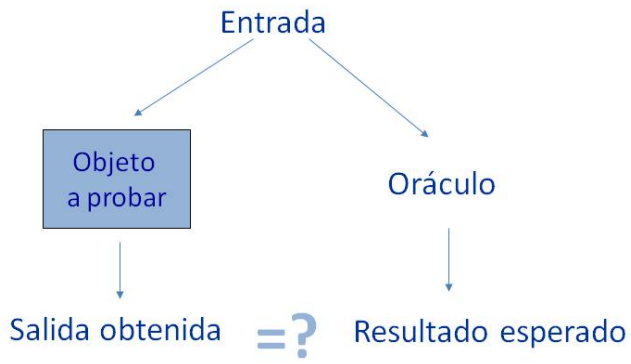

Figura 3.2: Evaluar un caso de prueba

En la figura 3.2 se ilustra cómo se evalúa un caso de prueba.

Diseño de casos de prueba Al diseñar casos de prueba, el objetivo es seleccionar del universo de "casos de prueba posibles" un subconjuto que sea eficaz para detectar defectos. Se diseñan casos de prueba a partir del conocimiento que se tiene del soft-

ware, de los problemas que imaginamos pueden existir y de los objetivos de calidad definidos.

Se buscan combinaciones de entradas y estados del sistema que tengan la más alta probabilidad de encontrar incidentes, dentro de un conjunto inmenso que no es posible probar exhaustivamente.

Es posible crear un modelo del sistema, que ayuda a simplificar y tomar los aspectos relevantes a los efectos del testing. Muchas veces el proceso de crear el modelo es lo más valioso, ya que en ese proceso es donde se aprende cómo funciona el sistema. Cuanto más modelos se conozcan, existen mayores posibilidades de optimizar las pruebas y de que las mismas sean más eficaces y efectivas.

3.2. Técnicas de pruebas unitarias

En esta sección se describen técnicas de pruebas unitarias y algunas posibles formas de clasificarlas. Se describirán 2 posibles clasificaciones, según el enfoque dinámico, estático o híbrido y según el conocimiento del código del programa.

3.2.1. Técnicas estáticas vs dinámicas

Los enfoques estático, dinámico o híbrido se usan para lograr objetivos similares, y proporcionar información sobre la calidad del software. Estos enfoques se utilizan para obtener información diferente (detectar diferentes tipos de defectos) de la calidad del software y son enfoques que se complementan. A continuación se describirán estos enfoques y algunas técnicas dentro de cada enfoque.

Técnicas estáticas (analíticas)

Las técnicas estáticas analizan el producto para comprobar si se comportará de acuerdo a lo esperado. No requieren ejecución del código a analizar. No es necesario contar con una versión ejecutable del código del programa. Dentro de las técnicas estáticas tenemos el “Análisis de código”, “Análisis automatizado de código fuente”, y la “Verificación formal”.

El *Análisis de código* consiste en revisar manualmente el código buscando defectos, esta técnica puede ser llevada a cabo en grupos o individualmente. Se recorre e inspecciona el código buscando problemas en algoritmos y otros defectos. Algunas técnicas de análisis de código son:

- Revisión de escritorio
- Recorridas e Inspecciones

El *Análisis automatizado de código fuente* utiliza herramientas de software que analizan el código, estas herramientas reciben como entrada el código fuente que se quiere analizar y como salida una serie de defectos detectados. Estas herramientas recorren código fuente y detectan posibles anomalías y faltas, mayormente analizan sintácticamente el código complementando al compilador. Detectan instrucciones mal formadas, infieren posibles defectos analizando el flujo de control.

La *Verificación formal* parte de una especificación formal y busca demostrar que el programa cumple con la misma. El objetivo de la verificación formal es demostrar que el programa es correcto a partir de una especificación formal.

Técnicas dinámicas

Las técnicas dinámicas experimentan con el comportamiento de un producto para conocer si el producto actúa como es esperado. Estas técnicas requieren ejecución del código que se quiere probar.

Las pruebas dinámicas son la verificación del comportamiento de un programa contra el comportamiento esperado, usando un conjunto finito de casos de prueba, seleccionados de manera adecuada desde el dominio infinito de ejecución [Sab99].

Ejecución simbólica

Esta técnica es un híbrido entre el enfoque estático y el dinámico. Es una técnica experimental, en la cual se ejecuta el código manualmente .

3.2.2. Técnicas de caja blanca, caja gris y caja negra

Otra forma de clasificar las pruebas es según el conocimiento del código del programa, en técnicas de:

- caja negra
- caja blanca
- caja gris

Su nombre está dado por el grado de conocimiento del código. En la figura 3.3 se ilustra la relación entre las técnicas de caja blanca, de caja gris y de caja negra.

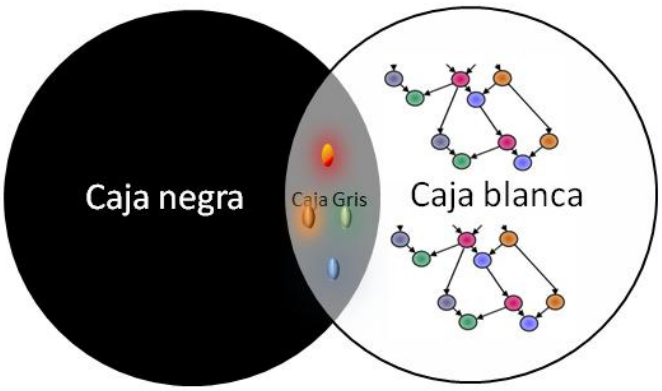

Figura 3.3: Clasificación según el grado de conocimiento del código

Definición previa: **Cubrimiento de código** cuánto código es ejercitado con las pruebas.

Las técnicas de caja blanca son técnicas que necesitan conocer el código del programa para usarlas. Las técnicas de caja blanca buscan ejercitar los caminos posibles de un código, cuando se ejecuta un camino dentro del código se cubre dicho camino. Algunas de estas técnicas son:

- **Basadas en el flujo de control del programa:** definen un criterio para ejercitar el código y según este criterio el cubrimiento que se obtiene. Expresan los cubrimientos del pruebas en términos del grafo de flujo de control del programa descrito en la sección 2.3.

  - Criterio de cubrimiento de decisión/condición
  - Criterio de cubrimiento de condición múltiple
  - Criterio de cubrimiento de trayectorias independientes

  Dependiendo del criterio de cubrimiento usado, las secciones de código que son ejercitadas por los casos de prueba.

- **Basadas en el flujo de datos del programa:** Expresan los cubrimientos de las pruebas en términos de las asociaciones definición-uso del programa

- **Mutation testing:** Es otro tipo de pruebas de caja blanca, se modifica el código inyectando defectos, el programa resultante recibe el nombre de mutante. Luego se ejecuta el conjunto de casos de prueba diseñado. Si los casos de prueba detectan los defectos inyectados se concluye que es un conjunto "útil" de casos de prueba.

Las técnicas de caja negra son técnicas que se basan en especificaciones, manuales y todo el conocimiento que se obtenga de los requerimientos de usuario para cuales deben ser los resultados esperados. Mediante esta información se determinan los resultados esperados. Luego de la ejecución de los casos de prueba se compara el resultado obtenido con el resultado esperado. Estas técnicas no usan en absoluto la estructura del código del programa.

También están las llamadas pruebas de caja gris, son una mezcla de las anteriores. Se establece un diálogo programador-tester o bien se estudia la documentación técnica para conocer algo de la estructura interna del sistema.

3.2.3. Experimentos formales en verificación de software

Culminando con la reseña a técnicas de pruebas unitarias se destaca otra forma de obtener información acerca de la calidad del código y de los defectos que éste contiene, mediante la experimentación formal. Existen numerosas investigaciones empíricas en el área, la primera de la cual se tiene conocimiento data de 1978 [Mye78].

Existen muchos experimentos formales, en el artículo *A Look at 25 Years of Data* publicado por IEEE Software, se examinan y comparan algunos de experimentos.

El artículo hace referencia a una gran dificultad para comparar experimentos, debido a que se diferentes experimentadores utilizan programas de distintos tamaños, codificados en diferentes lenguajes de programación y la documentación suele ser insuficiente, las técnicas utilizadas en la experimentación se encuentran parcialmente definidas. En este articulo se plantean estas dificultad pero no se plantea una solución [MSJV09].

El grupo de Ingeniería de software de la Facultad de Ingeniería está llevando a cabo experimentos formales, con el objetivo de obtener datos sobre la efectividad y el costo de técnicas de verificación, es posible profundizar en estos experimentos en *Effectiveness of five verification techniques* [VAL+09], *Effectiveness and cost of verification techniques: Preliminary conclusions on five techniques* [VH09].

Es importante destacar que los diferentes tipos de prueba se complementan entre sí, cada tipo de prueba es útil para evidenciar diferentes tipos de defectos. Es especialmente recomendable combinar técnicas al diseñar casos de prueba de forma de evidenciar la mayor cantidad y variedad de defectos.

3.3. JUnit

JUnit es una herramienta de pruebas unitario para código Java. Para cada clase se crea otra que es la que va a testear el funcionamiento de la anterior. JUnit nace en 1997, fue escrito por Kent Beck, precursor de XP (eXtreme Programming) y TDD (Test Driven Development) y por Eric Gamma experto en patrones de diseño en Orientación a Objetos [Far09]. Es una herramienta de código abierto es posible integrarla Eclipse y con Netbeans. Su utilización es sencilla, las clases de prueba deben heredar de TestCase y los métodos de prueba deben comenzar con el prefijo "test".

@test es una etiqueta que si se la escribe antes de un método, ésta indica que este método debe ser considerado como un caso de prueba. Los métodos deben ser **public void**, luego de ejecutar el método, si termina correctamente la ejecución se asume que la prueba fue exitosa. Cualquier excepción que lance el método es tratado como un incidente en JUnit.

Algunas validaciones disponibles para las pruebas son: assertTrue, assertEquals, assertSame, assertNull, fail(msg).

Además JUnit provee etiquetas para indicar el orden de ejecución de los casos de prueba, algunas de estas etiquetas son: @Before, @After, @BeforeClass, @AfterClass. Para más información se puede leer la página web:

`http://sourceforge.net/projects/junit/`

En la etapa de desarrollo de éste proyecto se hicieron pruebas unitarias utilizando JUnit.

3.4. Importancia de las pruebas unitarias

A menudo los equipos de desarrollo no prueban en profundidad el código que generan debido a que se sienten respaldados por el equipo de testing. El espíritu de un equipo de desarrollo de software debería apuntar a la calidad del producto que libera a testing. Si cada integrante del equipo desarrollo entrega su código con la mejor calidad que le es posible, permite a testing, centrarse en los defectos más sutiles. Un desarrollador de software que prueba sus unidades de código, conoce los errores que comente y le es posible utilizarlos para mejorar su propio trabajo.

Ventajas de las pruebas unitarias:

- Ayuda a pensar en los requerimientos y responsabilidades de cada método de cada clase que construye
- Contribuye a encontrar defectos tempranamente en el desarrollo
- El desarrollador construye productos más confiables y/o conoce las limitaciones de éste
- El retrabajo posterior del desarrollador disminuye

Capítulo 4

Framework

Definición de framework (Marco de trabajo) Es un conjunto de librerías que interactúan cooperativamente para dar solución a una necesidad específica. Un framework puede ser usado y extendido por cualquier aplicación. Se caracterizan por ser robustos y portables.

El objetivo principal de este proyecto es desarrollar un framework para seleccionar estrategias de testing unitario de forma automática a partir de código fuente escrito en Java. El nombre dado a dicho Framework es FSet (Framework para la Selección de Estrategias de Testing unitario).

Para la selección de estrategias se utilizan métricas del código que se quiere testear, datos históricos del programador, estos datos relacionan métricas del código con técnicas de verificación que han sido efectivas en clases similares (similares en relación a las métricas consideradas) y datos experimentales que relacionan métricas de código con tipos de defectos, y tipos de defectos con técnicas de verificación.

Actualmente no se cuenta con datos históricos y datos experimentales, por este motivo para esta parte del desarrollo se optó por utilizar datos simulados. En la selección que se hace en FSet se supone que la relación entre métricas, tipo de defectos, y técnicas de verificación es lineal. Esta relación es una simplificación de la realidad, la relación podría modelarse como una ecuación compleja en la que cada métrica de código y cada dato histórico del programador tienen un peso asociado. Algunas de estas posibles relaciones se presentaron en la sección 2.4.

El propósito de FSet en esta versión es generar un archivo "resultados.txt" en el que se presentan las medidas de la clase Java que se quiere probar, una lista de técnicas que históricamente han sido útiles en programas similares, los tipos de defectos que experimentalmente han sido evidenciados en clases similares y las técnicas que experimentalmente han sido apropiadas para evidenciar estos tipos de defectos. Por ejemplo para el siguiente código:

```
public class Euclides_for_test{

    public Euclides() {
        super();
    }

    public int euclides(int x, int y){
         while (x!=y){
```

```
            if (x>y)
                x=x-y;
            else
                y=y-x;
        }
        return x;
    }

    public void print (int x){
        System.out.print("El máximo común divisor es: " + x);
    }
}
```

el archivo de resultados contendrá:

```
Nombre de la clase Euclides
Locs 23

Metodos:
    Método Nombre: euclides locs: 7 complejidad ciclomática: 3
    Método Nombre: print locs: 2 complejidad ciclomática: 1
    Método Nombre: <init> locs: 2 complejidad ciclomática: 1
   (constructor)

Técnicas de testing que fueron útiles para probar clases
similares, según datos históricos:

    Técnica 1: 23

Tipos de defectos que experimentalmente presentaron clases
con características similares:

    defecto 1

Técnicas de testing que experimentalmente fueron útiles
para evidenciar los defectos descriptos anteriormente:

    técnica 1
```

Cabe mencionar que existen diversos factores que influyen en la elección de técnicas para probar una clase. Esta elección no es trivial, tampoco es posible afirmar que una técnica es “la mejor” para probar una clase dada. La opinión de un programador experimentado es de gran valor en estos casos. FSet recomienda una técnica que ha sido efectiva para probar clases similares, teniendo en cuenta la estructura y complejidad de la clase. Se hace hincapié en que ésta técnica no es la única que puede ser usada. Se recomienda usar FSet como una guía para seleccionar técnicas, pero es altamente recomendable que el programador que lo utilice para probar sus unidades tenga nociones de testing unitario.

En las siguientes secciones se presentan los requerimientos, el diseño y las principales características de implementación de FSet.

4.1. Documentación

En esta sección se describen los requerimientos del sistema, tanto funcionales como no funcionales.

4.1.1. Requerimientos funcionales

A continuación se describen las principales funcionalidades del sistema.

Cálculo de métricas de código

Dada la ruta donde se encuentra físicamente una clase Java, se deberá crear una estructura de datos que contenga las métricas de la clase. Éstas se usaran para seleccionar estrategias para probar dicha clase.

Escribir en un archivo métricas de código de una clase

Dada la ruta completa de una clase, se deberá crear un archivo "resultados.txt" que contenga las métricas a utilizar para seleccionar estrategias para probar una clase.

Obtener información histórica

Dadas métricas de código obtener del repositorio de información histórica un listado de técnicas útiles para probar una clase con estas características. Agregar esta información al archivo "resultados.txt", si no existe lo crea previamente.

Obtener tipos de defectos

Dadas métricas de código obtener del repositorio con datos experimentales un listado de tipos de defectos que han sido evidenciados en clases similares. Agregar esta información al archivo "resultados.txt", si no existe lo crea previamente.

Obtener técnicas

A partir de una lista de tipos de defectos obtener del repositorio con datos experimentales un listado de técnicas que experimentalmente han sido apropiadas para evidenciar estos defectos. Agregar esta información al archivo "resultados.txt", si el archivo no existe lo crea previamente.

4.1.2. Requerimientos no funcionales

Es deseable que el software tenga una calidad aceptable en cuanto a confiabilidad, mantenibilidad, usabilidad, portabilidad y reusabilidad.

Confiabilidad

El sistema deberá brindar las funcionalidades requeridas con un nivel de confiabilidad aceptable. Para esta primer versión del prototipo se definió suficiente que se efectuarán pruebas unitarias con JUnit3.3.

Usabilidad

El sistema deberá ser fácil de comprender, aprender y utilizar. La instalación del sistema debe poder comprenderse, configurarse y utilizarse mediante el tutorial para el usuario descrito en B. FSet debe poder usarse desde diferentes entornos de desarrollo, como mínimo en Eclipse y Netbeans.

Para cumplir con lo anterior se definió que el FSet se empaquete en formato jar de modo que sea portable y fácil de instalar.

Mantenibilidad

El sistema deberá ser capaz de adaptarse a modificaciones que se realicen tanto a nivel de cambios tecnológicos (software o hardware) como de requerimientos, e incluso de mejoras aplicadas sobre él.

Con mantenibilidad nos referimos a un diseño modular de forma que sea "simple" o poco costoso agregarle modificaciones y/o nuevos componentes.

Para cumplir con lo anterior el diseño debe pensarse en términos de reparabilidad y evolucionabilidad y buscar evitar acoplamientos poco flexibles entre distintos componentes.

Reusabilidad

Con reusabilidad nos referimos a que el software debe poder integrarse en nuevos desarrollos sin hacer modificaciones importantes en él. Que sea reusable también refiere a la facilidad de extender. Para esto se definen clases e interfaces que podrán ser re implementadas agregando el comportamiento deseado.

Portabilidad

Como último requerimiento no funcional es deseable que el software sea portable, a diferentes Sistemas Operativos. En otras palabras el sistema debe ser multiplataforma.

Para cumplir el requerimiento anterior se consideró suficiente que funcione correctamente en un entorno Windows y en un entorno Linux; se seleccionaron 2 Sistemas Operativos Windows Vista home premium y Ubuntu 8.0.

4.2. Aspectos de arquitectura y diseño

En esta sección se describen los aspectos de diseño considerados. Así como también se detalla la arquitectura y el diseño creado.

4.2.1. Patrones de diseño

Un patrón de diseño registra un problema atacado en el pasado y una forma de solucionarlo. El propósito de los patrones de diseño es reutilizar soluciones en lugar de re-descubrirlas.

Luego de analizar las características del problema se decidió aplicar los patrones de diseño Fachada, Template Method e Iterator.

Fachada

Este patrón de diseño se utiliza para acceder a parte de la funcionalidad de un sistema más complejo, definiendo una interfaz que permita acceder solamente a esa funcionalidad. Con este patrón ofrecemos un acceso sencillo y desacoplamos al máximo nuestro sistema cliente (el que accede a la fachada) de los sistemas ocultos.

Se utilizó fachada para ocultar los detalles de implementación de las clases Metricas e Historico, no accediendo directamente a ellas si no a IMetricas e IHistorico.

Template Method

Es un patrón de diseño que define una estructura algorítmica delegando la implementación a las subclases. Es decir, define una serie de pasos, en donde los pasos serán redefinidos en las subclases. Se utilizó este patrón para definir la estructura principal del método "obtenerEstrategia" este método delega parte de su comportamiento a IMetricas, a IHistorico, y a IExperimentales pudiendo éstas ser implementadas de diferentes maneras.

El esqueleto definido es el siguiente:

```
public void obtenerEstrategia(String path_java, String path_class){
    IMetricas m = new Metricas();
    m.calcularMetricas(path_java, path_class);

    IHistorico his = new Historico(m.getLocs());
    his.conectarseRepositorio();
    his.obtenerTecnicas();

    IExperimentales e = new Experimentales(m.getLocs());
    e.obtenerTecnicas();
    e.printTecnicas();

}
```

Iterator

Iterator o Iterador es un patrón de diseño que define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección. En otras palabras permite recorrer una estructura de datos sin que sea necesario conocer la estructura interna de la misma. Se utilizó el Iterador definido en la API de Java.

4.2.2. Arquitectura del sistema

Esta sección presenta la arquitectura propuesta para el sistema. Se buscó satisfacer tanto los requerimientos funcionales como los no funcionales.

Se diseñó en módulos bien definidos y poco acoplados. Esto se logró con clases especificas para interactuar con las herramientas externas y clases especificas para acceder a los repositorios de datos externos. En caso de agregar nuevas herramientas o nuevas funcionalidades es necesario extender la clase "EstrategiaTesting" redefiniendo el método "obtenerEstrategia" e implementar las interfaces IHistórico, IMétricas, IExperimentales según se quiera.

En la figura 4.1 se muestra el diagrama de clases de diseño con las clases más relevantes para la solución planteada:

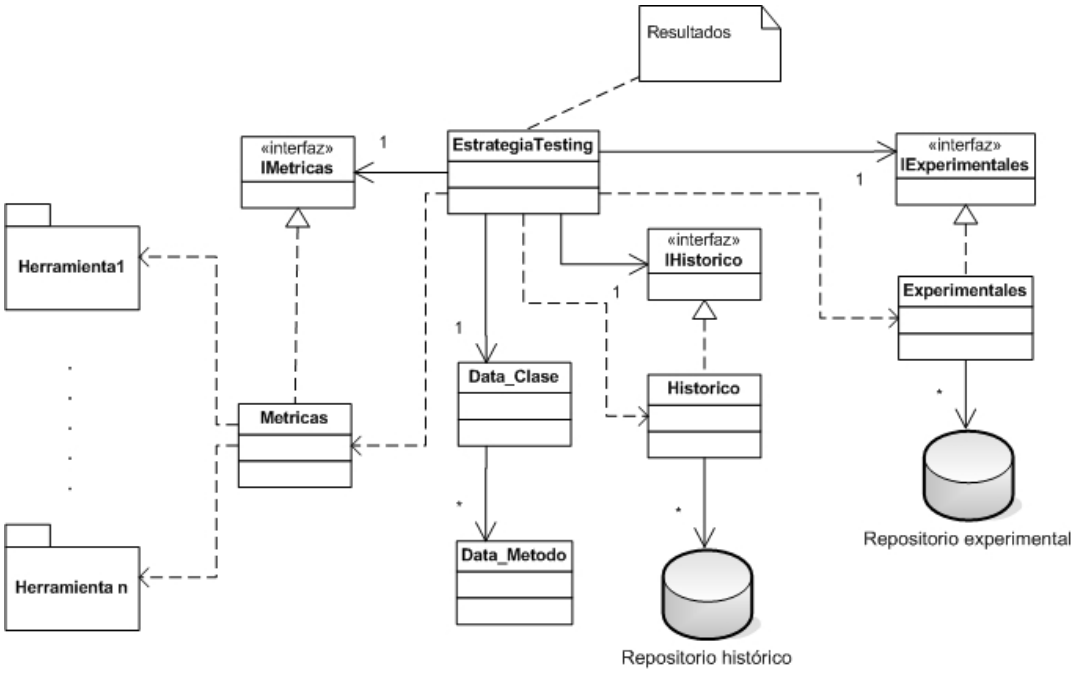

Figura 4.1: Diagrama de clases de diseño de FSet

4.3. Aspectos de implementación

A continuación se describe el lenguaje de programación y el entorno de desarrollo definido; también se describen los pasos necesarios para extender FSet.

4.3.1. Lenguaje y entorno de Codificación

El objetivo de FSet hace referencia a probar unidades de código Java y se creyó conveniente utilizar Java como lenguaje de programación. Específicamente se usó Java Development Kit 6 Update 17.

Se utilizó la última versión disponible de Eclipse al momento de comenzar la codificación Eclipse GANYMEDE, release 3.4.2.

4.3.2. Codificación

Luego de ser empaquetado en formato jar, FSet brinda la clase EstrategiaTesting para ser invocado. En la figura 4.2 se ilustra esta clase y las principales interfaces que utiliza.

El método público “obtenerEstrategia” de la clase “EstrategiaTesting” recibe como argumentos la ruta del archivo “.java” y la ruta del archivo compilado en byte code (“.class”) correspondiente a la clase que se quiere probar. Este método utiliza las Interfaces: IHistorico, IMetricas, e IExperimentales para obtener:

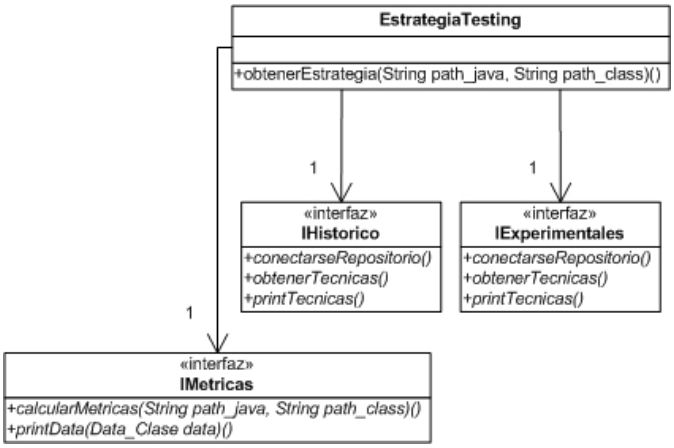

Figura 4.2: Principales clases de FSet.

- métricas de código,
- tipos de defectos posilbes,
- estrategias de verificación unitaria que históricamente fueron útiles,
- estrategias que experimentalmente fueron útiles.

La interfaz IMétricas se implementó mediante la clase Metricas, está clase utiliza las herramientas de código presentadas en el capítulo 2. Para extender o redefinir esta clase simplemente se debe implementar IMetricas, pudiendo de esta forma utilizar otras herramientas y/o otras metricas de código.

La interfaz IHistorico se implementó mediante la clase Historico, los datos Históricos utilizados son datos simulados. El objetivo de esta clase es conectarse con un repositorio de datos históricos, este puede ser una simple tabla, una base de datos o incluso un Data Mining. Luego utilizando algunas métricas de código consultar en dicho repositorio qué técnicas fueron de utilidad para probar clases Java similares. Finalmente se escribe en un archivo "resultados.txt" las técnicas obtenidas en las consultas.

La interfaz IExperimentales se implementó mediante la clase Experimentales, los datos Experimentales son datos simulados. El objetivo de esta clase es conectarse con un repositorio de datos experimentales, este puede ser una simple tabla, una base de datos o incluso un Data Mining. Luego utilizando algunas métricas de código consultar en dicho repositorio qué tipos de defectos y qué técnicas fueron de utilidad para probar experimentalmente clases Java similares. Finalmente se escribe en un archivo "resultados.txt" los tipos de defectos y las´técnicas obtenidas en las consultas.

En la sección "Extendiendo FSet" se describe con más detalle cómo extender FSet. A continuación se detallan las firmas de las operaciones que se deben desarrollar para implementar cada una de las interfaces mencionadas:

```
public interface IMetricas {

     //Dada la ruta completa de una clase Java y
     //la ruta de su correpondiente archivo compilado
     //en byte code
```

```
     //Calcula métricas de código dicha clase,
     //guarda las métricas en una estructura de datos
     //Invoca al método "printData"
     public void calcularMetricas(String path_java,
         String path_class);

     //Escribe en un archivo log.txt las métricas de código
     //contenidas en el parámetro "data"
     public void printData(Data_Clase data) throws IOException;

}


public interface IHistorico {

     //Se conecta a los repositorios de datos historicos
     public void conectarseRepositorio();

     //consulta los repositorios de datos histórico
     //PRE: Al consultar el repositorio se deben tener
     //métricas del código para utilizar en las consultas
     public void  obtenerTecnicas() throws IOException;

     //Escribe en el archivo resultados:
     //un conjunto de técnicas de verificación unitaria
     public void printTecnicas() throws IOException;



public interface IExperimentales {

    //Se conecta al repositorio de datos experimentales
    public void conectarseRepositorio();

    //Consulta el repositorio de datos experimentales
    //PRE: Al consultar el repositorio se deben tener
    //métricas del código para utilizar en las consultas
    public void  obtenerTecnicas()throws IOException;

    //Escribe en el archivo resultados:
    // un conjuntos de tipos de defectos posibles
    // un conjunto de técnicas de verificación unitaria
    public void printTecnicas() throws IOException;
}
```

Notar que los aspectos de implementación son transparentes para quien invoca a FSet, y que redefinir sus interfaces significa implementar IMetricas y/o IHistorico y/o IExperimentales. En la figura 4.3 se ilustra como se ve FSet empaquetado y que funcionalidad provee.

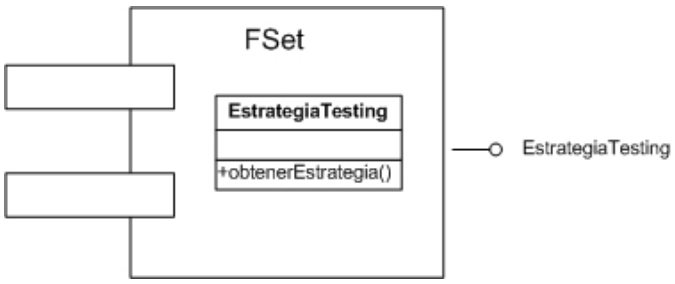

Figura 4.3: Invocar a FSet.

4.3.3. Extendiendo FSet

En esta sección se describe como extender FSet en caso de querer usar una nueva métrica de código (medir otro atributo de la clase), y/o herramientas para la selección de estrategias mediante la información histórica o información experimental.

La clase EstrategiaTesting actúa como un controlador con las interfaces IMetrics, IExperimentales e IHistorico. Ésta crea instancias de quién las implementa y las invoca según sea necesario, como se muestra en el código de la sección 4.2.1.

Para agregar nuevas herramientas que miden código y/o nuevas métricas se debe implementar IMetrics, esta clase debe almacenar las métricas calculadas en estructuras de datos para su posterior utilización. La implementación debe ser transparente externamente. Internamente se deben definir métodos para conectarse con las herramientas, se deben calcular métricas y redefinir los métodos: calcularMetricas y printData.

Posteriormente en la clase EstrategiaTesting se invocarán estos métodos, cabe destacar que puede ser necesario agregar operaciones “gets” para obtener las métricas calculadas.

Agregar una métrica

Medir un nuevo atributo de una clase puede representar un “valor numérico” o varios valores numéricos. Por ejemplo en caso de querer medir cuantas líneas de comentarios tiene la clase, la nueva métrica puede ser representada por un valor numérico. En caso de querer medir una propiedad de cada método, la nueva métrica será representada por una estructura que contenga varios valores numéricos (uno por cada método). A continuación se presenta el caso más complejo: Agregar una métrica que mida propiedades de cada método. Su complejidad radica en que es necesario agregar una estructura de datos para almacenar esta información.

La clase Metricas implementada declara:

```
public class Metricas implements IMetricas{
    private LinkedList<Data_Metodo> metodos;
    private String nombreClase;
    private int locs;
```

aquí se debería agregar un atributo (o lista de atributos) por ejemplo:

```
LinkedList<nueva_metrica> nuevasMetricas;
```

Luego implementar un método nuevo “obtenerNuevaMetrica” por ejemplo, este método invocará a una herramienta que calcula la nueva métrica, y carga los resultados en la estructura “nuevasMetricas” definida anteriormente.

Es necesario además redefinir los métodos:

```
public void calcularMetricas(String path_java, String path_class);
public void printData(data);
```

En la implementación del método “calcularMetricas” se le debe agregar la invocación a “obtenerNuevaMetrica”. El método “printData” escribe en el archivo “resultados.txt” las métricas calculadas, en su implementación además debe escribir en el archivo resultados.txt las nuevas métricas.

Luego si la métrica será usada por IHistorico y/o IExperimentales se deberán modificar los constructores de Historico y/o Experimentales. Los constructores implementados son:

```
Experimentales(locs);
Historico(locs);
```

Estos utilizan las Locs para elegir una estrategia de testing, para utilizar una nueva métrica es necesario modificarlos por ejemplo:

```
Experimentales(locs, nueva_metrica);
Historico(locs, nueva_metrica) ;
```

y además redefinir los métodos

```
obtenerTecnicas();
printTecnicas();
```

de las clases Historico y/o Experimentales para que utilicen la nueva métrica y escriban en el archivo “resultados.txt”.

Recordar que “obtenerTecnicas” consulta repositorios de datos, estos métodos posiblemente utilicen la nueva métrica en sus consultas.

Cambiar una herramienta

En caso de querer cambiar una herramienta por otra que calcule lo mismo es necesario redefinir el método correspondiente. Por ejemplo el método

```
obtenerLocs(path_java, path_class);
```

Invoca a la herramienta cyvis mencionada en el capítulo 2, que calcula las líneas de código de la clase y las almacena en el atributo Locs de la clase Metricas.

Es suficiente con redefinir el método

```
obtenerLocs(path_java, path_class);
```

de forma que calcule las locs utilizando la nueva herramienta.

Para el resto de las clases esto es transparente y por lo tanto no es necesario modificar nada.

Para obtener datos históricos se debe implementar la interfaz IHistorico, aquí se deberan definir todas las consultas al repositorio necesarias para obtener las técnicas. Las métodos relevantes en esta interfaz son conectarseRepositorio, obtenerTecnicas y

printTecnicas. Redefiniendo estas operaciones es posible cambiar el repositorio, obtener e imprimir en el archivo “resultados.txt” los datos calculados utilizando un nuevo repositorio de datos. Pudiendo agregar tanta complejidad como se quiera.

Para obtener datos experimentales se debe implementar la interfaz IExperimentales, aquí se deberan definir todas las consultas al repositorio necesarias para obtener tipos de defectos y técnicas. Los métodos relevantes en esta interfaz son conectarseRepositorio, obtenerTecnicas y printTecnicas. Redefiniendo estas operaciones es posible cambiar el repositorio, obtener e imprimir en el archivo “resultados” los datos calculados utilizando un nuevo repositorio de datos. Notar que es posible agregar tanta complejidad como se quiera.

Las interfaces IHistorico e IExperimentales necesitan las métricas de código calculadas por IMetrics para en base a ellas realizar sus consultas en los repositorios correspondientes.

4.4. Problemas encontrados y acciones tomadas

En la construcción de FSet se encontraron algunas limitaciones, en esta sección se describen algunas de ellas y las acciones tomadas al respecto.

4.4.1. Herramientas para medir código

La mayor cantidad de problemas con las herramientas surgió en la evaluación de las herramientas de métricas de código. Los problemas encontrados fueron referidos a la instalación, el uso y la correctitud de cada herramienta.

Las acciones tomadas para cada herramienta fueron:
En caso que la documentación encontrada no fuera suficiente para la instalación o para el uso de la herramienta en evaluación se enviaron correos electrónicos a los referentes de la herramienta solicitando documentación y/o recomendaciones. Para evaluar la correctitud, se definieron 3 programas sencillos (escritos en Java) con los cuales se evaluaron los resultados brindados por la herramienta.

4.4.2. Acceso a los resultados de JavaNCC

La herramienta JavaNCC seleccionada calcula como se comentó en el capítulo 2.1, complejidad ciclomática, Locs por clase, Locs por método y cantidad de métodos por clase. Retorna estos resultados en una lista de Strings, dónde cada String contiene los datos de cada método y no provee funcionalidades para acceder a cada dato. Por ejemplo para la clase Euclides mencionada anteriormente en la sección 4.4

Ésto se resolvió estudiando dicha estructura e implementando un parser para obtener cada métrica en forma aislada.

Parser: Es un analizador sintáctico. Es el proceso de análisis de un texto, constituido por una secuencia de símbolos (por ejemplo, palabras), para determinar su estructura gramatical con respecto una determinada gramática formal.

4.4.3. Empaquetado jar

Como se mencionó en la sección 4.1.2, FSet se empaquetó en formato jar de modo que sea fácil de instalar y portable. Para ésto fue necesario empaquetar toda la apli-

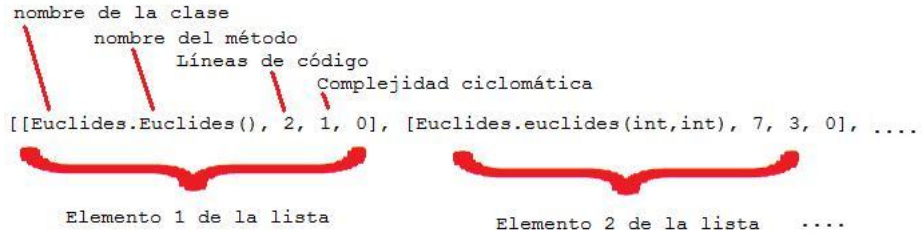

Figura 4.4: Ejemplo del resultado brindado por JavaNCC

cación, incluyendo las herramientas utilizadas para medir código seleccionadas en el capítulo 2.

Para generar un empaquetado incluyendo estas librerías externas fue necesario instalar un plugin llamado Fat, que genera el empaquetado incluyendo todas las librerías utilizadas.

Plugin Fat "Fat Jar" es un plugin para Eclipse, que genera en un único archivo "jar" incluyendo todas las clases y librerías que el software utilice. Ayuda a generar un archivo Jar auto-contenido, es decir, que incluyendo en un solo archivo todas las librerías necesarias para el funcionamiento de FSet.

4.4.4. Multiplataforma

Un inconveniente común al desarrollar aplicaciones que deban funcionar en Sistemas Operativos Windows y Linux son las rutas de los archivos. Para asegurarnos no tener problemas en este sentido se investigó la API de Java y se utilizó la funcionalidad "File.separator" para las URLs definidas en el código.

4.5. Testing

En esta sección de detallan las decisiones tomadas en relación a las pruebas unitarias y del sistema FSet.

4.5.1. Pruebas Unitarias

Se decidió probar FSet diseñando y codificando un conjunto de pruebas unitarias en JUnit 3.3. Se diseñó e implementó una clase de JUnit para probar cada clase implementada en FSet.

Se implementaron las clases "Euclides.java" y "PrintIdioma.java", estas clases se utilizaron para obtener sus métricas de código, estas clases son utilizadas por las clases test. El test suite de pruebas se ejecutó en los 2 Sistemas Operativos mencionados.

Luego de corridas dichas pruebas se corrigieron los defectos detectados. No se consideró necesario documentar los mismos.

4.5.2. Pruebas del Sistema

Para las pruebas del sistema se codificó una clase JUnit, que utiliza las funcionalidades públicas de FSet. El objetivo de estas pruebas fue que FSet cree el archivo

“resultados.txt” con los datos esperados y que funcione en los Sistemas Operativos Windows Vista home premium y Ubuntu 8.0.

Se realizaron pruebas de regresión cada vez que se terminó un componente o una funcionalidad crítica en los sistemas operativos definidos.

Cabe destacar que posteriormente en la construcción del plugin se ejecutaron pruebas utilizando FSet midiendo el código de nuevas clases. Se extenderá en 5.2.8

Capítulo 5

Plug-ins

Definición de plug-in Un plug-in es una aplicación que interactúa con otra aplicación para aportarle una función o utilidad específica. Esta aplicación adicional es ejecutada por la aplicación principal. Se utiliza como una forma de expandir programas de forma modular, de manera que se puedan añadir nuevas funcionalidades sin afectar a las ya existentes ni complicar el desarrollo del programa principal.

Otro de los objetivos de este proyecto es desarrollar un plugin para Eclipse que utilice FSet para seleccionar estrategias de pruebas unitarias. Se definió dentro del alcance del proyecto desarrollar un plugin para Eclipse y se co-dirigió el desarrollo de un plug-in para Netbeans, como forma de complementar FSet. Este desarrollo se llevó a cabo por un estudiante en forma paralela en una asignatura (Módulo de taller), que consistió en implementar un plugin para Netbeans que utiliza FSet. Se llevó a cabo bajo la supervisión Diego Vallespir, la construcción de este plugin se considera una forma de demostrar la portabilidad lograda para FSet. La figura 5.1 ilustra como interacciona FSet con diferentes plugins.

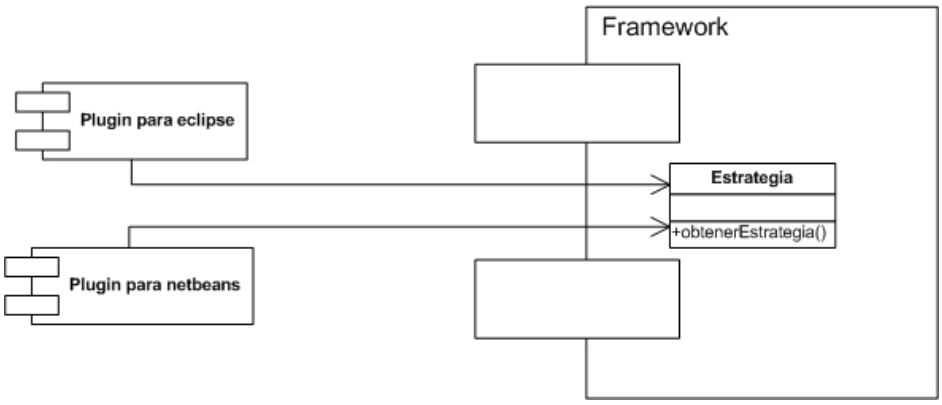

Figura 5.1: Interacción del framework FSet con diferentes plug-ins

5.1. Ventajas que ofrece un plug-in

A continuación se describen algunas ventajas del uso de plug-ins.

5.1.1. Sencillez y transparencia en la instalación

Por su arquitectura modular, un plug-in se conecta con la aplicación "anfitriona" en forma transparente, sin necesidad de configuración adicional. Por regla general, un plug-in puede ser eliminado de igual modo, sin que esto afecte al funcionamiento habitual del programa anfitrión. Para instalar un plug-in solamente se debe copiar el archivo en la carpeta indicada.

5.1.2. Integración total

Como se mencionó un plug-in se comunica con la aplicación destino utilizando los mecanismos proporcionados por ésta (y sólo estos), por este motivo la integración con la plataforma es total. El conjunto aplicación más los plug-ins son visibles al usuario como un TODO, en forma transparente. En todo momento, el usuario tendrá la sensación de estar utilizando las funcionalidades suministradas por la aplicación anfitriona mientras que en realidad se trata de un conjunto de elementos independientes trabajando en armonía.

5.1.3. Multiplataforma

Si la aplicación anfitriona se encuentra disponible para varias plataformas, la migración de los plug-ins es trivial, por regla general, utilizarán la misma interfaz de comunicación en todas las plataformas.

5.2. Plug-in para Eclipse

En esta sección se describen los requerimientos y el proceso de construcción de PlugSet (plugin para Eclipse que utiliza FSet).

5.2.1. Definiciones previas

En esta sección se definen algunos aspectos relacionados con plugins para Eclipse, útiles para comprender la construcción PlugSet. Un plugin para Eclipse es una unidad mínima de funcionalidad que puede ser distribuida de manera separada.

IDE: Entorno de desarrollo integrado

Un IDE es un entorno de programación integrado que se empaqueta como una sola aplicación. Contiene un editor de código, un compilador, un depurador y un constructor de interfaz gráfica. Su objetivo es facilitar la tarea del desarrollador integrando varias funcionalidades necesarias para el desarrollo de software.

Eclipse es un IDE abierto y extensible, (multiplataforma) que soporta una gran cantidad de plataformas Windows, Linux, Solaris, HP, Mac OS. Eclipse es una plataforma universal para la integración de herramientas de desarrollo. Contiene un editor de texto, compilación en tiempo real, resaltado de sintaxis, pruebas unitarias, control del versiones, asistentes para la creación de proyectos y refactorización.

Eclipse consta de un pequeño núcleo, el resto de las funcionalidades de la plataforma está implementada e integrada como plug-ins. Eclipse emplea plug-ins para proporcionar toda su funcionalidad, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no.

PDE: Plug-in Development Environment

PDE es el entorno de desarrollo de plugins que brinda Eclipse, el cual asiste en el desarrollo de plug-ins. El componente PDE UI provee un set de herramientas para crear, desarrollar, testear y desplegar plugins de Eclipse. PDE UI también proporciona herramientas OSGi (Estándar OSGi =>Especificación para una plataforma abierta, Orientando a proporcionar servicios), lo que lo convierte en un entorno ideal para la programación de componentes, no sólo para desarrollo de plug-ins de Eclipse [GLW08].

5.2.2. Requerimientos funcionales

A continuación se describen las funcionalidades que debe cumplir el plug-in.

Leer el archivo de configuración

La ubicación del framework FSet debe ser configurable. Para ésto se debe proveer una funcionalidad para obtener del archivo config.txt la ubicación de FSet.

Obtener la ruta de la clase

El plugin debe ser capaz de obtener la ruta completa de la clase que se desea testear, tanto la ruta del código fuente (“.java”), como del archivo compilado (“.class”).

Comunicación con FSet

Se debe poder comunicar con FSet. FSet se presenta como un ejecutable tipo jar, mediante la clase:

```
public class EstrategiaTesting{
     public EstrategiaTesting();
     public void obtenerEstrategia(String path_java, String path_class);
}
```

Se deberá invocar a FSet enviandole las rutas mencionadas. Esta invocación debe ocurrir al hacer click en botón derecho =>Testing =>Estrategia Testing sobre la clase a testear en la vista Navigator de Eclipse ilustrada en la figura 5.2.

5.2.3. Requerimientos no funcionales

A continuación se describen los requerimientos no funcionales de PlugSet, algunos de estos requerimientos se corresponden con los requerimientos de FSet.

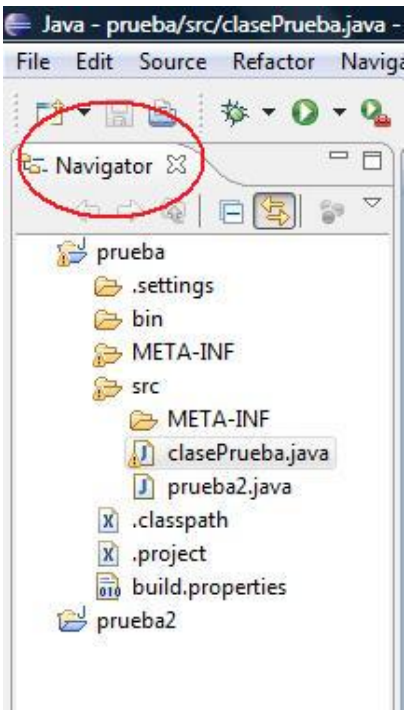

Figura 5.2: Vista Navigator de Eclipse

Portabilidad

Es deseable que el plug-in sea portable, a diferentes Sistemas Operativos. En otras palabras PlugSet debe ser multiplataforma. Para cumplir el requerimiento anterior se consideró suficiente que funcione correctamente en un entorno Windows y en un entorno Linux; se seleccionaron 2 Sistemas Operativos Windows Vista home premium y Ubuntu 8.0.

Sistema Operativo y lenguaje de codificación

Se eligió uno de los Sistemas Operativos utilizados para testear FSet, el Sistema Operativo Windows Vista home Premium; y versión de Java seleccionada para la construcción de FSet, el plug-in debe ser codificado en Java Development Kit 6 Update 17.

5.2.4. Arquitectura y diseño en Eclipse

La arquitectura plug-in de Eclipse permite escribir cualquier extensión deseada en este ambiente.

La base para Eclipse es la Plataforma de cliente enriquecido (RCP) que está compuesta por los siguientes componentes:

- OSGi: Plataforma abierta para interconexión de servicios
- Standard Widget Toolkit (SWT): Librería de componentes gráficos

- JFace: Manejo de archivos, manejo de texto, editores de texto
- Workbench de Eclipse: Vistas, editores, perspectivas, asistentes

Cómo se conectan los plug-ins

Los plugins de Eclipse se conectan mediante puntos de extensión. Un buen ejemplo para ilustrar el funcionamiento de esta característica es el de los conectores eléctricos. Un punto de extensión se compara a un toma de corriente al que conecta un enchufe, una lamparilla o cualquier dispositivo que se nos ocurra. La figura 5.3 ilustra como a un punto de extensión se le puede conectar una extensión.

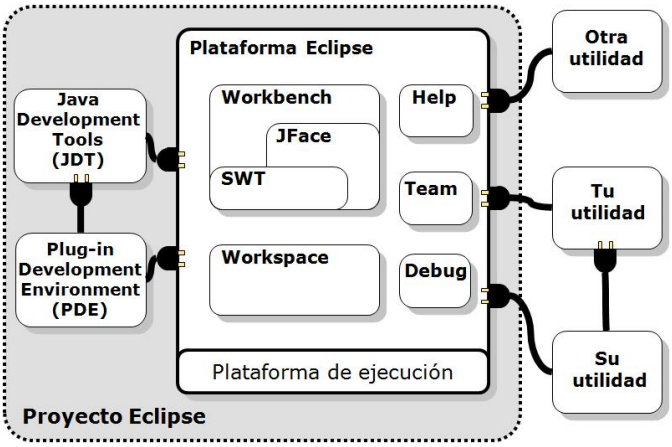

Figura 5.3: Puntos de extensión

En la definición de un punto de extensión está implícita la manera en que los componentes se conectan a él, proporcionando nuevas funcionalidades. El espacio de trabajo presenta una serie de puntos de extensión a los que los plugins pueden aportar nuevas funcionalidades o proporcionar nuevas implementaciones de editores y vistas concretas. La figura 5.4 muestra algunos de ellos indicando donde están localizados dentro del espacio de trabajo.

Cada plug-in tiene un archivo denominado “MANIFEST.MF” en el cuál se declara sus interconexiones con otros plug-ins. La interconexión sigue el modelo: un plug-in declara

- puntos de extensión
- extensiones para uno o más puntos de extensión de otros plug-ins

5.2.5. Patrones de diseño

En las secciones anteriores se especificaron los requerimientos del plug-in, y se describió la arquitectura general que utiliza Eclipse. En esta sección se detallan los principales patrones de diseño utilizados en la solución.

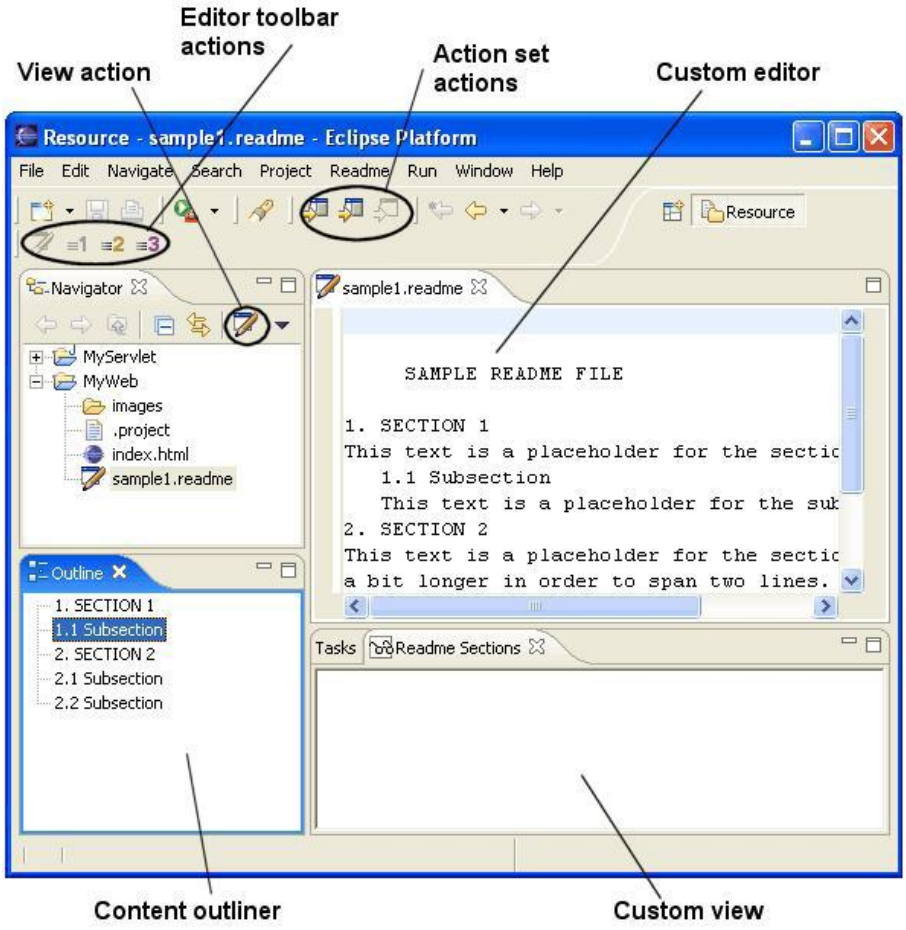

Figura 5.4: Ejemplos de puntos de extensión

Singleton

El patrón Singleton es útil cuando se quiere asegurar que una clase tenga una sola instancia y proveer un acceso global a ella. Singleton provee una operación de clase getInstance() que permite acceder a la única instancia de la clase.

La utilización de Singleton surgió de la necesidad de asegurar una única instancia de la clase "EstrategiaTesting" de FSet.jar. Esto se debe a que no está diseñado para trabajar con multiples instancias y no es posible asegurar su comportamiento en estas circunstancias.

Iterator

Este patrón fue detallado en la sección 4.2.1. Se utiliza el Iterador definido en la API de Java.

5.2.6. Diseño del plug-in

Como se describió en la sección anterior para diseñar un plug-in en Eclipse es necesario decidir como interconectar éste con la plataforma Eclipse. Para esto se debe seleccionar uno de los puntos de extensión brindado por Eclipse y conectarse a él. En la figura 5.5 se muestra el diagrama de diseño generado.

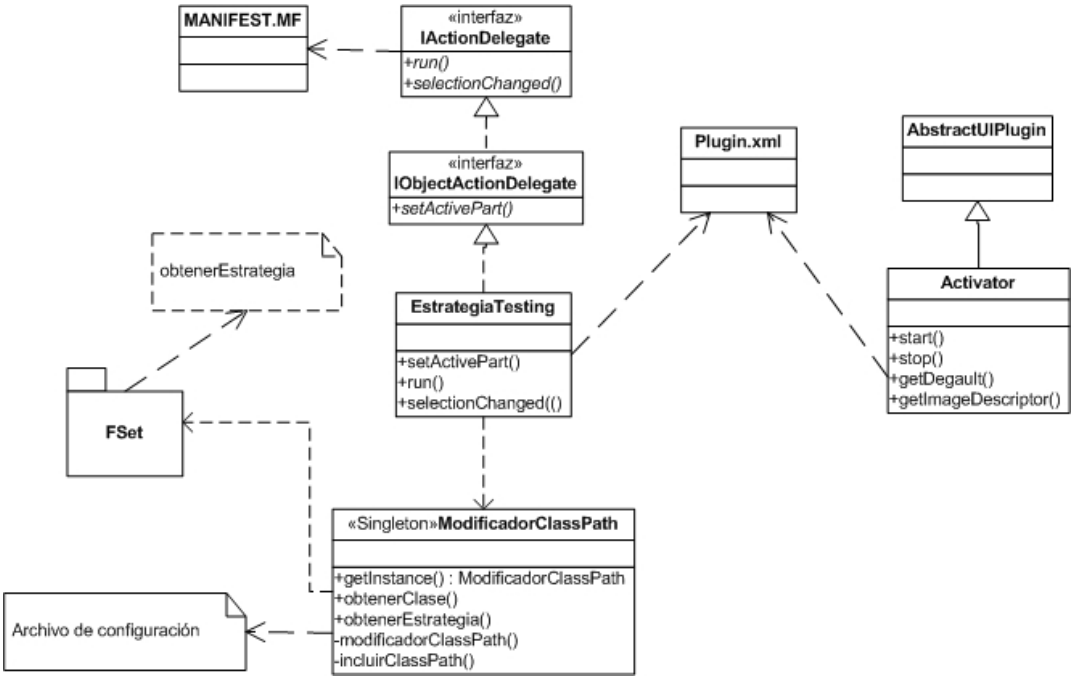

Figura 5.5: Diagrama de clases de diseño

Basandonos en los requerimientos de PlugSet, éste debe ser invocado haciendo click derecho sobre una clase en el menú Navigator de Eclipse, para esto es necesario conectarse al punto de extensión "org.eclipse.ui.popupMenus".

Para esto debemos:

- editar un archivo llamado MANIFEST.MF
- editar un archivo llamado plugin.xml
- extender la clase AbstractUIPlugin
- implementar la interfaz IObjectActionDelegate

El entorno de desarrollo PDE asiste en la genereación de estos archivos mediante templates. Se debe seleccionar qué punto de extensión se quiere extender y PDE genera en forma automática el archivo plugin.xml, el MANIFEST.MF, la clase que extiende AbstractUIPlugin y la clase que implementa IObjectActionDelegate.

Editar MANIFEST.MF

Este archivo se crea automáticamente al crear un nuevo proyecto plug-in con PDE. Dentro de cada MANIFEST, hay entradas para definir el nombre, identificador, la versión, el "activador"(nombre de la clase que activa el plug-in), el proveedor, una lista de bibliotecas que utiliza. A continuación se muestra el MANIFEST.MF definido:

```
Manifest-Version: 1.0
   Bundle-ManifestVersion: 2
   Bundle-Name: PluginTesting Plug-in
   Bundle-SymbolicName: pluginTesting; singleton:=true
   Bundle-Version: 1.0.0
   Bundle-Activator: plugintesting.Activator
   Require-Bundle: org.eclipse.ui,org.eclipse.core.runtime,
                   org.eclipse.core.resources,
                   org.eclipse.ui, org.eclipse.core.runtime,
                   org.eclipse.core.resources
   Bundle-ActivationPolicy: lazy
   Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

Editar plugin.xml

En este archivo se indica que punto de extensión se va a extender, se definen los nombres con los cuales se va a acceder al plug-in, que imagen se despliega y a que clase invoca el plug-in. En caso qué se quiera qué el plugin a su vez sea un nuevo punto de extensión para Eclipse, se define en este archivo. A continuación se muestra el contenido del archivo plugin.xml definido:

```
<plugin>
   <extension point="org.eclipse.ui.popupMenus">
      <objectContribution icon="icons/sample.gif"
            objectClass="org.eclipse.core.resources.IFile"
            id="pluginTesting.contribution1"
            setEnable = "true"
            menubarPath = "true">
         <menu
               label="Testing"
               path="additions"
               id="pluginTesting.menu1">
```

```
            <separator
                  name="group1">
            </separator>
         </menu>
         <action icon="icons/rojo.png"
               label="Estrategia Testing"
               class="uy.edu.fing.popup.actions.EstrategiaTesting"
               menubarPath="pluginTesting.menu1/group1"
               enablesFor="1"
               id="pluginTesting.newAction">
         </action>
      </objectContribution>
   </extension>
</plugin>
```

En la figura 5.6 se muestra graficamente las principales declaraciones en plugin.xml y con que clase o elemento gráfico de Eclipse está relacionada.

Extender AbstractUIPlugin

Con esta clase se controla el cíclo de vida del plug-in. Los principales métodos que se deben implementar son: “start” este método es invocado cuando es llamado el plug-in y “stop”este método es invocado cuando se para la ejecución del plug-in.

La clase que extiende AbstractUIPlugin es Activator como se ilustra en el diagrama de clases de diseño 5.5.

Implementar IObjectActionDelegate

Se debe implementar esta Interfaz para brindar comportamiento al plugin. Los métodos que se deben implementar son:

- setActivePart(IAction action, IWorkbenchPart targetPart)
- run(IAction action)
- selectionChanged(IAction action, ISelection selection)

setActivePart: En este método se establece el contexto de trabajo sobre el que se ejecutaran las acciones (como por ejemplo información inicial para el plug-in). Este método será llamado cada vez que se invoca el plug-in. Los métodos run y selectionChanged son heredados de IActionDelegate.

run: Lleva a cabo una acción. Este método es llamado cuando se invoca al plug-in. Aquí se implementa el comportamiento que se quiere obtener al invocar el plug-in.

selectionChanged: Notifica que la selección ha cambiado (la clase seleccionada en este caso). Cuando cambia la selección, el contexto de trabajo se debe actualizar según los criterios especificados en el archivo plugin.xml. A continuación, se notifica el cambio de selección. Aquí se implementa el comportamiento que se quiere obtener en este caso.

La clase EstrategiaTesting implementa esta Interfaz como se ilustra en el diagrama de clases de diseño 5.5.

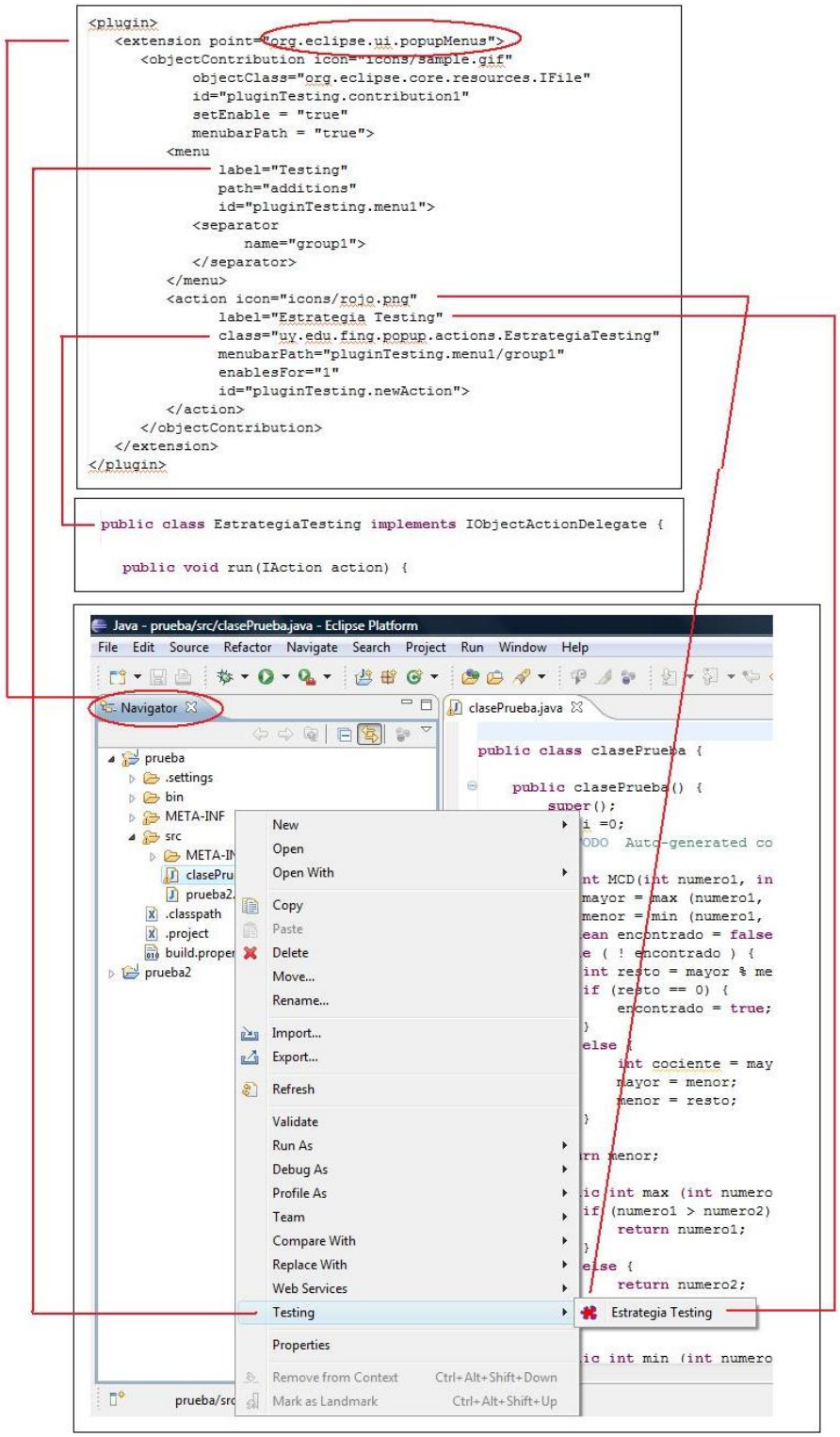

Figura 5.6: Archivo plugin.xml

Archivo de configuración y ModificadorClassPath

En tiempo de compliación de PlugSet no conoce la ubicación de FSet, la ubicación está definida en el archivo config.txt. En este archivo se encuentra la ruta completa en que reside FSet.

5.2.7. Problemas encontrados y acciones tomadas

En esta sección se describen algunos de los problemas encontrados en el diseño y la construcción PlugSet.

Falla en las rutas de archivos en Sistema Linux

Al comienzo del desarrollo del plugin se encontraron fallas al ejecutar el mismo en Ubuntu 8.0. Esta falla se debió a que las rutas de archivos en Sistemas Linux difieren de las rutas en Sistemas Windows. En Unix o linux el separador de rutas de archivos difiere del usado en Sitemas Windows. Para resolver este problema se investigó la API de Java y se utilizó la funcionalidad "File.separator" para las URLs definidas en el código.

Falla por no encontrar FSet

Se produjo una falla debida a que la ubicación de FSet no pertencía al Classpath de Java.

Definiciones previas:

- Una variable del entorno es un valor dinámico cargado en la memoria al iniciar el Sistema Operativo. Son utilizadas por el sistema operativo para la ubicación de algunas bibliotecas o de los archivos ejecutables del sistema. Un programa puede utilizarlas para encontrar la ubicación de datos u objetos en tiempo de ejecución.
- El ClassPath de Java es un conjunto de directorios que indican al JDK (Java Development Kit) dónde debe buscar los archivos a compilar o ejecutar, sin tener que escribir en cada ejecución la ruta completa.

Como se comentó en 5.2.6 en tiempo de compliación PlugSet no conoce la ubicación de FSet, la ubicación es leída del archivo config.txt. La ubicación de FSet podría no estar dentro del Classpath de Java, para poder utilizarlo es necesario incluir su ubicación en el Classpath de Java. Esto debe hacerse en tiempo de ejecución, dado que no se conoce su ubicación hasta este momento.

La clase ModificadorClassPath soluciona el problema, ésta incluye la ruta de residencia de FSet en el ClassPath de Java. En esta clase también se comunica con FSet en forma no acoplada.

Se estudiaron alternativas para resolver este problema y se decidió utilizar la API Reflexion que brinda Java.

API Reflection de Java

La API Reflection de Java brinda una manera de obtener información de una clase y/o un objeto. Por ejemplo es posible obtener de una clase el nombre, qué métodos declara, qué atributos declara. Es posible obtenerlos en tiempo de ejecución. El entorno Virtual de Java no necesita que todas las clases sean cargadas en memoria al momento al iniciar la aplicación. De esta forma, usando "reflection" es posible cargar clases en tiempo de ejecución.

Se logró crear e invocar dinámicamente a FSet. Así como también se logró incluir dinámicamente la ruta en que reside FSet en el Classpath de Java.

5.2.8. Testing

Pruebas de unitarias

Unitariamente se diseñaron pruebas para probar la correcta obtención de la ruta de la clase a probar. Estas pruebas fueron ejecutadas en los Sistemas Operativos Windows Vista home premium y Ubuntu 8.0.

Pruebas de integración

Luego se diseñaron pruebas de integración de PlugSet con FSet, en particular se ejecutaron estas pruebas en diferentes ubicaciones de FSet.jar y con diferentes clases a probar.

Pruebas del sistema

Se realizaron pruebas no funcionales de regresión cada vez que se terminó un componente o una funcionalidad crítica en los Sistemas Operativos mencionados.

Para finalizar la etapa de testing se generó y se instaló el plugin en eclispe. Se repitieron las pruebas diseñadas para probar la integración con el plugin ya instalado.

5.3. Plug-in para Netbeans

En esta sección se describe el trabajo realizado por un estudiante en forma paralela en una asignatura (Módulo de taller), este trabajo consistió en implementar un plugin para Netbeans que utiliza FSet. Este trabajo se llevó a cabo bajo la supervisión Diego Vallespir y M^{a} Elisa Presto.

Además la construcción de un plug-in en otro entorno de programación (Netbeans) contribuyó a demostrar la portabilidad lograda para FSet.

5.3.1. Requerimientos del módulo de taller

Este trabajo se realizó para contribuir al proyecto de grado Framework para Selección de Estrategias de Verificación Unitaria. Se debe construir un plugin para Netbeans que invoque a dicho framework, al presionar un botón sobre una clase en la vista Proyects de Netbeans.

El plug-in debe ser multiplataforma, como mínimo deberá funcionar en los Sistemas Operativos Windows Vista home premium y Ubuntu 8.0. Para esto se recomienda utilizar “File.separator” para el manejo de rutas de archivos.

5.3.2. Enunciado del problema

El problema a resolver por el módulo de taller fue el siguiente:

Construir un plugin para Netbeans, éste deberá ser capaz de enviar la ruta completa de la clase que se desea testear, tanto la ruta de la “clase.java”, como de la “clase.class” Se debe poder comunicar con FSet. FSet se presenta como un ejecutable tipo jar.

1. Primero deberá obtener la ruta completa de FSet.jar del archivo “config.txt”, este se encontrará en la carpeta en la cual está instalado Netbeans.

2. Luego se deberá agregar ésta ruta al ClassPath de Java y obtener una instancia de éste para poder ser invocado. (En este punto se recomienda utilizar la API Reflection de Java)

3. FSet brinda la clase pública:

```
public class Estrategia {
   public void obtenerEstrategia(String s1, String s2);
}
```

   Donde s1 es la ruta de la “clase.java” y s2 es la ruta de la clase compliada en byte code “clase.class”, éste método devuelve los resultados en un archivo “’resultados.txt’.

En el Apéndice A se presenta el informe final del módulo de taller Informe plug-in para Netbeans.

Capítulo 6

Conclusiones y trabajo a futuro

La motivación de este proyecto fue brindar al desarrollador una herramienta para asistirlo en el proceso de selección de técnicas de verificación unitaria. Invertir en evidenciar defectos en el código en etapas tempranas disminuye el costo de evidenciarlo en etapas posteriores del desarrollo del software y aumenta la calidad del producto final. Asistir al desarrollador en la tarea de evidenciar sus propios errores, mejora la calidad del código que éste produce y de todo el software.

En el presente capítulo se presentan las conclusiones y se describen los posibles trabajos futuros complementando y/o extendiendo el prototipo implementado.

6.1. Conclusiones

Se concluye que las principales contribuciones de este proyecto a los problemas planteados son:

- **Un estudio de herramientas que calculan métricas de código Java**
  Se estudiaron varias herramientas de libre distribución que calculan métricas de código Java, de cada una se evaluó la facilidad de instalación, la posibilidad de ser invocada desde código Java y las características y mediciones que proporciona.

- **Un análisis de trabajos relacionados en el área Métricas de código como predictoras de defectos**
  Se estudiaron varios trabajos que atacan la efectividad de las métricas de código como predictoras de defectos desde distintos puntos de vista.

- **Prototipo implementado**
  Se implementó un prototipo de software que se compone de un framework y un plug-in de Eclipse que utiliza dicho framework. En este proceso se investigaron nuevas tecnologías, se resolvieron problemas técnicos.

  Siendo las principales características funcionales del prototipo:

  - **FSet**
    Integración con las herramientas que miden código Java: JavaNCSS y Cyvis descriptas en la sección 2.3, se obtención de métricas de código definidas para la selección de estrategias de testing.

Simulación de datos históricos y experimentales para generar un resultado ficticio. Al momento de contar con datos reales se deberá implementar Interfaces en forma transparente como se menciona en la sección 2.1.

Diseño modular asistido por patrones de diseño facilitando su mantenimiento y extensión. Portabilidad testeada en Sistemas Operativos Windows Vista home premium y Ubuntu 8.0

- **PlugSet**
  Integración con FSet en tiempo de ejecución, PlugSet obtiene la ubicación de FSet del archivo config.txt. PlugSet se instala en la plataforma Eclipse y mediante un click derecho sobre la clase que se quiere testear, obtiene su ruta completa, invoca a FSet con ésta ruta. FSet genera el archivo resultados.txt dónde escribe las métricas del código y las técnicas sugeridas para testear dicha clase.

- **Se co-dirigió la construcción de un plug-in complementario**
  En conjunto con el tutor del proyecto se definió y supervisó un trabajo realizado por un estudiante en una asignatura (Módulo de taller), este trabajo consistió en implementar un plugin para Netbeans que utiliza el framework implementado.

6.2. Trabajos futuros

El grupo de proyecto tiene la intención que FSet se convierta en una herramienta de gran utilidad para desarrolladores de código Java en el ámbito académico.

Para esto se identificaron algunos puntos interesantes como trabajo futuro:

- **Contar con datos experimentales**
  En una primera instancia se cree necesario contar con datos experimentales reales de la relación entre tipos de defectos y técnicas de verificación y métricas de código y tipos de defectos mencionados en 1.2. Para hacer esto se identifican 2 caminos continuar estudiando artículos de otros investigadores y/o llevar a cabo experimentos formales con el fin de obtener resultados útiles a la hora de seleccionar una técnica de verificación unitaria en base a los tipos de defectos que pudiera contener el código.

- **Nuevas herramientas y/o nuevas métricas de código**
  Se cree interesante continuar con el desarrollo de FSet, integrándole nuevas herramientas que provean métricas de código para predecir defectos.

- **Datos históricos del programador**
  Cómo se comentó es interesante contar son los registros históricos del programador de la unidad de código. Sería interesante integrar una herramienta que permita al desarrollador registrar los errores que comente a lo largo de sus diferentes desarrollos. Esta herramienta se puede desarrollar o quizás integrar una existente de libre distribución que sirva para este propósito.

/

Apéndice A

Informe plug-in para Netbeans

En este apéndice se agregó el informe final del módulo de taller “Construcción de un plug-in para Netbeans” desarrollado por Nicolás Farías. Este módulo fue tutoriado por Diego Vallespir y co-tutoreado por María Elisa Presto.

A.1. Introducción

El presente documento constituye el informe de cierre del módulo de taller. Se detalla el contexto del trabajo y el desarrollo del mismo. Contexto y Objetivo El módulo de taller se desarrolló en el contexto de una ampliación al proyecto de grado “Construcción de Framework para Selección de Estrategias de Verificación Unitaria”. Proyecto en el cual se construyó un framework que a partir de una clase Java y, mediante una serie de cálculos, sugiere una estrategia de verificación unitaria para dicha clase. El objetivo del módulo de taller es construir un plugin para el IDE Netbeans (www.netbeans.org) que permita invocar fácilmente al framework mencionado. Paralelamente se venía trabajando (por parte de la tutora del módulo de taller) en la construcción de un plugin con la misma funcionalidad para el IDE Eclipse (www.eclipse.org), lo que permitió reutilizar parte de su código en la construcción del plugin para Netbeans. El siguiente diagrama ilustra la situación planteada.

A.2. Desarrollo del plugin

En esta sección se describen las desiciones más importantes tomadas en relación al estudio e implementación del plug-in.

A.2.1. Estudio

La primera parte del trabajo consistió en estudiar sobre el desarrollo de plugins para Netbeans. Para esta tarea la principal fuente de información fueron los tutoriales:

`"http://netbeans.org/kb/trails/platform.html"`

y la API de la plataforma Netbeans

`"http://bits.netbeans.org/dev/javadoc/index.html"`

.

A.2.2. Funcionamiento

El plugin básicamente consta de un “Action” de Netbeans que se invoca desde el menú contextual del IDE. De esta manera, cuando el usuario hace click derecho sobre un archivo .java en la ventana de navegación del IDE, puede invocar al framework activando la opción .Estrategia Testing". En la figura A.1 se muestra una captura de pantalla del IDE con el plug-in instalado.

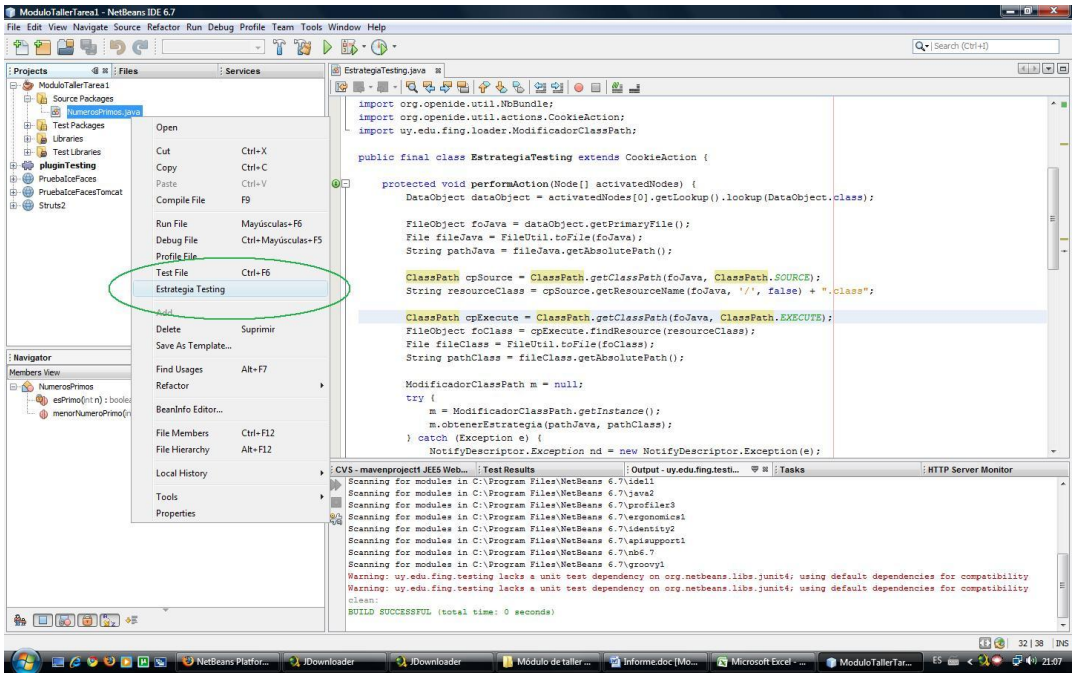

Figura A.1: Plug-in para Netbeans

A.2.3. Implementación

Si bien el código del plugin luego de completado y depurado es conciso, la dificultad mayor del trabajo estuvo en encontrar las funciones indicadas, dentro de la API de Netbeans, para realizar las tareas requeridas.

El framework a invocar, recibe como argumentos la ruta del archivo java y la ruta del archivo class correspondiente a la clase a procesar. La primer parte del código se encarga de obtener el archivo que representa la clase seleccionada, para luego obtener la ruta en el sistema de archivos. Se utiliza el método “toFile” de la clase “FileUtil” (brindada por API de Netbeans) para obtener el objeto File adecuado. Luego se utiliza el método “getAbsolutePath” de la clase File de Java para obtener la ruta absoluta del archivo en el sistema de archivos. De esta forma construimos la ruta correcta independientemente de la plataforma o sistema operativo en que se esté ejecutando. En la figura A.2.3 se detalla e´l fragmento de código correspondiente.

La siguiente sección del código utiliza la clase Classpath de Netbeans para, a partir del objeto “.java” seleccionado, encontrar el objeto .class correspondiente, es decir la clase compilada. Aquí es donde surgieron los principales problemas, dado que no

```
DataObject dataObject = activatedNodes[0].getLookup().lookup(DataObject.class);


FileObject foJava = dataObject.getPrimaryFile();

File fileJava = FileUtil.toFile(foJava);

String pathJava = fileJava.getAbsolutePath();
```

era muy claro el funcionamiento de este paquete de la API de Netbeans. El aspecto fundamental fue entender cuáles son los distintos tipos de Classpath que maneja la plataforma y qué recursos buscar en cada uno de ellos. Luego de eso, se logró una solución eficiente y elegante al problema. Es importante destacar que este mismo problema no fue posible resolverlo en Eclipse de una manera similar, lo que obligó a implementar una búsqueda más exhaustiva sobre el sistema de archivos.

```
ClassPath cpSource = ClassPath.getClassPath(foJava, ClassPath.SOURCE);

String resourceClass = cpSource.getResourceName(foJava, '/', false) + ".class";


ClassPath cpExecute = ClassPath.getClassPath(foJava, ClassPath.EXECUTE);

FileObject foClass = cpExecute.findResource(resourceClass);
```

Luego de obtenido el objeto .class correspondiente, es necesario obtener la ruta del archivo en el sistema de archivos al igual que con el archivo ".java".

```
File fileClass = FileUtil.toFile(foClass);

String pathClass = fileClass.getAbsolutePath();
```

Por último, se invoca al framework (utilizando la clase desarrollada para el plugin de Eclipse) pasándole como parámetros las rutas del archivo .java y ".class" de la clase en cuestión. En el caso que se produzca una excepción, se muestra una notificación con el mensaje de error.

A.2.4. Configuración

Para que el plugin funcione correctamente, es necesario contar con un archivo de nombre "config.txt" en el directorio "user.dir". En este archivo hay que especificar la ruta del archivo .jar correspondiente al framework que se invoca desde el plugin. Por ejemplo:

```
"C:\Users\Nicolas\Documents\Facultad\Cursos\Quinto\MT\FSet_fat.jar"
```

.

```
ModificadorClassPath m = null;

try {

    m = ModificadorClassPath.getInstance();

    m.obtenerEstrategia(pathJava, pathClass);

} catch (Exception e) {

    NotifyDescriptor.Exception nd = new NotifyDescriptor.Exception(e);

    DialogDisplayer.getDefault().notifyLater(nd);

}
```

Un detalle importante a tener en cuenta es que el directorio “user.dir”, en el caso de Windows, es el directorio en cual está instalado Netbeans. Por ejemplo,

```
"C:\Program Files\NetBeans 6.7"
```

. Sin embargo, en sistemas Linux, el directorio “user.dir” corresponde al directorio del usuario. Por ejemplo,

```
"/home/nicolas"
```

. Luego de ejecutar el plugin, el framework genera un archivo de nombre “log.txt” en el directorio “user.dir” con los resultados del procesamiento de la clase.

A.3. Investigación sobre Eclipse

Como parte del trabajo también se investigó sobre la API de Eclipse para intentar mejorar un aspecto del plugin desarrollado para ese IDE. El problema a mejorar era la obtención del archivo .class correspondiente a un archivo .java. Si bien en Netbeans ese problema fue resuelto de una manera elegante utilizando la API de Classpath que el propio Netbeans provee, no se encontró una funcionalidad similar en la API de Eclipse. Como se mencionó anteriormente, la solución alternativa consistió en implementar una búsqueda exhaustiva navegando en una sección del sistema de archivos.

A.4. Gestión de esfuerzo

A continuación se presenta en A.2 el detalle de las horas dedicadas a las distintas tareas del módulo de taller.

Fecha	Horas	Descripción
04/09/2009	2	Estudio sobre plugins en Netbeans
06/09/2009	2	Estudio sobre plugins en Netbeans
14/09/2009	2	Estudio sobre reflexión y el código del plugin construido por Elisa
15/09/2009	2	Construcción de un plugin de prueba y estudio sobre la API de Netbeans
22/09/2009	3	Acceso a los objetos de Netbeans y obtención de las rutas de los directorios de los fuentes y de los .class
30/09/2009	1	Estudio sobre API ClassPath de Netbeans para obtener las rutas
05/10/2009	1	Ajustes finales al tema de las rutas e invocación al framework
31/10/2009	4	Armado de módulo instalable de Netbeans y estudio del proyecto de Eclipse
14/11/2009	4	Estudio sobre mejora en la búsqueda de archivos en Eclipse
08/12/2009	2	Armado de informe de cierre
14/12/2009	3	Armado de informe de cierre
25/01/2010	2	Correcciones y profundización del informe de cierre
27/01/2010	2	Correcciones y profundización del informe de cierre

Figura A.2: Gestión de esfuerzo del módulo de taller

Apéndice B

Tutorial de instalación de PlugSet

PlugSet fue implementado por María Elisa Presto en el marco del proyecto de grado "Framework para la selección de estrategias de testing unitario". Se integra con FSet utilizandolo para la selección de técnicas de testing unitario.

Para instalar el plugin se debe copiar el archivo "plugSet.jar" en la carpeta plugins del paquete eclipse como se muestra en la imagen B.1.

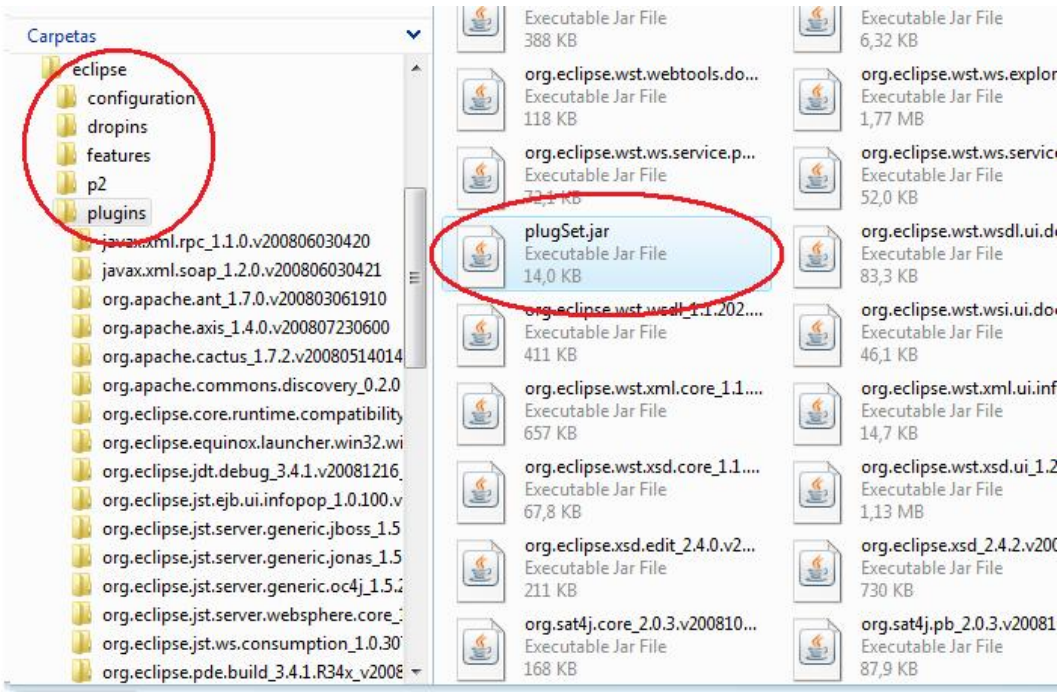

Figura B.1: Carpeta plugins de eclipse

Luego iniciar o reiniciar Eclipse, y ya queda instalado. Para visualizarlo se debe crear un proyecto, seleccionar "Windows ->Show view ->Navigator". Como se ilustra

en la figura B.2.

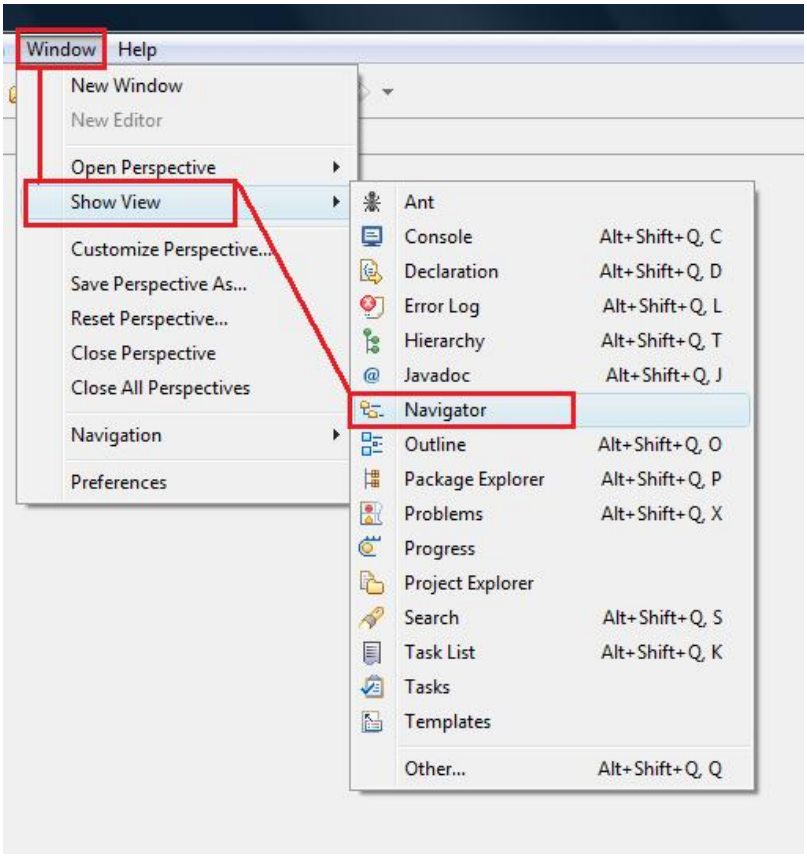

Figura B.2: Vista Navigator de eclipse

Luego hacer click derecho sobre una clase y a continuación se visualiza “Testing ->Estrategia Testing”, en la figura B.3 se muestra una captura de pantalla del IDE con el plug-in instalado.

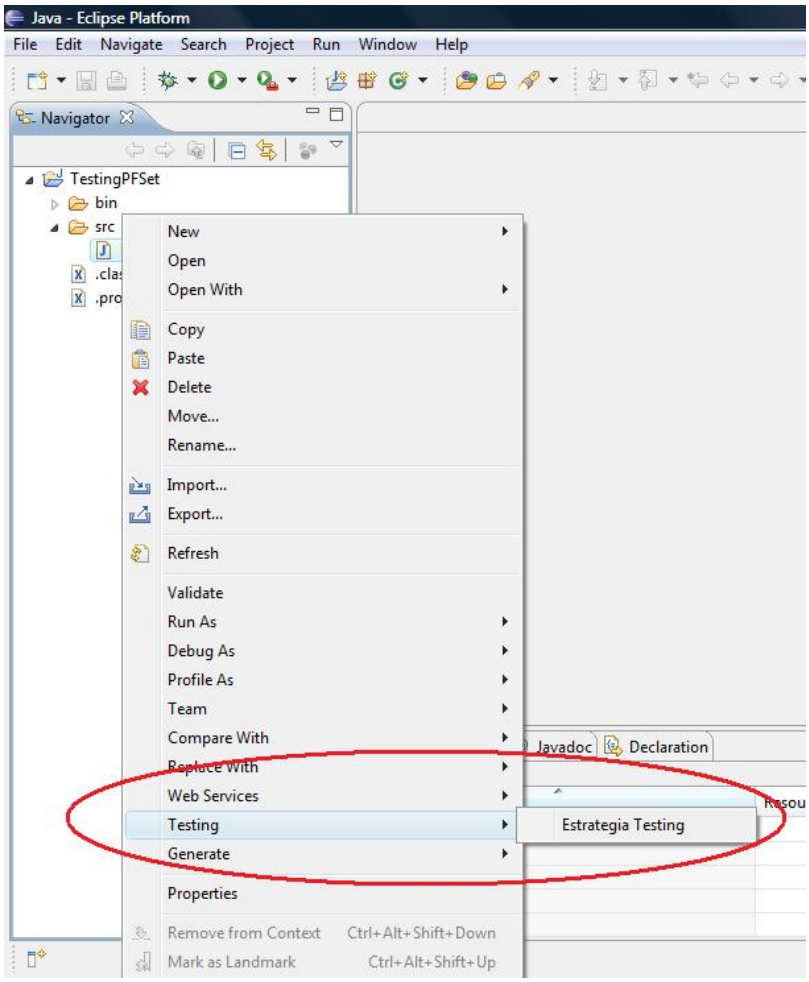

Figura B.3: Invocación al plugSet

Bibliografía

[Cas] Andrea Casarella. Proximity alert. Università della Svizzerai Italiana - Facoltà di scienze informatiche. 2.3.1

[Far09] Mauricio Farías. Testing para desarrolladores. Centro de Ensayos de Software, 2009. 3.3

[FO99] Norman E Fenton and Nicolas Ohlsson. Quantitative analysis of faults and failures in a complex software system. 1999. 2.4.3

[GLW08] Erich Gamma, LeeNackman, and John Wiegand. *Eclipse Plug-ins*. Addison-Wesley, 2008. 5.2.1

[JCMB] Yue Jiang, Bojan Cukic, Tim Menzies, and Nick Bartlow. Comparing design and code metrics for software quality prediction. 2.4.1

[McC76] T. J McCabe. A complexity measure. 1976. 2.1

[MGF07] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. 2007. 2.4.4

[MSJV09] Ana Moreno, Forrest Shull, Natalia Juristo, and Sira Vegas. A look at 25 years of data. *IEEE Software*, 26(1):15–17, Jan.–Feb. 2009. 3.2.3

[Mye78] Glenford J. Myers. A controlled experiment in program testing and code walkthroughs/inspections. September 1978. 3.2.3

[NBM] Nachiappan Nagappan, Thomas Ball, and Brendan Murphy. Using historical in-process and product metrics for early estimation of software failures. 2.4.2

[Sab99] Robert Sabourin. I am a bug. Software Engineering Body of Knowledge, 1999. 3.1.1, 3.2.1

[Som05] Ian Sommerville. *Ingenieria del Software*. Prentice Hall, 2005. 3.1

[TW] Mariana Travieso and Mónica Wodzislawski. Testing de software - casos de prueba. Centro de Ensayos de Software. 3.1.1

[VAL+09] D. Vallespir, C. Apa, S. De León, R. Robaina, and J. Herbert. Effectiveness of five verification techniques. 2009. 3.2.3

[VH09] D. Vallespir and J. Herbert. Effectiveness and cost of verification techniques: Preliminary conclusions on five techniques. 2009. 3.2.3

[Vis] Marcelo Visconti. Ingeniería de software avanzada. Universidad Técnica Federico Santa María. 2.1

[Wal] Lance Walton. Eclipse metrics plugin. 2.3.1