FACULTAD DE INFORMATICA
UNLP

TESINA DE GRADO

TITULO: Refactoring asistido mediante la detección automática de code smells

AUTOR: Matías Castilla

DIRECTOR: Alejandra Garrido

CARRERA: Licenciatura en Sistemas

Resumen

El ciclo de refactoring de código consiste en tres etapas: primero se identifica una parte del sistema cuyo diseño es deficiente, luego se elige el refactoring que resuelve el problema y, finalmente, se transforma el código fuente. Para asistir a los programadores en esta tarea, todos los IDEs modernos cuentan, en mayor o menor medida, con facilidades para aplicar refactorings de forma automática. Sin embargo, esta automatización se centra en la etapa final –es decir, en la transformación del código– cubriendo parcialmente, o no cubriendo en absoluto, la detección del problema y la selección del refactoring adecuado.

En esta tesis se estudia la posibilidad de asistir al programador a lo largo de todo el ciclo, identificando automáticamente los problemas de diseño –definidos en base a la noción de code smell y detectados por medio de reglas lint–, sugiriendo los refactorings que podrían solucionarlos y ejecutando la transformación del código de forma interactiva. Como prueba de concepto, se desarrolla una extensión del ambiente Pharo Smalltalk que provee refactoring asistido en virtud de la vinculación del framework Small Lint con el framework de refactoring.

Líneas de Investigación

- Refactoring
- Métricas
- Análisis de código
- Metodologías ágiles

Conclusiones

Mediante el análisis de la bibliografía y el desarrollo de la herramienta se demuestra que, dentro de ciertas limitaciones planteadas por la naturaleza del concepto de code smell y del análisis estático de código, es posible detectar fallas de diseño y guiar al programador en su solución. La herramienta desarrollada es fácilmente extensible y sirve como base para continuar la investigación en distintas direcciones.

Trabajos Realizados

- Documentación analítica de los frameworks Small Lint y de Refactoring de Pharo Smalltalk.
- Mejoras al diseño del framework Small Lint mediante la introducción de un modelo de fallas de diseño.
- Extensión de los frameworks de Small Lint y Refactoring con nuevas reglas lint y refactorings.
- Diseño e implementación de una herramienta de refactoring asistido.

Trabajos Futuros

- Introducción en Pharo Smalltalk de un modelo de base de datos de programa y métricas, para facilitar la detección de code smells.
- Mayor integración de la herramienta con el ambiente de desarrollo de Pharo.
- Visualización gráfica de los problemas de diseño.

Fecha de la presentación: Octubre de 2014

Índice

1. Introducción …………………………………………………………………….. 8
   1.1. Contexto ………………..……………………………………………….. 8
   1.2. Motivación ……………………………………………………………….. 8
   1.3. Objetivo ………………………………………………………………….. 10
   1.4. Contribuciones …………..………………………………………………….. 11
   1.5. Organización de la tesis ……………………………………………….. 11
   1.6. Convenciones …………..………………………………………………….. 12

2. Trabajos relacionados ……..…………………………………………………... 14
   2.1. Background ……………………………………………………………….. 14
      2.1.1. Refactoring ……..……………………………………………….. 14
      2.1.2. Refactoring browser ..…………………………………………...... 15
      2.1.3. Code smells y análisis de código ………………………………… 16
   2.2. Trabajos de investigación ………………………………………………… 18
      2.2.1. Refactorización basada en búsqueda: Code-Imp ……………… 18
      2.2.2. Estrategias de detección: reglas basadas en métricas para la detección de fallas de diseño ……………………………….. 18
      2.2.3. QA en Java mediante la detección de Code Smells ……………. 19
   2.3. Herramientas ……………………………………………………………… 20
      2.3.1. IntelliJ IDEA ……………………………………………………... 20
      2.3.2. iPlasma ……………………………………………………………. 21
      2.3.3. inFussion e inCode ……………………………………………… 21
      2.3.4. Cincom VisualWorks …………………………………………….. 22

3. Arquitectura de base …………………………………………………………… 23
   3.1. El framework de refactoring ………………………………………….. 23
      3.1.1. Uso de la herramienta ………………………………………….. 23
      3.1.2. Creación de un nuevo refactoring ………………………………… 25
      3.1.3. Refactorings ……………………………………………………… 26
      3.1.4. Condiciones ………………………………………………………. 28
      3.1.5. Ejecución de las transformaciones: los cambios y el modelo …. 30
      3.1.6. Reescritura de código ……………………………………………… 31
      3.1.7. Patrones de diseño ………………………………………………. 32
   3.2. Small Lint ……………………………………………………………………. 34
      3.2.1. El framework de reglas lint ………………………………………… 35
      3.2.2. Ejemplo: Refused Bequest ………………………………………… 36
      3.2.3. Ejemplo: Long Class ……………………………………………….. 37

3.2.4. Ejecución de las reglas 38
3.2.5. Patrones de diseño 39

4. Marco de trabajo 40
4.1. Definición de Code Smell 40
4.2. Relación entre code critics de Pharo y code smells 42
4.3. Code Smells catalogados 45
4.3.1. Dead Code 45
4.3.2. Long Parameter List 45
4.3.3. Large Class 46
4.3.4. Long Method 46
4.3.5. Message Chain 47
4.3.6. Duplicate Code 47
4.3.7. Temporary Field 48
4.4. Code Smells no catalogados 48
4.4.1. Inconsistent method classification 48
4.4.2. Excessive inheritance depth 49
4.4.3. Method defined in all subclasses but not in superclass 49
4.4.4. Rewrite super messages to self messages when both refer to same method 49
4.4.5. Refer to class name instead of self class 50
4.4.6. Variable is only assigned a single literal value 50
4.4.7. Resumen de asociación de Code Critics con Refactorings predefinidos 51
4.5. Code smells catalogados (Fowler) no implementados en Pharo 51
4.5.1. Duplicate Code 51
4.5.2. Divergent Change 52
4.5.3. Shotgun Surgery 52
4.5.4. Feature Envy 53
4.5.5. Data Clumps 53
4.5.6. Primitive Obsession 53
4.5.7. Switch Statements 54
4.5.8. Parallel Inheritance Hierarchies 54
4.5.9. Lazy Class 54
4.5.10. Speculative Generality 54
4.5.11. Middleman 55
4.5.12. Inappropriate Intimacy 55
4.5.13. Alternative Class with different Interfaces 55
4.5.14. Data Class 55
4.5.15. Refused Bequest 55

4.5.16. Comments ………………………………………………………………... 56
4.5.17. Resumen de Code Smells no detectados mediante Code Critics ………………………………………………………….. 57

5. Diseño de la herramienta ……………………………………………………….. 58
5.1. Diseño ………………………………………………………………………… 58
5.1.1. Asociación de code critics con refactorings ………………………. 58
5.1.2. Aplicación de un refactoring a partir del resultado de la ejecución de un critic …………………………………………………… 59
5.1.3. Precondiciones ……………………………………………………….. 64
5.1.4. Parámetros adicionales …………………………………………… 66
5.1.5. Integración con el Class Browser ……………………………….. 69
5.2. Implementación de asociaciones entre code critics y refactorings ……. 70
5.2.1. Remove Class Not Referenced ………………………………….. 70
5.2.2. Push Down Method with Refused Bequest ……………………….. 71
5.2.3. Replace Variable Only Assigned a Single Literal Value with Query. 75
5.2.4. Extract when Long Method ………………………………………... 76
5.3. Diagrama de clases de la herramienta ……………………………………. 78

6. Uso de la herramienta …………………………………………………………… 79
6.1. Ejecución de code critics para una clase ………………………………… 79
6.2. Menú de refactoring en Critic Browser ……………………………………. 80
6.3. Diálogo de refactoring ………………………………………………………. 82
6.4. Ejemplo de uso ……………………………………………………………… 82

7. Conclusiones y Trabajos Futuros ……………………………………………... 84
7.1. Conclusiones ………………………………………………………………… 84
7.2. Contribuciones ……………………………………………………………….. 85
7.3. Limitaciones …………………………………………………………………. 86
7.4. Trabajos Futuros …………………………………………………………….. 87

8. Bibliografía ……………………………………………………………………… 88

Índice de tablas y figuras

Tablas

2.1. Ejemplos de Code Smells (catálogo de Fowler) …………………………….. 17
3.1. Patrones utilizados por el framework de refactoring 34
3.2 Patrones utilizados por el framework de Small Lint …………………………… 40
4.1. Code Smells detectados por Pharo …………………………………………… 41
4.2. Asociación de code critics con refactorings predefinidos en Pharo ……….. 52
4.3. Resumen de code smells no detectados mediante code critics …………… 58
5.1. Clase del resultado de una regla lint según el enviroment al que aplica …. 63

Figuras

3.1. Refactoring en Pharo Smalltalk ……………………………………………….. 26
3.2. Esquema de la clase `RBRefactoring` ………………………………………. 27
3.3. Jerarquía de refactorings ……………………………………………………… 28
3.4. Diagrama de secuencia del *template method* **execute** …………………… 29
3.5. Modelo de la refactorización ……………………….............................. 25
3.5. Critics Browser ………………………………………………………………….. 36
3.6. Diagrama de clases del framework de reglas lint …………………………... 37
3.7. Ejecución de un code critic …………………………………………………… 40
4.1. Relación de code critics con code smells …………………………………… 43
5.1. Punto de entrada para el framework de asociaciones ……………………… 60
5.2. Resultados de reglas lint ……………………………………………………… 61
5.3. Abstracción de resultados de reglas lint ……………………………………… 62
5.4. CriticRefactoringFactory: asociación de critic con refactoring ……………. 64
5.5. Diagrama de secuencia de la ejecución de un refactoring para un code critic ……………………………………………………………………….. 65
5.6. Diagrama de la figura 5.5 actualizado con la verificación de precondiciones ………………………………………………………….... 66
5.7. Diálogo para refactoring de code critics …………………………………….. 68
5.8. Parámetros de CriticRefactoring ……………………………………………... 69
5.9. Diagrama de secuencia final de la ejecución de un refactoring ………….. 70
5.10 Menú de contexto para refactoring de code critics ………………………... 71
5.11 Ejemplo de Refused Bequest en la jerarquía de Timespan ………………… 73
5.12 Ejemplo de Refused Bequest refactorizado ………………………………... 74
5.13. Diagrama de clases de la herramienta ……………………………………… 78
6.1. Plugin para análisis de code critics de una clase …………………………... 80
6.2. Critic Browser adaptado para el análisis de una clase …………………… 80

6.3. Diálogo de refactoring en Critic Browser ……………………………………… 81
6.4. Diálogo de refactoring de critics …………………………………………….. 82
6.5. Extract Method a través del Critic Refactoring Browser ………………….. 83

Agradecimientos

Quiero agradecer a mi directora de tesis, la Dra. Alejandra Garrido, por su dedicación y paciencia, y por los valiosos aportes que hizo a mi formación durante el desarrollo de este trabajo.

Capítulo 1. Introducción

1.1 Contexto

El *refactoring* es una técnica de programación que consiste en reestructurar el código fuente de un software sin modificar su comportamiento observable. El objetivo del refactoring es mejorar los aspectos de la calidad relacionados a la mantenibilidad y la reusabilidad del código, sin alterar la funcionalidad.

Heredera del concepto de reestructuración de software, esta técnica surge formalmente a principios de los años 90, a partir del trabajo de un grupo de investigadores de la Universidad de Illinois, entre ellos William Opdyke, quien en 1992 publica su tesis doctoral sobre el tema [Opdyke92]. Opdyke trabaja el concepto de refactoring desde el punto de vista del desarrollo de frameworks. En este contexto, observa que el desarrollo orientado a objetos se realiza en forma iterativa a través de "ciclos de reutilización". Cada ciclo de reutilización supone un serie de reestructuraciones, cuya aplicación manual puede ser compleja y riesgosa. Por lo tanto, Opdyke enfoca su trabajo en la definición de un conjunto de reestructuraciones --llamadas *refactorings*-- que garanticen la preservación del comportamiento y que puedan ser aplicadas de forma *automática* por una herramienta de desarrollo.

Sin embargo, no fue hasta finales de la década, cuando se desarrollan las primeras metodologías ágiles--en especial *Extreme Programming*--, que el refactoring llega a alcanzar una amplia difusión [Beck99]. Las metodologías ágiles se basan en procesos incrementales e iterativos, dentro de los cuales el refactoring se vuelve en una pieza fundamental. Es en este marco que Martin Fowler publica su libro *Refactoring. Improving the design of existent code* [Fowler99], que terminó convirtiéndose en la principal referencia sobre el tema; y que John Brant y Don Roberts desarrollan el Refactoring Browser, la primera herramienta de refactoring automatizado, implementada en VisualWorks Smalltalk [Roberts97]. Desde entonces, la popularidad del refactoring fue extendiéndose hasta convertirse en una práctica de programación estándar. Actualmente, todas las herramientas de desarrollo --desde los editores más austeros hasta los ambientes de desarrollo más complejos-- en todos los lenguajes de programación, cuentan en alguna medida con utilidades de refactoring automático.

1.2 Motivación

Contando con un conjunto de refactorings automáticos, el programador se enfrenta al problema de cuándo y dónde aplicarlos. Fowler y Beck abordaron esta cuestión introduciendo el concepto de *code smell* [Fowler99]. Los code smells proveen una guía

para identificar problemas de diseño que podrían ser resueltos aplicando uno o más refactorings. Mediante los code smells se define un ciclo de refactoring en dos etapas:

Identificación de un code smell → Aplicación de refactoring.

Por lo tanto, la aplicación automática de refactorings a través de las herramientas provistas por los IDEs resuelve sólo una parte del problema: aún resulta difícil decidir qué refactorizaciones aplicar frente a la necesidad de hacer una reestructuración, e incluso detectar las partes del código que requieren de refactorización.

Ahora bien, si reconocemos que es importante que la aplicación de refactorings sea automática, también debería serlo la detección de code smells y la aplicación de refactorings a partir de éstos. En este sentido, distintos estudios han analizado la posibilidad de automatizar la detección de code smells. De hecho, existen herramientas capaces de detectar algunos code smells, generalmente dentro de un conjunto más amplio de análisis idiomáticos o de bugs (chequeos *lint*). También existen herramientas que permiten aplicar refactorings a partir del resultado del análisis del código. Sin embargo, estas se orientan en general a corregir aspectos idiomáticos y bugs de los lenguajes que conforman el *stack* de desarrollo, antes que a resolver problemas de diseño.

IntelliJ IDEA es la herramienta más avanzada en este sentido. Sin embargo, los análisis que realiza tienden a centrarse más en cuestiones técnicas de los distintos lenguajes del *stack* de desarrollo de aplicaciones Java, que en aspectos de diseño. No detecta algunos code smells básicos y para aquellos que detecta no siempre sugiere un refactoring. IntelliJ es una herramienta propietaria, y si bien existe una versión de código abierto, ésta no cuenta con un framework documentado que permita al programador agregar fácilmente nuevos code smells y asociarlos a refactorings.

Por otro lado, existen líneas de investigación orientadas a desarrollar herramientas que permitan la refactorización automática de un sistema aplicando métricas ponderadas y algoritmos de búsqueda, produciendo una nueva versión optimizada del sistema sin la intervención directa del programador [Moghadam11], [O'Keefe08]. Las conclusiones a las que llegan estos estudios son, en cierto modo, desalentadoras, puesto que los sistemas refactorizados que resultan de aplicar el algoritmo adolecen de problemas de legibilidad, siendo la legibilidad del código un objetivo esencial de la refactorización. Esta tesis descarta la idea de la automatización completa y se orienta a la asistencia al programador, suscribiendo al criterio de Opdyke y Roberts para quienes el diseño es una actividad esencialmente humana que no puede ser automatizada por completo [Roberts99].

Son justamente Opdyke y Roberts quienes sientan las bases para el desarrollo de la primera herramienta de refactoring en el lenguaje Smalltalk, bases que fueron heredadas por todas las herramientas actuales [Roberts97][Roberts99].

Por lo tanto, consideramos que el terreno adecuado para trabajar sobre estos problemas debería ser el mismo lenguaje Smalltalk. A nuestro entender, este lenguaje cuenta con un conjunto de características adicionales que lo hacen idóneo para la tarea:

* Smalltalk fue la usina de la cuál surgieron, no sólo la primera herramienta de refactoring, sino también la programación extrema, los tests de unidad, etc. Esto se debe a que Smalltalk promueve un estilo de programación altamente interactivo.
* La estructura original del Refactoring Browser se mantiene, con algunas modificaciones, en las principales implementaciones de Smalltalk. Esta estructura está documentada en la tesis doctoral de Don Roberts [Roberts99].
* Todas las implementaciones del Refactoring Browser cuentan con un framework de transformaciones y un framework de análisis de código (lint) sobre los cuales podemos basar nuestro trabajo. Ambos frameworks son maduros y abiertos.
* En Smalltalk es fácil implementar distintas formas de visualizar información.
* Pharo Smalltalk es un sistema de código abierto y con una comunidad de desarrollo extensa y activa, que cobró renovado impulso en los últimos años; lo cual lo convierte en terreno adecuado para el desarrollo de investigaciones.

1.3 Objetivo

El objetivo de esta tesis es estudiar la viabilidad de desarrollar una herramienta con las características que consideramos esenciales para el refactoring asistido: que permita automatizar todo el proceso de reestructuración que va desde la detección de un *code smell* hasta la aplicación de un refactoring adecuado para ese problema, pero que a diferencia de las herramientas totalmente automatizadas, brinde asistencia al programador durante el proceso de desarrollo; o dicho en otros términos, que funcione en forma interactiva.

A tal fin, elegimos el ambiente Pharo Smalltalk; investigamos las técnicas para detectar automáticamente *code smells* y sus limitaciones, tomando como punto de partida el

framework de *lint rules* provisto por el ambiente Pharo; buscamos la forma de vincular un *code smell* con la refactorización que soluciona el problema, recabando, de ser necesario, información adicional para su aplicación, para finalmente ejecutarla mediante el framework de *refactoring*.

1.3 Contribuciones

- Relevamiento de las investigaciones disponibles sobre detección de code smells y asociación de code smells con refactorings.
- Documentación analítica de los frameworks de refactoring y reglas lint de Pharo.
- Diseño e implementación de una herramienta de refactoring asistido integrada con las herramientas de desarrollo de Pharo Smalltalk y que cumple con los objetivos propuestos.
- Mejoras al modelo del framework Small Lint a través del modelado del concepto de “falla de diseño”.
- Incorporación de extensiones, tanto al framework de Small LInt con nuevos code smells, como al framework de refactoring con nuevos refactorings.

1.4 Organización de la tesis

Esta tesis se organiza en los siguientes capítulos:

Capítulo 2. Trabajos Relacionados

Se hace una breve reseña de los conceptos y aplicaciones más importantes sobre los cuales se basó la tesis.

Capítulo 3. Arquitectura base

Se describen los frameworks de Pharo Smalltalk sobre los cuales se basa el desarrollo de nuestra herramienta: el framework de refactoring y Small Lint.

Capítulo 4. Marco de trabajo

Se analiza desde un punto de vista teórico el concepto de code smell, sobre la base de los trabajos existentes. Se estudia la posibilidad de automatizar la detección de code smells, de resolver automáticamente los code smells detectados y se hace referencia a las fuentes que pueden utilizarse para la implementar las reglas lint y los refactorings.

Capítulo 5. Diseño de la herramienta

Se describen los detalles de implementación de la herramienta y explica las principales decisiones de diseño.

Capítulo 6. Uso de la herramienta

Se ilustra el uso del a herramienta mediante un ejemplo.

Capítulo 7. Conclusiones y trabajos futuros.

Se realiza un resumen del trabajo presentando la conclusión final, enumerando las contribuciones y limitaciones, y definiendo trabajos futuros que pueden derivarse de la tesis.

1.5 Convenciones

Este documento adopta las siguientes convenciones.

Nombres

Los nombres de los refactorings y los code smells se escriben con las palabras capitalizadas. Dado que los refactorings y los code smells son conocidos por sus nombres en inglés, decidimos no traducirlos.

Diagramas

Los diagramas de clases incluidos en este documento no incluyen los accessors para las variables de clase y de instancia (getters y setters). En cambio, se usa el nivel de visibilidad de las propiedades (pública, privada) de UML para determinar su accesibilidad.
La herramienta utilizada para crear los diagramas no permite ingresar los nombres de los métodos con la sintaxis de Smalltalk, por lo tanto se expresan según el siguiente patrón:

```
fromRule: aRule andSelection: anObject
```

se escribe

```
fromRuleAndSelection(aRule, anObject)
```

Tipografía

Utilizamos fuente de acho fijo para las referencias elementos del código (clases, métodos, etc.) y para el código fuente. Por ejemplo: `RBRefactoring`.

Capítulo 2. Trabajos Relacionados

2.1 Background

2.1.1 Refactoring

Un refactoring es un cambio que se aplica a la estructura interna de un software para hacerlo más fácil de comprender y modificar —es decir, para mejorar su diseño— sin alterar su comportamiento observable [Fowler99]. También se denomina refactoring al proceso o la técnica de aplicar estas transformaciones para mantener saludable el diseño del software.

Los refactorings abarcan desde cambios simples, como renombrar una variable, hasta cambios más complejos, como reemplazar lógica condicional por polimorfismo. El denominador común está en que todos los refactorings preservan el comportamiento del programa y de alguna forma mejoran el diseño y la inteligibilidad del código.

El concepto de refactoring se originó en el ámbito de la programación orientada a objetos, y fue acuñado por William F. Opdyke en [Opdyke92]. El foco de la tesis de Opdyke está puesto en proveer soporte automatizado para los refactorings, garantizando que el comportamiento del código sea preservado. Un refactoring puede abarcar cambios en varias clases, que a su vez pueden disparar nuevos cambios en otras clases. Trazar estas dependencias y realizar los cambios manualmente es una tarea ardua e ineficiente que abre la puerta a la introducción de errores. Es por eso que la aplicación sistemática del refactoring no es posible sin la ayuda de herramientas automatizadas.

Para garantizar la preservación del comportamiento, Opdyke asoció a cada refactoring un conjunto de *precondiciones* definidas en términos de propiedades del programa. La ejecución de un refactoring será legal —esto es, preservará el comportamiento externo del programa— siempre y cuando el programa cumple con las precondiciones. Garantizando de esta forma que los refactorings preserven el comportamiento, se sientan las bases para la creación de una herramienta automatizada de refactoring.

Opdyke elabora su tesis con la mirada puesta en el desarrollo de frameworks, dado que éstos necesariamente pasan por varias fases de rediseño, cada una de las cuales involucra un proceso de reestructuración. Más adelante, con la entrada en escena de la Programación Extrema y la posterior popularización de las metodologías ágiles, el refactoring cobró un nuevo protagonismo [Back05]. Las metodologías ágiles se basan en procesos iterativos e incrementales. Al final de cada iteración se obtiene una versión

funcional del sistema, que sólo implementa la funcionalidad pautada para esa iteración. El diseño siempre es el más simple posible para la funcionalidad implementada. Por otro lado, estas metodologías son adaptativas, lo que significa que los cambios son bienvenidos, incluso en etapas tardías del desarrollo. Todo esto implica que el software requiera ser reestructurado permanentemente, tanto para extender la funcionalidad a lo largo de las iteraciones, como para incorporar los cambios. Un enfoque semejante no puede funcionar sin la aplicación continua y agresiva del refactoring.

Con su libro *Refactoring: improving the design of existing code*, Martin Fowler hizo un gran aporte a la divulgación de la técnica de refactoring [Fowler99]. El libro tiene un enfoque práctico e incluye un largo catálogo de refactorings, que llegó a hacerse popular. Además introduce el importante concepto de *code smell*, que veremos más adelante.

2.1.2 Refactoring Browser

En este el contexto de expansión del uso del refactoring se desarrolló en Smalltalk la primera herramienta profesional de refactoring automatizado, el Refactoring Browser [Roberts97]. Luego de una fallida primera implementación que funcionaba como una herramienta aparte, el Refactoring Browser se desarrolló como un Browser extendido con operaciones de refactoring y otras mejoras.

El Browser (también conocido Class Browser o System Browser) es la principal herramienta de desarrollo del ambiente de Smalltalk. Se utiliza para explorar y editar el código fuente del programa y típicamente está compuesto por una serie de paneles que permiten navegar el código en forma jerárquica. El Refactoring Browser es una versión extendida del Browser que permite ejecutar refactorings desde los menús de contexto de los distintos elementos del programa.

A partir de la tesis doctoral de Opdyke, Don Roberts desarrolla una teoría sobre las herramientas de refactoring y explica la arquitectura del Refactoring Browser [Roberts97]. Esta arquitectura, con algunas variaciones, ha sido heredada por todos las implementaciones posteriores de Smalltalk, y en particular por Pharo Smalltalk. En el Capítulo 3 describiremos detalladamente la arquitectura de la implementación de Pharo del Refactoring Browser, señalando aquellas partes donde difiere de la implementación original.

El Refactoring Browser ofrece varios componentes útiles para investigar el análisis y la transformación de programas. A saber: un parser especial del lenguaje Smalltalk, que extiende la sintaxis para permitir la búsqueda y la transformación sobre los árboles de

sintaxis; un framework para el análisis de programas; y un framework para la ejecución de las transformaciones (ver Capítulo 3).

2.1.3 Code smells y análisis de código

Code smell es una metáfora para referirse a patrones asociados con un diseño deficiente o con malas prácticas de programación. La presencia de un code smell indica la necesidad de aplicar un refactoring para eliminarlo. Un ejemplo básico de code smell es el código duplicado. El código duplicado dificulta la comprensión y el mantenimiento del software. La forma más común de abordarlo es mediante el refactoring Extract Method. Esto es: se crea un método donde se pone el código duplicado y cuyo nombre describe la intención del código. Luego, se reemplaza cada ocurrencia del código duplicado por una invocación al nuevo método. En la Tabla 2.1 se muestran otros ejemplos de code smells.

El concepto de code smell, atribuido a Kent Beck, fue introducido en un capítulo escrito en colaboración por Beck y Fowler, en libro de este último [Fowler99]. Luego de describir la técnica de refactoring y los principios que la sustentan, el libro ofrece una lista de code smells cuyo objeto es ayudar al programador a determinar *cuándo y dónde* se debería aplicar un refactoring.

Los autores afirman que con los code smells no pretenden dar métricas precisas, como por ejemplo la cantidad máxima de líneas que debería tener un método. Creen que nada puede batir a la intuición humana y que cada diseñador debería desarrollar su propia idea de conceptos como método demasiado grande. Sin embargo, consideramos que la intuición no alcanza en caso de programadores con poca experiencia o cuando el sistema es demasiado grande para ser revisado manualmente. Una solución intermedia entre la revisión manual y la revisión automática es la posibilidad de configurar los aspectos métricos de los code smells.

Code Smell	Descripción
Long Method	Los métodos largos son difíciles de leer y de entender. Por el contrario, los métodos cortos son más fáciles de entender y si se les asigna nombres adecuados el código se vuelve autodocumentado.
Lazy Class	La existencia de una clase implica un costo de comprensión y mantenimiento. Es por eso que una clase que no hace lo suficiente debe ser eliminada.
Long Parameter List	Los métodos con demasiados parámetros son difíciles de entender, tienden a volverse inconsistentes, son difíciles de usar, y requieren ser cambiados permanentemente cuando se necesitan más datos.

Tabla 2.1. Ejemplos de Code Smells (catálogo de Fowler).

Small Lint es un componente del Refactoring Browser que permite detectar varios bugs clásicos del lenguaje Smalltalk. Para analizar el código, se vale las mismas herramientas de análisis utilizadas para chequear las precondiciones de los refactorings. Su nombre hace referencia al analizador estático de C desarrollado por Ian F. Darwin [Darwin88].

Small Lint no es una herramienta de métricas propiamente dicha, sino de chequeos. Permite detectar los distintos elementos del código que están afectados por una falla. Desde luego, los chequeos se hacen en base a métricas, pero no existe una representación explícita de éstas.

Como Small Lint está implementado mediante un framework, permite a los programadores agregar nuevos chequeos (este proceso será explicado en detalle en el Capítulo 3). Dentro de las limitaciones inherentes al análisis estático de programas —que por ejemplo no permite analizar el flujo del programa o definir si una variable de instancia es un componente— es posible crear chequeos que ya no detecten bugs ni violaciones a reglas de estilo, sino que busquen code smells. De hecho, en las últimas versiones de Pharo Smalltalk, entre los chequeos disponibles se incluye una lista de fallas de diseño, algunas de las cuales pueden ser identificadas como code smells clásicos, como veremos en el Capítulo 4.

2.2 Trabajos de investigación

2.2.1 Refactorización basada en búsqueda: Code-Imp

Basándose en la idea de abordar la mejora del diseño de un software como un problema de optimización combinatoria [O'Keefe03], un grupo de Investigadores del University College Dublín, desarrolló una herramienta de refactorización automática llamada Code-Imp [O'Keefe08]. Esta herramienta cuenta con un conjunto de métricas de software orientado a objetos que se utilizan para medir la calidad del diseño, y con un conjunto de operaciones de refactorización automáticas. En base a las métricas, el usuario define una función de aptitud (*fitness function*), estableciendo los pesos relativos para cada métrica, o utilizando un criterio llamado *optimalidad de pareto*. La búsqueda de una solución cercana a la óptima se realiza aplicando en forma semi aleatoria una secuencia de refactorings tentativos, cuya aceptación o no, depende del efecto que éstos tengan sobre el diseño, medido según la función de aptitud. Los distintos métodos de búsqueda y su eficacia relativa han sido materia de investigaciones posteriores.

Dado que Code-Impl realiza todo el proceso de mejora del diseño en forma automática, sin más intervención del desarrollador que la que se produce al inicio del proceso, cuando se define la función de aptitud, no es de sorprender que los autores hayan llegado a la conclusión de que la técnica sólo resulta práctica en áreas donde el programador no tiene que entender la salida del proceso de refactorización. Mientras que este trabajo descansa sobre la convicción de que la refactorización automatizada puede lograr mejores resultados que la automatización de las refactorizaciones individuales, nosotros seguimos la línea de Opdyke y Fowler según la cual el diseño es una actividad esencialmente humana que no puede ser automatizada totalmente, y que considera que la legibilidad es un objetivo primordial de la refactorización. De aquí que nuestro trabajo se oriente a brindar asistencia al programador en el proceso de reestructuración del diseño y no a automatizarlo por completo.

2.2.2 Estrategias de detección: reglas basadas en métricas para la detección de fallas de diseño

En [Marinescu04] se propone un mecanismo para definir reglas que, basadas en métricas de código, capturen desviaciones de los principios del buen diseño. Como las métricas arrojan información cuantitativa sobre distintos aspectos del sistema, a menudo no brindan indicios que señalen claramente los problemas de diseño ni las

entidades del programa afectadas por estos problemas. Para superar esta limitación los autores introducen las denominadas *estrategias de detección*: reglas que, valiéndose de mecanismos de filtrado y composición, ofrecen una interpretación de las métricas.

El framework de reglas lint de Smalltalk también provee información a nivel de problema -- code critic, que puede ser o bien un bug o una falla de diseño-- e identifica el elemento del programa afectado. La diferencia con las estrategias de detección reside en la implementación. Mientras que éstas aplican filtros sobre una base de datos de métricas del sistema, en Smalltalk las reglas se escriben en código Smalltalk, el cual accede al metamodelo del programa mediante las características reflexivas propias del lenguaje y el framework de AST (una facilidad del lenguaje que permite acceder a los árboles de sintaxis de los métodos) [Foote88]. Este hecho, obedece a que Smalltalk es un entorno estrechamente integrado, y la base de datos del programa se mantiene siempre actualizada por la compilación dinámica del código [Roberts99, sección 6.2.1].

Otro aporte del paper es un método para obtener una estrategia de detección a partir de un conjunto de reglas de diseño informales, expresadas en lenguaje natural.

2.2.3 QA en Java mediante la detección de Code Smells

Eva van Emden y Leon Moonen, del CWI de Ámsterdam, propusieron en 2002 un método para detectar y visualizar code smells en el código de sistemas desarrollados en Java [Emden07]. Para esa época, la mayoría de las herramientas disponibles sólo eran capaces de detectar aspectos técnicos del código, mientras que unas pocas empezaban a introducir análisis más complejos.

Si bien los autores afirman que una vez detectados los code smells mediante la técnica propuesta, podrían ser utilizados tanto por programadores para mejorar el código, como por herramientas de refactoring automático y por revisores de código, el foco del trabajo está puesto en este último objetivo, esto es, en facilitar las inspecciones de software.

Un prototipo de herramienta, llamado jCOSMO, fue desarrollado para verificar la viabilidad del enfoque. jCOSMOS contenía un par de code smells relativamente simples y ofrecía distintos tipos de visualización de los resultados basados en grafos.

La asociación de code smells con refactorings se planteaba como una posible línea de desarrollo futura, que finalmente no fue abordada.

2.3 Herramientas

2.3.1 IntelliJ IDEA

IntelliJ IDEA es un IDE para Java, desarrollado por la empresa JetBrains [IntelliJ]. Se encuentra disponible tanto en un versión comunitaria (bajo licencia Apache 2) como en una versión comercial. IntelliJ IDEA fue uno de los primeros productos en incorporar capacidades integradas de refactoring para el lenguaje Java.

La versión actual (13.1) permite aplicar una larga lista de refactorings, muchos de los cuales son sugeridos y aplicados directamente en el editor de código fuente. Los refactorings pueden aplicarse no sólo desde el código fuente sino también a partir de los diagramas UML.

Análisis de código

IntelliJ IDEA también cuenta con una herramienta de análisis de código. La herramienta es un típico analizador lint, con opciones de refactorización integradas. El analizador es muy completo, posee una larga lista de reglas lint --llamadas inspecciones-- y permite refactorizar el código a partir de los resultados del análisis, solicitando información cuando el refactoring lo requiere (ejemplo, ver casos complejos). Sin embargo, no detecta problemas de diseño elementales como Middleman, Lazy Class, Data Class, etc.
Las inspecciones que podemos asociar con code smells son las siguientes:

- Feature Envy
- Clase demasiado grande: con demasiados métodos, con demasiadas propiedades o demasiado compleja (complejidad ciclomática).
- Clase demasiado acoplada
- Clase utilitaria
- Refused Bequest: detecta un caso marginal: cuando un método sobreescribe un método concreto pero no lo invoca a través de super.
- Demasiados parámetros (Long Parameter List)
- Método demasiado largo (Long Method)
- Chained method calls (Message Chain)
- Distintas formas de Dead Code
- Switch statements
- Duplicate Code (sólo en la edición comercial).

Las inspecciones pueden ser parametrizadas. Por ejemplo, para *clase con demasiados métodos* se puede definir la cantidad máxima de métodos; o para *clase demasiado compleja* se puede establecer la complejidad ciclomática límite. Además, cada inspección tiene un nivel de severidad (v.g. warning, error, etc.). También es posible seleccionar las inspecciones que serán incluidas en el análisis.

Refactorings

IntelliJ IDEA soporta una lista bastante completa de refactorizaciones. Algunas de estas son refactorizaciones complejas incluidas en el catálogo de Fowler, por ejemplo:

- Extract Delegate (Extract Class)
- Extract Parameter Object
- Extract Method Object
- Remove Middleman
- Replace Inheritance with Delegation
- Replace Temp with Query

Previo a la ejecución de un refactoring, el IDE realiza una búsqueda y muestra todas las partes del código que se van a modificar. El programador puede explorar estos cambios antes de aplicarlos.

IntelliJ IDEA es la herramienta que más se aproxima al objetivo de nuestro trabajo. Sin embargo tiende a centrarse en cuestiones técnicas de los distintos lenguajes que se utilizan en el desarrollo de aplicaciones Java que en cuestiones de diseño, y no detecta algunos code smell básicos. Además, para code smells como Too Many Parameters, Long Method (Extract Method), Refused Bequest, Feature Envy, etc., no sugiere ningún refactoring.

Por otro lado si bien, posee una versión de código abierto, no cuenta con un framework documentado que le permita al programador agregar fácilmente nuevos code smell y asociarlos a refactorings.

2.3.2 iPlasma

iPlasma [iPlasma] es un entorno para el análisis de calidad de sistemas de software orientados a objetos, basado en la técnica de estrategias de detección mencionada más arriba. Permite analizar sistemas desarrollados en Java y C++, es customizable (con respecto a estrategias de detección y los modelos de análisis), y ofrece un front-

end que integra distintas vistas y análisis. Este front-end puede ser extendido mediante plugins. iPlasma es una herramienta de análisis de código, no ejecuta refactorings.

2.3.3 inFussion e inCode

inFussion e inCode [inFussion] son plataformas de análisis de código fuente de la compañía Intooitus. Permiten detectar code smells y ofrecen distintas visualizaciones para el análisis. Destaca entre éstas la posibilidad de exportar el resultado del análisis para ser visualizado mediante CodeCity, una herramienta que muestra gráficamente el tamaño respectivo de los distintos componentes de un sistema. Un sistema es representado gráficamente usando la metáfora de ciudad, donde las clases se muestran como edificios y los paquetes como distritos donde éstos edificios residen.

Estas herramientas ayudan en el análisis del código y la detección de code smells, pero no hacen refactorings.

2.3.4 Cincom Visual Works

Las últimas versiones de Visual Works permiten refactorizar una variable a partir del resultado del análisis del código. Específicamente, se puede hacer *pull up* o *push down* de una variable que ha sido señalada con un code critic. Las opciones de *pull up* y *push down* aparecen habilitadas, sólo si se cumplen las precondiciones para ejecutar esos refactorings. Sin embargo, esta utilidad no está implementada genéricamente como una relación entre code critics y refactorings, sino que está implementada directamente en la vista para el caso particular del refactoring de variables.

Capítulo 3. Arquitectura de Base

3.1 El Framework de Refactoring

Todos las implementaciones actuales de Smalltalks permiten ejecutar refactorings en forma automática a través del Browser. Esta característica está implementada mediante un framework, llamado Framework de Transformaciones [Roberts97], heredado del Refactoring Browser original de Roberts y Brant. El hecho de que esté implementada mediante un framework, nos permite extender la herramienta de desarrollo agregando nuevos refactorings.

Cabe destacar que el Refactoring Browser fue el primer framework de refactoring desarrollado, y que todas las herramientas actuales se basan en los mismos conceptos fundamentales del Refactoring Browser y en su manera de realizar las transformaciones. Algunos de estos conceptos fundamentales del framework son:

- Condiciones: permiten analizar las precondiciones para que un refactoring sea legal, es decir, para que su aplicación preserve el comportamiento del código.
- Cambios: ejecutan las transformaciones sobre la base de datos del programa. Quedan registrados y pueden ser revertidos.
- Refactorings: consisten de un conjunto de condiciones y un conjunto de cambios.
- Búsqueda y reescritura de árboles de sintaxis.

Cada ambiente Smalltalk implementa el Refactoring Browser con pequeñas variantes. En este apartado describiremos los principales componentes de la versión de Pharo del Framework de Transformaciones, a lo largo del proceso de creación de un nuevo refactoring. Más adelante, en el Capítulo 5 explicaremos en detalle la implementación de otros refactorings.

3.1.1 Uso de la herramienta

La herramienta permite ejecutar los refactoring a través de los menús de contexto disponibles en el Browser para los distintos elementos del programa: paquetes, clases, variables, métodos y código fuente. El refactoring toma la información de contexto para determinar el elemento sobre el cual se aplica. Además, el refactoring puede requerir información adicional, que nos será solicitada mediante un cuadro de diálogo. Por ejemplo, al ejecutar el refactoring Rename Method –que se utiliza para cambiar el

nombre de un método por otro más adecuado o más legible--, el Browser nos pide que ingresemos el nuevo nombre.

Cuando ejecutamos un refactoring, el Browser verifica que las precondiciones sean válidas. Como explicamos en el Capítulo 2, las precondiciones de un refactoring son las que determinan si su aplicación es legal, es decir, si preservará el comportamiento del programa. Si las precondiciones son válidas, entonces el refactoring puede ejecutar la transformación del programa. Caso contrario, el Browser muestra un mensaje de error que indica que no puede realizar el refactoring porque no es seguro y por lo tanto no puede ser aplicado.

Una transformación se compone de conjunto de cambios. En el caso de Pharo Smalltalk, estos cambios se muestran en un diálogo antes de ejecutar el refactoring, dándonos la posibilidad de decidir cuáles de ellos queremos aplicar. El diálogo lista los cambios concretos que se van a aplicar y es equivalente a la búsqueda que hace IntelliJ.

En la Figura 3.1 vemos el Browser de Pharo Smalltalk al momento de ejecutar el refactoring Push Up aplicada al método `methodThatRefusesBequest` de la clase `FlawedSubclass2`. Para lanzar esta refactorización se presionó el botón secundario del mouse sobre el método `methodThatRefusesBequest` y se eligió la opción “Push up” del menú "Refactoring".

La ventana cuyo título es Change Browser muestra los cambios concretos que realizará el refactoring. Seleccionando un cambio, podemos ver cómo será aplicado sobre el código fuente.

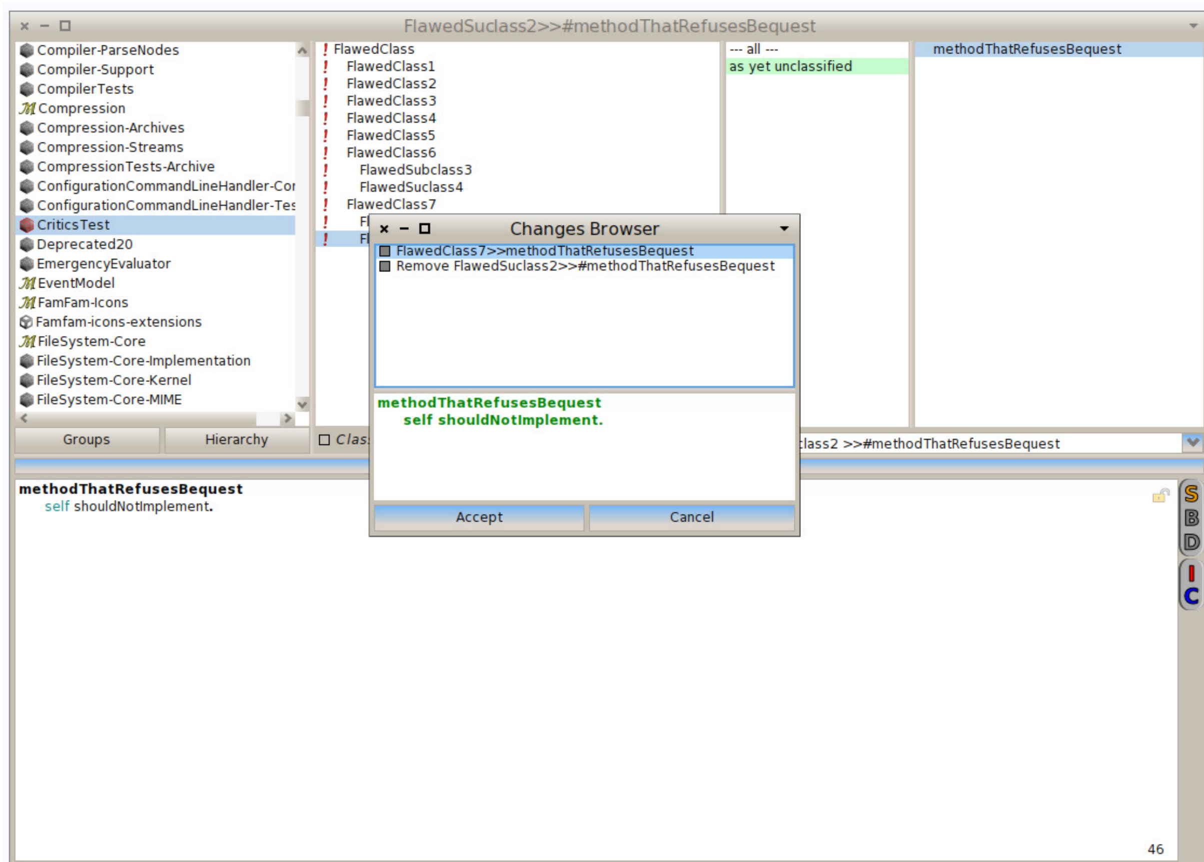


Figura 3.1. Refactoring en Pharo Smalltalk

3.1.2 Creación de un nuevo refactoring

En su tesis doctoral, *Practical Analysis for Refactoring* [Roberts99], Don Roberts ofrece una descripción de la arquitectura del Refactoring Browser donde identifica cinco componentes principales: (i) refactorings, (ii) condiciones, (iii) parser, (vi) árbol de reescritura de código fuente y (v) objetos que representan los cambios.

A continuación repasaremos estos conceptos sobre la implementación del Refactoring Browser de Pharo Smalltalk. Lo haremos desde el punto de vista del programador que necesita extender el framework. Para ello crearemos un nuevo refactoring y lo utilizaremos como ejemplo a lo largo de nuestra exposición. Este refactoring tendrá como objetivo definir un método abstracto en una superclase abstracta, de forma tal que quede establecido el protocolo de las subclases y éstas se vean obligadas a definir el método. Para asegurarnos de que se pueda ejecutar el refactoring, deberemos verificar que el método esté en todas las subclases y que no esté definido en la clase abstracta sobre la que se ejecuta el refactoring ni en ninguna de sus ascendientes.

Llamaremos al refactoring Define Abstract Method y lo definimos de la siguiente forma:

Nombre:	Define Abstract Method		
Nombre de la clase:	RBDefineAbstractMethodRefactoring		
Parámetros:	className	:	String
Precondiciones:	El método está definido en todas las subclases. El método no está definido en la superclase sobre la cual se aplica el refactoring (directa o indirectamente).		
Transformaciones:	Definir un nuevo método en la clase que se llama de la misma forma que los que están en las subclaes y que contenga siguiente código: `^self subclassReponsibility.`		

3.1.2 Refactorings

Cada uno de los refactoring definidos en el Refactoring Browser se implementa como una subclase de la clase abstracta `RBRefactoring`. Esta clase utiliza el patrón Template Method para brindar el punto de extensión principal del framework (*hotspot* en la jerga de frameworks). Esto significa que la clase define el esqueleto de un algoritmo en un método denominado *template method* y delega en las subclases la implementación de algunos de sus pasos. Los pasos que se delegan en las subclases se definen mediante métodos abstractos que éstas deben implementar. A estos métodos se los llama *hook methods* [Gamma94].

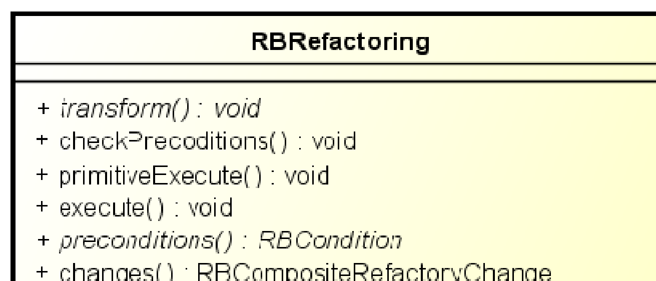


Figura 3.2. Esquema de la clase `RBRefactoring`

Debajo de `RBRefactorig` existen otras clases abstractas que, en líneas generales, organizan los refactorings en una jerarquía que los agrupa según el elemento del programa al cual se aplican (Figura 3.3).

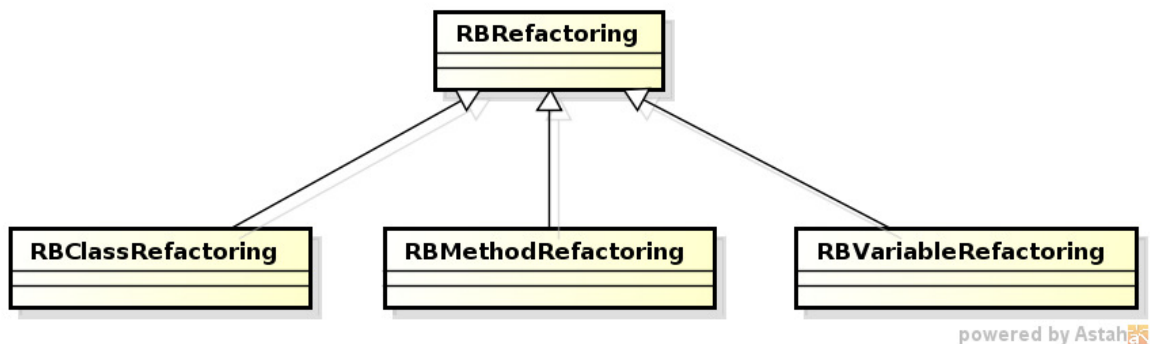


Figura 3.3. Jerarquía de refactorings.

Para crear un nuevo refactoring, debemos extender `RBRefactoring` o alguna de sus subclases e implementar los *hook methods*.

- `preconditions`: Devuelve una instancia de `Conditions` que representa las precondiciones del refactoring
- `transform`: Ejecuta los cambios mediante objetos `RefactoryChange`.

La Figura 3.4. muestra cómo, a partir del método execute, se invocan los métodos *hook*.

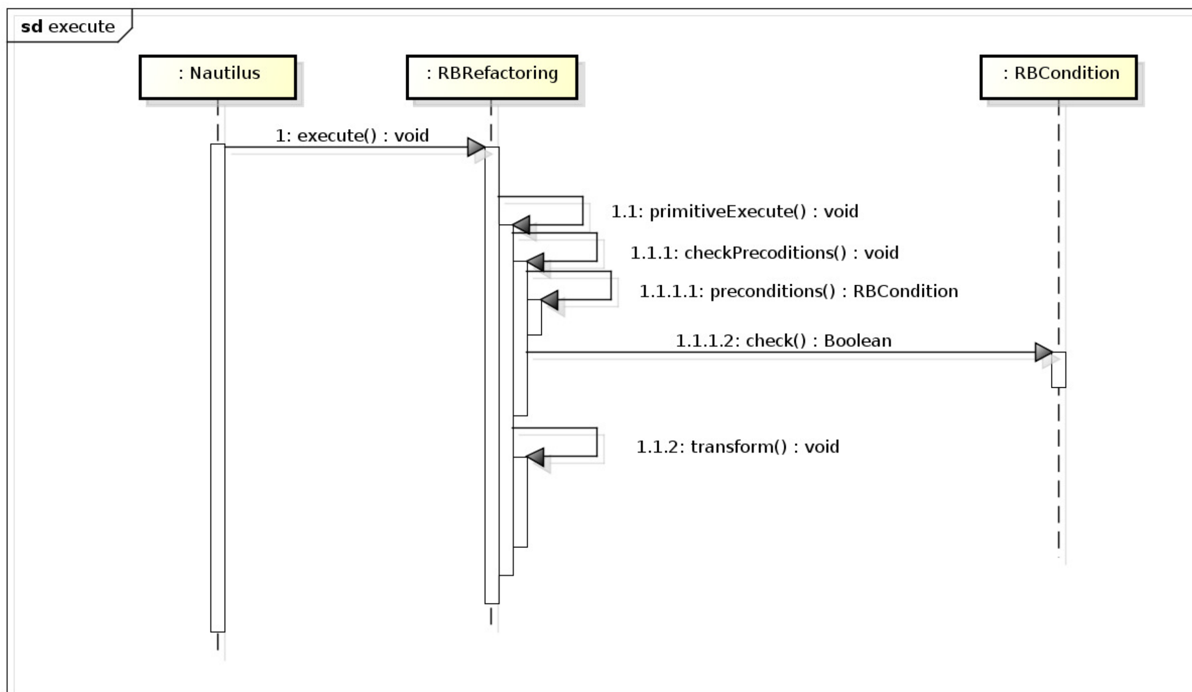


Figura 3.4. Diagrama de secuencia del *template method* **execute**

Antes de implementar los *hooks* es necesario definir un constructor. El constructor debe tomar la información del elemento sobre el cual se aplica el refactoring. Además, el constructor siempre toma como parámetro el modelo del Refactoring Browser . Como veremos en la sección 3.1.5, el modelo es una representación del sistema que permite realizar cambios en el programa mediante objetos `RefactoryChange`.

3.1.4 Condiciones

El siguiente paso en la creación de un refactoring es la definición de las precondiciones. Las precondiciones son una instancia de `RBAbstractCondition` y se definen en el hook method `conditions` del refactoring.

Las condiciones representan los distintos chequeos que los refactorings realizan para determinar la legalidad de su aplicación. En su forma más simple, una precondición es una instancia de `RBCondition` inicializada con un bloque (closure) que devuelve un valor booleano. El bloque es evaluado al invocarse el método `check` y el resultado de la evaluación dice si la condición se cumple o no. Opcionalmente se puede especificar el mensaje de error que será mostrado si la condición falla.

Asimismo, la clase RBCondition provee un conjunto de métodos de clase que permiten crear precondiciones estándar. Por ejemplo, la condición más simple es la condición vacía, que devuelve siempre verdadero y se crea enviando el mensaje empty a la clase RBCondition. Otros ejemplos de condiciones son definesSelector: aSelector in: aClass, que se utiliza para chequear si la clase aClass define el método aSelector, hierarchyOf: aClass canUnderstand: aSelector, que dice si la jeraquía de aClass implemenenta el método: aSelector.

Mediante estos métodos de creación de instancias se pueden definir en forma clara y legible, las propiedades del programa que se deben cumplir.

También es posible crear reglas compuestas gracias a que el modelo de condiciones implementa el patrón Composite [Gamma94]. La composición permite combinar condiciones mediante los operadores lógicos *y* (&) y *o* (|) y negar reglas mediante el operador unario *no* (not). Cuando se envía el mensaje & a una condición, se obtiene una instancia de RBConjunctiveCondition que contiene tanto la condición receptora del mensaje como la condición pasada como argumento. El ejecutarse el método check de RBConjunctiveCondition, se evaluan ambas condiciones y se devuelve el resultado de la conjunción.

Análogamente, se puede utilizar el método | para crear condiciones disyuntivas. Este método crea una RBConjunctiveCondition negada con sus componentes izquierdo y derecho también negados, dado que $P \lor Q \Leftrightarrow \sim(\sim P \land \sim Q)$ (leyes de De Morgan).

Las condiciones de nuestro refactoring RBDefineAbstractMethodRefactoring deben chequear que, en primer lugar, el método que vamos a agregar no esté previamente implementado. En segundo lugar, debemos asegurarnos de que el método esté implementado en todas las subclases. En este caso, usamos las condiciones estándar definesSelector:in: y hierarchyOf:canUnderstand:. En código Smalltalk, esto se define de la siguiente forma:

```
RBDefineAbstractMethodRefactoring >> preconditions
      | cond |
      cond := (RBCondition definesSelector: selector in: class) not.
      ^ class subclasses
            inject: cond
            into: [ :c :subclass | c &
            (RBCondition hierarchyOf: subclass canUnderstand: selector)
            ]
```

3.1.5 Ejecución de las transformación: los cambios y el modelo

Una vez validadas las precondiciones, el refactoring puede ejecutar la transformación del programa. Para ello, el framework provee el *hook method* `transform` donde se codifica la transformación. La transformación se define mediante objetos `RefactoryChange`, que implementan los cambios sobre los elementos del programa.

La clase `RefactoryChange` (`RBRefactoryChange` en Pharo), es un ejemplo del pattern Command. Este diseño permite implementar fácilmente la operación de reversión (deshacer) de los refactorings.

En la versión original del Refactoring Browser, los refactorings creaban directamente los objetos RefactoryChange y ejecutaban la transformación. Actualmente, los refactorings operan sobre un *modelo del sistema* que a su vez se encarga de crear y registrar los cambios.

En Pharo, el modelo es una instancia de `RBNamespace` y usa las clases `RBClass`, `RBMetaclass` y `RBMethod` para representar los elementos del programa. Cuando es manipulado desde un refactoring, el modelo registra los cambios dentro de una instancia de `RBCompositeRefactoryChange` y modifica su estado interno, simulando el efecto que tienen los cambios sobre el sistema (por ejemplo registrando las clases que fueron eliminadas). Otra función del namespace es la de cachear el resultado de algunas consultas para hacer más eficientes las transformación, como por ejemplo aquellas que se utilizan para determinar los usos de un mensaje (*senders* e *implementors*). Por último, el modelo tiene una referencia a un *environment* (`RBBrowserEnvironment`). Un environment es básicamente una colección de clases que sirve para determinar el alcance del refactoring y permite hacer consultas sobre la base de datos del programa.

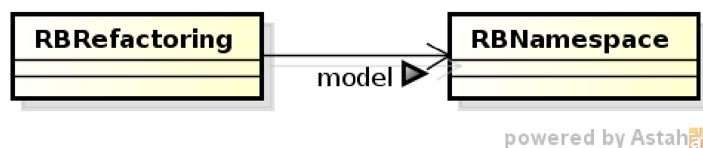


Figura 3.5. Modelo de la refactorización.

En nuestro ejemplo, la transformación consiste en crear un método en la clase sobre la cual se ejecuta el refactoring, que se limite a levantar la excepción “subclass

responsibility". Esto se realiza creando una instancia de `RBMethod` con el código `^self subclassResponsibility` y pidiéndole a la clase que lo compile.
El código de la transformación es el siguiente:

```
transform
	| method code|
	code := selector asString, String cr, '^self
subclassResponsibility'.
	method := RBMethod for: class source: code selector: selector.
	class compile: code withAttributesFrom: method.
```

En Pharo, los refactorings son ejecutados y registrados por la clase `RBRefactoringManager`. Esta clase es un caso del patrón "Singleton", es decir que de ella existe una sola instancia que funciona como una variable global del sistema. Trabaja en colaboración con `RBRefactoryChangeManager`, otro Singleton que a su vez se encarga de ejecutar los cambios y registrar las operaciones para deshacerlos.

3.1.6 Reescritura de código

En muchos casos, no alcanza con realizar cambios sobre las estructura del programa a través del modelo del sistema. En cambio, puede ser necesario modificar el código dentro de los métodos. Un ejemplo de este tipo de transformación puede hallarse en el refactoring Extract Method.

A estos efectos, los ambientes Smalltalks en general y Pharo en particular, cuentan con una herramienta de reescritura de código fuente, basada en la transformación de árboles de sintaxis (AST).

La reescritura se realiza a través de un objeto `RBParseTreeRewriter`. Esta clase, junto con `PathTreeSearcher` implementan el patrón Visitor. Este patrón permite desacoplar las operaciones que se realizan sobre determinada estructura: en este caso la búsqueda y la reescritura que se aplican sobre los árboles AST.

El objeto se instancia y se configura la regla de reescritura utilizando el método `replace:with:`, *o* `replace:with:when:`. Luego, se ejecuta sobre el AST de un método. La ejecución arroja como resultado un nuevo AST modificado, que se usa para recompilar el método.

El siguiente ejemplo corresponde a la configuración de la regla de reescritura que utiliza el refactoring Inline Method para eliminar del método que se va a intercalar las ocurrencias del símbolo de retorno (que en Smalltalk se identifica con un '^').

```
removeReturns
      | rewriter |
      rewriter := RBParseTreeRewriter new.
      rewriter replace: '^``@object' with: '``@object'.
      (rewriter executeTree: inlineParseTree)
            ifTrue: [inlineParseTree := rewriter tree]
```

La regla busca en el código los objetos precedidos por un caret (^) y elimina este último. El árbol resultante se inserta en el método que envía el mensaje, que luego es recompilado utilizando el modelo del sistema (a través de `RBClass`). Al hacerse a través del modelo, se genera el `RBRefactoryChange` correspondiente, lo que da la posibilidad de revertir el cambio.

La sintaxis de las reglas de reescritura, permite escribir expresiones que matcheen distintos elementos del árbol y utilizar las correspondencias para crear el árbol resultante.

Las reglas de reescritura no son más que código Smalltalk con una sintaxis especial para expresar los patrones. Puede encontrarse una referencia de la sintaxis de reescritura en la tesis de Don Roberts [Roberts99].

Originalmente, Roberts y Brant extendieron el parser de Visual Works para soportar las reglas de reescritura. Luego, por cuestiones de portabilidad, decidieron que era mejor implementar un parser propio. En Pharo, las reglas de reescritura son procesadas por el mismo parser que se usa para compilar el código Smalltalk, volviendo de esta forma la implementación original de Roberts y Brant.

3.1.7 Patrones de diseño

Como un aporte mas para la descripción de estos frameworks, hemos identificado los patrones que estan presentes en el diseño de los mismos. Como dicen Beck y Johnson [Beck94][Johnson92], los patrones de diseño son muy útiles al momento de documentar el diseño complejo de un framework orientado a objetos.

Patrón	**Comentarios**
Signleton	Se aplica en las clases `RBRefactoringManager` y `RBRefactoryChangeManager`
Template Method	Lo aplica la clase `RBRefactoring` y para proveer el punto principal de extensión del framework.
Visitor	Se utiliza paral la implementación de la búsqueda y reescritura sobre árboles ASTs.
Command	Los `RBRefactoryChange` son comandos que transforman el programa.
Composite	Se aplica en la jerarquía de `RBCondition` para la construcción de condiciones complejas con conectores lógicos, y en la jerarquía de environments.

Tabla 3.1 Patrones utilizados por el framework de refactoring.

3.2 Small Lint

Small Lint es un componente del Refactoring Browser que permite detectar bugs y errores del lenguaje Smalltalk. El análisis del código se realiza recorriendo los elementos del programa y aplicándole chequeos llamados *code critics* o *reglas lint*.

Small Lint está implementado mediante un framework, lo que permite permite al programador agregar code critics.

Además de incluir los bugs clásicos, la implementación de Pharo cuenta con un amplio conjunto de reglas predefinidas, que incluye la detección de varios problemas de diseño bajo la categoría *design flaws*.

En Pharo, los chequeos se corren mediante un browser especial llamado Critic Browser (ver figura 3.5). Este browser nos permite seleccionar los paquetes que queremos analizar y los code critics que vamos a incluir en el análisis. El browser ejecuta el análisis y muestra la lista de critics, señalando aquellos que detectaron errores. Al hacer click sobre uno de estos critics, se muestra la lista de elementos del programa (clases o métodos, según el critic) que no pasaron el chequeo.

Desde el punto de vista del análisis del código para el refactoring, el uso de un browser separado es incompatible con el criterio establecido por Roberts según el cual las herramientas de refactoring tienen que estar lo más integradas posible con el entorno de desarrollo.

Figura 3.5. Critic Browser

3.2.1 El framework de reglas lint

Los code critics son instancias de la clase `RBLintRule`. Las reglas pueden ser compuestas. Esto supone la implementación del patrón Composite en la jerarquía de `RBLintRule`, como se observa en la Figura 3.6. Dentro de las reglas simples, la jerarquía define dos grupos: las reglas de bloque y las reglas de árboles de parseo. A diferencia de aquellas, estas últimas definen un *matcher* que analiza el AST de los métodos.

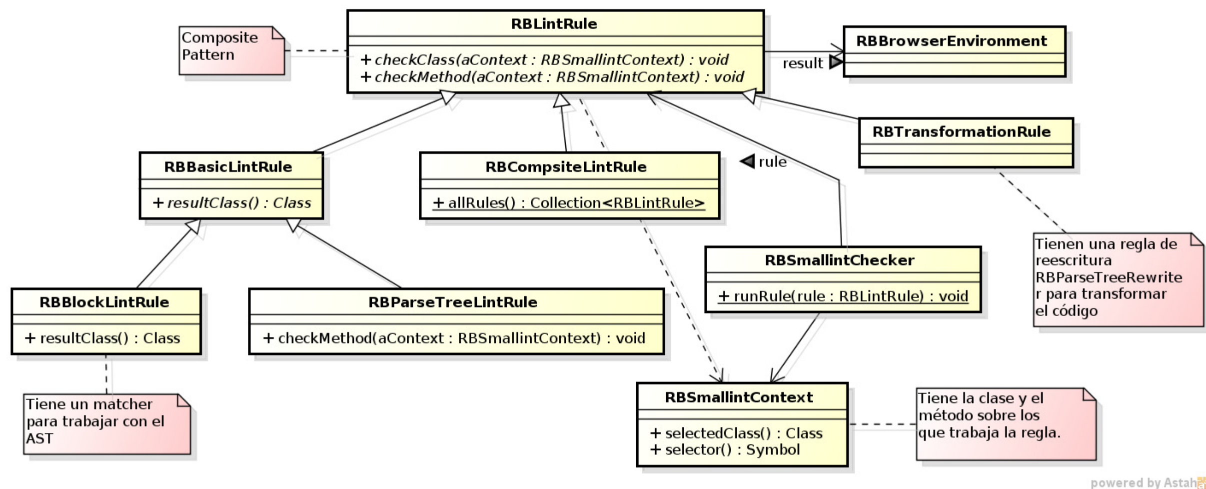


Figura 3.6. Diagrama de clases del framework de reglas lint.

Por último existe un tercer tipo de regla, las reglas de transformación, que permiten definir un patrón de búsqueda sobre el código fuente y un patrón de reemplazo. De esta forma es posible definir refactorings simples utilizando la notación extendida de Smalltalk que permite analizar el código utilizando expresiones regulares (*).

Para agregar un nuevo critic debemos crear una nueva subclase de `RBLintRule`. Si la regla va a analizar el código de los métodos, entonces nuestra clase debe extender de `RBParseTreeLintRule`. Por el contrario, si los chequeos sólo se harán usando reflexión sobre las clases y los métodos, debe extender de `RBBlockLintRule`.

A continuación crearemos una regla para cada caso, explicando el proceso y los detalles del framework.

3.2.2 Ejemplo: Refused Bequest

Refused Bequest es un code smell que se manifiesta cuando una clase hereda un método o una variable que no utiliza. Una forma sencilla, aunque limitada, de reconocer esta situación en Smalltalk es buscando los métodos que envían el mensaje `shouldNotImplement` a `self`. El método `shouldNotImplement` está implementado en la clase `Object` y levanta una excepción `ShouldNotImplement`. Mediante su invocación una clase rechaza explícitamente parte de la semántica heredada.

Nuestra clase se llamará `RBRefusedBequestRule` y heredará de `RBParseTreeLintRule`, porque necesitamos verificar si los métodos contienen la invocación a `shouldNotImplement`.

A continuación debemos implementar una serie de métodos que utiliza el Critic Browser para mostrar información sobre a regla y la clasifican: `category`, `group`, `name` y `rationale`.

Luego, debemos definir el matcher. Este se crea en el método initialize y luego es utilizado por el framework cuando se aplica la regla.

```
initialize
      super initialize.
      self matcher
            matchesAnyOf: #(
                  '`@object shouldNotImplement' )
            do: [ :node :answer | node ]
```

Para definir el matcher se usa la sintaxis especial del parser del Refactoring Browser definida en la tesis de Don Roberts [Roberts99]. El matcher es una instancia de `RBParseTreeSearcher`. Esta clase usa el patrón Visitor para recorrer el arbol AST y buscar correspondencias en los nodos. El bloque que se pasa en el segundo argumento permite construir una respuesta a partir de los nodos que se corresponden con el patrón (como si fuese un `inject:into:`, la implementación de Smalltalk de la función *reduce*). Sin embargo, esta característica no es utilizada por el framework lint.

3.2.3 Ejemplo: Long Class

A diferencia de las reglas sobre el árbol de parseo, que sólo operan sobre los métodos, las reglas de bloque también pueden detectar errores asociados a una clase. A continuación mostraremos el ejemplo del critic “cantidad excesiva de métodos”, que es una forma del code smell Big Class [Fowler99].

En este caso, luego de implementar los métodos de información de la regla, tenemos que implementar `checkClass: aContext`. Este método toma como argumento una instancia `RBSmallLintContext` y realizar el chequeo sobre la clase. Análogamente, se usa el método `checkMethod: aContext` para analizar un método. El contexto contiene la clase y el método (el selector) que están siendo chequeados (este último solo está presente cuando se invoca `checkMethod:`). Además, el contexto colabora con la regla calculando propiedades de la clase y el método.

```
checkClass: aContext
	aContext selectedClass selectors size >= self methodsCount
		ifTrue: [ result addClass: aContext selectedClass ]
```

En este ejemplo, a través contexto se obtiene la clase, se le envía el mensaje `selectors` para obtener la cantidad de métodos y se compara el valor obtenido con el máximo definido por la regla. Si la clase no pasa el chequeo, entonces se guarda en la variable de instancia `result`.

Esta variable puede contener una instancia de `RBSelectorEnivornment` o `RBClassEnviroment` dependiendo de si la regla chequea clases o métodos. Por defecto es una instancia de `RBSelectorEnviroment`. Para cambiar ese comportamiento debemos sobreescribir el método `resultClass`.

```
resultClass
	^ RBClassEnvironment
```

3.2.4 Ejecución de las las reglas

Las reglas se ejecutan desde la clase `RBSmallLintChecker`. Esta clase es responsable de crear el contexto, aplicar las reglas a las clases de un *enviornment* y guardar el resultado. La regla puede ser una `RBCompositeLintRule`, en cuyo caso el `RBSmallLintChecker` aplica todo el conjunto de reglas al environment. De hecho, esa es la función que cumple el pattern Composite en el framework, y es lo que hace el Critic Browser para aplicar el conjunto de reglas seleccionadas. A su vez, esto es posible porque la clase `RBCompositeLintRule` contiene métodos de clase que devuelve, agrupadas por categorías, todas las reglas que heredan de `RBBasicLintRule`.

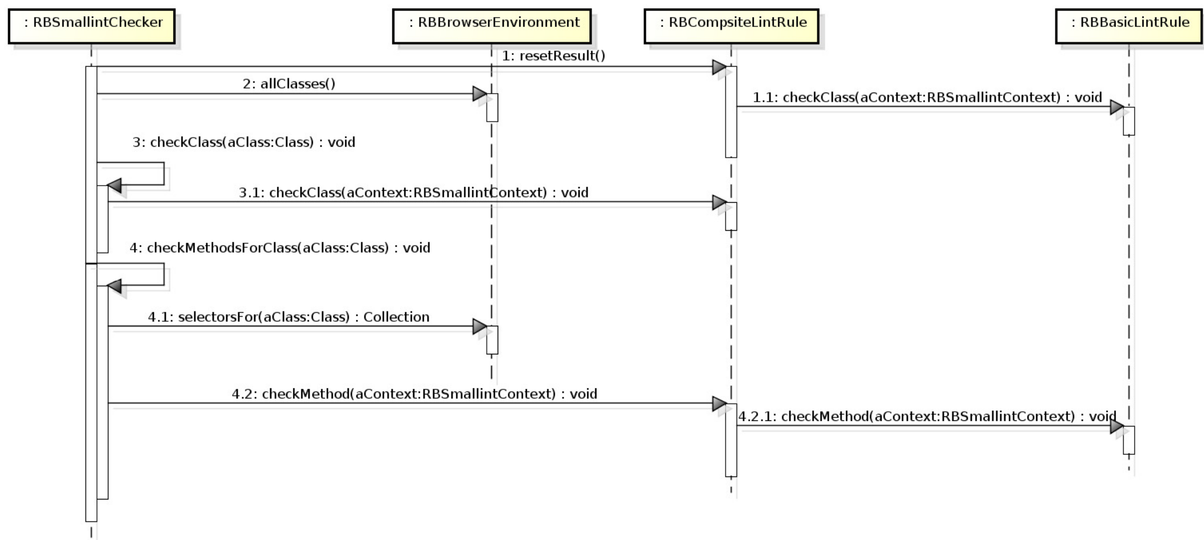


Figura 3.7 Ejecución de un code critic

3.2.5 Patrones de diseño

A contunación detallamos los patrones utiliziados por el framework.

Patrón	**Comentarios**
Signleton	Se aplica en las clases `RBRefactoringManager` y `RBRefactoryChangeManager`
Template Method	Lo utilizan las clases `RBRefactoring` y `RBLintRule`.
Visitor	Se utiliza paral a implementación de la búsqueda y reescritura sobre árboles ASTs.
Composite	Este patrón es aplicado por el framework de rafactoring en la jerarquía de `RBRefactoryChange` y por el farmework de lint para la construcción de reglas compuesta, en la jerarquía de `RBLintRule`.

Tabla 3.2 Patrones utilizados por el framework de Small Lint.

Capítulo 4. Marco de trabajo

4.1 Definición de Code Smell

En este capítulo analizaremos los code critics definidos por Pharo Smalltalk para determinar cuáles de ellos pueden considerarse code smells, y entre éstos, cuáles pueden asociarse con uno o más de los refactoring predefinidos en Pharo.

A tal fin, volveremos sobre nuestros pasos para repasar y precisar el concepto de code smell, de forma tal de poder determinar con precisión cuándo un critic es un code smell.

A continuación repasaremos las definiciones que pueden encontrarse en la bibliografía utilizada para nuestro trabajo.

- Fowler y Beck [Fowler99]: Code smell es una noción que describe el “cuándo” de un refactoring. Los code smell detectan la presencia de *potenciales* problemas que pueden ser resueltos con refactoring, mediante la identificación de ciertas estructuras de código.

- Wake [Wake03]: Da una definición análoga y hace énfasis en el hecho de que si bien no todos los code smells señalan un problema, la mayoría amerita una inspección. Algunos de los problemas son obvios, mientras que otros enmascaran otros problemas. Wake utiliza el término fallas de diseño como sinónimo de code smell. El catálogo de Wake --basado en el de Fowler-- agrupa los smells bajo un conjunto de categorías que arrojan luz sobre la posibilidad de automatizar su detección. Por ejemplo, los smells relacionados con nombres (agrupados bajo la categoría “nombres”) o aquellos que se vuelven evidentes cuando se intenta modificar el código (categoría “acomodando el cambio”) no pueden ser detectados mediante el análisis estático del código, sin recurrir a información adicional.

- Lanza y Marinescu [Lanza06]: Definen los code smells como factores que indican que se están aplicando malas prácticas, o que el diseño es malo o está empezando a volverse frágil. Según los autores, el libro de Fowler no ofrece un método sistemático para detectar posibles bad smells. Los autores usan los términos code smell, falla de diseño, antipattern y discordancia --*disharmony* en inglés-- como intercambiables.

- Stefan Slinger [Slinger05]: Slinger distingue los errores de sintaxis y otras advertencias del compilador de los code smells, dado que éstos señalan

problemas de diseño o malas prácticas de programación, que podrían generar dificultades al momento de modificar el código. Al igual que Wake, advierte que no siempre indican la presencia de un problema y les da un carácter relativo al mencionar que pueden variar de un proyecto a otro o de un dominio a otro. Singer divide los code smells en tres categorías:

- Primitivos: son aquellos que pueden ser observados directamente en el código.
- Derivados: puede deducirse de hechos extraídos del código.
- Code smells que pueden ser detectados por el ojo humano o posiblemente usando una herramienta de CVS.

- Fontana et al [Fontana12]: Estos autores advierten sobre el carácter informal y subjetivo de las definiciones de code smell. Por su parte, entienden por code smell una característica estructural del software que puede indicar la presencia de un problema de código o de diseño y que puede hacer que la evolución y el mantenimiento del software se tornen difíciles. Como los síntomas pueden ser causados por distintos problemas, o incluso por ningún problema, el juicio humano es indispensable para detectar los smells en el contexto de un proyecto dado. Sin embargo, las herramientas automatizadas pueden ayudar a aliviar la tarea de encontrar los smells en sistemas con una base de código grande.

- Van Endment et al [Endment07]: Según estos autores, code smell es una metáfora que describe patrones que generalmente son asociados con un mal diseño o con malas prácticas de programación. Los code smells se usan para encontrar los lugares donde un sistema puede beneficiarse de la aplicación del refactoring. Se diferencian de las inspecciones de bajo nivel que buscan bugs, que comúnmente detectan los inspectores de código (problemas de aritmética de punteros, alocación de memoria, referencias a null, errores en los rangos de los arrays, etc).

A partir de los puntos en común de las definiciones expuestas podemos concluir que:

- No existe una definición formal que permita establecer taxativamente cuándo un problema es un code smell. El concepto mismo de code smell encierra cierta cuota de subjetividad y relatividad.
- Los code smells apuntan a detectar problemas de diseño --problemas estructurales, a diferencia de otros problemas de bajo nivel--, a corregir el

deterioro del diseño y los problemas que pueden hacer difícil el mantenimiento del software (comprensión y modificación), y a forzar el cumplimiento de las buenas prácticas de programación (heurísticas).
- Si bien deben ser considerados como una advertencia, puede haber casos en los que su presencia no constituya un problema.
- Señalan los lugares donde el sistema puede beneficiarse de la aplicación del refactoring. La solución a un code smell puede consistir en la aplicación de uno o más refactorings. También puede ocurrir que no sea necesario aplicar un refactoring. En este caso, una herramienta de análisis debería ser capaz de identificar y marcar estos “falsos positivos”.
- Sólo un subconjunto de los code smells pueden ser observados directamente en el código.

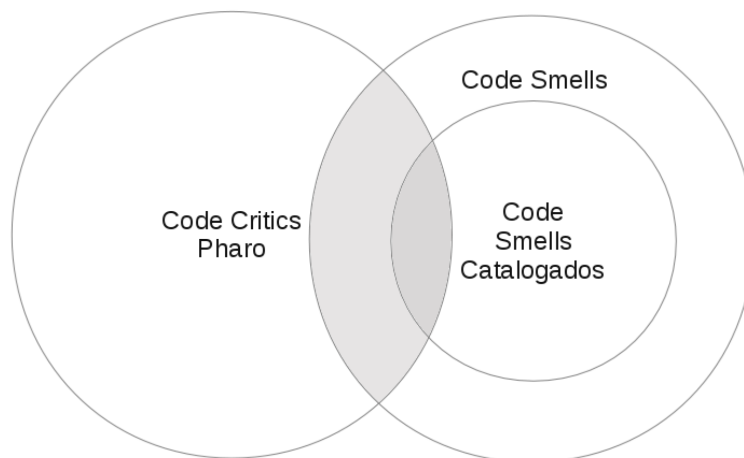


Figura 4.1 En este capítulo se investigarán los code critics que pueden considerarse code smells.

4.2 Relación entre code critics de Pharo y code smells

Como vimos en el capítulo anterior, Pharo Smalltalk agrupa los code critics por categoría. Esta categorización nos sirve como un primer filtro, dado que separa los problemas de diseño de los bugs, los problemas de estilo y las optimizaciones.

Dentro de los problemas de diseño detectados mediante code critics, podemos identificar los siguientes como code smells. Dentro de lo posible, asociaremos cada code smell con el catálogo donde puede hallarse. No siempre es posible puesto que muchos code smells van surgiendo en el desarrollo de las herramientas y no están definidos en un catálogo.

Code Critic	**Code Smell**	**Catálogo**
Class not referenced	Dead Code	Wake
Excessive inheritance depth		
Excessive number of arguments	Long Parameter List	Fowler
Excessive number of methods	Large Class	Fowler
Excessive number of variables	Large Class	Fowler
Inconsistent method classification	-	-
Instance variables defined in all subclasses	Duplicate Code	Fowler
Law of demeter	Message Chain	Fowler
Long methods	Long Method	Fowler
Method defined in all subclasses but not in superclass.	-	-
Methods equivalently defined in superclass	Duplicate Code	Fowler
Methods implemented but not sent	Dead Code	Wake
Refer to class name instead of self class.	-	-
Rewrite super messages to self messages when both refer to same method	-	-
Sends deprecated messages to a known global	Dead Code	Wake
Sends a different super message	-	-
Variable is only assigned a single literal value	-	-
Variable referenced in only one method and always assigned first	Temporary Field	Fowler
Variables not referenced	Dead Code	Wake

Tabla 4.1 Code Smells detectados por Pharo

En las sección 4.3 describiremos los code critics que pueden ser identificados con code smells catalogados. En la sección 4.4 analizaremos los code critics que no pudimos identificar con code smells catalogados a fin de determinar si pueden ser considerados code smells. Por último, en la sección 4.5 estudiaremos la viabilidad de implementar en Pharo los code smells del catálogo de Fowler que no están implementados como code critics.

Como estas secciones hacen referencia a los refactorings que pueden aplicarse para solucionar los code smells, a continuación ofrecemos una lista con los refactorings más relevantes implementados por Pharo.

- Abstraer una variable
- Agregar clase
- Renombrar una clase
- Reemplazar herencia con delegación: RBChildrenToSibilinRefactoring (Fowler)
- Renombrar una clase
- Agregar método
- Renombrar método (Fowler)
- Agregar parámetro a un método (Fowler)
- Eliminar parámetro de un método (Fowler)
- Hacer "inline" de un parámetro
- Extraer método
- Extraer método a un componente
- Extraer una expresión a variable temporal
- "Inline" de un método (Fowler)
- "Inline" de un método desde un componente
- "Inline" de una variable temporal (Fowler)
- Mover un método. (Fowler)
- Mover la definición de una variable (Fowler, Move Field)
- "Pull up" de un método
- "Push down" de un método (Fowler)
- Eliminar un método
- Convertir variable temporal en variable de instancia
- Eliminar una clase
- Dividir una clase
- Abstraer variable
- Agregar variable
- Pull up de variable (Fowler, Push Up Field)

- “Push down” de variable (Fowler, Push Down Field)
- Eliminar variable
- Renombrar variable (Fowler)

4.3 Code Smells catalogados

4.3.1 Dead code

Este code smell identifica elementos del código que no se utilizan. Cada porción del código tiene un costo de mantenimiento, y por lo tanto los elementos no utilizados se deben eliminar.
Los code critics que podemos identificar con este code smell pueden resolverse fácilmente mediante los refactoring que eliminan clases, métodos y variables. El critic “class not referenced” puede resolverse con el refactoring Remove Class, “method implemented but not sent” con Remove Method, y “variable not referenced” con Remove Variable”. Los refactorings podrían aplicarse directamente a partir del resultado de la aplicación del code critic.

4.3.2. Long parameter list

Las listas de parámetros largas deben ser refactorizados porque se vuelven difíciles de entender y tienden a crecer y a volverse inconsistentes. Fowler sugiere que este smell puede eliminarse aplicando alguno de los siguientes refactorings:

1) Replace Parameter With Method: el dato provisto por parámetro se obtiene de un objeto que es conocido.
2) Preserve Whole Object: los datos de varios parámetros están siendo obtenidos de un objetos y pasados como parámetros al método, en cuyo caso puede usarse el objeto directamente como argumento.
3) Introduce Parameter Object: Un mismo grupo de parámetros aparece recurrentemente en distintos métodos. En general existe una lógica subyacente que hace que vayan juntos, y por lo tanto resulta razonable agruparlos en una clase y reemplazar los parámetros individuales por un parámetro que representa al grupo.

Ninguno de estos refactorings se corresponden con un refactoring de Pharo.

4.3.3 Large class

Cuando una clase se vuelve demasiado grande significa que está teniendo demasiadas responsabilidades. Las clases deben mantenerse cohesivas. La cohesión es un concepto fundamental del diseño orientado a objetos. Una clase es altamente cohesiva si las tareas que ejecutan sus métodos están relacionadas y forman un todo coherente. La cohesión facilita la comprensión del código y promueve la reutilización. Cuando una clase crece demasiado, inevitablemente pierde cohesión.

Las clases demasiado grandes suelen tener un número elevado de métodos o de variables. Los code critics “excessive number of methods” y “excessive number of variables” detectan estas desviaciones.

La solución más común para este problema es el refactoring Extract Class. Pharo cuenta, por un lado, con el refactoring Split Class (análogo a Extract Class de Fowler), que crea una nueva clase y la asocia a una variable de instancia; y, por otro lado, con los refactoring Extract Method to Component y Extract Variable to Component, mediante los cuales se pueden pasar a otra clase métodos y variables respectivamente. No existe un refactoring que abarque la operación entera tal como la concibe Fowler.

4.3.4 Long Method

Los métodos largos son difíciles de leer y de mantener. Por el contrario, los métodos cortos son más fáciles de entender y si se les asigna nombres adecuados vuelven el código autodocumental.

Según Fowler, el noventa por ciento de los casos de Long Method se resuelven con el refactoring Extract Method. Pharo cuenta con este refactoring. La ejecución del refactoring implica que el usuario ingrese determinada información, como la porción del código que se va a extraer, el nombre del nuevo método y su lista de parámetros. Pharo también cuenta con el refactoring Extract to Component que permite ubicar el método extraído en la clase de una variable.

En ambos casos, para ejecutar el refactoring a partir del code smell se necesita que el usuario proporciona cierta información adicional. La ejecución del code critic determina la clase y el método afectados. Por su parte, el usuario debe seleccionar en primer lugar el refactoring a aplicar: o bien “extract method” o bien “extract method to component”. En segundo lugar debe seleccionar la porción del método que va a extraer

y definir el número y los parámetros del nuevo método, como se hace actualmente en Pharo.

4.3.5 Message Chain

Este smell se presenta cuando un objeto navega a través de una cadena de objetos para obtener un determinado valor. El problema de estas cadenas de mensajes reside en que cualquier cambio en la configuración de las relaciones entre los objetos puede romper el código.

La ley de demeter es un principio de diseño que pretende evitar la introducción de este tipo de cadenas. Se la define en forma coloquial como “sólo habla con tus amigos, no hables con extraños”. La idea es la misma que la del smell Message Chain: evitar cadenas de mensajes. En Pharo, existe un code critic llamado “Law of Demeter”.

Los refactorings que propone Fowler para este smell son:

- Hide delegate: En uno o más puntos de la cadena se crea un método que oculta las relaciones intermedias.
- Extract method + Move method: en función del uso que se le da al valor obtenido a través de la cadena de mensajes, se extrae un método que aplique esa funcionalidad y se lo mueve a lo largo de la cadena de relaciones.

La herramienta de refactoring podría aplicar la segunda alternativa en un nivel, ofreciendo la ejecución extract method en el método que tiene la cadena de mensajes.

4.3.6 Duplicate Code

El código duplicado es el code smell más común. Existe código duplicado cuando la misma estructura de código se encuentra en más de un lugar.

Según Lanza y Marinescu [Lanza06] el código duplicado daña la calidad de un sistema en dos aspectos: por un lado, aumenta el tamaño del sistema, y por otro, al eliminarse la unicidad de las entidades del sistema, requiere que distintas partes deban evolucionar paralelamente y duplica los errores. Para prevenir estos problemas, existen varios principios de programación cmoo DRY (do not repeat yourself), la regla de sólo una vez y el principio del único punto de control.

Si bien hay distintas estrategias para detectar automáticamente duplicación dentro del código de los métodos, éstas no están implementadas en Pharo Smalltalk. Ahora bien,

podemos considerar código duplicado a una variable que está definida en todas las subclases o a un método que está definido exactamente con el mismo código en una superclase. Estos casos son detectados por los critics “Instance variables defined in all subclasses” y “methods equivalently defined in superclass” respectivamente.

El code critic Instance Variables Defined in all Subclasses puede resolverse directamente con la aplicación del refactoring “pull up instance variable” mientras que Methods Defined Equivalently in Superclass, puede resolverse eliminando el método de la clase, esto es mediante el refactoring Remove Method.

4.3.7 Temporary Field

Temporary Field surge cuando una variable de instancia es asignada y utilizada sólo bajo determinadas circunstancias. Esto es una señal de que la variable debería estar en otra clase.

El code critic “variable referenced in only one method and always assigned first”, puede considerarse como un caso particular de Temporary Field en el cual la variable puede ser eliminada y reemplazada por una variable temporal.

Pharo no cuenta con un refactoring que transforme una variable de instancia en una variable temporal. Esta situación podría resolverse introduciendo una variable temporal del mismo nombre mediante el refactoring “agregar variable de instancia”. Sin embargo, esta solución oscurecería el código, y por lo tanto la descartamos.

Existen otros algoritmos más generales que cuentan la cantidad de métodos que referencian a una variable que eventualmente podrían implementarse como code critics [Griffith11].

4.4 Code Smells no catalogados

4.4.1 Inconsistent method classification

Este code critic detecta inconsistencias en la clasificación de los métodos de una clase respecto de la clasificación de los métodos de la superclase. Si en una clase un método está en una categoría, en sus subclases el método debe permanecer en la misma categoría. Este problema podría resolverse mediante un refactoring que recategorice un método, dándole la posibilidad al usuario de adoptar la clasificación de la superclase o de la clase. No existe este refactoring en Pharo.

4.4.2 Excessive inheritance depth

En [Riel] se afirma que en la práctica, una jerarquía de clases debería ser de un tamaño tal que pueda ser retenido por la memoria de corto plazo de una persona, puesto que de otra forma la jerarquía se vuelve difícil de mantener. En este sentido podemos considerar que Excessive Inheritance Depth es un code smell.

Los refactorings que podrían aplicarse para resolver este smell son Replace Inheritance With Delegation y Collapse Hierarchy, pero no existen en Pharo.

4.4.3 Method defined in all subclasses but not in superclass.

Si un método está definido en todas las subclases de una clase y no está definido en la clase ni en sus ancestros entonces hay un concepto que no está siendo reflejado en el diseño. Este problema podría resolverse simplemente agregando un método abstracto en la clase.

En Smalltalk, un método abstracto es un método que contiene la sentencia `self subclassResponsiblity` (que levanta una excepción cuando el método es invocado). Este método puede agregarse mediante el refactoring Add Method, haciendo que el nombre y el contenido del método sean fijos en lugar de ser definidos por el usuario.

4.4.4 Rewrite super messages to self messages when both refer to same method

Detecta los casos en que se envía un mensaje a super, pero el mensaje no está redefinido en ninguna subclase, por lo cual el efecto sería el mismo si se enviara el mensaje a self. Este code critic parece ser más un bug potencial que un problema de diseño.

Este critic está implementado como una regla de reescritura. Este es un tipo de regla que permite especificar un patrón de búsqueda y un patrón de reescritura (ver Capítulo 3). El usuario puede elegir transformar el código que contiene el problema. De esta forma, el code critic queda contenido el refactoring que lo soluciona.

4.4.5 Refer to class name instead of self class.

Una clase que hace referencia a sí misma podría hacerlo invocando `self class` en lugar de usar el nombre de la clase. Usar el nombre de la clase puede traer problemas en el futuro. La solución consiste en reemplazar la referencia a la clase por `self class`. Si bien no existe un refactoring que aplique directamente a este problema, la regla podría reformularse como una regla de reescritura, de forma tal que el código pueda ser corregido automáticamente.

4.4.6 Variable is only assigned a single literal value

Una variable es asignada una sola vez y el valor asignado es un literal. Por lo tanto, la variable no funciona como tal sino como una constante. Este problema puede resolverse creando un método que devuelva el literal --aplicando Extract Method--, eliminando la asignación y reemplazando el resto de las ocurrencias de la variable con la invocación al método. No existe un refactoring que realice la operación completa, pero podría ser implementado.

4.4.7 Resumen de asociación de Code Critics con Refactorings predefinidos.

Code Smell	Code Critic	Refactoring	Aplicación
Dead code	Class not referenced	Remove Class	Directa
Dead code	Method implemented but not sent	Remove Method	Directa
Dead code	Variable not referenced	Remove Variable	Directa
Long methods	Long method	Extract Method	Input adicional del programador
Duplicate code	Instance variables defined in all subclasses	Pull Up Instance Variable	Directa
Duplicate code	Methods defined equivalently in superclass	Remove Method	Directa
Temporary Field	Variable referenced in only one method and always assigned first	-	-

Tabla 4.2 Asociación de code critics con refactorings predefinidos en Pharo

4.5 Code smells catalogados (Fowler) no implementados en Pharo

En este apartado reseñaremos los code smells del catálogo de Fowler que no son detectados por Pharo, analizando la viabilidad de su implementación como code critics.

4.5.1 Duplicate Code

Como mencionamos más arriba, existen distintas técnicas para detectar código

duplicado. Ejemplo de éstas pueden encontrarse en [Baxter99], [Riega96] y [Lanza06]. Estas técnicas son costosas y difíciles de implementar sin una infraestructura de métricas y por lo tanto consideramos que caen fuera del alcance de este trabajo.

4.5.2 Divergent Change

Wake agrupa este code smell junto con Shotgun Surgery y Parallel Inheritance Hierarchies, bajo la categoría “adaptación al cambio”. A diferencia del resto de los smells, que pueden ser detectados observando el código estáticamente, estos smells suelen hacerse visibles cuando se intenta modificar el código.

Divergent Change se manifiesta cuando una clase necesita ser modificada por distintas razones. Por ejemplo, una clase que exporta datos puede requerir ser modificada cuando cambia el formato de los datos o cuando cambia el destino de la exportación. Esto es una señal de que la clase está teniendo demasiadas responsabilidades (idealmente una clase debería tener una sola responsabilidad: Single Responsibility Principle [Martin05]) que deben ser delegadas.

Para poder detectar la presencia de Divergent Change se necesitaría extender el análisis estático del código en con una dimensión temporal. Asimismo, sería necesaria alguna forma de representar o inferir los motivos de cambio. No es posible implementar la detección de este smell con las capacidades actuales de Small Lint.

En [Palomba13] puede encontrarse un ejemplo de detección de este code smell que usa las herramientas de configuración del código fuente como dimensión temporal.

4.5.3 Shotgun Surgery

En contraposición a lo que ocurre con Divergent Change, Shotgun Surgery se presenta cuando un mismo motivo dispara cambios en varias clases.

En [Lanza06] se discute una estrategia de análisis estático que *anticipa* la aparición de este smell. Si un método es invocado desde muchos otros métodos y estos residen en otras clases, es probable que al cambiar el método invocado se haga necesario realizar cambios en los métodos de las otras clases (por ejemplo, si se modifica la signatura).

Sin embargo, esta técnica puede llevar a la detección de falsos positivos, como en el caso del patrón Visitor, donde los *visitantes* invocan el método `accept` --aunque se

puede evitar si en vez de métodos de distintas clases se consideran los métodos de distintas jerarquías--.

4.5.4 Feature Envy

Feature Envy ocurre cuando un método se enfoca más en la manipulación de los datos de otra clase que en los de la clase a la cual pertenece. Si existe Feature Envy, entonces cuando se haga un cambio en la clase que tiene los datos, estos cambios deberán propagarse a la clase que los manipula. Si, por el contrario, los datos están en la misma clase que los manipula, los cambios en la representación de éstos quedan encapsulados dentro de la clase.

Feature Envy es detectado por distintas herramientas, y la técnica de detección consiste básicamente en medir el nivel de uso de atributos de otras clases por parte de un método, aplicando distintas métricas. En Smalltalk, esto se puede medir contando la cantidad de llamadas a *accessors* de otras clases que realiza un método.

Análogamente a lo que ocurre con Shotgun Surgery, la detección de este smell puede revelar casos donde el comportamiento es retirado como en el caso de los patrones Strategy y Visitor.

4.5.5 Data Clumps

Los data clumps son grupos de dos o tres datos que suelen aparecer juntos como campos en varias a clases o como parámetros en la signatura de varios métodos. Esto implica una duplicación que puede resolverse introduciendo una clase que contenga el grupo de datos.

A partir de un análisis de las variables de las clases y de los parámetros de los métodos podría detectarse este smell, en la medida en que los items estén identificados con nombres equivalentes (es decir, nombres iguales o con características comunes).

4.5.6 Primitive Obsession

Primitive Obsession señala el uso de datos primitivos (strings, números, etc.) en lugar de clases para representar una abstracción. Por definición, no es posible detectar este smell automáticamente automáticamente; sin embargo a veces puede manifestarse

como un caso de Switch Statements , es decir mediante la presencia de sentencias condicionales asociadas a un código (ver apartado siguiente).

4.5.7 Switch Statements

El uso de sentencias de tipo switch es en general una mala práctica dentro de la programación orientada a objetos, ya que a menudo se utiliza para simular el polimorfismo. En Smalltalk no existe una sentencia switch, pero puede ser emulada utilizando sentencias condicionales anidadas.

La detección de este smell puede lograrse fácilmente mediante un análisis estático de los métodos.

4.5.8 Parallel Inheritance Hierarchies

Parallel Inheritance Hierarchies es un caso especial de Shotgun Surgery que se presenta cuando cada vez que se agrega una subclase en una jerarquía es necesario agregar una subclase en otra jerarquía. A menudo, este problema se refleja en los nombres de las clases de ambas jerarquías: las subclases de las dos jerarquías tienen los mismos prefijos. Este caso podría detectarse mediante análisis de código.

4.5.9 Lazy Class

La existencia de una clase implica un costo de comprensión y mantenimiento. Es por eso que una clase que no hace lo suficiente debe ser eliminada.

Existen distintos enfoques para determinar si una clase es *lazy*. En [Slinger05] se cuentan la cantidad de variables, métodos y líneas de código. En [Lanza06] se define una estrategia de detección para clases grandes (God Class) que toma en cuenta el uso intensivo de datos de otras clases, la complejidad funcional de la clase y el nivel de cohesión.

Esta métrica podría usarse en forma inversa para determinar si una clase no está haciendo lo suficiente para garantizar su existencia.

4.5.10 Speculative Generality

Este smell está relacionado con el principio de *simplicidad* de las metodologías ágiles, que promueve la implementación del conjunto de requerimientos actuales sin intentar

anticipar la implementación de requerimientos futuros. Esta anticipación generalmente se manifiesta mediante la presencia de clases, métodos o variables que no tienen clientes, o cuyos únicos clientes son tests.

La naturaleza abstracta de este smell y su relación con los requerimientos hace difícil su traducción a métricas de código. Sus distintas manifestaciones pueden asociarse también con otros smells como Lazy Class o Dead Code.

4.5.11 Middleman

Se dice que una clase es un Middleman (intermediario), cuando la mayor parte de sus métodos se limitan a delegar en otra clase. Fijando un umbral, esta característica puede ser chequeada fácilmente mediante análisis estático, y de hecho, distintos herramientas son capaces de detectar este smell [Landa06] y [Slinger05].

4.5.12 Inappropriate Intimacy

Intimacy, en este caso, significa acoplamiento. Puede darse porque existe una interdependencia grande entre dos clases, por ejemplo, mediante una relación bidireccional; o porque una clase accede a aspectos “privados” (de representación interna) de otra clase.

4.5.13 Alternative Class with different Interfaces

Este smell ataca inconsistencias en los nombres de las interfaces y por lo tanto es indetectable mediante análisis de código.

4.5.14 Data Class

Data Class identifica clases que sólo poseen datos y no realizan ningún cómputo. En Smalltalk, estas clases se caracterizan por ser todos sus métodos *accesors*. Esto puede ser comprobado fácilmente por medio de una regla lint.

4.5.15 Refused Bequest

Refused Bequest surge cuando una subclase hace poco o ningún uso de los métodos y variables heredados de su superclase, lo que revela que existe algún tipo de problema en el diseño de la jerarquía, la cual debería ser reestructurada. Si bien este problema es algo sutil, se han investigado técnicas para detectarlo basadas en la cantidad de métodos y variables heredados que una clase utiliza [Slinger05].

Una forma más flagrante de Refused Bequest --Wake [Wake03] la llama rechazo "honesto"-- se da cuando una clase rechaza explícitamente el protocolo definido por la superclase, ya sea levantando una excepción (especialmente en el caso de Smalltalk, cuando la excepción es "should not implement") o dejando el cuerpo del método vacío. Esta forma es fácilmente detectable mediante un análisis simple del AST de los métodos.

4.5.16 Comments

Los programadores usan con frecuencia los comentarios dentro de los métodos para subsanar un código que es difícil de entender. Un diseño correcto debería conducir a métodos cortos y autoexplicativos. Por consiguiente, la presencia de comentarios dentro de los métodos puede interpretarse como un potencial problema de diseño.

Desde el análisis estático es fácil detectar la presencia de comentarios dentro de los métodos. Las solución más común a este problema consiste en hacer Extract Method de un bloque de código explicado, lo que requiere la intervención del usuario.

4.5.17 Resumen de Code Smells no detectados mediante Code Critics

Code Smell	**Puede ser implementado como code critic**	**Comentarios**	**Referencia implementación**
Duplicate Code	No	Algoritmos complejos	
Divergent Change	No	Requiere un método para manejar la dimensión temporal	
Shotgun Surgery	Algunos casos	Si bien requiere de la dimensión temporal, existen algoritmos que permiten anticipar su aparición (cf. [Lanza06])	[Lanza06]
Feature Envy	Sí		[Lanza06]
Switch Statements	Sí		Directo
Parallel Inheritance Hierarchies	Algunos casos	Subclases de distintas jerarquías con los mismos prefijos.	
Lazy Class	Sí	Clases con poco código	Directo
Speculative Generality	No	Se define en relación con los requerimientos	
Middle Man	Sí		[Lanza06] y [Slinger05]
Data Class	Sí	Clases que sólo contienen accesors	
Primitive Obsession	No	Se define en relación a la representación de abstracciones en el programa.	
Refused Bequest	Algunos casos	-Utilización de métodos de la superclase -Rechazo explícito de la semántica definida por la superclase	[Slinger05]
Comments	Sí		Directo
Inappropriate Intimacy			
Alternative Class with different Interface	No	Apunta a un aspecto semántico indetectable	

Tabla 4.3 Resumen de code smells no detectados mediante code critics

Capítulo 5. Diseño de la herramienta

En este capítulo analizaremos el diseño y la implementación de la herramienta de asistencia durante el proceso de refactoring.

5.1. Diseño

5.1.1 Asociación de code critics con refactorings

El primer paso para desarrollar la herramienta consiste en encontrar una forma de relacionar un code critic con los refactorings capaces de solucionarlo y ejecutar uno de estos refactorings a partir de los resultados de la aplicación del code critic.

Para lo primero, se necesita una estructura tabular que define para cada code critic, cero o más refactorings capaces de resolverlo, como se muestra en los siguientes ejemplos.

Code Critic	**Refactoring**
Class not referenced	Remove Class
Long method	Extract Method, Extract Method to Component

El Critic Browser debe poder acceder a esta estructura para sugerir los refactorings en base a la regla que detectó encontró el problema. En Smalltalk suelen implementarse este tipo de estructuras globales aplicando alguna forma de patrón Singleton [Gamma94]. Por ejemplo, Pharo hace disponibles las reglas lint a través de un conjunto de métodos de clase implementados en `RBCompositeLintRule`.

Para los refactorings de code critics, comenzaremos por definir una clase que contenga la información de ésta tabla y que implemente el patrón Singleton. La llamamos `CriticRefactoringAssociations`. Esta clase contendrá un diccionario con las asociaciones. Las claves del diccionario son las reglas lint, mientras que los valores son colecciones de refactorings. El diccionario se crea en el método initialize de esta clase.

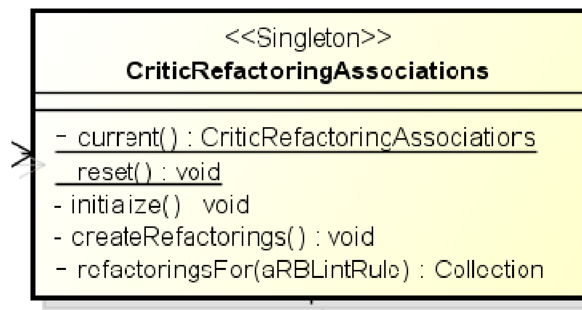


Figura 5.1 Punto de entrada (Facade) para el framework de asociaciones.

Cuando un usuario selecciona el resultado de un critic, el Critic Browser puede usar esta clase para buscar los refactorings disponibles para solucionar el problema. A tal fin, se agrega el método `refactoringFor: aRBLintRule`.

5.1.2 Aplicación de un refactoring a partir del resultado de la ejecución de un code critic

Hasta aquí hemos definido la asociación entre code critics y refactorings. Ahora evaluaremos cómo se podría aplicar un refactoring elegido para un elemento señalado por un code critic.

En primer lugar, veamos cómo representa el Critic Browser los resultados de la aplicación de una regla.

En líneas generales, cuando el Critic Browser obtiene la lista de reglas a través de `RBCompositeLintRule`, esta clase recorre la jerarquía de `RBLintRule` y crea una instancia para cada una de las reglas. Luego, este conjunto de reglas es aplicado al *enviroment* sobre el cual se está trabajando, como se explicó en el Capítulo 3.

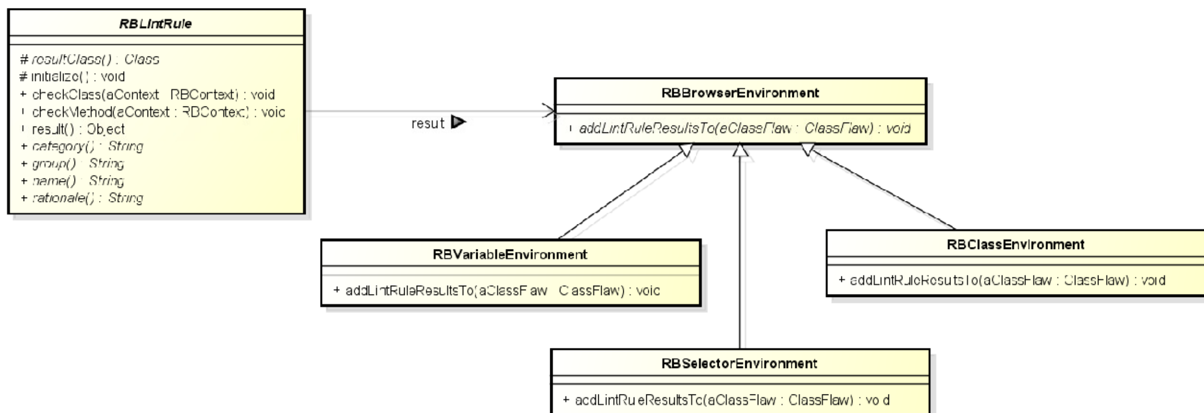


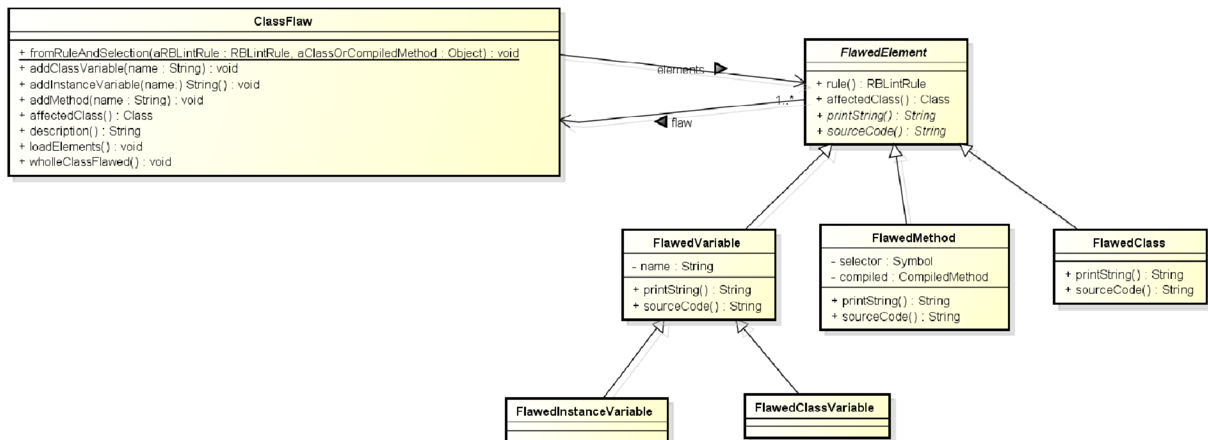
Figura 5.2. Resultados de reglas lint

Una vez ejecutada, la regla guarda en la variable de instancia `result` los elementos que no pasaron el chequeo. Estos elementos están representados mediante un `RBBrowserEnvironment`. Según el tipo de elemento que analiza la regla, el environment será un `RBVariableEnviroment`, un `RBSelectorEnviroment`, un `RBParseTreeEnviroment` o un `RBClassEnviroment`. Las reglas definen el tipo de su resultado mediante el método `resultClass`. Luego, los métodos `checkMethod:` y `checkClass:` son los encargados de agregar los elementos al enviroment. Asimismo, la clase `RBLintRule` contiene el método `critics`, que extrae del enviroment los elementos afectados. Estos elementos son los que se muestran cuando se selecciona una regla en el Critic Browser, y su clase depende del enviroment. Si el enviroment es un `RBClassEnviroment`, entonces los elementos serán clases; si el enviroment es un `RBSelectorEnviroment`, entonces los elementos serán métodos compilados (`CompiledMethod`); y por último, si el enviroment es un `RBVariableEnviroment`, entonces los elementos serán clases, y dentro del enviroment se guardan los nombres de las variables afectadas asociadas a esa clase. Como podemos observar, no hay una representación consistente de los resultados de las reglas. Esta es una deficiencia del diseño original que tendremos que remediar en nuestra herramienta.

Entonces, tenemos por un lado un refactoring, es decir una instancia de `RBRefactoring`, y por otro lado un objeto que puede ser un `CompiledMethod`, una clase o una colección de nombres de variables. El hecho de contar con objetos de distintas clases como elementos a refactorizar presenta las siguientes desventajas:

- En algún punto será necesario preguntar por la clase del elemento seleccionado.
- En el caso de las variables, no se puede elegir a cuál de ellas aplicar el refactoring.

Por lo tanto, comenzamos por definir una abstracción para representar los resultados de los critics.



Figrua 5.3 Abstracción de resultados de reglas lint

`ClassFlaw` representa un defecto o problema en una clase. `ClassFlaw` tiene una referencia a la lista de elementos de la clase --métodos, variables o la clase como un todo-- afectados por la falla. Cuando un critic afecta a una clase entera (por ejemplo, en el caso de Class Not Referenced), la falla tendrá un solo elemento de la clase FlawedClass. Cada subtipo de elemento provee información concreta sobre el elemento (`name` devuelve el nombre de la variable, `selector` devuelve el selector del método, etc.)

Junto con esta nueva jerarquía, definimos en la jeraquía de `RBBroweserEnviroment` el método `addLintRuleResultTo: aClassFlaw`, el cual utiliza la ténica de *double-dispatching* para recolectar los resultados de la regla en una instancia de `ClassFlaw`. Por lo tanto, el tipo de *environment* define el tipo de elemento.

Clase del resultado	Clase del elemento
RBVariableEnvironment	FlawedVariableInstanceVariable FlawedClassVariable
RBSelectorEnviroment	FlawedMethod
RBClassEnviroment	FlawedClass

Tabla 5.1 Clase del resultado de una regla lint según el enviroment al que aplica

Ahora, al momento de ejecutar el refactoring contamos por un lado con un refactoring (una instancia RBRefactoring) y por otro lado con un FlawedElement.
Tomemos como ejemplo el critic Method Implemented but Not Sent y el refactoring Remove Method. El Refactoring Browser de Pharo instancia el refactoring RBRemoveMethodRefactoring de la siguiente forma:

```
^ RBRemoveMethodRefactoring
      model: environment
      removeMethods: selectors
      from: class
```

Si comparamos este código con el código que se utiliza para instanciar otros refactorings, observamos que no existe un protocolo estándar para realizar la instanciación. Por ejemplo, el código que utiliza el Refactoring Browser para instanciar el refactoring Pull Up Method es el siguiente:

```
^ RBPullUpMethodRefactoring
      model: environment
      pullUp: selectors
      from: class
```

De este hecho podemos concluir que no nos basta con mantener una asociación entre code critics y refactorings a través de un diccionario. Necesitamos una representación de la asociación que además sirva como *Adapter* para instanciar el refactoring a partir del resultado de la aplicación de la regla y del elemento seleccionado por el programador. En la próxima sección veremos otro comportamiento de las asociaciones que nos induce a modelarlos usando herencia.

A tal efecto, creamos una nueva clase, a la cual llamamos `CriticRefactoringFactory`. `CriticRefactoringFactory` define el método abstracto `refactoringFor: aFlawedElement`. Cada asociación entre code critic y refactoring deberá definirse como una subclase de `CriticRefactoringFactory`. La clase `CriticRefactoringAssociations` será la encargada de instanciar las asociaciones usando un mecanismo similar al que se usa para instanciar los reglas lints (ver Capítulo 3).

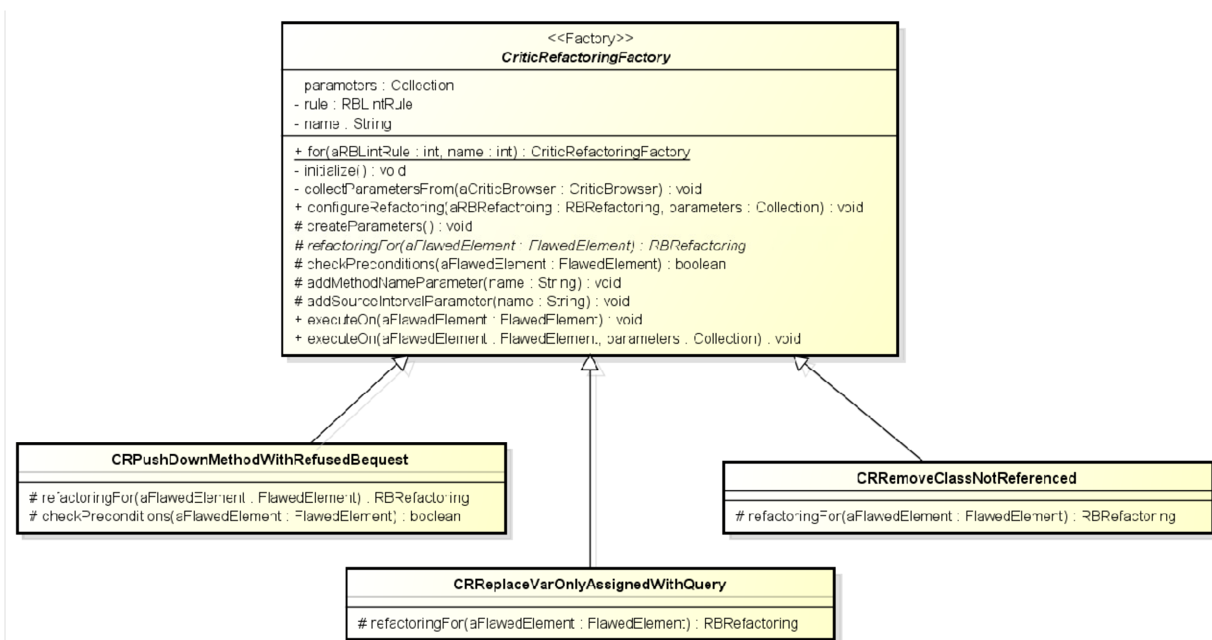


Figura 5.4. `CriticRefactoringFactory`: asocia un code critic con un refactoring

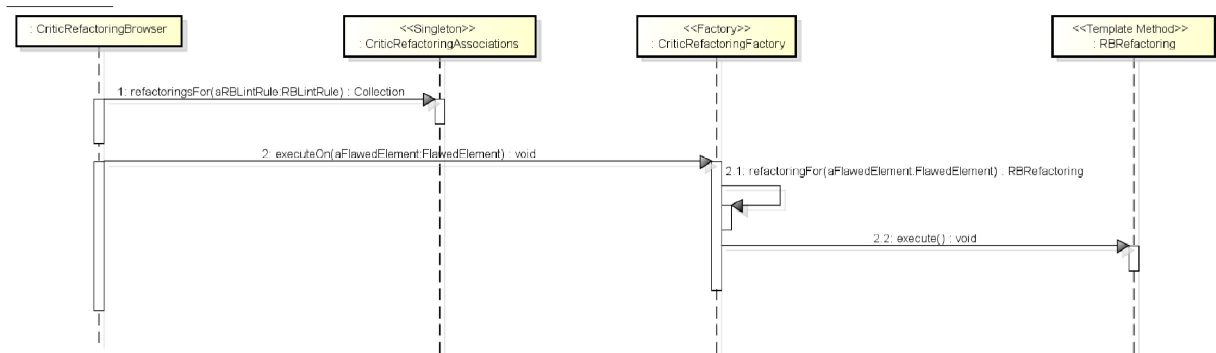


Figura 5.5 Diagrama de secuencia de la ejecución de un refactoring para un code critic

Volviendo sobre ejemplo de Remove Method, definimos una subclase `CRRemoveMethodNotSent` para la asociación, e implementamos el método `refactoringFor:`.

```
CRRemoveMethodNotSent >> refactoringFor: aFlawedMethod
^RBRemoveMethodRefactoring
        removeMethods: (Array with: aFlawedMethod selector)
        from: aFlawedMethod affectedClass.
```

Como veremos en la sección 5.2, puede ocurrir que para resolver un code critic necesitemos aplicar más de un refactoring. Como el framework de refactoring no permite componer refactorings, deberemos extenderlo, agregando una subclase de `RBRefactoring` que implemente el patrón Composite.

5.1.3 Precondiciones

En algunos casos podemos encontrarnos con la necesidad aplicar algún tipo de restricción a la posibilidad de aplicar un refactoring para solucionar un code critic. Tomemos por ejemplo la aplicación del refactoring Push Down Method para solucionar el code critic Refused Bequest. Más adelante mostraremos cómo implementamos este critic. Por el momento, basta con saber que podemos detectar un método de una clase que contiene ese code smell. Según hemos visto, una forma de resolver este problema es haciendo Push Down del método en la superclase que lo define y eliminándolo de la

clase que tiene el Refused Bequest. Ahora bien, si el método está definido en una superclase lejana, el alcance del cambio puede ser muy grande, y con consecuencias difíciles de calcular. Por lo tanto, resultaría conveniente limitar los casos en que se sugiere la aplicación automática de este refactoring para solucionar un Refused Bequest.

Esto nos lleva a definir un nuevo método en la clase `CriticRefactoringFactory`, que será utilizado para verificar si el refactoring es aplicable al elemento señalado.

```
CriticRefactoringFactory >> checkPreconditions: aFlawedElement
    ^true.
```

Por defecto, el método devuelve verdadero. Las subclases de `CriticRefactoringFactory` pueden redefinirlo eventualmente para verificar precondiciones adicionales antes de aplicar el refactoring. Estas precondiciones no son las mismas que las de los refactorings. A diferencia de las precondiciones de los refactorings, que verifican que su aplicación preserve el comportamiento, las precondiciones de la asociaciones entre code critics y refactorings acotan los casos en que es conveniente resolver el code critic automáticamente.

A continuación mostramos el diagrama de secuencia actualizado.

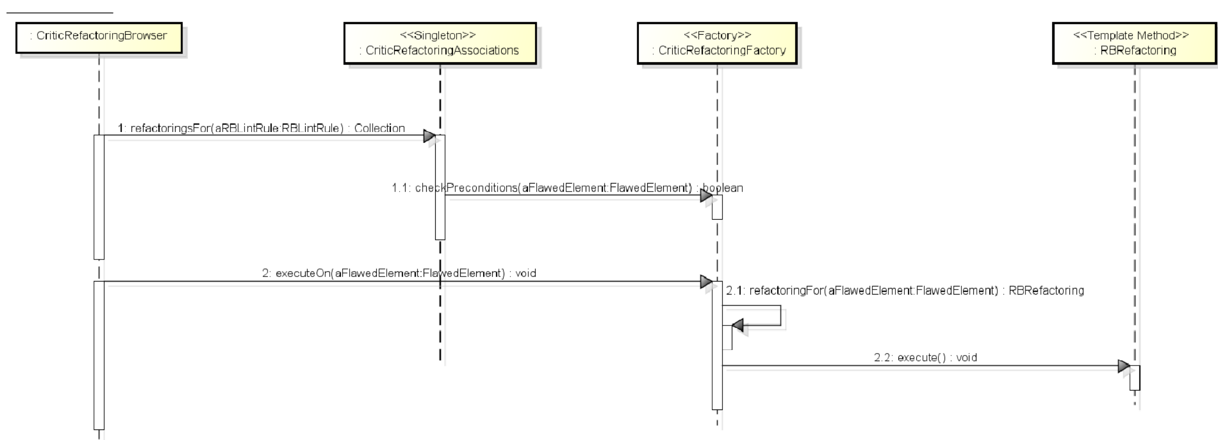


Figura 5.6: Diagrama de la figura 5.5 actualizado con la verificación de precondiciones

5.1.5 Parámetros adicionales

Otro aspecto a resolver es el de la información adicional que se requiere para la aplicación de ciertos refactorings (ver Capítulo 4). Por ejemplo, para aplicar el refactoring Extract Method a un método marcado con el critic Long Method el programador debe seleccionar la porción del código que quiere extraer, el nombre del nuevo método y las variables. Otro caso de refactoring que requiere input adicional es Extract Method To Component, que también puede aplicarse al critic Long Method. En este caso, además de los parámetros ya mencionados, el programador debe indicar la variable componente que contendrá el nuevo método junto con su clase.

El Refactoring Browser setea estos datos utilizando las opciones de los refactorings. Cada opción se configura mediante un *block closure* que toma dos parámetros. El primero es el refactoring y el segundo es un valor adicional que se puede utilizar como valor por defecto. Por ejemplo, el Refactoring Browser usa el siguiente código para establecer el nombre del método a ejecutar:

```
aRefactoring
    setOption: #methodName
    toUse: [ :ref :name | self requestMethodNameFor: name ];
```

Al ser ejecutado, el refactoring invoca este bloque y le pasa a través de `:name` la signatura del método que pudo deducir en función de la porción de código seleccionada.

La siguiente sentencia abre el diálogo que permite cargar la signatura del nuevo método (nombre del método y parámetros).

En el caso de la refactorización de critics, debemos considerar como adicional toda aquella información que no se encuentre contenida en el resultado de la aplicación de la regla, es decir, en la instancia de `FlawedElement` que le pasamos al `CriticRefactoringFactory`. Por ejemplo, en el caso de Extract Method necesitamos, además de la signatura del nuevo método, la porción del código que se va a extraer. Esto obedece a que en el Refactoring Browser el refactoring se ejecuta en el contexto de la edición del código fuente, mientras que en nuestra herramienta se ejecuta desde la detección de un defecto.

Consideramos que si implementamos la recolección de estos datos dentro del Critic Browser corremos con algunas desventajas:

- El Critic Browser no muestra los elementos a refactorizar en forma clara.
- La secuencia de pasos para la refactorización puede ser confusa, puesto que una vez que se selecciona el refactoring se pierde el contexto.
- El Critic Browser es complejo y el resultado de extenderlo resultaría poco modular.

Por lo tanto, optamos por desarrollar un nuevo diálogo o *browser* para la ejecución de los refactoring de critics. Este diálogo se abre cuando el programador selecciona un resultado de un code critic y elige la opción *Refactoring* en el menú de contexto o en la barra de herramientas.

Dentro del diálogo, el programador puede seleccionar el elemento a refactorizar y el refactoring a aplicar. A continuación, el diálogo solicita la información adicional, en caso de ser necesario.

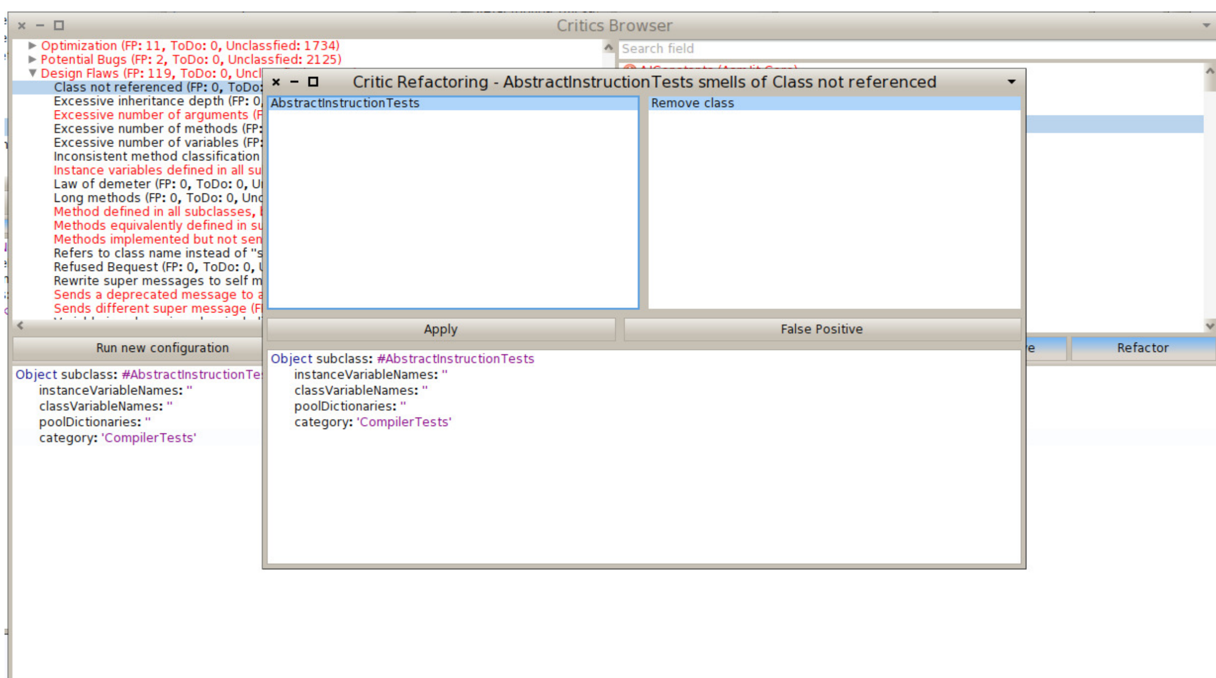


Figura 5.7 Diálogo para refactoring de code critics

Hasta aquí hemos visto el aspecto del problema relacionado a la interfaz gráfica. Con respecto al modelo, para que nuestra herramienta sea genérica y la asociaciones entre code critic y refactorings puedan ser agregadas fácilmente, necesitamos establecer alguna forma de definir los parámetros adicionales que necesita un refactoring. Por

ejemplo, tomando el caso de Long Method -> Extract Method, deberíamos definir en la clase que representa la asociación que el refactoring necesita los siguientes parámetros para ser ejecutado:

- La porción de código que se va a extraer.
- La signatura del nuevo método.

A tal fin, debemos agregar a la clase `CriticRefactoringFactory` una variable de instancia que contenga una colección con la definición los parámetros adicionales. Los parámetros serían instancias de una clase `CriticRefactoringParameter` que sería especializada para cada tipo de parámetro posible. Por ejemplo: signatura de un método, porción de código de un método, etc. Cada parámetro se recolectaría colaborando con el Critic Refactoring Browser (usando *double dispatching*). El mecanismo de recolección de parámetros debe ser implementado en forma análoga a las opciones de los refactorings, es decir, mediante el uso de bloques, dado que éstos permiten desacoplar el mecanismo de interfaz gráfica que se utiliza para el ingreso de los parámetros, de la aplicación del refactoring.

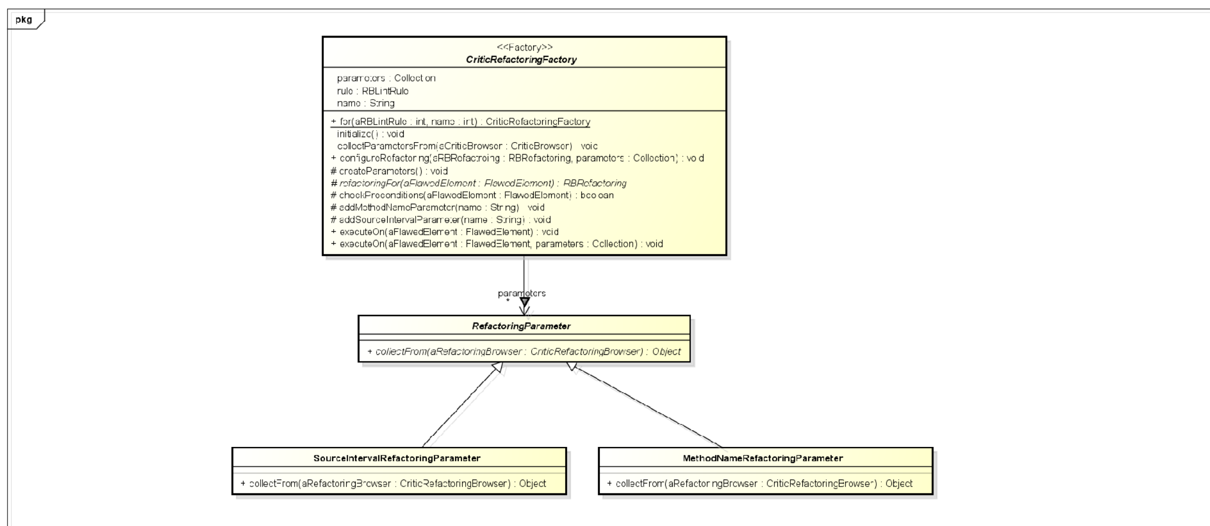


Figura 5.8 Parámetros de CriticRefactoring

Por último mostramos el diagrama de secuencia actualizado con la recolección de parámetros.

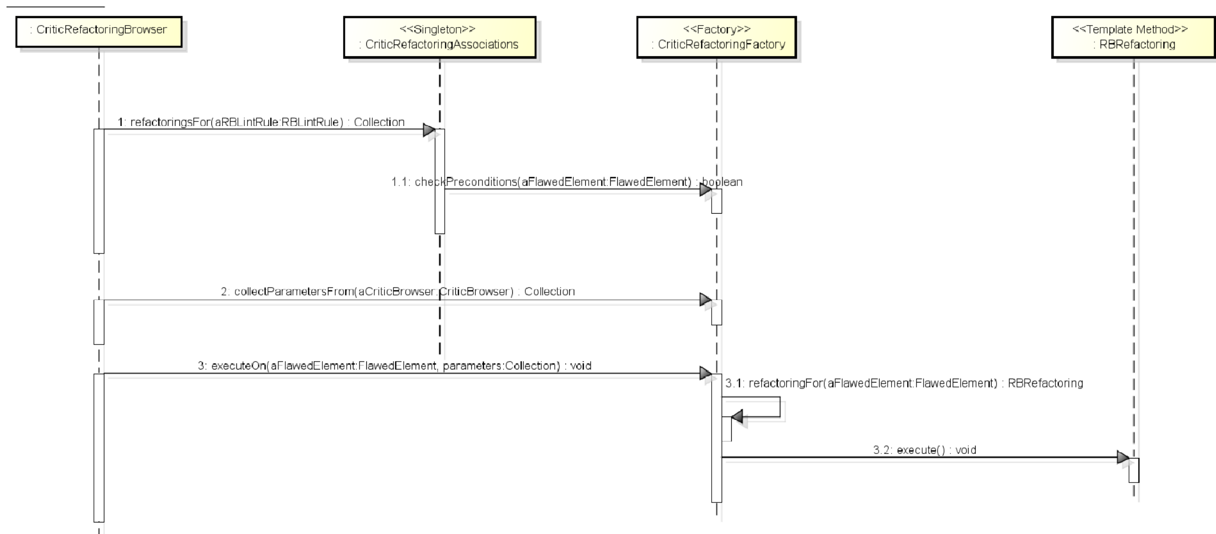


Figura 5.9. Diagrama de secuencia final de la ejecución de un refactoring

5.1.5 Integración con el Class Browser

Entre los criterios técnicos para las herramientas de refactoring que postula Don Roberts [Roberts99] se encuentra el de integración con las herramientas de desarrollo. Es decir, para que la herramienta sea práctica, tiene que estar integrada en el ambiente de desarrollo.

Podemos afirmar que el Code Critic Browser no cumple totalmente con este requerimiento. Por un lado, la ejecución de los critics es lenta, lo que desalienta su utilización durante el desarrollo. Por otro lado, supongamos que un programador está trabajando sobre una clase. Para recibir una sugerencia de refactoring debe abrir el Critic Browser y entre los problemas detectados buscar aquellos que se refieran a la clase en cuestión. Claramente, esta situación incumple el requerimiento de integración. Por lo tanto, consideramos necesario introducir algún mecanismo para realizar en el análisis de una clase en forma directa. Por ello, incluimos un botón que permite analizar la mediante el Critic Browser la clase que se está editando e incorporamos una alerta en el Class Browser.

En el Capítulo 6 se describe mediante un ejemplo el uso de la herramienta.

5.2 Implementación de asociaciones entre code critics y refactorings

Diseñada la herramienta, el próximo paso consiste en implementar un conjunto de asociaciones entre critics y refactorings que cubran los distintos casos analizados en el capítulo anterior para, de esta forma, probar su viabilidad. Estos casos son:

1. Asociar un critic existente con un refactoring existente
2. Crear un nuevo code critic y asociarlo a un refactoring existente.
3. Crear un nuevo refactoring para solucionar un code critic existente.
4. Asociar un code critic existente con un refactoring existente que requiera parámetros adicionales.

En la siguiente tabla detallamos las asociaciones que implementaremos para cubrir cada uno de los casos:

Caso	**Critic**	**Refactoring**
1	Class Not Referenced	Remove Class
2	Refused Bequest (nuevo)	Push Down Method
3	Variable is Only Assigned a Single LIteral Value	Replace Instance Variable With Query (nuevo)
4	Long Method	Extract Method

Tabla 5.1 Asociaciones implementadas

5.2.1 Remove Class Not Referenced

Creamos `CRRemoveClassNotReferenced` y la hacemos heredar de `CriticRefactoringFactory`.
Luego, implementamos el método `refactoringFor:`, como se ve en el siguiente cuadro:

```
CRRemoveClassNotReferenced >> refactoringFor: aWholeFlawedClass
^RBRemoveClassRefactoring
            classNames: (Array with: aWholeFlawedClass
affectedClassName).
```

Luego, necesitamos agregar el refactoring al diccionario de asociaciones. Esto se codifica en el método `createAssociations` de la clase `CriticRefactoringAssociations`., que mostramos a continuación:

```
CriticRefactoringAssociations >> createAssociations
self add: (CRRemoveClassNotReferenced name: 'Remove class')
     for: RBClassNotReferencedRule.
```

Hecho esto, el refactoring queda disponible en el menú de contexto del Critic Browser.

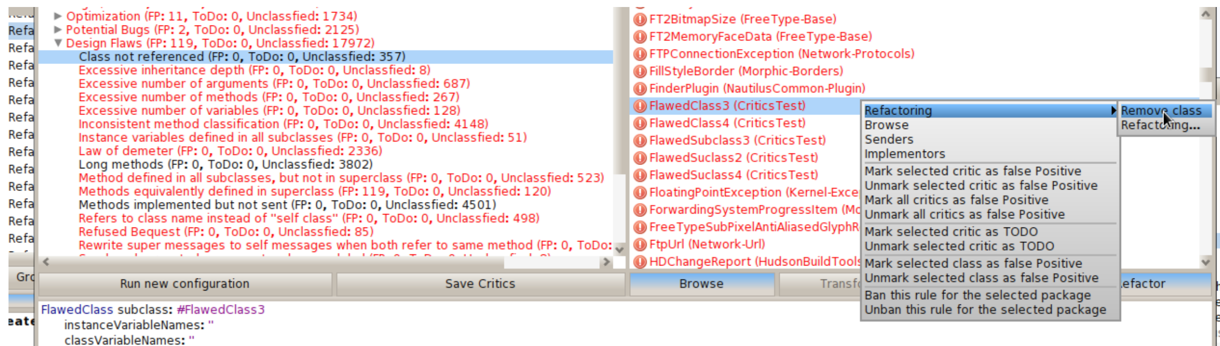


Figura 5.10 Menú de contexto para refactoring de code critics

5.2.2 Push Down Method with Refused Bequest

Como hemos visto en la sección 4.5.15, existen distintas estrategias para detectar un Refused Bequest. En nuestro ejemplo, nos limitaremos a detectar el caso en el cual una clase rechaza explícitamente el protocolo heredado levantando una excepción que indica que no puede manejar el mensaje recibido.

Recordemos que en Smalltalk este caso se presenta generalmente en forma de un método heredado que se envía a `self` el mensaje `shouldNotImplement`. El método `shouldNotImplement`, definido en la clase `Object`, se limita a levantar una excepción, abortando la ejecución del programa, ante lo que sería un error de programación.

En el Capítulo 3,en el contexto de la introducción al framework Small Lint, ya hemos implementado un code critic Refused Bequest que detecta la invocación a `shouldNotImplement`.

Fowler propone dos métodos para solucionar este bad smell. El primero consiste en crear una clase hermana y hacer Push Down de los métodos que no se utilizan, mientras que el segundo --más adecuado si bien más complejo-- supone reemplazar la herencia con delegación. Para nuestro ejemplo haremos la asociación con el primer refactoring.

Tomemos como ejemplo la jerarquía de `Timespan` de Pharo Smalltalk. Un timespan es una magnitud y se define como "una duración de tiempo que comienza en una fecha específica". Las instancias de `Timespan` entienden el mensaje `daysInMonth`, el cual se utiliza para saber la cantidad de días que tiene el mes en el que está inserto el periodo de tiempo en cuestión. Ahora bien, dentro de la jerarquía de `Timespan` podemos encontrar la clase `Year`. Queda claro que para un año no tiene sentido preguntar cuántos días tiene el mes. Los diseñadores resolvieron esta situación haciendo que la clase `Year` rechace el método `daysInMonth`. En la figura 5.12 se ilustra el problema mediante una versión simplificada de la jerarquía de `Timespan`. En la figura 5.13 podemos ver la jerarquía refactorizada según propone Fowler.

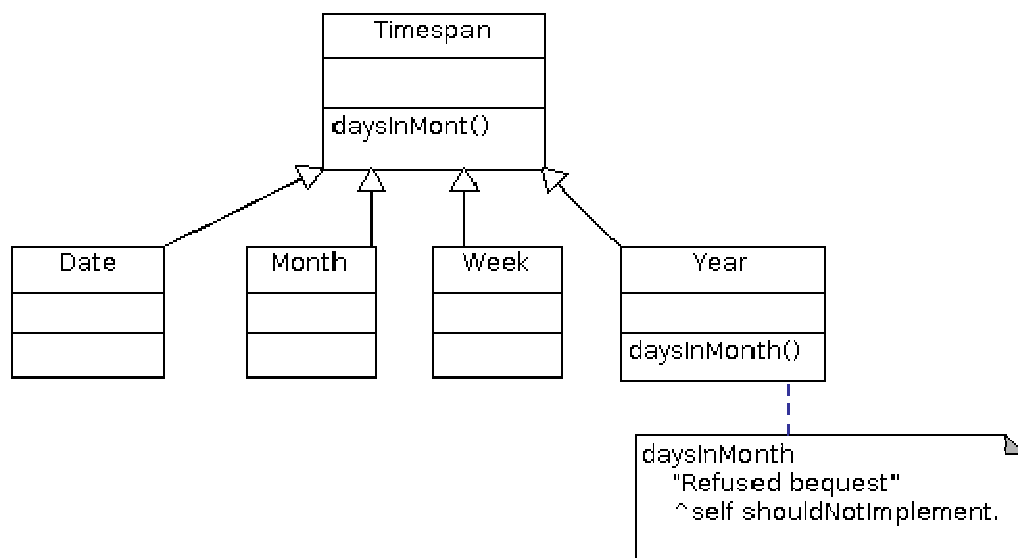


Figura 5.11 Ejemplo de Refused Bequest en la jerarquía de Timespan

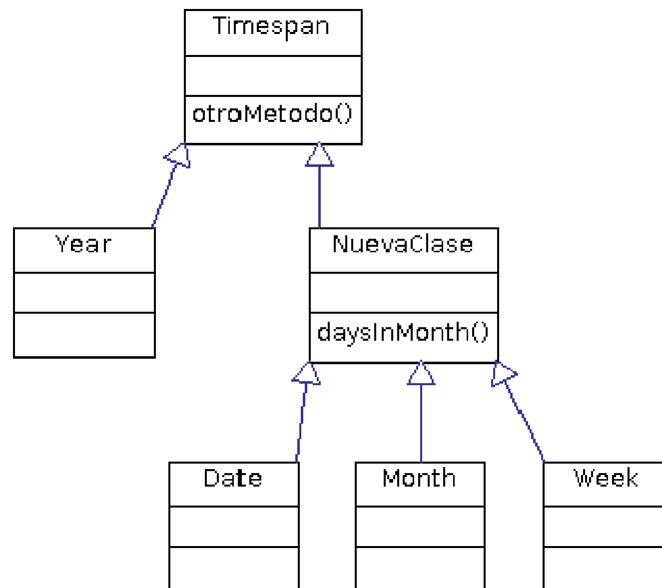


Figura 5.12 Ejemplo de Refused Bequest refactorizado.

Con la refactorización de Refused Bequest buscamos demostrar que es posible implementar un nuevo code critic y asociarlo a un refactoring existente. Por consiguiente, vamos a implementar una versión simplificada del refactoring de Fowler, de forma tal de evitar complejidades que abordaremos en las siguientes asociaciones. Nos limitaremos a hacer el Push Down del método en la superclase para luego eliminar el método con el Refused Bequest. La creación de la nueva clase hermana podría realizarse en un segundo paso.

Comenzamos por crear la clase `CRPushDownMethodWithRefusedBequest` haciéndola extender de `CriticRefactoringFactory`. A continuación tenemos que definir el método `refactoringFor:`. Ahora bien, la transformación no involucra un solo refactoring como en los casos vistos hasta ahora, sino que consta de dos pasos: hacer el Push Down del método en la superclase y eliminar el método de la clase que tiene el Refused Bequest. Para salvar esta situación creamos una subclase de `RBRefactoring` llamada `RefactoringSequence` que implemente el pattern Composite. De esta forma, los clientes de la clase `CriticRefactoringFactory` continuarán viendo la secuencia como un refactoring simple. El siguiente cuadro muestra el código que crea la secuencia de refactorings:

```
RPushDownMethodWithRefusedBequest >> refactoringFor:
aFlawedMethod
    |sequence|
    sequence:= RefactoringSequence new.
    sequence addRefactoring:
        (RBPushDownMethodRefactoring
            pushDown: (Array with: aFlawedMethod selector)
            from: aFlawedMethod affectedClass super).
    sequence addRefactoring:
        (RBRemoveMethodRefactoring
            removeMethods: (Array with: aFlawedMethod selector)
            from: aCompiledMethod methodClass).
```

Este refactoring asume que el método en cuestión está definido en la superclase directa de la clase que tiene el Refused Bequest. Una implementación más general llevaría realizar cambios riesgosos con un alcance demasiado amplio, difícil de controlar. Especificamos esta restricción mediante las precondiciones mencionadas más arriba. Sólo si se cumplen éstas, la herramienta ofrecerá la opción de aplicar el refactoring. El código que crea estas precondiciones puede verse en el siguiente cuadro:

```
RPushDownMethodWithRefusedBequest >> checkPreconditions:
aFlawedMethod
    (RBClass
        existingNamed: aFlawedMethod affectedClassName
asSymbol)
     directlyDefinesMethod: aFlawedMethod selector.
```

5.2.3 Replace Variable Only Assigned a Single Literal Value with Query

Entre los code critics de fallas de diseño de Pharo se encuentra Variable Only Assigned a Single Literal Value. Este critic busca las variables que sólo son asignadas con un solo valor literal. Una variable a la que sólo se le asigna un valor literal no tiene sentido como tal y puede ser reemplazada por un método que devuelva el valor. Como Pharo

no cuenta con un refactoring que extraiga una variable de instancia a un query, tenemos que crearlo. Lo llamamos `RBReplaceInstVariableWithQuery`.

Precondiciones:

- La variable es asignada una sola vez y el valor asignado es un literal

Transformaciones:

- Extraer el literal a un método que funcione como query.
- Eliminar la asignación.
- Reemplazar las ocurrencias de la variable con la invocación del método.
- Eliminar la variable.

Opciones:

- Nombre del método query (ver sección 5.2.3).

Una vez implementado el refactoring, la asociación se hace en forma análoga a los casos vistos previamente.

5.2.4 Extract when Long Method

Por último, vamos a implementar la asociación entre el code critic Long Method y el refactoring Extract Method. Esta asociación cubre dos aspectos importantes de la herramienta. Por un lado, el Extract Method es considerado en los trabajos sobre refactoring como caso testigo: si una herramienta permite hacer refactorings, entonces deberá permitir hacer Extract Method. Por otro lado, este refactoring requiere que el usuario ingrese información adicional, como la porción de código a extraer y la signatura del nuevo método.

Definimos la clase `CRExtractLongMethod` haciendola extender de `CriticRefactoringFactory`. En el método `intialize` definimos los parámetros.

```
CRExtractLongMethod >> initialize
self
        addMethodIntervalParameter: ‘intervalToExtract’
        description: ‘Code to extract’.
self
        addMethodNameParameter: ‘methodName’
        description: ‘New method name’.
```

Al momento de crear el refactoring se utilizan los parámetros creados en el método `initialize`, como se muestra en el siguiente cuadro:

```
CRExtractLongMethod >> refactoringFor: aFlawedMethod
parameters:
        aDictionary
    |refactoring|
    refactoring :=
        RBExtractMethodRefactoring
            extract: (aDictionary at: ‘intervalToExtract’)
value
            from: aFlawedMethod selector
            inClass: aFlawedMethod  affectedClass.
        refactoring
            setOption: #methodName
            toUse: (aDictionary at: ‘methodName) block.
```

Los parámetros son recolectados previamente desde el Critic Refactoring Browser.

5.3 Diagrama de clases de la herramienta

La figura 5.13 muestra el diagrama de clases completo de la herramienta desarrollada para este trabajo.

Figura 5.13. Diagrama de clases de la herramienta (ver última página).

Capítulo 6. Uso de la Herramienta

En este capítulo explicaremos las características de la herramienta e ilustraremos su uso.

La herramienta incluye:

1. Un plugin para el browser de Pharo que analiza automáticamente una clase y permite visualizar los code critics de esa clase.
2. Menús de contexto en el Critic Browser para refactorizar code smells (critics).
3. Un diálogo para ejecutar los refactorings más complejos.

6.1 Ejecución de Code Critics para una clase

La herramienta permite ejecutar el análisis para una clase en particular, mediante un botón disponible en el Class Browser. Al presionar este botón se abre el Critic Browser y se aplican a la clase las reglas configuradas en la última ejecución global (Figura 6.1 y Figura 6.2). Como mencionamos anteriormente, esta característica es importante desde el punto de vista de los criterios para herramientas de refactoring desarrollados por Roberts [Roberts97]. Según este autor, para que las herramientas de refactoring resulten prácticas y sean utilizadas por los programadores, deben estar integradas en el ambiente de desarrollo. Sin esta característica, el análisis de código quedaba disociado de la principal herramienta de desarrollo que es el Class Browser.

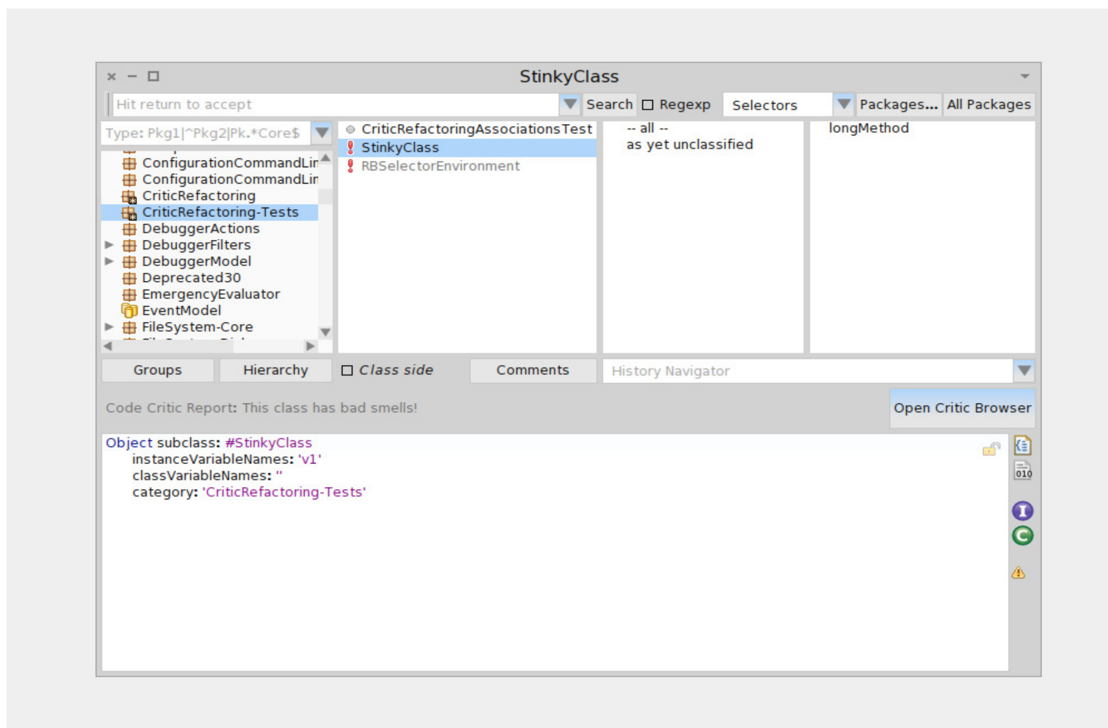


Figura 6.1 Plugin para análisis de code critics de una clase

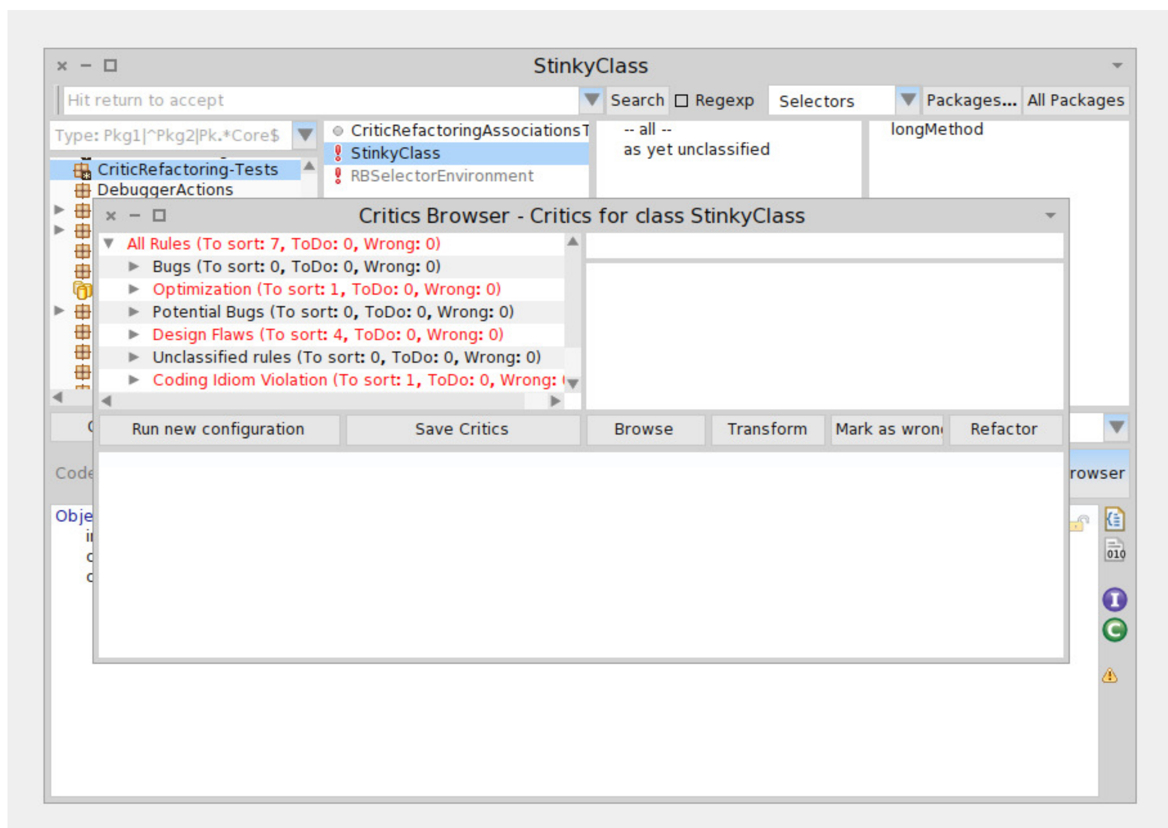


Figura 6.2 Critic Browser adaptado para el análisis de una clase.

6.2 Menú de refactoring en Critic Browser

La herramienta agrega una nueva entrada de refactoring al menú de contexto del Critic Browser. Cuando existen uno o más refactorings asociados al critic, y el elemento seleccionado cumple con las precondiciones de la asociación, el menú despliega los refactorings disponibles (Figura 6.3).

Para el caso de las critics que tienen como resultado un conjunto de variables, se despliega otro nivel del menú que permite seleccionar la variable que se quiere refactorizar.

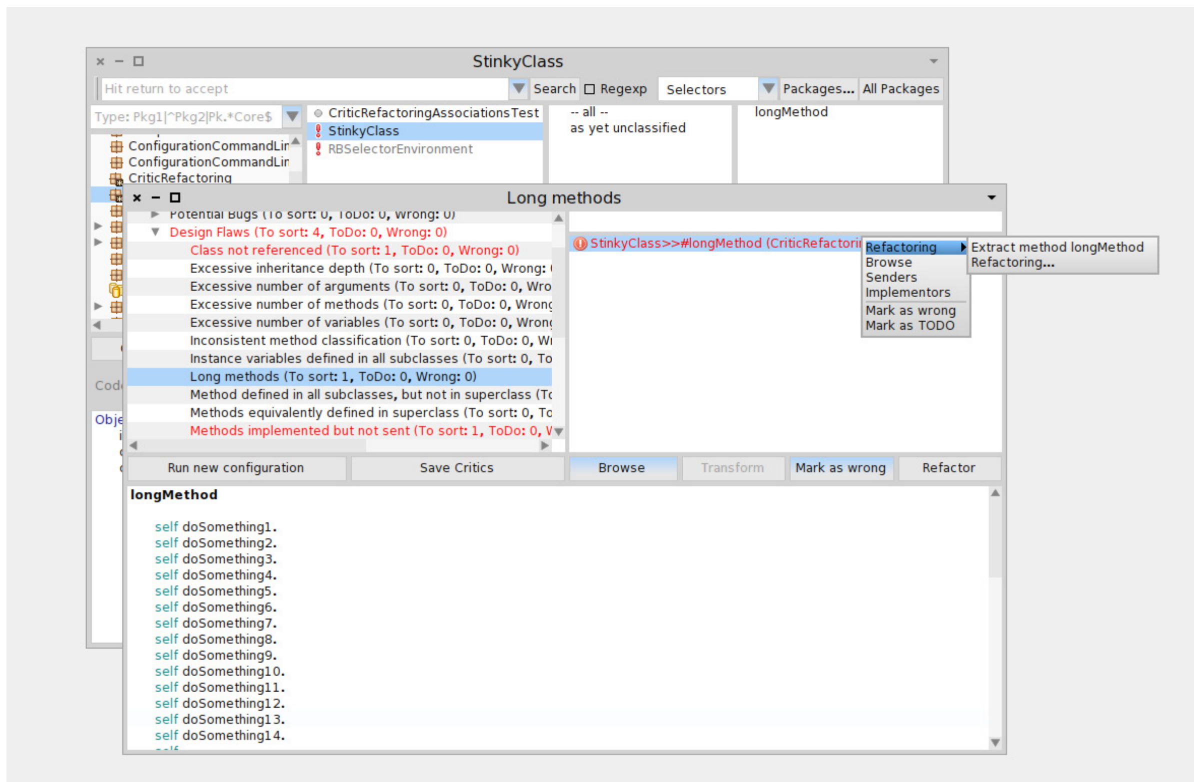


Figura 6.3 Diálogo de refactoring en Critic Browser

6.3 Diálogo de refactoring

Por último, la herramienta agrega un diálogo para refactoring de critics, que llamamos Critic Refactoring Browser. Este diálogo permite ejecutar los refactorings más complejos, que requieren *input* adicional del usuario, como por ejemplo Extract Method y lista en detalle los elementos afectados de una clase, como se explicó en el Capítulo 5. (Figura 6.4).

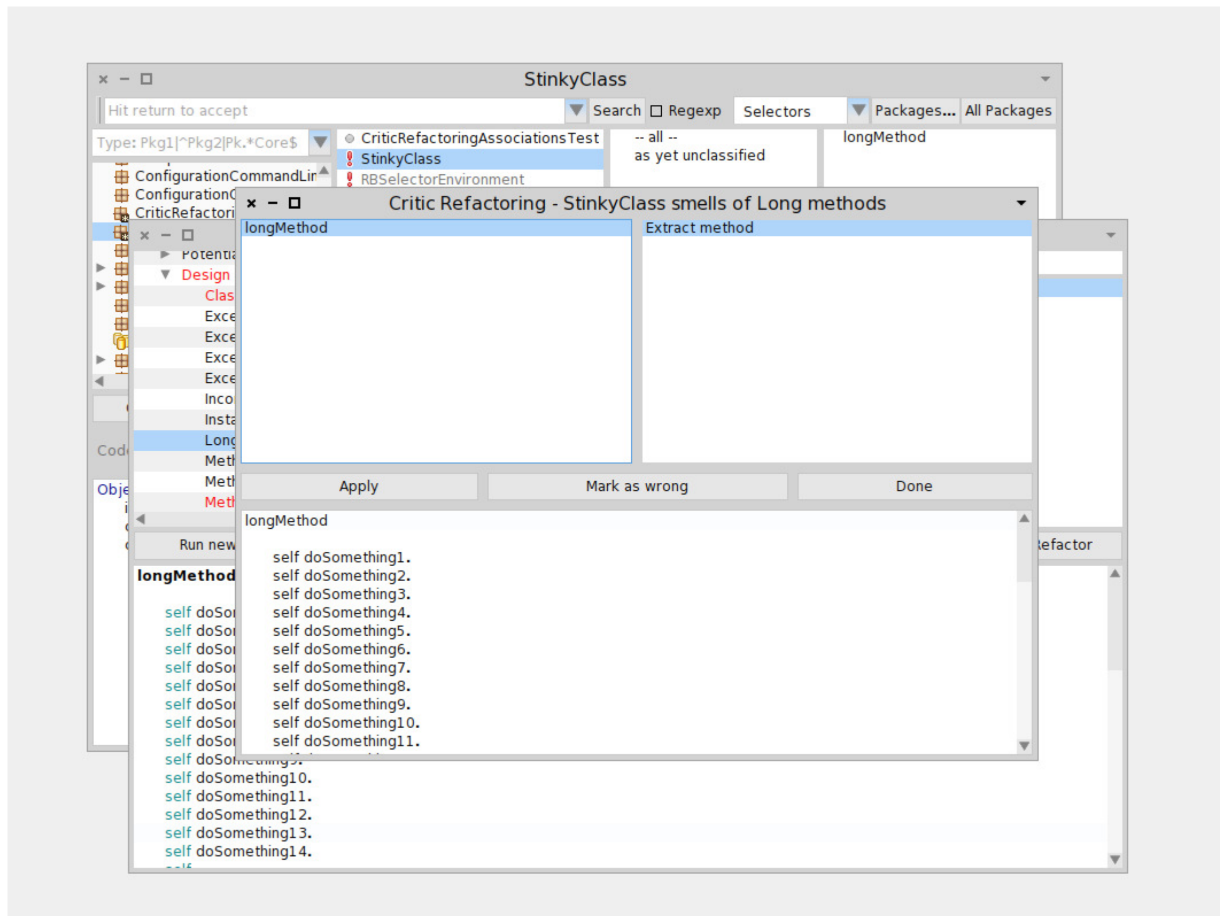


Figura 6.4. Diálogo de refactoring de critics

6.4 Ejemplo de uso

Tomemos como ejemplo una clase `StinkyClass` que tiene un método demasiado largo llamado `longMethod`. Cuando seleccionamos la clase en el Class Browser, el plugin nos muestra el resultado del análisis indicando que existen code smells para la clase (Figura 6.1).

Si hacemos click en el botón “Open Critic Browser” podemos ver el detalle de los critics encontrados para la clase (Figura 6.2). Al seleccionar el critic “Design Flaw -> Long Methods” el

Critic Browser nos muestra el método afectado, `longMethod`, en el cuadro de la derecha . Ahora podemos ejecutar el refactoring Extract Method seleccionándolo en el menú de refactoring o haciendo click en el botón “Refactor” (Figura 6.3).

Como el refactoring Extract Method requiere parámetros adicionales, en lugar de ser ejecutado desde el Critic Browser se abrirá automáticamente el diálogo de refactoring de critics (Critic Refactoring Browser. Figura 6.4).

En el diálogo que muestra la Figura 6.4 seleccionaremos el método en cuestión (`longMethod`) y el refactoring a aplicar (Extract Method). Luego elegiremos en el cuadro inferior, que muestra el código fuente del método, la porción de código a extraer. Hecho esto, hacemos click en el botón “Refactor” y un diálogo nos solicitará el nombre y los argumentos del nuevo método, como se muestra en la Figura 6.5. Esto es exactamente el mismo comportamiento que se obtiene al seleccionar Extract Method desde el Class Browser, es decir que hemos logrado reutilizar el framework de refactoring de manera de preservar en la medida de lo posible las interfaces gráficas del ambiente a las que el usuario está acostumbrado. Esto permite que un usuario de Pharo Smalltalk pueda utilizar la herramienta desarrollada en este trabajo de una manera natural.

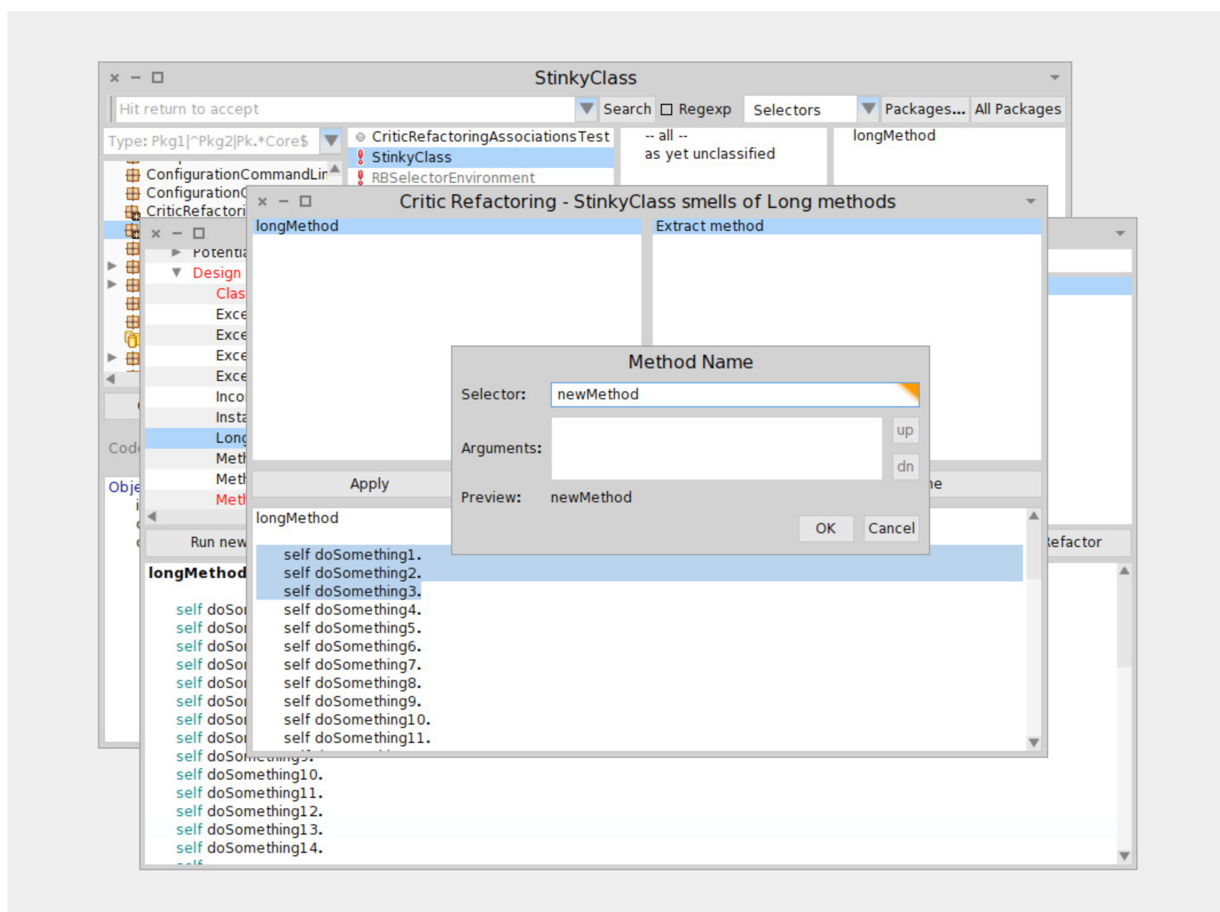


Figura 6.5 Extract Method a través del Critic Refactoring Browser.

7. Conclusiones y Trabajos Futuros

El presente capítulo se enfoca en aportar una pequeña reseña de lo que fue esta tesis, presentando la conclusión final, enumerando las contribuciones y limitaciones, y definiendo trabajos futuros que pueden desprenderse.

7.1 Conclusiones

El refactoring es una técnica de programación que desde su aparición ha ido ganando popularidad hasta convertirse en una práctica universal, aceptada en la industria del software e investigada en los ámbitos académicos. La reestructuración continua del software durante el ciclo de vida es consecuencia de la naturaleza del desarrollo de software —donde la incorporación de cierta medida de cambio es inevitable—. Sobre este punto se ha llegado a un virtual consenso durante los últimos años.

Podemos ver reflejada esta realidad, por un lado, en las metodologías ágiles, donde el refactoring constituye un pilar fundamental, y por otro lado, en la presencia universal de las herramientas de refactoring automático. Desde las utilidades de renombramiento múltiple de los editores texto hasta los complejos refactoring automáticos de los IDEs, todos los entornos de desarrollo cuentan con algún tipo de herramienta de refactoring automático.

Contando con un conjunto de refactoring automatizados, el programador se enfrenta al problema de dónde y cuándo aplicarlos. Para ayudar en esta tarea, Fowler y Beck introdujeron el concepto de code smell, mediante el cual definieron un conjunto de reglas para identificar problemas en el código que podían ser resueltos mediante la aplicación de uno o más refactorings. Distintos estudios han analizado la posibilidad de automatizar la detección de los code smells. De hecho, existen herramientas capaces de detectar varios de éstos, generalmente dentro de un conjunto más amplio de análisis idiomáticos y de bugs.

Ponderando la importancia de ofrecer al programador una automatización completa del ciclo de refactoring, esta tesina se centró en el estudio de la posibilidad de vincular los code smells detectados mediante análisis de código con la ejecución automática de los refactorings. Utilizamos Pharo Smalltalk como plataforma de trabajo, por contar este lenguaje con una larga tradición en el terreno del refactoring, y por ser Pharo una herramienta abierta, que dispone de frameworks de análisis y refactoring y de una comunidad de desarrollo activa.

Con el relevamiento de las investigaciones y herramientas existentes como punto de partida, tomamos el framework de reglas *lint* y analizamos hasta qué punto podía ser utilizado para detectar code smells. Luego comprobamos que es viable agregar nuevos refactoring cuando ninguno de los predefinidos sirve para resolver un code smell. Por último, a modo de prueba de concepto, implementamos una herramienta que permite al programador ejecutar refactorings sugeridos a partir del resultado de la aplicación de las reglas. Incluimos en la herramienta un conjunto mínimo de asociaciones entre code critics y refactorings que contempla los casos que consideramos necesarios para verificar su viabilidad. La implementación está basada en un framework, lo que permite su extensión y desarrollo futuro. .Con esto logramos demostrar que la propuesta de este trabajo es viable, y además fácilmente extensible, siguiendo el método propuesto para cada uno de los casos.

7.2 Contribuciones

7.2.1 Relevamiento de las investigaciones disponibles sobre detección de code smells y asociación de code smells con refactorings.

En el Capítulo 2 introdujimos los conceptos de refactoring y code smell y reseñamos los trabajos de investigación existentes sobre detección automática de code smells y ejecución automática de refactorings a partir del análisis del código. Asimismo, relevamos las herramientas, tanto comerciales como experimentales, que implementan en alguna medida estas características.

7.2.2 Documentación analítica de los frameworks de refactoring y reglas lint de Pharo.

En el Capítulo 3 describimos los frameworks de refactoring y reglas lint de Pharo Smalltalk. La descripción la realizamos a través de ejemplos que muestran cómo agregar nuevos refactoring y “code critics”.

7.2.3 Diseño e implementación de una herramienta de refactoring asistido integrada con las herramientas de desarrollo de Pharo.

Finalmente desarrollamos la herramienta de refactoring asistido como prueba de concepto. En el Capítulo 4 analizamos la viabilidad, los requerimientos y el alcance mínimo de la herramienta, mientras que en el Capítulo 5 explicamos su diseño e implementación. La herramienta se integra con en el IDE de Pharo y constituye un framework en sí misma, ya que permite su fácil extensión.

7.2.4 Mejoras al modelo del framework Small Lint a través del modelado del concepto de "falla de diseño".

Al implementar nuestra herramienta, en el Capítulo 5, comprobamos que el framework Small Lint tenía algunas deficiencias en el diseño en la representación de los resultados de las reglas. Para solucionarlas, modelamos estos resultados explícitamente como "fallas de diseño" que pueden afectar a uno o más elementos de una clase, o a una clase en su totalidad.

7.2.5 Incorporación de extensiones, tanto al framework de Small LInt con nuevos code critics, como al framework de refactoring con nuevos refactorings.

Otro producto secundario de la implementación de la herramienta fue la extensión de los frameworks de refactoring y Small Lint. En la segunda parte del Capítulo 5 agregamos el code critic Refused Bequest y un refactoring que llamamos Replace Instance Variable with Query. Con esto demostramos que se pueden agregar code smells que se resuelven con los refactorings existentes, y que algunos code smells existentes pueden resolverse con nuevos refactorings.

7.3 Limitaciones

- En nuestro análisis, hemos llegado a la conclusión de que el análisis estático de programas presenta limitaciones a la hora de detectar problemas de diseño.
- Dentro del análisis estático, los lenguajes dinámicos presentan limitaciones adicionales. Al desconocerse los tipos de las variables, es menor la cantidad de información que se puede obtener del código.
- No existe una definición formal de code smell. Todas las definiciones incluyen cierto grado de relatividad y subjetividad.
- Incluso en los casos en que es posible detectar un code smell, generalmente se detecta una forma específica éste. Por ejemplo, cuando detectamos Refused Bequest, lo hicimos sólo para el caso llamado "honesto", por las dificultades que suponía abordarlo en forma general.
- En Pharo no existe una representación explícita de las métricas del código, lo que vuelve más compleja la detección de code smells.

7.4 Trabajos Futuros

Durante el desarrollo de la tesina hemos podido identificar distintas direcciones en las cuales podía profundizarse el trabajo:

- Introducir en Pharo modelo explícito de métricas y de base de datos del programa puede facilitar la detección de code smells. Esto probablemente aportaría también a mejorar la velocidad de ejecución de las reglas.
- La herramienta de análisis de Pharo funciona en un browser aparte. Si bien, incorporamos la posibilidad de analizar una clase por vez en lugar de correr el proceso de análisis para todo el sistema o para un paquete, una mayor integración con el System Browser es necesaria para cumplir con el criterio de Roberts de integración con el browser.
- Yendo un poco más lejos, la herramienta podría extenderse para detectar los errores y sugerir los refactorings directamente sobre el código que se está editando, como hacen IntelliJ IDEA y otros IDEs.
- La herramienta podría extenderse en la dirección del análisis de código, ofreciendo formas gráficas de visualizar los code smells, como en los ejemplos que hemos visto de inFussion o iPlasma. En este sentido, Pharo cuenta con un motor de visualización de datos llamado Rossal, que puede servir a ese fin.

Capítulo 8. Bibliografía

[Baxter99] BAXTER Ira D. et al. *Clone detection using abstract syntax trees.* En: *Proceedings of ICSM'98.* Noviembre de 1998*.* Bethesda, Mayland*,* Estados Unidos.

[Beck94] BECK, Kent; JOHNSON, Ralph. *Patterns Generate Architectures*..

[Beck99] BECK, Kent. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999.

[Darwin88] DARWIN, Ian. *Checking C programs with lint*. O'Reilly. 1988.

[Emden02] EMDEN, Eva; MOONEN, Leon. *Java Quality Assurance by Detecting Code Smells*. En: *Proceedings of the 9th Working Conference on Reverse Engineering. IEEE* Computer Society Press, 2002.

[Fontana12] FONTANA, Francesca Arcelli; BRAIONE Pietro; ZANONI Marco. *Automatic detection of bad smells in code: An experimental assessment*. En: *Journal of Object Technology*, vol. 11, no. 2, 2012.

[Foote89] FOOTE, B. y JOHNSON Ralph. *Reflective Facilities in Smalltalk-80*. En: *Anales de OOPSLA'98*. ACM.

[Fowler99] FOWLER, Martin. *Refactoring: Improving the design of existing code.* Addison Wesley, 1999.

[Gamma94] GAMMA Erich et al. *Design patterns: Element of resuable object-oriented software.* Addison-Wesley. 1994. Estados Unidos.

[Griffith11] GRIFFITH, Isaac; WAHL, Scott; IZURIETA, Clemente. *TrueRefactor: An Automated Refactoring Tool to Improve Legacy System and Application Comprehensibility*. Computer Science Department Montana State University Bozeman. 2011, Massachusetts, Estados Unidos.

[inFussion] http://www.intooitus.com/products/infusion. Consultado el 07/10/2014

[IntelliJ14] www.jetbrains.com/idea. Consultado el 07/10/2014.

[iPlasma] http://loose.upt.ro/reengineering/research/iplasma. Consultado el 07/10/2014

[Johnson92] JOHNSON Ralph. E. *Documenting Frameworks using Patterns*. Presentado en OOPSLA'92. 1992

[Klimas02] KLIMAS, Edward J., SKUBLICS Suzanne y THOMAS David A. *Smalltalk with Style*. Prentice Hall, 2002.

[Lanza06] LANZA, Michele; MARINESCU, Radu. *Object-oriented metrics in practice*. Springer-Verlag Berlin Heidelberg, 2006.

[Moghadam11] MOGHADAM, Iman Hemati y Ó CINNÉDE, Mel , *Code-Imp: A Tool for Automated Search-Based Refactoring*. En: *Workshop of Refactoring Tools '11*. 2011, Hawai, Estados Unidos.

[Marinescu04] MARINESCU, Radu. *Detection Strategies: Metrics-Based Rules for Detecting Design Flaws*. En: *Analaes de la vigésima Conferencia Internacional IEEE sobre Mantenimiento de Software (ICSM'04)*. 2004.

[Opdyke92] OPDYKE, William F. *Refactoring object-oriented frameworks*: s. n. 1992. Tesis doctoral. Universidad de Illinois, en Urbana-Champaing, Estados Unidos.

[O'Keefe03] O'KEEFE Mark y Ó CINNÉDE, Mel. *A Stochastic Approach to Automated Design Improvement*. En : *Anales de la segunda conferencia internacional sobre Principios y Práctica de Programación en Java, PPJ '03*, 2003.

[O'Keefe08] O'KEEFE Mark y Ó CINNÉDE, Mel. *Search-Based Refactoring: an empirical study*. En: Journal of software maintenance and evolution: research and practice, 2008.

[Palomba13] PALOMBA, Fabio et al. *Detecting Bad Smells in Source Code Using Change History Information* The College of William and Mary. 2013, Williamsburg, Estados Unidos.

[Riel96] RIEL, Arthur J. *Object-Oriented Design Heuristics*. Addison-Wesley. 1996.

[Roberts97] ROBERTS, Don; BRANTH, John; JOHNSON, Ralph. *A refactoring tool for Smalltalk. Theory and Practice of Object Systems*. En: *Special issue object-oriented software evolution and re-engineering*, Volume 3 Issue 4, 1997, Pages 253 – 263.

[Roberts99] ROBERTS, Don. *Practical analysis for refactoring*. Tesis de doctorado. University of Illinois at Urbana Campaign. 1999.

[Slinger05] SLINGER, *Code Smell Detection in Eclipse*. Reporte de Tesis. Delft University of Technolog. 2005.

[Wake03] WAKE, William C. *Refactoring Workbook.* Addison-Wesley. 2003.