UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Refactorings portables para soportar la evolución automática de código que utiliza componentes externos.

Autores: Juan Cruz Gardey.

Director: Alejandra Garrido.

Carrera: Licenciatura en Sistemas.

Resumen

La reutilización de código agiliza considerablemente el desarrollo, pero hace que los sistemas dependan de los componentes que reúsan (librerías, frameworks, servicios, etc.). Estos componentes proveen APIs (Application Programming Interface) que los sistemas utilizan para interactuar con ellos. Estas APIs sufren cambios con mucha frecuencia (los métodos cambian de nombre, se vuelven obsoletos, aparecen nuevos, etc.), lo cual impacta en los programas que las usan.

Considerando la gran utilidad de los refactorings y la posibilidad de ejecutarlos automáticamente, una forma de actualizar un componente de software usado dentro de un sistema es aplicar cada modificación sobre el componente por medio de un refactoring, y luego exportar estos refactorings para que puedan ser reproducidos automáticamente en los sistemas que dependan de éste.

El objetivo del trabajo es desarrollar una herramienta usando Pharo Smalltalk, para exportar los cambios sobre componentes en forma de refactorings, y luego poder reproducirlos en el código que hace uso de estos componentes, con el fin de aplicar automáticamente actualizaciones de software a través de los refactorings.

Palabras Claves

- *Refactoring.*
- *Actualización de aplicaciones.*
- *Pharo Smalltalk.*
- *Componentes de software reusables.*

Conclusiones

Tomando como base el framework de Refactorings de Pharo Smalltalk, se implementó una herramienta que permite grabar un subconjunto de refactorings hechos en un ambiente de trabajo y re-crearlos en otro.

A partir del análisis del concepto de dependencias entre refactorings, se desarrolló un mecanismo para identificarlas de una forma práctica, con el objetivo de asegurar que la re-ejecución de los refactorings sea exitosa.

Trabajos Realizados

- *Descripción de la arquitectura del framework de Refactoring de Pharo.*
- *Análisis de las dependencias entre refactorings.*
- *Incorporación de extensiones a los refactorings de Pharo.*
- *Diseño e implementación de una herramienta para exportar modificaciones de código.*

Trabajos Futuros

- *Introducir más refactorings a la herramienta desarrollada.*
- *Desarrollar pruebas de usuario para comprobar la usabilidad de la herramienta.*
- *Mayor integración de la herramienta con el ambiente de trabajo de Pharo.*
- *Realizar la exportación de refactorings a distintos formatos de archivos.*
- *Visualizar los detalles de cada uno de los refactorings exportados.*

Fecha de la presentación: **Noviembre de 2017**

Agradecimientos

En primer lugar, quiero agradecerle a la Dra. Alejandra Garrido por su tiempo dedicado y por los aportes hechos durante el desarrollo del trabajo.

Agradezco a mis padres y a mi hermano, por el apoyo incondicional de siempre, y por ayudarme a llegar esta instancia.

Por último, agradecerle a mi novia por estar siempre.

¡A todos ellos, Muchas Gracias!

Juan Cruz Gardey

Índice

Índice de Figuras

1 Introducción

Los sistemas de software evolucionan constantemente. Esta es una de las leyes propuestas por Manny Lehman en los ’70, que llamó “Ley del Cambio Continuo” [Lehman78]. Además de los nuevos requerimientos que surgen en respuesta a nuevas tecnologías o contextos, los diseños evolucionan hacia componentes más reusables y mantenibles. A pesar de la gran cantidad de estudios sobre el mantenimiento de software y las herramientas desarrolladas, el costo de mantenimiento continúa siendo muy elevado. Además, la incapacidad de cambiar el software de manera rápida y segura implica que se pierden oportunidades de negocio [Kotter12].

El refactoring surge a principios de los ’90 como una técnica que modela algunas transformaciones que ocurren en la evolución de un framework orientado a objetos [Opdyke92]. Estas transformaciones preservan el comportamiento del componente al que se aplican, pero asumen un “mundo cerrado”, es decir, el comportamiento se preserva siempre y cuando no haya otros sistemas que hagan uso de ese componente [Henkel05]. El problema que surge es que este contexto de “mundo cerrado” en la actualidad es cada vez más irreal, a partir de la sostenida tendencia hacia un mayor reúso de software, no sólo a través de frameworks sino también de librerías y servicios. En este trabajo se estudia este problema y se propone una solución para soportar la evolución de componentes reusables sin que esto implique un costo elevado de mantenimiento de las aplicaciones cliente.

1.1 Motivación

Generalmente, los ingenieros de software no desarrollan un sistema desde cero, sino que reutilizan componentes de software ya realizados. El término componentes en este trabajo hace referencia a librerías, frameworks, servicios, etc. que son implementados por terceros. Cabe destacar que la utilización de servicios implementados por terceros y disponibles en la nube es cada vez mayor, así como el uso de librerías de código abierto implementadas por una comunidad.

La reutilización agiliza considerablemente el desarrollo, pero hace que los sistemas dependan de los componentes que reúsan. Estos componentes proveen APIs (*Application Programming Interface*) que los sistemas utilizan para interactuar con ellos. Estas APIs sufren cambios con mucha frecuencia (los métodos cambian de nombre, se vuelven obsoletos, aparecen nuevos, etc.), lo cual impacta en los programas que las usan. Frente a una modificación de un componente de software surgen dos posibles soluciones:

- ✓ Una posible solución al problema descripto antes podría ser que los desarrolladores de los componentes, luego de hacer las modificaciones correspondientes, mantengan tanto

la versión "nueva" del componente como también la versión anterior. Esto permitiría que los usuarios puedan seguir usándolo sin introducir ningún cambio en su código, con la desventaja de no tener las características, funcionalidades, correcciones de bugs, etc. que incorpora la actualización. Esta opción tiene un altísimo costo de mantenimiento para los desarrolladores del componente, que se ven obligados a mantener distintas versiones de un mismo software cada vez que sale una nueva versión, además de que se acumula cada vez más código obsoleto, etc.

- ✓ La otra alternativa, que de hecho es la que comúnmente se utiliza, es que se cambie el código de los sistemas que hacen uso del componente para utilizar la versión más reciente. El problema es que los programadores suelen hacer estos cambios en forma manual, lo cual es muy tedioso y propenso a errores. Esto hace que se transfiera el alto costo de mantenimiento a los programadores del sistema. Lo ideal sería que las modificaciones puedan aplicarse automáticamente para facilitar el proceso de mantenimiento de los sistemas y también de los componentes.

Dentro de la orientación a objetos, los *refactorings* son una herramienta ampliamente utilizada durante el desarrollo y mantenimiento de un programa, ya que realizan transformaciones sobre el código (preservando su comportamiento) con el fin de incrementar su mantenibilidad, reusabilidad y legibilidad. Actualmente, la mayoría de los IDE (*Entorno de Desarrollo Integrado*), proveen herramientas que permiten aplicarlos automáticamente. Esto significa que el usuario selecciona el refactoring a ejecutar e ingresa los datos correspondientes, la herramienta chequea si la transformación subyacente al refactoring es legal y en caso de serlo, la ejecuta.

Considerando la gran utilidad de los refactorings y la posibilidad de ejecutarlos automáticamente, una forma de actualizar automáticamente un componente de software usado dentro de un sistema sería aplicar cada modificación sobre el componente por medio de un refactoring (siempre y cuando sean modificaciones que no alteren el comportamiento), y luego exportar estos refactorings para que puedan ser reproducidos automáticamente en los sistemas que dependan de éste. De esta manera, cada actualización de un programa de una versión a otra, puede verse como una secuencia de refactorings.

Tal como sostiene Roberts en su tesis doctoral [Roberts99], un único refactoring suele introducir cambios mínimos en el código, y para lograr cambios más grandes, generalmente se ejecutan una serie de refactorings en cadena, por lo tanto, es muy probable que cada actualización de software implique varios refactorings, no solamente uno. Esto plantea la necesidad de ejecutar una secuencia de refactorings como si fuera uno único, en el sentido de que, o se ejecutan todos los refactorings de la secuencia o no se ejecuta ninguno, dado que, si la ejecución de la cadena es

parcial, la actualización del programa quedará inconsistente y es muy probable que éste no funcione.

Al momento de pensar en la re-ejecución de los refactorings, surgen algunas cuestiones:

- ✓ *¿Qué sucede si la ejecución de alguno de los Refactorings de una secuencia falla?*: Cada refactoring posee determinadas precondiciones que deben cumplirse para que sea válido, es decir, para que pueda ser ejecutado exitosamente. Por ejemplo, un refactoring que incorpora una nueva clase requiere que no exista previamente una clase con el mismo nombre.

  Puede ocurrir que al momento de ejecutar algún refactoring de una cadena, sus precondiciones no se cumplan. Esto implicaría deshacer los refactorings de la secuencia anteriores, o de algún modo hacer que las precondiciones del refactoring se cumplan para poder continuar la ejecución de la secuencia.

- ✓ *Dependencias entre Refactorings*: Esta cuestión está relacionada con el punto anterior debido a que tiene que ver con la validez de las precondiciones de los refactorings. Cuando se ejecuta una cadena de refactorings, es muy posible que la ejecución de uno habilite las precondiciones de otro, lo cual produce una dependencia entre estos. Esta dependencia impone ciertas restricciones en la ejecución de ambos refactorings. Una restricción posible es que la ejecución se tenga que realizar en un determinado orden. Resolver estas dependencias, implica reacomodar los refactorings de alguna forma para que todos puedan ser ejecutados correctamente.

La exportación y re-ejecución de refactoring ya ha sido demostrada en CatchUp! [HD05], pero esta herramienta no proporciona una solución a las cuestiones planteadas anteriormente; durante la re-ejecución de los refactorings si alguno de ellos no se puede realizar, la re-ejecución se cancela quedando la aplicación en un estado inconsistente debido a que la secuencia se ejecutó parcialmente. Por otro lado, al poder elegir cuáles refactorings re-ejecutar, el hecho de no contemplar las posibles dependencias entre estos puede hacer que la re-ejecución genere errores en la aplicación. Posteriormente Danny Dig en su tesis doctoral *Automated Upgrading of Component-Based Applications* [Dig07] enriqueció el grabado y la re-ejecución de refactorings desarrollando una serie de herramientas para solucionar las limitaciones de CatchUp!

Tanto CatchUp! como las herramientas de Dig están implementadas como un plugin de Eclipse, el IDE para desarrollar en Java. La mayoría de los refactorings soportados son específicos del lenguaje Java, el ejemplo más común es el refactoring *Change Method Signature* que permite modificar el nombre de un método, su tipo de retorno, su visibilidad, entre otras cosas. Este refactoring no es aplicable en lenguajes donde no existen los tipos de datos, como es el caso de Smalltalk donde todo es un objeto.

Teniendo en cuenta los desarrollos existentes, la motivación principal de este trabajo es implementar la captura y re-ejecución de refactorings sin depender de un lenguaje en particular, proporcionando soporte para los refactorings catalogados por Fowler en [Fowler99] que pueden ser aplicados en cualquier programa orientado a objetos.

1.2 Objetivos

El objetivo del trabajo es desarrollar una solución para exportar los cambios sobre componentes en forma de refactorings, y luego poder reproducirlos en el código que hace uso de este (teniendo las cuestiones planteadas en la motivación), con el fin de aplicar automáticamente actualizaciones de software a través de ellos. Estas actualizaciones automáticas proporcionan la ventaja de disminuir los costos de mantenimiento tanto para los desarrolladores de componentes como también para los programadores usuarios de estos.

Para el desarrollo de la solución planteada se utilizará el ambiente de desarrollo Pharo Smalltalk. La elección se debe a las siguientes razones:

- ✓ En Smalltalk está implementado el Refactoring Browser, la primera herramienta de refactoring. Refactoring Browser contiene implementados los refactorings propuestos por Fowler y también su diseño permite cambiar los refactorings existentes o incorporar nuevos sencillamente. En los demás Refactorings Engines esto suele ser más difícil.
- ✓ Pharo Smalltalk es un sistema de código abierto que puede ser fácilmente modificado. Además, mantiene la arquitectura original del Refactoring Browser. Esta arquitectura está documentada detalladamente en la tesis de Don Roberts [Roberts99].
- ✓ Smalltalk es un lenguaje con amplias facilidades de reflexión desde su especificación en el año 80, lo cual lo hace altamente extensible [Foote89] y permite una implementación de la propuesta mucho más transparente y limpia que con otras opciones. Si bien Smalltalk no es un lenguaje de los más usados en la actualidad, la capacidad de reflexión se ha incorporado incrementalmente en lenguajes muy utilizados como Java, y los nuevos lenguajes como Python o Ruby han surgido con facilidades de reflexión incorporadas. De esta manera, la implementación de esta propuesta utilizando reflexión será fácilmente aplicable a otros lenguajes con la misma capacidad.

1.3 Contribuciones

- ✓ Descripción detallada de la arquitectura del Refactoring Browser. Dado que este framework se utiliza como base para el desarrollo, es necesario entender detalladamente cada uno de sus componentes y su funcionamiento.

- ✓ Diseño e implementación de una herramienta para guardar y re-ejecutar un subconjunto de los refactorings implementados en Pharo Smalltalk. Para determinar cuáles refactorings soportar, se analizaron algunas investigaciones que revelan los refactorings más utilizados.
- ✓ Análisis del concepto de dependencias entre refactorings introducido inicialmente por Don Roberts [Roberts99]. Teniendo en cuenta que cada refactoring es diferente, es fundamental determinar bajo qué condiciones un refactoring específico depende de otro.
- ✓ Incorporación de Extensiones a los refactorings soportados por la herramienta. Para cada refactoring se agregó la posibilidad de poder determinar, dado otro refactoring, si depende de éste o no. Además, a cada uno de los refactorings se le incorporó la opción de exportarlo.
- ✓ Se añadió soporte para tratar de manera uniforme todos los refactorings, incluido el que permite agregar un nuevo método (*Add Method*). Si bien *Add Method* fue definido como un refactoring desde un comienzo en la tesis de William Opdyke [Opdyke92], estando implementado en Pharo, no estaba siendo utilizado como tal. Este tratamiento uniforme es fundamental para lograr extensibilidad en la herramienta desarrollada, de manera que cada vez que un desarrollador agregue un método a una clase, se instancie un refactoring de este tipo para poder exportar este cambio realizado.

1.4 Organización de la Tesina

- ✓ ***Capítulo 2, Trabajos Relacionados:*** Se describen los conceptos básicos necesarios para entender este trabajo y se describen brevemente aquellos trabajos que motivaron la realización de esta Tesina.
- ✓ ***Capítulo 3, Arquitectura de Base:*** Se detalla la arquitectura del Refactoring Browser, enumerando sus componentes principales, sus puntos de extensión y sus limitaciones. Se muestra la secuencia desde que un usuario selecciona un refactoring hasta que el mismo se instancia y finalmente se ejecuta.
- ✓ ***Capítulo 4, Grabado y re-ejecución de Refactorings:*** En este capítulo se explica la solución implementada para poder capturar los refactorings y luego re-ejecutarlos en otro ambiente de Pharo. Se destacan las consideraciones de diseño e implementación importantes y se listan las principales limitaciones de la solución elegida.
- ✓ ***Capítulo 5, Validación de Refactorings:*** Se analizan los beneficios y las desventajas de la forma en la que el Refactoring Browser chequea la validez de los refactorings. Luego de describe cómo la herramienta desarrollada ejecuta estas validaciones y muestra los resultados al usuario.

- ✓ ***Capítulo 6, Dependencias entre Refactorings:*** En este apartado inicialmente se presenta el concepto de dependencias entre refactorings con algunos ejemplos. Posteriormente, se describe el mecanismo utilizado en la herramienta desarrollada para detectar estas dependencias de forma práctica.
- ✓ ***Capítulo 7, Uso de la herramienta:*** Tomando un pequeño framework como ejemplo, se muestra el uso típico de la herramienta capturando una serie de refactorings en un ambiente de trabajo y luego reproduciéndolos en otro.
- ✓ ***Capítulo 8, Conclusiones y Trabajos Futuros:*** Este capítulo resume las contribuciones de esta tesina y presenta futuras extensiones.

2 Trabajos Relacionados

2.1 Conceptos Básicos

2.1.1 Refactoring

Refactoring es el proceso de modificar un programa sin alterar su comportamiento externo con el objetivo de mejorar su código [Opdyke92]. Mejorar el código significa hacer que sea más fácil de mantener y de reutilizar. Es una técnica que se aplica después de desarrollar el código, por lo tanto, su objetivo es proveer un mecanismo sistemático de cambiar este código minimizando la introducción de errores o bugs.

El concepto de refactoring fue introducido por William Opdyke [Opdyke92]. Posteriormente fue ampliado por Martin Fowler [Fowler99] que propuso un catálogo de transformaciones aplicables a un programa orientado a objetos, donde cada una de estas también recibe el nombre de refactoring. Un refactoring es una transformación simple como por ejemplo renombrar una variable o mover un método de una clase a otra, sin embargo, una secuencia de refactorings encadenados pueden lograr grandes cambios de diseño.

El principio fundamental de la técnica de refactoring es preservar la funcionalidad de un programa. Esto significa que, si el programa es ejecutado antes y después de un refactoring con los mismos valores de entrada, los resultados de salida deben ser los mismos. Para lograr este efecto, un refactoring no puede aplicarse en cualquier momento, sino que se deben dar determinadas condiciones. Cada refactoring posee precondiciones que deben cumplirse para que se pueda realizar. Las precondiciones son propiedades que el programa debe cumplir, un ejemplo de precondición podría ser que no exista una clase con un nombre determinado. Si el programa no reúne las precondiciones del refactoring, la aplicación del mismo no asegura que el programa mantenga su comportamiento.

Más allá de no alterar el funcionamiento del programa, los refactorings se realizan para obtener una mejora en el diseño del código tal como se dijo anteriormente. Aplicar un refactoring arbitrariamente aun conservando el comportamiento del programa en lugar de enriquecer el diseño lo deteriora. Fowler en su libro [Fowler99], además del listado de refactorings, también describe cuándo usar cada uno de estos mediante la definición de *Code Smells*. El término *Code Smell* se usa para indicar posibles deficiencias de diseño que presenta un fragmento de código. Fowler notó que determinados problemas de diseño se repetían en diferentes proyectos y es por eso que elaboró un listado asignándole un nombre a cada uno. El más común es *Duplicate Code*

que indica que una misma porción de código está repetida en distintos lugares. Cada *Code Smell* sugiere la aplicación de un refactoring específico para mitigar la deficiencia de diseño.

Sabiendo los beneficios de los refactorings y en qué momento deben ser aplicados ya se podría comenzar a utilizarlos. Sin embargo, los programas orientados a objetos suelen ser difíciles de modificar debido a que cambiar un elemento (clase, método, etc.) implica modificar el código en distintas partes (el renombramiento de un método requiere alterar todos los métodos en los cuales se referencia este método). Identificar todos los fragmentos de código que tienen que transformarse como consecuencia de un refactoring y aplicar estas modificaciones manualmente, es un trabajo tedioso y propenso a errores. Una de las razones por la cual no se suelen realizar refactorings es justamente por el riesgo que se corre de introducir errores en el código.

Como solución a la problemática anterior, Opdyke propuso automatizar la ejecución de los refactorings con el objetivo de asegurar que se preserven las funciones del programa. Finalmente, Donald Roberts y John Brant [Roberts99] fueron quienes desarrollaron el *Refactoring Browser*, la primera herramienta que permitió aplicar automáticamente un conjunto de refactorings que fue hecha para *Smalltalk*. Esta herramienta se ha ido mejorando con el paso del tiempo (incluso actualmente sigue siendo muy usada en las distintas implementaciones de Smalltalk) y lo más importante es que su arquitectura y diseño sirvió como base para la creación de Refactoring Engines de otros ambientes de desarrollo.

El Refactoring Engine de Eclipse se parece mucho al Refactoring Browser en cuanto a la forma en la que se validan las precondiciones de los refactorings y a la manera en la que se aplican las transformaciones de código. Sin embargo, esta herramienta no proporciona la variedad de Refactorings provista por Refactoring Browser y además muchos de los Refactorings que provee están relacionados con cuestiones específicas del lenguaje, por lo que no son aplicables a cualquier programa orientado a objetos. Otra gran ventaja del Refactoring Browser sobre el Refactoring Engine de Eclipse está en la facilidad de extensión y modificación; Refactoring Browser permite modificar cualquier Refactoring existente e incluso incorporar nuevos, este es el motivo principal por el cual se decidió usar Smalltalk para desarrollar este trabajo.

2.2 Trabajos de Investigación

2.2.1 CatchUp!

Johannes Henkel y Amer Diwan fueron quienes propusieron por primera vez capturar y re-ejecutar refactorings con el objetivo de automatizar la evolución de las APIs de librerías y frameworks. Los autores de esta herramienta sostienen que los desarrolladores de componentes

suelen no restructurar el código (o lo hacen de forma muy limitada) para no causar problemas en las aplicaciones clientes de estos. Para demostrar esto realizaron un estudio de los métodos marcados como obsoletos (*deprecated*) y descubrieron que estos métodos muy pocas veces son eliminados debido a cuestiones de compatibilidad y al alto costo que implica actualizar el código de las aplicaciones.

Con la captura y re-ejecución de los refactorings, Henkel y Diwan pretenden animar a los desarrolladores de componentes a que reestructuren y modifiquen su código para reducir su complejidad y hacerlo más fácil de modificar. Por ejemplo, que se eliminen los métodos y clases marcados como obsoletos para que no se acumule este tipo de código.

CatchUp! está implementado como un plugin para Eclipse. Básicamente el desarrollador de la librería aplica sobre la misma los refactorings y los cambios manuales que considera necesarios. Posteriormente CatchUp! genera un archivo JAR con la nueva versión de la librería y un archivo XML con la información de cada refactoring realizado. En la aplicación cliente, la herramienta reemplaza el JAR de la nueva versión por el de la versión anterior y luego ejecuta los refactorings contenidos en el archivo XML previamente generado. Los refactorings actualizan la API de la librería dentro de la aplicación mientras que el reemplazo del JAR asegura que las modificaciones que se hicieron en la librería que no son cambios en la API (por ejemplo, modificar el cuerpo de un método) también sean transportadas a la aplicación cliente. Se utiliza el formato XML para exportar los refactorings para que la información asociada a estos sea de fácil lectura, lo que le da a los desarrolladores la posibilidad de analizar cada refactoring ejecutado y reproducirlos manualmente cuando la re-ejecución automática no es posible por alguna razón.

CatchUp! fue desarrollado como un prototipo para mostrar la idea de capturar y re-ejecutar refactorings. El prototipo provee soporte completo (exportación y re-ejecución) para tres refactorings: *Rename Type* (se usa para renombrar una clase o una interface), *Moving Java Elements* (mover una clase o interface de un paquete a otro) y *Change Method Signature*. Hay otros refactorings para los cuales únicamente se implementó la exportación o captura, de manera que no pueden ser reproducidos automáticamente. La idea de los autores de CatchUp! era convertir el prototipo en una herramienta libre que pueda ser utilizada en aplicaciones reales por cualquier desarrollador, sin embargo, esta idea no prosperó.

2.2.2 Actualización Automática de aplicaciones

Danny Dig [Dig07] desarrolló un conjunto de herramientas para automatizar la actualización de aplicaciones que reúsan otros componentes. En principio para determinar cuáles son los cambios que causan problemas en las aplicaciones observó cuatro frameworks y librerías de código abierto

y descubrió que de aquellas modificaciones que hacen que un sistema deje de funcionar, más del 80% son originadas por refactorings.

A pesar de encontrar que el uso de refactorings genera inconvenientes al momento de actualizar componentes, Dig reconoce el refactoring como una técnica muy poderosa para mejorar el código, y por eso propone utilizarla como un medio para expresar la evolución de un componente, debido a que los refactorings contienen la semántica del cambio realizado, no solamente la sintaxis. De esta manera, la diferencia entre una versión y otra de un mismo componente podría representarse como un conjunto de refactorings.

Partiendo de la base que existen Refactoring Engines que dan la posibilidad de ejecutar refactorings automáticamente, su solución, al igual que CatchUp!, consiste en determinar los refactorings que se aplican sobre un componente y luego incorporarlos automáticamente en las aplicaciones que hacen uso de éste asegurando que no se altera su funcionamiento. La principal diferencia con CatchUp! es que CatchUp! es un prototipo que solamente muestra la idea de capturar y re-ejecutar refactorings. Cuando Dig quiso realizar una implementación real de esta idea, se encontró con algunas limitaciones que superó mediante la implementación de un conjunto de herramientas. Estas herramientas son *RefactoringCrawler*, *MolhadoRef* y *ReBa*, las mismas fueron desarrolladas como una extensión del Refactoring Engine de Eclipse.

RefactoringCrawler

Para sacarle el mayor provecho al refactoring como una operación que describe la evolución de un componente, es necesario que se realicen todos los cambios en el código por medio de refactorings, o al menos aquellos para los cuales existe un refactoring. Si un refactoring se aplica manualmente sobre un componente, no podrá ser exportado y luego incorporado a la aplicación porque no queda registro de su aplicación. Esto es lo que ocurre la mayoría de las veces dado que los desarrolladores suelen editar el código directamente en lugar de utilizar el Refactoring Engine.

RefactoringCrawler es una herramienta que se encarga de detectar automáticamente refactorings realizados en un determinado programa. El algoritmo que usa la herramienta compara dos versiones de un mismo componente para calcular los refactorings que se aplicaron. Primero realiza un rápido análisis sintáctico para encontrar fragmentos similares en los diferentes archivos de código y luego un análisis semántico preciso para refinar los resultados obtenidos en el análisis sintáctico. En total detecta siete refactorings diferentes que son aquellos que más se usan en la práctica.

MolhadoRef

Para poder reproducir en una aplicación refactorings que fueron hechos anteriormente en un componente, es necesario que esta aplicación reúna sus precondiciones, sin embargo, puede

ocurrir que estas precondiciones no se cumplan. Los Refactorings Engines, cuando no se satisfacen las precondiciones de un refactoring, generan un error y cancelan su ejecución. En el caso de la re-ejecución, los cambios subyacentes al refactoring tienen que realizarse de alguna manera dado que, si no se aplican, la actualización de la aplicación quedará inconsistente y es muy probable que el sistema deje de funcionar.

Además de los refactorings, el código de los componentes y de las aplicaciones evolucionan mediante ediciones manuales de los desarrolladores. Este tipo de cambios suele ocasionar conflictos con los refactorings.

Para resolver las cuestiones mencionadas antes, Dig desarrolló MolhadoRef, una herramienta que realiza un *merging* entre los cambios hechos en un componente y aquellos realizados en la aplicación. MolhadoRef mantiene un registro de las entidades de un programa y las operaciones que las modifican, es decir que contempla la presencia de refactorings en los distintos cambios que se presentan en el código. Esto le permite hacer un merging semántico de los refactorings que, si es exitoso, asegura que el programa resultante compila correctamente y no contiene errores de compilación. La gran diferencia con las herramientas de merging tradicionales es que estas lo hacen sintácticamente, comparando archivos de texto plano línea por línea.

El merging textual tiene la característica de generar un conflicto cada vez que encuentra diferencias en una misma línea de código, aún si las diferencias se deben únicamente al agregado de nuevo código. Sin embargo, su principal desventaja es que no es confiable en el sentido de que el programa resultante puede llegar a tener errores de compilación y ejecución.

MolhadoRef hace el merging de los refactorings semánticamente y el de las ediciones manuales sintácticamente tal como lo hacen las herramientas basadas en texto. Si un refactoring no puede re-ejecutarse debido al incumplimiento de sus precondiciones, se considera una edición de código y se combina textualmente. Si se hacen todas las modificaciones manualmente en lugar de utilizar refactorings, MolhadoRef se comporta igual que las herramientas de merging tradicionales. Cuando más se usen los refactorings, mayor provecho se obtiene de la herramienta.

Resultados de un caso de estudio muestran que MolhadoRef resuelve automáticamente más conflictos de merging que las herramientas basadas en texto. A su vez, luego del merging originó menos errores de compilación y ejecución que estas herramientas.

ReBa

La última limitación con la que se encontró Dig al desarrollar la exportación y re-ejecución de refactorings es que a veces el código fuente de la aplicación que reúsa el componente no puede ser modificado porque no está disponible. Teniendo en cuenta la imposibilidad de acceder al

código, una alternativa para que las aplicaciones interactúen con la última versión del componente es utilizando un adaptador de compatibilidad como intermediario entre ambos.

ReBa es una herramienta que genera automáticamente un adaptador de compatibilidad entre una aplicación y un componente. A pesar de permitir usar la versión más reciente de un componente, Dig en un caso de estudio concluyó que ejecutar una aplicación con un adaptador de este tipo degrada considerablemente su performance. De manera que esta alternativa es una solución temporal, a largo plazo será necesario cambiar el código de la aplicación para actualizarla sin afectar a su rendimiento.

Por último, cabe destacar que las herramientas descriptas en esta sección son muy poderosas en cuanto al poder de automatización alcanzado, sobre todo la que permite realizar un merge correcto entre refactorings de distintas versiones de un componente. Sin embargo, estas herramientas no proveen flexibilidad ni poder de decisión al usuario para que pueda elegir refactorings que no desea aplicar. Esta tesina se focaliza en darle al usuario esa flexibilidad, además del soporte a la automatización.

3 Arquitectura de Base

Tal como se adelantó en la introducción, para el desarrollo de este trabajo se utiliza como base el Refactoring Browser de Smalltalk. Esta herramienta no sólo da la posibilidad de ejecutar refactorings automáticamente, sino que también permite modificar los refactorings existentes o incorporar nuevos debido a que está diseñada como un framework: posee claros puntos de extensión en los cuales los usuarios pueden definir su propio comportamiento.

En este capítulo, primero se explicará el uso básico de la herramienta y luego se analizarán los aspectos técnicos del framework. Tomando un refactoring como ejemplo, se describirán los principales componentes y los puntos de extensión mencionados antes. Es posible que el Refactoring Browser presente algunas variantes en las distintas implementaciones de Smalltalk. Por lo tanto, vale aclarar que en este apartado se detallará la versión de Pharo.

3.1 Uso de la herramienta

La herramienta permite ejecutar los refactorings a través de los menús de contexto disponibles en el Browser para los distintos elementos de un programa: paquetes, clases, variables, métodos y código fuente. Una vez que el usuario elige el refactoring que quiere aplicar, el Refactoring Browser toma la información del contexto para determinar sobre qué elemento se aplica. Es probable que los refactorings requieran información adicional para poder ejecutarse, como ejemplo, el refactoring que cambia el nombre de una clase necesita el nuevo nombre de la misma. Esta información es solicitada al usuario durante la aplicación del refactoring.

Una vez que se cuenta con todos los datos necesarios, la herramienta chequea que el refactoring sea válido. La validez de un refactoring está dada por sus precondiciones: si se cumplen sus precondiciones significa que puede ejecutarse correctamente, en caso contrario se muestra un mensaje de error informando que el refactoring no puede ser aplicado ya que no garantiza que se preserve el comportamiento del programa.

Un refactoring suele implicar modificaciones en distintas partes del código, por lo que se compone de un conjunto de cambios. Estos cambios son creados luego de validar las precondiciones del refactoring y son mostrados en pantalla antes de ser aplicados sobre el código para que el usuario pueda decidir cuáles de ellos ejecutar. La Figura 3.1 muestra una aplicación del refactoring *Rename Method*. En este caso se muestran tres cambios: el primero agrega un nuevo método con el nombre ingresado cuyo código es el mismo que el del método renombrado, el segundo cambio modifica el cuerpo de un método que invoca al método renombrado reemplazando el viejo nombre por el nuevo, y el último elimina el método renombrado. Cabe destacar que el Refactoring

Browser modifica un elemento de programa eliminándolo y creándolo nuevamente con los cambios correspondientes.

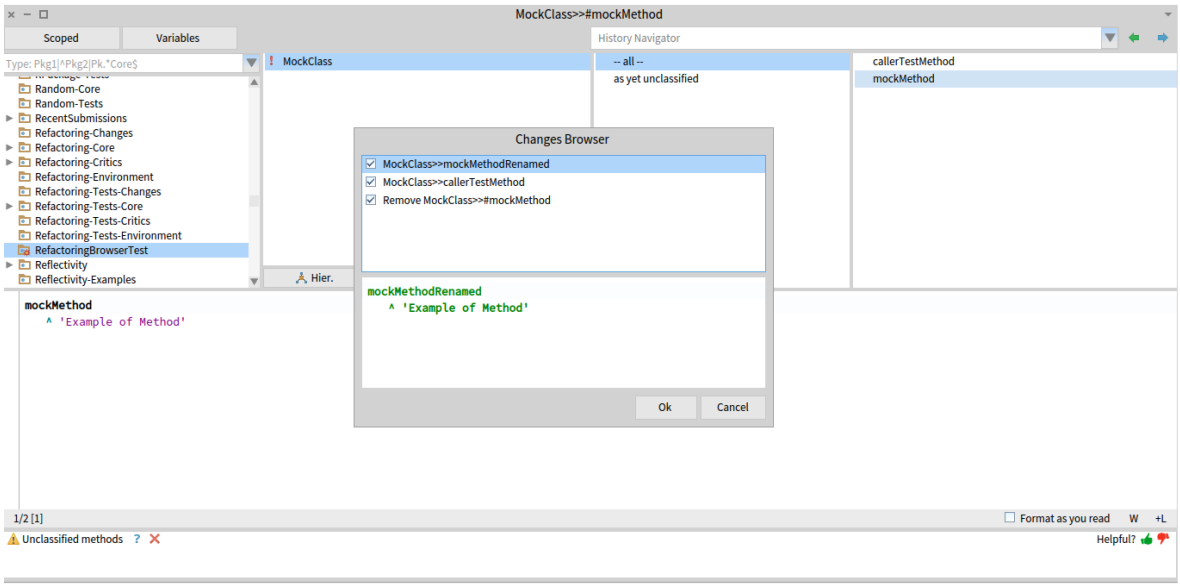

Figura 3.1. Ejemplo de un refactoring en Pharo.

3.2 El Framework de Refactoring

A continuación, se analizará la herramienta desde el punto de vista de un framework, enumerando sus componentes y señalando cómo se puede extender. Para explicar cada uno de estos componentes se utilizará como ejemplo el refactoring *Rename Class* ya que es bastante simple y esto permite comprender fácilmente cada una de las etapas de ejecución de un refactoring. En su tesis doctoral, *Practical Analysis for Refactoring* [Roberts99], Don Roberts destaca cinco componentes principales:

1. Refactorings.
2. Condiciones.
3. Objetos que representan cambios en el código.
4. Parser.
5. Reescritura del código fuente.

Los componentes más importantes que merecen una descripción detallada son los primeros tres, los últimos dos tienen que ver con la forma en que se aplican las transformaciones sobre el código. Dado que el objetivo de la herramienta que se pretende crear es re-ejecutar refactorings existentes y no crear nuevos refactorings, tanto el Parser como la reescritura del código fuente se describirán muy brevemente.

3.2.1 Refactorings

Cada uno de los refactorings soportados por la herramienta se implementa como una subclase de la clase abstracta `RBRefactoring`. Esta clase utiliza el patrón de diseño *Template Method* para definir el algoritmo de ejecución de los refactorings, delegando algunos de sus pasos en las subclases. En la Figura 3.2 se muestra el protocolo principal de `RBRefactoring`.

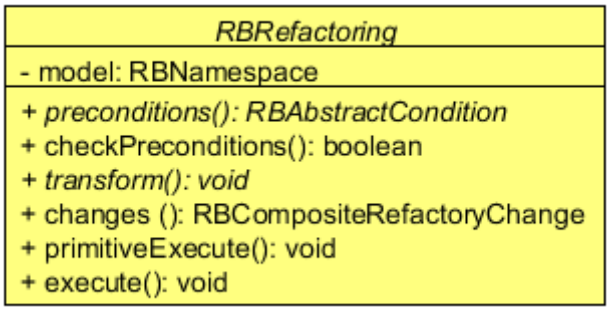

Figura 3.2. Protocolo de un Refactoring.

- ✓ `preconditions`: obtiene las precondiciones del refactoring. Cada refactoring posee sus propias precondiciones por lo cual este método debe ser redefinido en la clase de cada uno de ellos. En la siguiente sección se explicará cómo están implementadas las condiciones.
- ✓ `checkPreconditions`: Evalúa las condiciones obtenidas con el método anterior y retorna un valor booleano que indica si el refactoring es válido o no.
- ✓ `transform`: Este método se encarga de crear los cambios subyacentes al refactoring. Al igual que `preconditions` debe ser implementado en cada refactoring.
- ✓ `changes`: Luego de ejecutar `transform` este método obtiene los cambios que se deben ejecutar para aplicar el refactoring en el código.
- ✓ `primitiveExecute`: invoca a `checkPreconditions` para validar el refactoring y en caso de que la validación sea exitosa, utiliza `transform` para instanciar los cambios.
- ✓ `execute`: Realiza todo el ciclo de ejecución del refactoring. Invoca a `primitiveExecute` para validar y crear sus cambios, y luego ejecuta estos cambios.

Habiendo mostrado el protocolo común a todos los refactorings, se puede ver que para crear un nuevo refactoring se debe extender la clase `RBRefactoring` e implementar los métodos `preconditions` y `transform`. A medida que se avance en la descripción de los componentes del Refactoring Browser, se ampliará con mayor detalle qué función cumplen cada uno de los métodos anteriores.

Como se indicó anteriormente, los refactorings contienen los datos de los elementos del programa que modifican. Cabe destacar que, dado que en Smalltalk todo es un objeto, los elementos del programa también lo son y es por eso que los datos que tiene un refactoring son objetos que representan a los elementos sobre los cuales este se ejecuta. El refactoring *Rename Class* posee el objeto que equivale a la clase que será renombrada y una cadena de caracteres que contiene el nuevo nombre de la misma.

3.2.2 Condiciones

Las condiciones de un refactoring representan los distintos chequeos que se realizan para determinar su validez. Al igual que los elementos de programa y los refactorings, son objetos; la razón principal por la cual se representan mediante objetos es que hay refactorings que tienen chequeos en común, por lo tanto, incluir estáticamente en cada refactoring sus condiciones no es una buena idea. Una condición en su forma más simple es una instancia de `RBCondition` que puede ser evaluada mediante el método `check`. Un objeto `RBCondition` posee un bloque (*closure*) que es evaluado cuando se invoca a `check`, una cadena de caracteres que describe el error cuando la evaluación falla (*errorString*), y un tipo (*type*) que contiene información de la condición que permite compararla con otras condiciones. El hecho de poder comparar las condiciones es un gran beneficio que da la posibilidad de determinar las dependencias entre refactorings de una forma práctica. En el Capítulo *Dependencias entre Refactorings* se desarrollará en detalle este mecanismo.

Considerando que hay refactorings que comparten algunas precondiciones, `RBCondition` posee un conjunto predefinido de constructores para instanciar condiciones típicas. Por ejemplo, el método de clase `definesSelector:aSelector in:aClass` instancia una condición que chequea si la clase *aClass* define un método denominado *aSelector*. Otro ejemplo es el método `empty` que instancia una condición vacía (devuelve siempre verdadero).

Una característica interesante de las condiciones es que se pueden componer utilizando los operadores lógicos &, | y not. Cuando se envía el mensaje & a una condición, se obtiene una instancia de `RBConjunctiveCondition` que contiene tanto la condición receptora del mensaje como también la condición enviada por parámetro. El método `check` de la clase `RBConjunctiveCondition` evalúa las dos condiciones y retorna la conjunción de los resultados. Con el método | se pueden crear condiciones disyuntivas, usando la ley de De Morgan ($\neg(p \vee q) = \neg p \wedge \neg q$) crea una condición disyuntiva a partir de una condición conjuntiva, es decir que este método también retorna una instancia de `RBConjunctiveCondition`. Por

último, el método `not` crea una instancia de `RBNegationCondition` que representa a la condición receptora del mensaje negada.

Todas las condiciones son subclases de una clase abstracta denominada `RBAbstractCondition`, en la Figura 3.3 se muestra un pequeño diagrama que muestra la relación entre las clases mencionadas antes. Se puede observar que el método `check` debe ser redefinido por cada condición debido a que cada una de ellas se compone de una forma distinta: `RBConjuntiveCondition` posee una condición `left` que representa la condición izquierda de la conjunción y otra condición `right` que equivale a la parte derecha. El método `check` en esta clase evalúa `left` y `right` y en caso de que alguna falle devuelve falso. Por otro lado, `RBNegationCondition` se compone de una sola condición denominada `condition`, su implementación de `check` evalúa `condition` y retorna el resultado inverso de la evaluación. Por último, la condición más simple es `RBCondition` dado que no se compone de ninguna condición, su método `check` evalúa el bloque *(block)* que describe la condición.

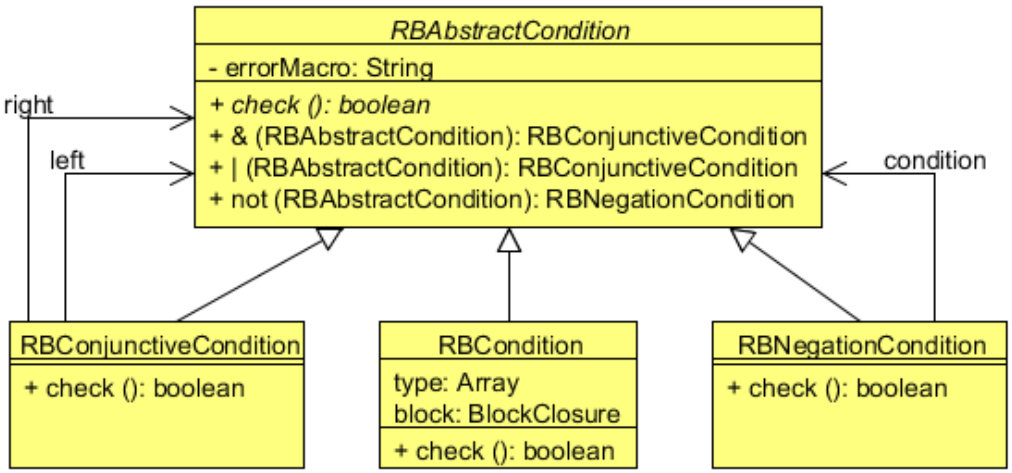

Figura 3.3. Condiciones del Refactoring Browser.

El método `preconditions` del refactoring *Rename Class* está definido de la siguiente forma:

```
preconditions
    ^(RBCondition withBlock: [class notNil and: [class isMeta not]]
        errorString: className , ' is not a valid class name')
            & (RBCondition isValidClassName: newName)
            & (RBCondition isGlobal: newName in: self model) not
```

Básicamente las precondiciones de *Rename Class* chequean que la clase que se quiere renombrar (*class*) sea una clase válida y que el nuevo nombre de la clase (*newName*) sea un nombre válido: esto implica que el nombre sea sintácticamente correcto y que no exista otra clase con el mismo nombre.

3.2.3 Ejecución de las transformaciones

Luego de haberse chequeado sus precondiciones, el refactoring está preparado para ejecutarse. En las primeras versiones del Refactoring Browser, después de validar sus precondiciones los refactorings con el método `transform` ejecutaban directamente los cambios en el código. Posteriormente, la ejecución de las transformaciones fue dividida en dos etapas:

- Creación de los cambios: es el primer paso en la ejecución de un refactoring. Este paso lo realiza el método `primitiveExecute`.
- Ejecución de los cambios: luego de haber instanciado los cambios en la etapa anterior, el usuario selecciona cuáles de ellos ejecutar y finalmente son aplicados al código. Esta etapa es realizada dentro del método `execute` del refactoring.

Creación de los cambios

Anteriormente se adelantó que los refactorings no trabajan directamente sobre el código del programa. En realidad, operan con un modelo del sistema que está representado con la clase `RBNamespace` y que se encarga de crear, registrar y simular los cambios que implica el refactoring. Todos los refactorings deben existir en el contexto de un `RBNamespace`, por lo que al momento de instanciar un refactoring se puede proporcionar el modelo sobre el cual este operará y en caso de no proveerlo, automáticamente se creará un modelo exclusivamente para ese refactoring.

Teniendo en cuenta que el modelo es una representación del sistema, no trabaja sobre los elementos de programa reales, sino que utiliza un conjunto de clases para representarlos, de esta manera, las clases se modelan con `RBClass` y los métodos con `RBMethod`. Cuando se crea un refactoring el modelo se encarga de crear la representación de cada uno de los elementos de programa provistos, de modo de asegurar que este refactoring únicamente opere dentro del modelo.

Los cambios subyacentes a un refactoring también están modelados como objetos siendo cada cambio una subclase de una clase abstracta `RefactoryChange`. El framework de refactoring define un total de doce subclases de `RefactoryChange` que se utilizan para implementar todos los refactorings. Entre estas subclases se encuentran `RBRenameClassChange` que permite

cambiarle el nombre a una clase del sistema y `RBAddMethodChange` que crea un nuevo método. Además de estas clases, teniendo en cuenta que un refactoring suele implicar varios cambios, la clase `RBCompositeRefactoryChange` representa a una colección de cambios.

El método `transform` del refactoring le indica al modelo cuáles son los cambios que debe crear y a partir de esto, el modelo los instancia y los almacena, posteriormente cuando el refactoring los necesita se los pide. Además de guardarlos, simula el efecto que tienen sobre el sistema registrando las clases que fueron incorporadas, modificadas y eliminadas. Las clases hacen exactamente lo mismo con los métodos y con las variables de instancia y de clase. El hecho de conservar los elementos que fueron modificados hace que el modelo actúe como una caché, evitando tener que buscar y crear nuevamente las representaciones de los elementos de programa que ya fueron accedidos.

Continuando con el ejemplo del refactoring *Rename Class*, a continuación, se muestra su implementación de `transform`:

```
transform
    self model
        renameClass: class
        to: newName
        around: [self renameReferences]
```

En el código se puede observar que el refactoring solicita al modelo, a través del método `renameClass: class to: newName around: aBlock`, que modifique el nombre de la clase representada por *class*. El modelo renombra *class* utilizando *newName* para simular el cambio y luego crea y registra una instancia de `RBRenameClassChange` para poder ejecutar la modificación sobre el código. El método `renameReferences` básicamente busca los fragmentos de código en los cuales se referencia la clase que se renombrará y actualiza su nombre. Para realizar esta acción nuevamente el refactoring invoca al modelo.

Ejecución de los cambios

La segunda etapa de la ejecución de los cambios a simple vista es más sencilla que la primera. En este punto, el refactoring ya fue validado y los cambios que involucra ya fueron creados, por lo cual solo resta persistirlos en el sistema. Para ello se utiliza un objeto `RBRefactoringManager` cuya función es llevar un registro de todos los refactorings que se ejecutan en el sistema y recuperar sus objetos `RBRefactoryChange` para que el administrador de los cambios (`RBRefactoryChangeManager`) finalmente los ejecute.

Una característica importante de los objetos `RBRefactoryChange` es que además de proporcionar un método para efectuar el cambio en el código, también proporcionan una operación que permite deshacer este cambio. Más específicamente cada subclase de `RBRefactoryChange` implementa el método `asUndoOperation` que construye un nuevo cambio que representa la operación inversa. En algunos cambios esta operación es bastante simple, en otros casos puede volverse bastante compleja como por ejemplo en el `RBCompositeRefactoringChange` que permite deshacer una secuencia completa de cambios. Esta posibilidad de deshacer un conjunto de cambios es lo que permite implementar la operación de deshacer un refactoring completo.

En la Figura 3.4 se muestra la relación entre las principales clases de la implementación del Refactoring Browser de Pharo.

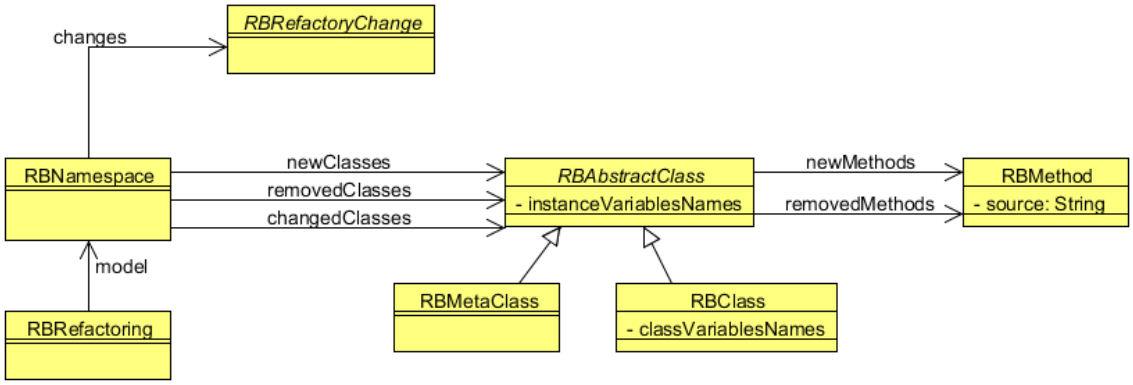

Figura 3.4. Principales clases del Refactoring Browser

3.2.4 Parser y reescritura del código fuente

Al momento de crear los cambios de un refactoring es cuando actúan el parser y la reescritura del código fuente. En el caso del *Rename Class*, para actualizar las referencias a la clase es necesario modificar el código de los métodos que la acceden. El código de cada método está representado por su árbol de sintaxis que contiene todos los elementos que lo componen.

Para editar un método es necesario obtener su árbol de sintaxis y modificar ciertas partes de éste, para poder finalmente recompilarlo. El Refactoring Browser provee un parser, representado por la clase `RBParser`, que permite entre otras cosas, construir el árbol de sintaxis de un método a partir de un string que contiene su definición mediante el método `parseMethod:`. La modificación del árbol se realiza por medio de la clase `RBParseTreeRewriter`, que da la posibilidad de definir reglas de reescritura que reemplazan determinados patrones de texto por otros. Estas reglas se aplican a un árbol de sintaxis en particular, y como resultado se obtiene el

árbol modificado. Esta clase utiliza el patrón Visitor para recorrer e inspeccionar cada uno de los nodos de un árbol sintáctico buscando los patrones de texto que debe reemplazar.

Los árboles de sintaxis modificados se usan posteriormente para instanciar los cambios correspondientes. Siguiendo con el refactoring *Rename Class*, se instanciará un `RBAddMethodChange` por cada método que haga referencia a la clase renombrada. En [Roberts99] se explica detalladamente con ejemplos cómo definir diferentes reglas de reescritura.

4 Grabado y re-ejecución de refactorings

4.1 Introducción

Desarrollar y mantener software es una tarea difícil de llevar adelante, principalmente porque suele ser difícil comprender código que ya fue escrito. El refactoring es una técnica que se utiliza para facilitar el mantenimiento de un sistema y consiste en realizar continuas reestructuraciones al código para mantener su diseño simple a medida que la complejidad aumenta. La idea de estos cambios es mejorar características internas del software, sin alterar las funciones que realiza.

Más allá de sus beneficios bien conocidos, el refactoring no puede alcanzar todo su potencial cuando se utiliza en componentes de software que son reutilizados. Si el desarrollador de un componente reusable quiere aplicar refactorings sobre su código, está limitado a cambiar su implementación interna o en todo caso a expandir su API; no puede eliminar o cambiar partes de la API debido a que esto hace que dejen de funcionar las aplicaciones que hacen uso del componente. Hacer cambios sobre un componente que invaliden sus aplicaciones clientes tiene un altísimo costo para los desarrolladores de estas aplicaciones y es por eso que muchas veces los programadores de los componentes dejan de realizar cambios que podrían reducir la complejidad de su código o incrementar su eficiencia. Un ejemplo de este tipo de cambios es la eliminación de métodos o clases *deprecated:* los elementos marcados como *deprecated* raramente son quitados del código para mantener la compatibilidad con las aplicaciones clientes. La acumulación de código obsoleto termina incrementando la complejidad de todo el código en general.

La solución a la problemática anterior que se plantea en el siguiente trabajo es grabar los cambios que se realizan sobre un componente y luego reproducirlos en las aplicaciones clientes de éste, siendo estos cambios refactorings. De esta manera, el código del componente debe ser cambiado por medio de refactorings usando Refactoring Browser. La información de cada refactoring aplicado es guardada y posteriormente es trasladada a las aplicaciones clientes para recrear estos refactorings en su código.

Cuando el responsable de un componente realiza un refactoring, su alcance está limitado al componente sobre el cual se aplica. Mientras que cuando se recrea en una aplicación cliente, el refactoring además de modificar el código del componente, también afecta al código del sistema que lo usa. El hecho que el alcance de un refactoring se determine en forma dinámica según el ambiente en el cual se ejecuta, lo convierte en una herramienta muy poderosa que da la posibilidad de actualizar automáticamente en una sola operación la API y el código interno de un componente, y los sistemas usuarios de este.

Con esta forma de trabajo, los desarrolladores generan una nueva versión de un componente realizando una secuencia de refactorings. Cuando consideran que se hicieron suficientes cambios para lanzar una nueva versión, exportan estos refactorings para que los usuarios puedan obtenerlos y re-ejecutarlos. Así, las diferencias entre una versión y otra de un componente se describen en cada uno de los refactorings que se aplicaron, lo cual es un beneficio importante para los usuarios ya que pueden entender fácilmente los cambios realizados.

Un punto a favor que tiene esta solución es que no requiere una infraestructura centralizada como un Sistema de Control de Versiones o un Sistema de Gestión de la Configuración, sino que está pensada para ser implementada como un plug-in de los IDE actuales, partiendo de la base que actualmente la mayoría de estos ofrecen una herramienta para realizar refactorings automáticamente.

4.2 Escenario de trabajo

El ambiente de desarrollo elegido para este trabajo es Pharo Smalltalk. La ventaja que presenta este entorno es que su implementación del Refactoring Browser ofrece una gran variedad de refactorings, de manera que prácticamente cualquier edición manual que se haga sobre el código puede realizarse también por medio de un refactoring, esto da la posibilidad de exportar cualquier cambio del código en forma sencilla y con un contenido semántico: más allá de los cambios sintácticos que implican, los refactorings también explican el significado de cada uno de estos cambios.

Una característica de Smalltalk que es importante mencionar es que es un lenguaje de programación basado en imagen. Los lenguajes tradicionales requieren que el código sea escrito en archivos de texto usando un editor de texto plano o un IDE, este código posteriormente se compila y se ejecuta. Una vez que finaliza su ejecución, el programa es eliminado de la memoria y por lo tanto su estado no se conserva entre sus distintas ejecuciones. Esto hace que se utilicen distintos repositorios de información (bases de datos, archivos XML, etc.) para persistir el estado del programa y/o cualquier información de éste. En este tipo de lenguajes, el estado del programa está separado de su código.

Smalltalk en cambio no diferencia el código de un programa y su estado. La forma de trabajar en este tipo de lenguajes es usando una *imagen* que representa a un sistema Smalltalk completo en un instante de tiempo. Una *imagen* no es más que un archivo que contiene una colección de objetos Smalltalk, que posee las siguientes características:

- ✓ Sistema Smalltalk completo: la *imagen* incluye todo el entorno necesario para trabajar; browser para inspeccionar los elementos de programa, una terminal para evaluar

expresiones, entre otras herramientas. Todo el código Smalltalk puede ser modificado libremente; en el capítulo anterior se mencionó que en Smalltalk todo está representado mediante un objeto, incluso las clases y los métodos. De manera que se podría cambiar por ejemplo la forma de agregar una nueva clase.

- ✓ Instante de tiempo: La noción tiempo tiene que ver con que la *imagen* en cualquier momento puede ser guardada y recuperada en el estado en el que se la dejó, aun estando en la mitad de un cálculo. Al momento de guardar la imagen sus objetos son persistidos en el archivo correspondiente y posteriormente pueden ser cargados otra vez en memoria conservando su estado anterior.

Si bien los lenguajes basados en imagen dan la posibilidad de mantener el estado de un sistema sin tener que usar recursos externos como una base de datos, exportar código en este tipo de lenguajes no es una tarea sencilla. En un lenguaje tradicional para compartir código alcanza con copiar y exportar el archivo de texto que contiene el código deseado y luego incluirlo en el paquete correspondiente. En Smalltalk en cambio, como el código está contenido en una imagen no es posible exportarlo en formato de texto, sino que se transporta en un formato de archivo específico que luego puede ser importado en otra imagen. De esta manera, el desarrollador necesariamente tiene que importar el código en una imagen para ver su contenido.

Considerando la limitación planteada antes de los lenguajes basados en imagen, un mecanismo que permita compartir código fácilmente como el caso del grabado y la re-ejecución de refactorings, cobra una mayor importancia en este tipo de lenguajes que en los lenguajes tradicionales.

Habiendo detallado el escenario de trabajo, se podría redefinir el objetivo de la herramienta a desarrollar: exportar los refactorings realizados en una imagen Smalltalk y posteriormente construirlos y re-ejecutarlos en otra. La imagen en la cual se aplican los refactorings (de ahora en adelante se denominará imagen origen) representa el ambiente de trabajo en el cual se desarrolla un componente de software reusable, mientras que la imagen en la que se recrean estos refactorings (imagen destino) es el sistema Smalltalk en el cual reside la aplicación que hace uso de este componente.

Es importante diferenciar además la forma en que suele reusarse un componente en este tipo de ambientes. En el caso de que el componente a reusar es un framework, hay dos formas de reutilizarlo: cuando el framework es de "caja negra", se reúsa creando configuraciones especiales que instancian clases existentes en el framework y conectan las instancias de una forma particular. En cambio, cuando el framework es de "caja blanca" la forma de reúso es creando nuevas subclases además de configuraciones. Con esto lo que se quiere remarcar es que no solo están

involucrados los cambios de API sino el código completo del componente y sus jerarquías de clases.

4.3 Grabado de refactorings

El primer paso en el desarrollo de la herramienta es encontrar la forma de capturar y grabar los refactorings una vez que son ejecutados. Posteriormente estos refactorings deberían ser exportados del algún modo para poder transportarlos a otras imágenes de trabajo.

4.3.1 Captura de refactorings

Ya se sabe que los refactorings son objetos, por lo que capturar un refactoring significa obtener el objeto que lo representa. En principio la alternativa que surgió para acceder a este objeto fue encontrar el método en el que se instancia el refactoring una vez que el usuario decide aplicarlo desde el menú de contexto y editar este método para agregar el objeto refactoring a una colección propia de refactorings ejecutados. Esta opción presenta una clara desventaja que es que por cada refactoring que se quiera incorporar a la herramienta, tendrá que cambiarse el código que lo instancia para poder guardarlo. La otra limitación que presenta está relacionada con los datos que el refactoring necesita para ejecutar: es probable que el refactoring solicite datos al usuario durante su ejecución para aplicarse correctamente. Considerando que esta alternativa obtiene el refactoring ni bien se instancia, habrá datos que el refactoring posteriormente necesite para su re-ejecución que todavía en ese punto no están disponibles. Por lo tanto, la captura debería realizarse una vez que el refactoring fue ejecutado para contar con todos sus datos.

Una forma correcta de obtener el refactoring después de haber finalizado su aplicación es al final del método `primitiveExecute` de `RBRefactoring`. Para ese momento terminó la primera etapa de ejecución del refactoring por lo que todos los datos que necesita para ejecutarse ya fueron solicitados y pueden ser accedidos. También podría ser capturado en su método `execute`: la característica que tiene este método es que ejecuta todos los cambios que implica un refactoring. En el Capítulo anterior cuando se describió el uso del Refactoring Browser se mostró que al ejecutar un refactoring es posible elegir cuáles de sus cambios aplicar. Considerando que `execute` no da la posibilidad de elegir los cambios, para lograr este comportamiento el menú de contexto de Pharo después de instanciar el refactoring no utiliza `execute`, sino que invoca a `primitiveExecute` y después ejecuta uno por uno los cambios elegidos por el usuario. Por lo tanto, la captura de los refactorings que el usuario realiza desde el menú de contexto necesariamente debe ser en `primitiveExecute`.

El punto en contra que tiene efectuar la captura de un refactoring en `primitiveExecute` es que como este método está definido en `RBRefactoring`, cualquier refactoring que se ejecute será capturado. Si bien Refactoring Browser posee una gran cantidad de refactorings, en la herramienta interesa exportar y re-ejecutar solamente aquellos que pueden afectar el código de la aplicación que usa un componente, de forma que hay varios de los refactorings que no deberían capturarse. Para descartar aquellos refactorings que no son de interés se incorporó un método denominado `isReplayable` que indica si el refactoring puede ser exportado o no; su implementación por defecto en `RBRefactoring` devuelve falso, por lo que debe ser redefinido en cada uno de los refactorings que se quieran exportar.

A continuación, se muestra el código resultante de `primitiveExecute`. Se puede observar que la captura se realiza después de `transform`, luego de haber solicitado todos los datos necesarios. La clase `ReplayableRefactoringManager` guarda cada uno de los refactorings capturados; esta clase es la clase principal de la herramienta que además de guardar los refactorings realiza otras funciones que se irán describiendo a lo largo del trabajo.

```
primitiveExecute
    self checkPreconditions.
    self transform.
    (self isReplayable) ifTrue: [ReplayableRefactoringManager instance addRecordedRefactoring: (self serialize)].
```

Un aspecto del código anterior que puede llamar la atención es que una vez que se captura el refactoring, no se guarda el objeto que lo representa, sino que se almacena el resultado de enviarle el mensaje `serialize` a este objeto. En la siguiente sección se explicará que función cumple este mensaje y por qué se guarda su resultado en lugar de guardar el objeto refactoring.

Hasta el momento cada vez que el usuario realiza un refactoring soportado por la herramienta este es capturado y guardado, sin importar el momento en el cual se haga. Sin embargo, lo deseable sería que el usuario pueda elegir en qué momento empezar a capturar refactorings y cuándo finalizar. Incluso una vez comenzada la captura de refactorings, el desarrollador podría querer efectuar algún refactoring que no tiene intención de exportar, por lo que también sería útil poder poner en pausa la captura para poder llevar a cabo alguna tarea específica y luego reanudarla en el estado en el cual se la dejó. El programador del componente a actualizar, en su imagen Smalltalk podría estar desarrollando otros sistemas además de este componente, de manera que sin la posibilidad de comenzar, pausar y finalizar una captura se podrían "mezclar" refactorings de distintos sistemas. Es importante resaltar que este caso podría darse porque se está trabajando sobre un lenguaje basado en imagen; en un lenguaje tradicional es muy probable que para cada programa que se desarrolla se utilice una instancia distinta de un IDE particular, por lo que los refactorings aplicados en cada sistema nunca residirían en el mismo espacio.

Para poder seleccionar cuáles son los refactorings que se quieren exportar, la captura de refactorings se implementó por medio de **sesiones**: el usuario puede iniciar una captura, aplicar algunos refactorings y luego pausarla, en cualquier momento puede reanudarla y seguir grabando refactorings hasta finalizar la sesión. En este punto es cuando los refactorings grabados son exportados para poder re-ejecutarse en otra imagen Smalltalk. Estando la sesión activa o pausada, el usuario también podrá descartar los refactorings grabados hasta el momento lo que genera que la sesión se cierre sin llevar a cabo la exportación.

Teniendo en cuenta que una sesión de trabajo puede estar en distintos **estados**, se utilizó el patrón de diseño *State* para modelar cada uno de ellos. Los estados posibles son tres: *Activa, Pausada y Cerrada*. Tal como se vio en el código de `primitiveExecute`, la captura de un refactoring se efectúa enviando el mensaje `addRecordedRefactoring:` a un objeto `ReplayableRefactoringManager`. Este objeto no guarda directamente el refactoring, sino que le envía el refactoring a la sesión que esté iniciada en ese momento y esta sesión dependiendo su estado almacena el refactoring o no. Cuando se tenga que hacer la exportación de los refactorings, el `ReplayableRefactoringManager` le solicitará a la sesión los refactorings que posee grabados.

4.3.2 Exportación de refactorings

Definida la captura de refactorings, el siguiente paso es encontrar la forma de exportarlos para que puedan ser re-ejecutados en otra imagen Smalltalk. El mecanismo de persistencia usado para guardar los refactorings debería permitir importarlos de forma sencilla en otra imagen de otra computadora, es por eso que se eligió utilizar **archivos**. De esta manera, el programador de un componente realiza una sesión de captura de refactorings para aplicar los cambios que crea necesarios en su imagen de trabajo, y luego estos refactorings son exportados a un archivo. Más tarde los usuarios del componente acceden al archivo y utilizando esta herramienta únicamente deben importar ese archivo y re-ejecutar sus refactorings.

Para exportar los refactorings a un archivo se usó una librería de Pharo denominada **Fuel**. Fuel es un framework de serialización binaria de objetos que fue creado como una solución a los problemas de performance que suelen tener los frameworks que hacen una serialización textual (archivos JSON, XML, etc.). La ventaja de esta herramienta que motivó su utilización para este trabajo, es que es muy sencilla de usar debido a que no requiere ningún tipo de configuración: Fuel guarda y reconstruye los objetos desde un archivo automáticamente, permitiendo serializar prácticamente cualquier objeto de Pharo, por ejemplo, un método compilado o una pila de ejecución. El programador únicamente debe indicar el objeto que quiere guardar y el nombre del

archivo destino en el caso de la serialización, y solamente el nombre del archivo a leer cuando se quiere reconstruir un objeto.

Con una herramienta como Fuel que da la posibilidad de guardar en un archivo cualquier objeto de Pharo, la primera idea que surge para exportar los refactorings es serializar los objetos que los representan; al serializar el refactoring también se serializan los elementos de programa que este necesita. Sin embargo, la limitación que presenta esta opción es que cada imagen Smalltalk posee sus propios objetos que son válidos únicamente en esta: todas las imágenes tienen por ejemplo un objeto que representa a la clase `Integer`, y si bien los valores de este objeto en cada imagen probablemente sean los mismos, los objetos son distintos en el sentido que tienen una identidad diferente. Si se serializa el objeto refactoring ejecutado en la imagen origen, en la imagen destino Fuel intentará reconstruirlo usando los objetos que pertenecen a la imagen origen lo cual derivará en un error. Sabiendo que cada imagen tiene sus propios objetos, los refactorings deberían ser reconstruidos en una imagen con los objetos propios de ella.

Aun suponiendo que la limitación anterior no existe, hay otra cuestión que tiene que ver con las dependencias entre refactorings que hace inviable la alternativa planteada. Considerando que puede existir una dependencia entre un par de refactorings cualquiera, si se guardan los objetos de estos refactorings, al momento de importar el archivo Fuel intentará reconstruir los dos refactorings, fallando la reconstrucción del segundo dado que la información que necesita para instanciarse todavía no está disponible: la proporciona la ejecución del primero. Este problema se verá con mayor detalle en los siguientes capítulos.

A partir de las restricciones encontradas, se determinó que la estrategia de exportación debe cumplir con dos propiedades:

1. El refactoring tiene que reconstruirse en cada imagen con los objetos propios de esta.
2. El refactoring no tiene que instanciarse ni bien se importa el archivo, sino que se debe poder elegir cuándo crearlo. Así, es posible crearlo recién cuando los datos que necesita están disponibles.

Para satisfacer las dos propiedades lo que se guarda en un archivo son los datos que cada refactoring necesita para instanciarse y ejecutarse. Como ya se sabe que estos datos son objetos y que se deben usar los objetos propios de cada imagen en la que se importe el archivo, en realidad se exportan los nombres de los elementos de programa que requiere el refactoring y a partir de estos nombres se buscan en cada imagen los objetos que los representan.

A modo de ejemplo, se muestran los datos que se deben guardar en el refactoring *Add Class*. Con esta información el refactoring puede ser construido y ejecutado en cualquier imagen Smalltalk.

`#refactoring`	**addClassRefactoring**
`#superclass`	Nombre de la clase que será la superclase de la clase nueva
`#subclasses`	Nombres de todas las clases que serán subclases de la clase nueva
`#newName`	Nombre de la clase nueva
`#category`	Nombre del paquete de Pharo en el que se agregará la clase.

Figura 4.1. Exportación del refactoring *Add Class*.

El mensaje `serialize` que se veía en el código de la captura de un refactoring se encarga de construir un objeto listo para ser exportado que posee todos los datos de un refactoring como el ejemplo anterior. El siguiente punto a determinar es a qué clase pertenece este objeto, para esto surgen dos posibilidades:

- ✓ Crear una clase por cada refactoring que represente la información que este requiere. Esta opción tiene la ventaja de ser muy simple, pero requiere crear una nueva clase por cada refactoring para el cual se quiera dar soporte en la herramienta, lo que hace que la solución sea menos escalable. Además, dado que el Refactoring Browser ya posee una clase por cada refactoring, implementando esta opción se estaría replicando su modelo lo cual no es una buena idea.
- ✓ Usar una única clase para guardar la información de cualquier refactoring. Presenta la ventaja que se pueden incorporar nuevos refactorings sin tener que replicar el modelo de Refactoring Browser. Esta estrategia es más compleja que la anterior pero la capacidad de reflexión de Smalltalk permite implementarla de forma sencilla.

Teniendo en cuenta su ventaja y su facilidad de implementación, se optó por la segunda posibilidad. La clase implementada se denomina `RefactoringData`; una instancia de esta clase posee una colección clave-valor para poder almacenar cualquier tipo y cantidad de datos de refactorings. En la siguiente sección se explicará cómo se reconstruye el refactoring a partir de estos datos.

4.4 Re-ejecución de refactorings

Para poder re-ejecutar un conjunto de refactorings en otra imagen Smalltalk lo primero que se debe hacer es reconstruir los refactorings en la imagen destino a partir de la información contenida

en el archivo importado. Al final de la sección anterior se mencionó que los datos de cada refactoring son guardados en un archivo por medio de objetos `RefactoringData`. Un objeto de este tipo además de los datos de un refactoring, contiene el nombre del refactoring que representa y un método por cada refactoring que lo instancia usando sus propios datos. Usando reflexión, el objeto se envía a sí mismo el mensaje denotado por el nombre del refactoring obteniendo como resultado el objeto refactoring listo para ejecutarse. El método `buildRefactoring:` de `RefactoringData` es el encargado de llevar a cabo esta acción y devuelve el refactoring instanciado.

Recorriendo los `RefactoringData` de un archivo y enviándole al resultado de `buildRefactoring:` el mensaje `execute`, se pueden re-ejecutar todos los refactorings en la imagen destino. Los refactorings fueron guardados en el archivo en el orden en que fueron aplicados originalmente, por lo tanto, la re-ejecucion se haría en el mismo orden en la imagen destino.

La Figura 4.1 muestra un diagrama con la secuencia realizada para instanciar y re-ejecutar un refactoring. El mensaje `perform:` en Smalltalk permite que un objeto se auto envíe un mensaje con un nombre determinado. En este ejemplo el nombre del mensaje es `#renameMethod` porque la instancia de `RefactoringData` representa a un refactoring *Rename Method*. Este método con los datos que posee el objeto, utiliza el mensaje `model:renameMethod:in:to` para instanciar el refactoring: este mensaje requiere el modelo sobre el cual trabajará, el nombre del método a renombrar, la clase en la que está definido y el nuevo nombre del método.

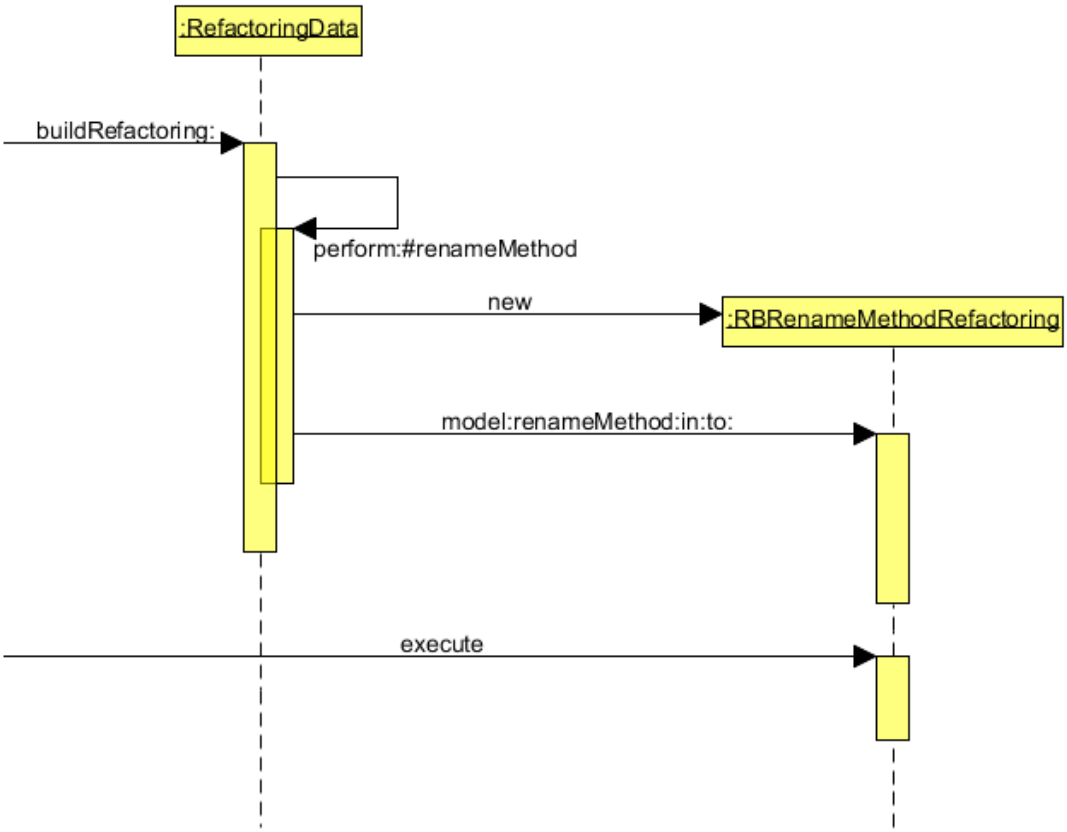

Figura 4.2. Secuencia de re-ejecución de un refactoring.

4.4.1 Información proporcionada por el usuario

Los refactorings suelen necesitar que el usuario ingrese determinados datos para poder aplicarse correctamente. Cuando el usuario realiza un refactoring en la imagen origen está bien que se proporcionen esos datos ya que de no ser así no se puede completar su ejecución. Sin embargo, cuando se va a re-ejecutar este refactoring en la imagen destino los datos que se requieren ya fueron provistos en su ejecución original y por lo tanto debería esos datos en lugar de solicitarlos nuevamente. Sería un error que esos datos vuelvan a ser ingresados dado que de esta manera es muy probable que los cambios que implica un refactoring sean diferentes en cada imagen que este se ejecuta lo que haría que la actualización del componente no sea consistente. Tomando como ejemplo el refactoring *Rename Method*, si cada vez que se re-ejecuta se ingresa un nombre distinto, cada imagen tendrá definido el mismo método con diferentes nombres.

Si el usuario de un componente decide re-ejecutar un refactoring determinado, la aplicación de un refactoring en la imagen destino debe ser exactamente igual a la ejecución en la imagen origen y para eso la re-ejecución debe usar la misma información que se utilizó en la ejecución. La única

diferencia entre la ejecución y la re-ejecución debe ser el alcance del refactoring: en la ejecución el refactoring modifica únicamente el componente que se está actualizando, y en la re-ejecución además del componente también se modifica la aplicación que lo usa.

Para lograr este comportamiento habría que diferenciar la ejecución de un refactoring de su re-ejecución de forma que cuando este se ejecute los datos sean solicitados al usuario, y cuando se re-ejecute se usen los datos pedidos previamente. Para utilizar los datos de la ejecución original en principio cuando se exporta el refactoring, habría que guardar no solamente los datos necesarios para su instanciación, sino que también se deben exportar los datos que se fueron pidiendo a lo largo de su ejecución. La siguiente cuestión es determinar en qué momento se solicitan los datos para poder reemplazar la petición al usuario por la información existente. Analizando los refactorings se encontró que la interacción con el usuario puede darse en dos etapas:

- ✓ Antes de la instanciación del refactoring: este caso se da en la mayoría de los refactorings. Primero se solicitan los datos y luego se instancia el refactoring con todos los datos que necesita. En el refactoring *Rename Method* se da este tipo de interacción.
- ✓ Durante su ejecución: El refactoring se instancia con un conjunto de datos mínimo y a medida que se va ejecutando pide los datos requeridos. De los refactorings soportados por la herramienta, este tipo de interacción se da solamente en dos: *Extract Method* y *Move Method.*

El primer caso no requiere diferenciar entre la ejecución y la re-ejecución de un refactoring debido a que como los datos se piden antes de crear el refactoring, en la imagen destino directamente se puede instanciarlo con los datos importados.

En el segundo caso, cuando se piden los datos el refactoring ya fue instanciado y por lo tanto si es necesario distinguir entre una ejecución original y una re-ejecución. Para hacer esta diferenciación en forma ordenada por cada refactoring que presenta este tipo de interacción se decidió crear una subclase que representa a la versión "re-ejecutable" de ese refactoring. Como es un refactoring re-ejecutable, se asume que fue previamente aplicado y por lo tanto cuando se instancia se deben proporcionar todos sus datos para que se ejecute sin la intervención del usuario. A continuación, se presenta un ejemplo de cómo funciona este mecanismo con un refactoring específico.

El refactoring *Extract Method* consiste en seleccionar un fragmento de código dentro de un método y extraerlo en un método nuevo con el objetivo de simplificar el método original. En Pharo cuando se invoca a este refactoring sobre una porción de código, el Refactoring Browser antes de pedir el nombre del nuevo método, primero chequea si existe otro método cuyo contenido sea equivalente al código extraído. Si existe ese método, directamente el código seleccionado se

reemplaza por una invocación a este método y no se crea uno nuevo. En caso de no existir un método equivalente, se solicita al usuario el nombre del método a crear. Dado que cuando se instancia el refactoring no se sabe si se creará un nuevo método o no, de ser necesario su nombre será pedido durante la ejecución de este refactoring.

Para poder re-ejecutar un refactoring de ese tipo, suponiendo que se necesitó crear un nuevo método, cuando se exporta el refactoring también se guarda el nombre de este método. Posteriormente en la imagen destino en lugar de instanciar el *Extract Method* estándar de Refactoring Browser, se crea una instancia de la versión re-ejecutable del refactoring a la que se le provee el nombre del método que debe producir y de esta manera el refactoring puede aplicarse sin que el usuario intervenga y respetando la información de la ejecución original.

4.4.2 Selección de refactorings a re-ejecutar

Cuando se actualiza un componente de software no solamente se corrigen problemas de la versión anterior, sino que también se incorporan nuevas funcionalidades y se eliminan aquellas que quedaron obsoletas. Teniendo en cuenta que la actualización se lleva a cabo por medio de refactorings en el código y que cada uno de estos refactorings describe los cambios que realiza, sería bueno que el usuario del componente pueda participar en el proceso de actualización pudiendo ver los refactorings que fueron aplicados y seleccionar aquellos que quiere re-ejecutar.

Observando cada uno de los refactorings que contiene la actualización, el usuario podría decidir si aplicarlo o no dependiendo del uso que le da al componente. Un ejemplo de uso puede ser que el desarrollador del componente quite cierta funcionalidad obsoleta mediante la eliminación de un método y que un usuario quiera seguir usándola: al ver que en el listado de refactorings aparece uno que elimina un método que para él es muy importante, decide no aplicar el refactoring y poder seguir utilizando ese método.

La posibilidad de poder elegir qué cambios quiere realizar posee la ventaja que le otorga un gran poder de decisión al usuario. Y esta característica es muy importante debido a que cuando un componente se actualiza por medio de la forma tradicional (modificando el código manualmente y luego distribuyendo el código compilado), los usuarios no pueden seleccionar solamente los cambios que desean; para utilizar la última versión del componente obligatoriamente se tienen que adaptar a los cambios que esta presenta. Muchas veces los usuarios por distintas razones deciden no adaptarse a la última versión del componente y por lo tanto utilizan una versión anterior de este con las desventajas que eso implica.

Si bien la selección de refactorings a re-ejecutar proporciona un gran beneficio para los usuarios de la herramienta, le agrega cierta complejidad al desarrollo de la misma que tiene que ver con

las posibles dependencias entre refactorings. Dado que hay refactorings que necesitan que otros se ejecuten previamente, si el usuario selecciona libremente los refactorings a aplicar podría darse el caso que elija un determinado refactoring y que descarte alguno de los cuales este necesita para realizarse correctamente, lo que terminará en un error. De forma que debería proveerse alguna solución para que estos casos no sucedan.

En caso de aplicar todos los refactorings y no dar la posibilidad de descartar alguno de ellos, no podría darse la situación anterior ya que los refactorings en el archivo exportado conservan el orden original en el que se aplicaron. De manera que, si se ejecutan respetando su orden, un refactoring nunca va a fallar por la falta de la aplicación de otro refactoring, si falla se debe a otros motivos que se estudiarán en el siguiente capítulo.

Por más que la selección de refactorings implique mayor complejidad en la herramienta, la flexibilidad que le otorga a los usuarios fue lo que motivó su inclusión en el desarrollo. Como veremos en los siguientes capítulos, cuando la herramienta recrea los refactorings en la imagen destino, primero realizará un chequeo de correctitud, y destacará los refactorings habilitados y los no habilitados (ver Capitulo 5). Una vez realizado este chequeo, se permite al usuario seleccionar aquellos refactorings que no quiera recrear, chequeando nuevamente la correctitud de la secuencia resultante (ver Capitulo 6).

4.5 Resumen

En este capítulo se describió el grabado y re-ejecución de refactorings como una alternativa para actualizar componentes de software que son reutilizados en la que los desarrolladores de estos componentes realizan y exportan las modificaciones a través de refactorings y los usuarios participan en estas actualizaciones decidiendo cuáles de estos cambios desean aplicar en sus programas.

Luego de presentar los beneficios de este mecanismo de actualización, se detalló el escenario de trabajo para desarrollar una herramienta que permita capturar y re-ejecutar refactorings en Pharo Smalltalk. Además de describir algunas características deseables de esta herramienta, se enumeraron las decisiones de diseño más relevantes que fueron tomadas para implementarla.

Hay otros aspectos a tener en cuenta al momento de desarrollar una herramienta de este tipo que serán desarrollados en los siguientes capítulos.

5 Validación de refactorings

5.1 Introducción

En el capítulo anterior se explicó el mecanismo utilizado para capturar un conjunto de refactorings en una imagen Smalltalk, exportarlos a un archivo, y luego poder recrearlos en otra imagen distinta. Hasta el momento se asume que las imágenes en las cuales se re-ejecutan los refactorings reúnen todas sus precondiciones y por lo tanto pueden aplicarse correctamente; si la imagen origen y la imagen destino poseen la misma versión de un determinado componente, los refactorings aplicados en una deberían poder también ser ejecutados en la otra y no tendrían por qué fallar.

Sin embargo, puede ocurrir que, teniendo la misma versión de un componente, un refactoring pueda ser aplicado en la imagen origen, pero no en la imagen destino. El motivo se debe a que en la imagen destino, el usuario desarrolla un sistema usando el componente y en el proceso de desarrollo toma decisiones que pueden invalidar los refactorings ejecutados en la imagen origen. Un ejemplo sencillo de este caso puede darse con el agregado de una clase: el responsable del componente incorpora mediante un refactoring una clase para soportar nueva funcionalidad y al mismo tiempo en la imagen destino el usuario crea una nueva clase con el mismo nombre. Cuando el usuario quiera actualizar el componente reproduciendo el refactoring que agrega la clase, este refactoring fallará porque ya existe en el sistema una clase con el mismo nombre. El ejemplo anterior es un caso muy simple porque la actualización del componente consiste en un solo refactoring. Si en lugar de tener uno solo la actualización tiene una secuencia de refactorings, un error en la re-ejecución de uno de ellos hará que los refactorings subsiguientes no se realicen.

Teniendo en cuenta que la re-ejecución de refactorings puede fallar, sería importante poder determinar previamente a su ejecución, los refactorings importados que pueden ser aplicados correctamente. Esto permitiría restringir la re-ejecución de refactorings a aquellos que son válidos en la imagen destino y así se aseguraría que todos los refactorings que se reproducen se hacen de manera exitosa. Al mismo tiempo, al validar los refactorings previo a su ejecución e informarle al desarrollador sobre los refactorings inválidos, se le da la oportunidad de corregir su código para que los refactorings puedan reproducirse exitosamente.

El objetivo de este capítulo es describir el método usado para validar un refactoring sin tener que aplicarlo en la imagen. Para entender su funcionamiento es necesario conocer la arquitectura del Refactoring Browser detallada en el Capítulo 3.

5.2 Ejemplo de motivación

Para mostrar cómo se validan los refactorings, se utilizará un pequeño framework como ejemplo. La idea es actualizarlo a través de dos refactorings y luego describir cómo se validan esos refactorings en una imagen que hace uso del framework.

Se trata de un framework que permite construir una red social de recomendación de objetos, donde estos objetos podrían ser autos, películas, libros, entre otros. El framework posee usuarios, ítems (los objetos) y calificaciones de los ítems hechas por los usuarios. La Figura 5.1 presenta un diagrama de clases.

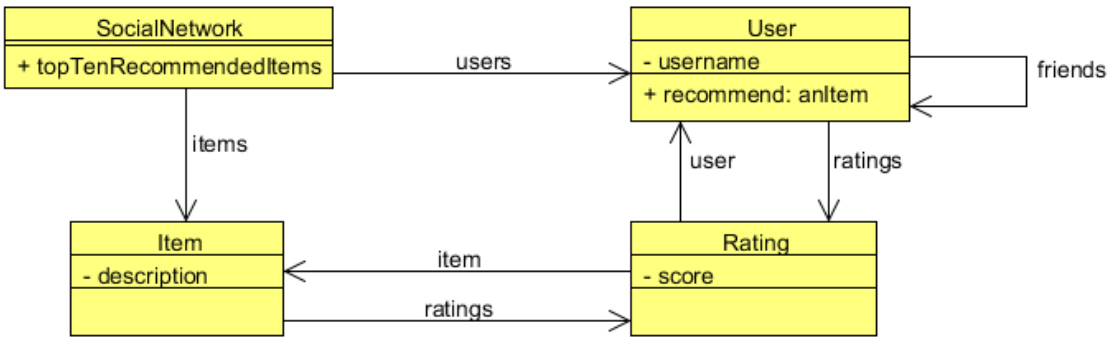

Figura 5.1. Diagrama de clases del framework.

Un usuario (`User`) posee un nombre y una colección con los usuarios que son sus amigos. Las calificaciones (`Rating`) tienen un puntaje (`score`), el ítem que se puntúa y el usuario que la realizó. El framework permite determinar si un usuario recomienda un determinado ítem mediante el mensaje `recommend:`. Un usuario recomienda un ítem si al menos tres de sus amigos efectuaron una calificación de este ítem. En base a las recomendaciones de los usuarios, `topTenRecommendedItems` obtiene los ítems más recomendados ordenados por cantidad de recomendaciones.

En la siguiente versión del componente, se pretende implementar que los ítems puedan ser recomendados siguiendo distintos mecanismos de recomendación. El objetivo es que la estrategia usada para recomendar pueda ser cambiada dinámicamente y que los usuarios del framework puedan definir sus propios métodos de recomendación. Para cumplir con estos requerimientos, la recomendación de un ítem por un usuario se implementará usando el patrón de diseño *Strategy*, esto implica en principio los siguientes refactorings:

1. *addClass* (`RecommendationStrategy`, `Object`, *subclases*): define una clase denominada `RecommendationStrategy`. `Object` es la superclase de la clase a agregar y *subclasses* denota al conjunto de las subclases de la nueva clase, en este caso no posee ninguna subclase.
2. *addMethod* (`RecommendationStrategy`,`#user:anUser recommend:anItem`)*:* En `RecommendationStrategy` se agrega el método `user:recommend`.

Se requieren algunas modificaciones más además de las anteriores para implementar *Strategy* pero para el propósito de este ejemplo alcanza con estos dos refactorings. A lo largo del capítulo se los llamará *R1* y *R2* respectivamente.

5.3 Precondiciones

Teniendo en cuenta que las precondiciones de un refactoring reflejan las propiedades que se deben cumplir para que se pueda aplicar, se podría determinar fácilmente su validez enviándole el mensaje `checkPreconditions`, si bien debería ser así, hay veces que con esto no es suficiente. Si se analiza cuáles son las precondiciones de *R2*, la precondición elemental es que debe existir una clase cuyo nombre sea `RecommendationStrategy`. Pero si se analizan las precondiciones definidas en Refactoring Browser para el refactoring *Add Method*:

```
preconditions
   | selector method |
   method := RBParser parseMethod: source
             onError:
                [:string :position |
                ^RBCondition
                   withBlock: [self refactoringFailure: 'The sources could not be parsed']].
   selector := method selector.
   selector isNil ifTrue: [self refactoringFailure: 'Invalid source.'].
   ^(RBCondition canUnderstand: selector in: class) not
```

Se puede observar que no existe una condición que chequee la validez de la clase en la que se agregará el método; únicamente se valida el código asociado al método y que no exista definido un método con el mismo nombre en dicha clase. Esta última condición permite ver que el refactoring asume que *class* ya está asociada a `RecommendationStrategy`. No se chequea la existencia de la clase porque forma parte de los argumentos que se necesitan para instanciar el refactoring, de manera que si la clase no es válida el refactoring directamente no debería crearse. De hecho si el refactoring se instancia utilizando como argumento cualquier otro objeto que no corresponda a una clase, la ejecución de `checkPreconditions` fallará indicando que *class* no entiende el mensaje `canUnderstand:in:`.

Como ya se indicó en el Capítulo 3, cuando el usuario decide realizar un refactoring primero selecciona en el browser el elemento de programa que quiere modificar y luego se crea una instancia del refactoring elegido utilizando el elemento elegido. De esta manera nunca puede ocurrir que un refactoring sea creado con un elemento de programa inválido, por lo que tiene lógica asumir dentro de los refactorings que los elementos sobre los cuales trabajan son legales. Pero cuando se trata de recrear refactorings realizados previamente en otra imagen, sí se puede dar que los elementos sean inválidos (como sucede en el ejemplo). Si se da este caso sería deseable que `checkPreconditions` devuelva falso en lugar de generar un error: en definitiva, que la clase sea válida es una precondición del refactoring *Add Method* por lo cual debería estar.

Como solución a la limitación anterior, cada uno de los métodos de `RefactoringData` que se encargan de crear los refactorings, primero chequean que los elementos sean válidos y si esto no se cumple los refactorings no se instancian. Además, se redefinió `checkPreconditions` en `RefactoringData` de manera que si el refactoring no pudo ser construido retorne falso y en caso que se haya creado se devuelve el resultado de `checkPreconditions` del refactoring. De esta manera, las precondiciones se chequean directamente sobre el `RefactoringData` y no sobre el objeto `RBRefactoring`.

Analizar las precondiciones de un refactoring sin que ocurra un error de ejecución, no siempre proporciona el resultado esperado. Continuando con el ejemplo y asumiendo que *R1* es válido, si se evalúan las precondiciones de *R2* el resultado será falso porque la clase `RecommendationStrategy` no está definida. Si bien este resultado es correcto, puede observarse que el refactoring es válido debido a que anteriormente a este hay otro denominado *R1* cuya aplicación valida la precondición que al momento del análisis no se cumple, de manera que, si las precondiciones de *R2* se chequean después de ejecutar *R1*, el resultado será que *R2* es válido.

Considerando que hay refactorings que necesitan de la ejecución de otros para aplicarse, no es posible determinar la validez de un conjunto de refactorings analizando sus precondiciones sin ejecutar ninguno de ellos. Como los refactorings importados en una imagen conservan el orden en el que se realizaron, una forma de validar un refactoring sería chequear sus precondiciones luego de ejecutar todos los refactorings anteriores. El problema que tiene esta alternativa es que para validar la colección completa de refactorings importada es necesario ejecutarlos todos, y como el usuario debe poder elegir cuáles de ellos aplicar, luego de realizar la validación habría que deshacer todos los refactorings para que posteriormente el usuario haga la selección.

En lugar de tener que ejecutar los refactorings para validarlos, una mejor opción es simular de alguna manera esta ejecución. De esta forma no sería necesario tener que deshacer la colección de refactorings cada vez que se realiza la validación.

5.4 Simulación de refactorings

Gracias a la arquitectura del Refactoring Browser el posible simular la ejecución de un refactoring sin tener que realizar ninguna modificación. En el Capítulo 3 se describió que la aplicación de los refactorings tiene dos partes: en la primera se crean los cambios subyacentes al refactoring y en la segunda estos cambios son persistidos en la imagen. Sabiendo que en la primera etapa no solo se instancian los cambios, sino que además se simula la aplicación de los refactorings, se podrían validar realizando únicamente la primera parte de la ejecución.

La simulación se realiza en una representación del sistema denominada **modelo.** Todo refactoring debe ser creado en el contexto de un modelo, que puede ser provisto junto con los demás datos del refactoring o puede ser generado directamente por el Refactoring Browser. Cuando el refactoring es instanciado, su modelo se encarga de crear una representación de cada elemento de programa que necesita para ejecutarse.

La Figura 5.2 muestra el estado del modelo luego de instanciar el refactoring *R1*. Este refactoring agrega `RecommendationStrategy` como subclase de `Object`, por lo que el único elemento que necesita además del nombre de la nueva clase, es la clase `Object`.

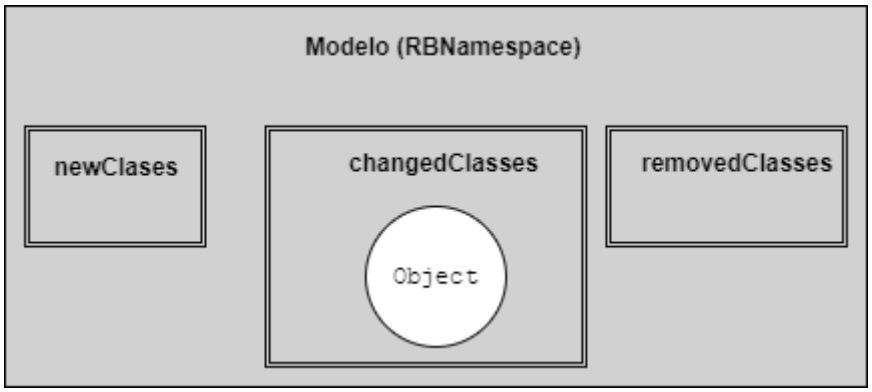

Figura 5.2. Creación del refactoring *R1*.

El modelo primero chequea si ya tiene creado un objeto que representa la clase `Object`, en ese caso el refactoring utiliza mismo objeto. Si es la primera vez que necesita la clase `Object`, la busca en la imagen y con la información obtenida crea un objeto que la simboliza. El objeto instanciado es conservado en una colección denominada `changedClasses` para no crearlo nuevamente si se vuelve a necesitar la clase `Object`.

El siguiente paso es realizar la primera etapa de ejecución del refactoring, para eso se le envía el mensaje `primitiveExecute` a *R1*. En este punto se evalúan las precondiciones; asumiendo que estas se cumplen, la Figura 5.3 muestra el estado del modelo luego de la ejecución. Para

simular que se agregó la clase `RecommendationStrategy`, el modelo crea un objeto que la representa y lo guarda en la colección `newClasses`.

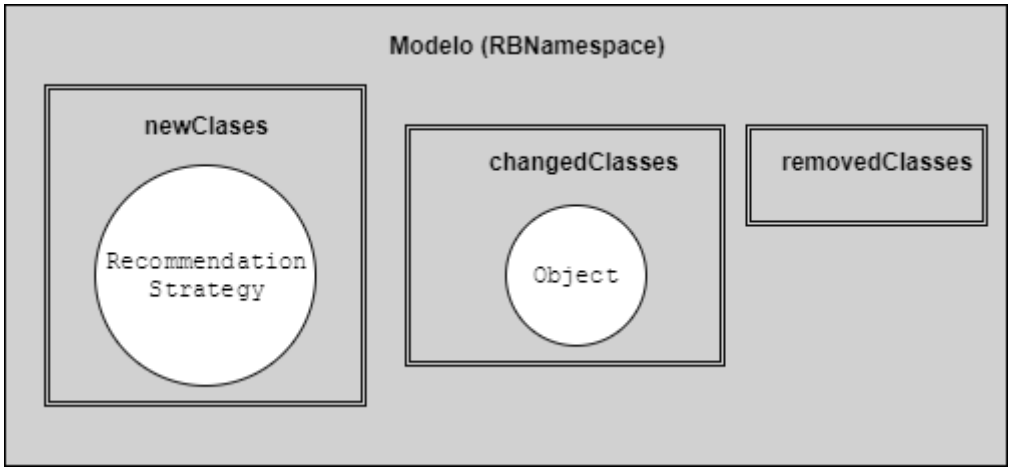

Figura 5.3. Simulación del refactoring *R1*.

Analizando la imagen de trabajo se podrá ver que no existe una clase llamada `RecommendationStrategy` porque se ejecutó solo la primera parte del refactoring que la agrega (resta ejecutar la segunda que persiste los cambios), sin embargo, el modelo registró la nueva clase. Si el refactoring *R2* se instancia en el mismo modelo que *R1*, la precondición que indica que debe existir una clase denominada `RecommendationStrategy` será válida a pesar de que la clase no esté definida en el sistema, ya que las precondiciones del refactoring analizan su modelo y este modelo tiene la clase incorporada.

El punto clave por el cual funciona la validación de *R2* sin haber ejecutado realmente *R1* es que los refactorings siempre trabajan en el contexto del modelo en el cual se instancian, es decir nunca acceden a objetos que estén fuera del alcance de éste. Si un refactoring necesita algún elemento de programa que el modelo no posee, el modelo lo busca en el sistema y crea una representación para éste. Por otro lado, si el modelo ya tiene una representación para un elemento como ocurre en *R2* con la clase `RecommendationStrategy`, el refactoring toma ese elemento como válido y no chequea su existencia en el sistema. De esta forma el modelo funciona como una arena de simulación.

El requerimiento entonces para poder simular la ejecución de un conjunto de refactorings es que todos se instancien en el mismo modelo y que cada uno de ellos sea creado posteriormente a la primera etapa de la ejecución del anterior. De esta manera, como *R2* necesita de la ejecución de *R1*, cuando se instancia *R2* su modelo ya posee la clase `RecommendationStrategy` y al evaluar sus precondiciones se obtiene que el refactoring es válido. Si *R2* se instancia en un modelo distinto o sin simular la ejecución de *R*1, el modelo buscará `RecommendationStrategy` en

el sistema y como no está definida, *R2* será determinado como inválido. El parámetro del mensaje `buildRefactoring:` de la clase `RefactoringData` presentado en el capítulo anterior es el modelo en el cual se debe crear el refactoring. Así, cuando se importa un archivo de refactorings se crea un nuevo modelo y cada refactoring es validado en éste.

La Figura 5.4 muestra el estado del modelo después de haber validado y simulado *R2*. El modelo registró que la clase `RecommendationStrategy` posee un nuevo método denominado `user:recommend:` agregando un nuevo objeto a la colección `newMethods` de la clase. Suponiendo que a los dos refactorings anteriores se le suma un tercero llamado *R3* que necesita el método `user:recommend:` para ejecutarse, si *R3* se instancia en este modelo también será válido ya que el modelo reconoce el método sin tener que evaluar el sistema real.

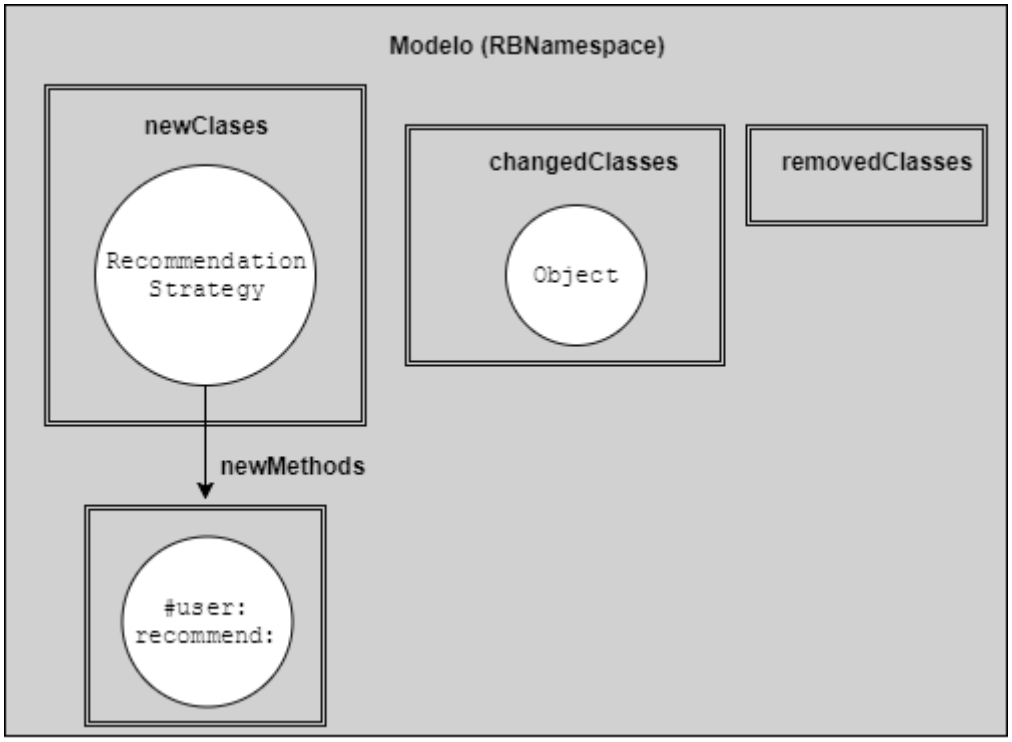

Figura 5.4. Simulación de *R2*.

Cada vez que se quiera repetir la validación de una colección de refactorings, nuevamente se deberá crear un modelo e instanciar estos refactorings en él ya que si se intenta volver a simular un refactoring en un modelo en el que ya fue ejecutado será determinado como inválido. Por ejemplo, si se chequean las precondiciones de *R1* en el modelo de la Figura 5.4 el resultado será que `RecommendationStrategy` no es un nombre válido para una clase ya que existe definida una con ese nombre.

La gran ventaja que tiene esta forma de validar los refactorings es que su ejecución es muy rápida porque no se realiza ningún cambio en la imagen, lo cual hace que el tiempo requerido para

realizar la validación no varíe demasiado conforme aumenta la cantidad de refactorings a chequear. Si habría que aplicar completamente los refactorings, cuanto más refactorings haya para validar, el tiempo de ejecución de la validación incrementaría considerablemente teniendo en cuenta que hay algunos refactorings que implican grandes modificaciones en el código, y además a este tiempo habría que agregarle el tiempo requerido para deshacerlos una vez que termina la validación.

Considerando que mediante la simulación es posible chequear una gran cantidad de refactorings sin que eso degrade la performance del sistema, se podría repetir la validación tantas veces como sea necesario. Que se pueda repetir es una característica esencial del chequeo teniendo en cuenta que después de hacer la validación de refactorings es probable que haya algunos de ellos que sean inválidos y que el usuario quiera re-ejecutarlos, pero aplicarlos deben ser válidos. Por lo cual el usuario debería poder realizar los cambios necesarios en la imagen para cumplir aquellas precondiciones que fallaron y luego hacer el chequeo nuevamente para actualizar el estado de cada refactoring.

El caso anterior podría darse en el ejemplo planteado en la introducción del capítulo, en el cual un componente se actualiza agregándole una clase para soportar nueva funcionalidad. Un usuario del componente podría tener definida en su imagen una clase con el mismo nombre por lo que cuando se chequee el refactoring de la actualización resultará inválido. El usuario, sabiendo que la actualización agrega una funcionalidad importante, podría renombrar su clase para que el refactoring sea válido y así poder re-ejecutarlo.

5.5 Validación en la herramienta

Con el mecanismo de simulación presentado anteriormente, la herramienta desarrollada valida cada uno de los refactorings ni bien se importa un archivo. Luego de realizar la validación, se muestra el resultado de cada refactoring y por cada uno de ellos que resultó inválido se muestra la precondición que no se cumplió. Sabiendo cuáles son los refactorings válidos, únicamente ellos pueden ser re-ejecutados, de esta manera se garantiza que la aplicación de todos ellos es exitosa.

Una vez que finaliza la validación de refactorings, se da la posibilidad de realizarla nuevamente con el objetivo que el usuario, a partir de los errores de la validación, pueda revertir el resultado de aquellos refactorings inválidos que le interesan re-ejecutar. Cuando el usuario realiza la validación, se actualiza el estado de cada refactoring y por ende también la lista de refactorings que se pueden reproducir. Este proceso puede repetirse la cantidad de veces que el usuario desee.

En algún momento el usuario decidirá no repetir la validación y continuar hacia la re-ejecución de los refactorings. Cuando eso ocurra se mostrarán los refactorings válidos para que el usuario

seleccione cuáles quiere re-ejecutar. Si bien solo se pueden re-ejecutar refactorings que fueron previamente chequeados, todavía esta re-ejecución puede fallar.

En el ejemplo planteado tanto *R1* como *R2* resultaron ser válidos, de manera que el usuario puede re-ejecutar ambos. Como se sabe que *R2* necesita que primero se ejecute *R1*, se puede notar que si selecciona solamente *R2* su ejecución fallará, lo cual no debería ocurrir ya que deja la secuencia de refactorings en un estado inconsistente. Si el usuario elige realizar *R2* de alguna forma habría que hacer que también se aplique *R1* para que *R2* se ejecute sin problemas. Hacer que se re-ejecute *R1* si se elige *R2* es bastante simple, el problema está en cómo determinar que *R2* necesita de la ejecución de *R1*. En el siguiente capítulo se explicará el mecanismo usado para resolver esta cuestión.

5.6 Resumen

En el capítulo anterior se planteó una forma de exportar refactorings a otras imágenes en la que se asume que todas las imágenes poseen la misma versión de un componente de software por lo que los refactorings pueden aplicarse correctamente a cualquiera de ellas. En este capítulo se analizó que no siempre es así ya que los desarrolladores en sus imágenes de trabajo toman decisiones de diseño que pueden invalidar refactorings que fueron aplicados en otra imagen con el mismo componente. Para poder determinar cuáles son los refactorings inválidos se decidió implementar un mecanismo que permita hacer un chequeo sin la necesidad de tener que ejecutarlos, para ello es necesario simular de algún modo los cambios que implican. La ventaja adicional que provee este mecanismo, además de darle la información al desarrollador sobre los refactorings válidos e inválidos, es que la ejecución de los refactorings no va a detenerse ante el primer refactoring inválido, sino que podrán ejecutarse refactorings posteriores que no dependan de refactorings inválidos anteriores en la secuencia.

Si bien con el método de validación planteado se descartan aquellos refactorings inválidos, al finalizar el capítulo se mostró con un ejemplo que igualmente la re-ejecución de refactorings puede generar errores debido a que hay refactorings que son válidos siempre y cuando se ejecuten otros anteriormente, por lo cual, si se decide ejecutar un determinado refactoring, obligatoriamente habría que ejecutar también aquellos que éste necesita.

Por último, también fue necesario agregar algunas precondiciones que el Refactoring Browser asume, las cuales se agregaron de forma transparente en la clase `RefactoringData`.

6 Dependencias entre refactorings

6.1 Introducción

Los refactorings conocidos son muy simples en el sentido que cada uno de ellos individualmente no realiza grandes modificaciones en el código. Generalmente para lograr una mejora de diseño importante es necesario realizar una secuencia de refactorings. En la herramienta propuesta, al usar el refactoring como una operación para actualizar un componente de software a una nueva versión, es muy probable que cada actualización consista de una secuencia de refactorings teniendo en cuenta que las nuevas versiones de un software suelen incluir diversos cambios. Dentro de una cadena de refactorings, es posible que la ejecución de un refactoring habilite las precondiciones de otro, en ese caso se dice que el segundo refactoring *depende* del primero, es decir, necesita que se ejecute antes.

Cuando se actualiza un componente usando esta herramienta, el desarrollador aplica los refactorings y estos se van guardando según el orden en que se realizaron. Como se realizan de a uno por vez, nunca podría ocurrir que la ejecución de un refactoring falle debido a que falta aplicar otro primero. Por lo tanto, las dependencias que puedan existir, no presentan ningún inconveniente cuando los refactorings se crean en la imagen origen. Las dependencias pueden generar conflictos cuando estos refactorings son recreados en otras imágenes.

Al momento de re-ejecutar una secuencia de refactorings luego de realizar la validación, las dependencias pueden hacer que la ejecución falle si se ejecuta solo un subconjunto de ellos. Si se intenta aplicar un refactoring *B* que depende de *A*, la ejecución de *B* fallará si antes no se realizó *A*. Para que no se de esta situación el camino más fácil es que obligatoriamente se hagan todos los refactorings; continuando con el ejemplo, primero se aplicaría *A* y luego *B* se ejecutaría sin problemas. De esta manera la re-ejecución nunca podrá fallar dado que los refactorings ya fueron validados y todos se aplican siguiendo el orden en el que se realizaron originalmente.

A pesar de esta facilidad, la prioridad de este trabajo es otorgarle una herramienta flexible y usable al desarrollador. Como se introdujo en el Capítulo 4, la posibilidad de seleccionar qué refactorings re-ejecutar le otorga una gran flexibilidad al usuario ya que puede observar cada uno de ellos y en función de los cambios que realizan y de sus necesidades, puede descartar aquellos que no les sean útiles. Debido a este requerimiento donde no necesariamente se aplicarán todos los refactorings de una actualización, habrá que analizar e identificar las posibles dependencias entre ellos, de modo que por cada refactoring seleccionado, previamente se ejecuten aquellos de los cuales depende y así asegurar que todos se realicen correctamente sin posibilidad de fallar. El análisis de las dependencias plantea dos problemas:

- ✓ Establecer cuándo un refactoring depende de otro. Para cualquier par de refactorings, se deberían analizar las condiciones que deben darse para que exista una dependencia.
- ✓ A partir de la arquitectura de base, encontrar un método que permita identificar las dependencias automáticamente.

6.2 ¿Cuándo existe una dependencia?

Para identificar las dependencias entre los refactorings, lo primero que se debe hacer es determinar cuándo pueden existir estas dependencias. Anteriormente se definió que un refactoring *R2 depende* de *R1* si la aplicación de *R1* hace que se cumplan las precondiciones de *R2*, por lo tanto, para saber si un par de refactorings cualesquiera pueden depender entre sí se debe analizar si la ejecución de uno puede modificar el estado de las precondiciones del otro.

Los dos refactorings del capítulo anterior sirven como ejemplo de una dependencia sencilla. Sean *R1* y *R2* los siguientes refactorings:

R1 = *addClass* (`RecommendationStrategy`, `Object`).

R2 = *addMethod* (`RecommendationStrategy`,`#user:anUser recommend:anItem`).

R2 depende de *R1* porque una de sus precondiciones es que debe existir una clase denominada `RecommendationStrategy` y esta clase es definida por el refactoring *R1*. La dependencia no se da en el sentido contrario porque el valor de las precondiciones de *R1* no puede ser alterado por la ejecución de *R2*.

Que un refactoring que agrega un método pueda depender de otro que define una nueva clase, no significa que siempre ocurra. Sean *R3* y *R4* los refactorings:

R3 = *addClass* (`MockClass`, `Object`).

R4 = *addMethod* (`Monolith, #foo`).

R4 para poder aplicarse correctamente necesita que esté definida la clase `Monolith`, y la ejecución de *R3* no cambiará el valor de esta precondición, por lo que se puede afirmar que *R4 no* depende de *R3*. Lo mismo ocurre al revés; la aplicación de *R4* no afecta las precondiciones de *R3*, de forma que *R3* no depende de *R4*. Como no existe una dependencia entre estos dos refactorings, se dice que son *independientes* y esto significa que pueden *conmutar*, es decir, se pueden realizar

en cualquier orden. Cuando hay una dependencia entre refactorings, necesariamente la ejecución se tiene hacer siguiendo un orden determinado.

Analizando los cambios que producen los refactorings y sus precondiciones se puede determinar cuándo existe una dependencia entre un par de refactorings específicos, tal como se hizo en los ejemplos anteriores. Así como cada tipo de refactoring tiene sus precondiciones definidas, también sería bueno poder describir de algún modo las modificaciones que cada uno realiza para poder calcular en forma genérica cuando un tipo de refactoring depende de otro. El objetivo es poder establecer las condiciones que deben darse por ejemplo para que un refactoring *Add Method* dependa de otro *Add Class*.

Para describir los cambios que realiza un refactoring se utilizarán *postcondiciones*. Las postcondiciones de un refactoring se expresan de la misma forma que sus precondiciones y enumeran las propiedades que el programa cumplirá luego de aplicar el refactoring. Cada refactoring está conformado por una serie de precondiciones que determinan su validez y un conjunto de transformaciones de código. Si se extiende su definición añadiéndoles sus postcondiciones, se pueden determinar fácilmente sus dependencias. El concepto de postcondición en el contexto de refactoring fue definido por primera vez por Donald Roberts en su tesis doctoral [Roberts99], aunque el concepto permaneció mayormente en la teoría de refactoring y no se utiliza en la práctica en las herramientas. En el contexto de este trabajo, el concepto pudo ser utilizado para calcular las dependencias entre refactorings.

A modo de ejemplo, se presentarán las precondiciones y postcondiciones de algunos refactorings y cómo es posible a partir de estas saber cuándo existe una dependencia entre estos refactorings. Para definir las precondiciones y postcondiciones, se usarán las funciones que se listan a continuación que permiten analizar las propiedades de un programa:

IsClass (*class*): verdadero si existe una clase llamada *class* en el sistema, de lo contrario es falso.

DefinesSelector (*class, selector*): verdadero si *class* define un método denominado *selector*.

DefinesInstanceVariable (*class*, *varName*): verdadero si *class* define la variable de instancia *varName*.

UnderstandsSelector (*class*, *selector*): verdadero si una instancia de *class* responde al mensaje *selector*.

Superclass (*class*): representa la superclase de *class*.

Subclasses (*class*): describe al conjunto de todas las subclases inmediatas de *class*.

Con las funciones anteriores, sean los refactorings *Add Class* y *Add Method* definidos de la siguiente manera:

Add Class = *addClass* (*class*$_1$, *superclass, subclasses*).

Add Method = *addMethod* (*class*$_2$, *methodName*).

La función *pre* (*Add Class*) denota las precondiciones de *Add Class* y se define como:

$$pre\ (Add\ Class) = isClass(superclass)\ \wedge\ \neg\ isClass\ (class_1)$$

$$\wedge\ \forall\ c\ \in subclasses: isClass(c)\ \wedge Superclass(c) = supeclass$$

Las postcondiciones de *Add Class* se describen mediante la función *pos* (*Add Class*):

$$pos\ (Add\ Class) = isClass(class_1)\ \wedge\ Superclass(class_1)\ = superclass$$

$$\wedge\ Subclasses(class_1) = subclasses$$

De la misma manera que se definieron para *Add Class*, a continuación, se definen las precondiciones y postcondiciones de *Add Method*.

$$pre\ (Add\ Method) = isClass(class_2)\ \wedge\ \neg\ definesSelector\ (class_2, methodName)$$

$$\wedge\ \neg\ understandsSelector\ (class_2, methodName)$$

$$pos\ (Add\ Method) = definesSelector\ (class_2, methodName)$$

$$\wedge\ understandsSelector\ (class_2, methodName)$$

Analizando los dos refactorings, se puede ver que la condición *isClass* (*class*) forma parte de las postcondiciones de *Add Class* y también de las precondiciones de *Add Method*. Si *class*$_1$ = *class*$_2$ significa que *Add Method* depende de *Add Class* ya que *isClass* (*class*$_1$) será válida luego de ejecutar *Add Class* y se necesita que esto ocurra para poder aplicar *Add Method*. Como las postcondiciones de un refactoring describen los cambios que este realiza, siempre serán falsas antes de su ejecución, por lo que la única forma de hacer que *isClass*(*class*$_1$) sea cierta es ejecutando primero *Add Class*. De esta manera, un refactoring *Add Method* dependerá de otro *Add Class* siempre y cuando *class*$_1$= *class*$_2$, en caso contrario serán independientes.

El ejemplo permite ver que se puede se determinar si existe una dependencia en un par de refactorings cualquiera simplemente comparando las postcondiciones de uno con las precondiciones del otro y si poseen alguna condición en común significa que dependen entre sí. Dos condiciones se consideran iguales cuando realizan el mismo análisis y a su vez sus argumentos también son equivalentes. Con el agregado de las postcondiciones a la definición de un refactoring, el concepto de dependencia entre refactorings podría redefinirse de la siguiente forma:

> Sean *R1, R2* dos refactorings cualesquiera, *R2* **depende** de *R1* si alguna de las precondiciones de *R2* está presente en las postcondiciones de *R1*.

Con esta forma de encontrar las dependencias, a partir de la definición de *Add Class* proporcionada antes también se puede notar que dos refactorings de este tipo pueden depender entre sí debido a que la condición *isClass*(*class*) está presente tanto en las precondiciones como en las postcondiciones. Dados los siguientes refactorings:

$R1 = addClass\ (class_1, superclass_1, subclasses_1)$.

$R2 = addClass\ (class_2, superclass_2, subclasses_2)$.

Si $superclass_2 = class_1$ entonces *R2* depende de *R1* ya que las postcondiciones de *R1* y las precondiciones de *R2* tienen en común la condición *isClass* ($class_1$). Si se analiza esta dependencia con algún ejemplo puntual se verá que es una dependencia lógica ya que para incorporar una clase al sistema se necesita una superclase y en este caso esa superclase la define el refactoring anterior.

Así como se estudió cuando puede darse una dependencia entre los refactorings *Add Class* y *Add Method*, se podría repetir el mismo análisis para todos los pares de refactorings posibles y así encontrar todas las dependencias que existen. Sin embargo, la ventaja que tiene utilizar este mecanismo para analizar las dependencias es que no sería necesario analizar cada caso en particular ya que las dependencias se derivan automáticamente a partir de las precondiciones y postcondiciones de cada refactoring. La clave para que la detección de las dependencias sea confiable está entonces en que cada refactoring tenga una definición completa de sus precondiciones y postcondiciones.

6.3 Detección de dependencias en Refactoring Browser

Al igual que ocurre con la simulación de refactorings, afortunadamente la arquitectura de Refactoring Browser da la posibilidad de implementar el mecanismo de detección de dependencias detallado anteriormente sin tener que realizar grandes modificaciones.

El primer paso para poder analizar las dependencias entre refactorings es poder extender cada uno de ellos para incluir sus postcondiciones, que deberían definirse de la misma manera que sus precondiciones. Cuando se describió el funcionamiento del Refactoring Browser en el Capítulo 3, se destacó que las precondiciones de los refactorings no están codificadas estáticamente en cada uno de ellos, sino que como muchas de ellas se repiten, se definen mediante objetos `RBCondition` que pueden ser compuestos para formar condiciones más complejas. Por lo tanto, esta clase puede ser reutilizada para agregarle a cada refactoring sus postcondiciones.

La segunda cuestión a realizar es completar las precondiciones de algunos refactorings. Cuando se analizó la validación de los refactorings en el Capítulo 5, se mostró que por ejemplo el refactoring *Add Method*, asume que la clase en la que se agregará el método es válida y no la analiza. Al momento de encontrar las dependencias, es necesario que esta precondición esté presente ya que de lo contrario algunas de ellas no serán detectadas.

Por último, con la definición completa de las precondiciones y postcondiciones, se necesita poder comparar las condiciones para poder saber si existe alguna condición en común entre las postcondiciones de un refactoring y las precondiciones de otro. Los objetos `RBCondition` poseen un atributo `type` que resume toda la información de la condición que representan y por lo tanto pueden ser fácilmente comparadas.

6.3.1 Definición de postcondiciones

Así como cada uno de los refactorings posee el método `preconditions` que tiene la definición de sus precondiciones, se agregó un método `postconditions` que describe sus postcondiciones. Las postcondiciones se definen mediante la composición de objetos `RBCondition`, de la misma forma que las precondiciones. A continuación, se muestra el método `postconditions` del refactoring *Rename Method*:

```
postconditions
    ^ (RBCondition definesSelector: newSelector in: class) &
    (RBCondition definesSelector: oldSelector in: class) not
```

El refactoring *Rename Method* renombra el método `oldSelector` definido en `class` a `newSelector`. Las postcondiciones indican que después de ejecutar el refactoring, existirá un método llamado `newSelector` en la clase `class` y que no tendrá definido un método con el nombre `oldSelector`.

6.3.2 Extensión de las precondiciones

La mayoría de los refactorings que requieren una clase para aplicarse no chequean que la misma sea válida. Esta limitación ya fue resuelta en la etapa de validación, por lo cual una vez que se creó el refactoring no es necesario validar nuevamente la clase. Sin embargo, como la detección de las dependencias se hace comparando precondiciones y postcondiciones, y hay muchas dependencias que basan en la validez de una clase (como ocurre en los refactorings *Add Class* y *Add Method*), es necesario que exista entre las precondiciones una condición que describa que la clase es requerida para poder identificar estas dependencias.

Como las precondiciones que se deben agregar no tendría sentido chequearlas en el contexto original de uso del Refactoring Browser (dado que siempre son válidas), para no modificar el método `preconditions` del Refactoring Browser, se definió un método denominado `implicitPreconditions` para representar aquellas condiciones que tienen que estar presentes al momento del análisis de las dependencias. De esta forma, el método `implicitPreconditions` de *Add Method* queda definido de la siguiente forma:

```
implicitPreconditions
    ^ (RBCondition existsClassNamed: class name in: self model)
```

La condición `existsClassNamed:in:` fue definida para representar la función *isClass(className)* ya que el Refactoring Browser no tiene implementada una condición que haga ese análisis.

Además de las precondiciones implícitas, hay otras precondiciones que no están definidas por una cuestión de diseño del Refactoring Browser, que es necesario que estén. En el capítulo 4 se mencionó que existen refactorings que solicitan datos al usuario durante su ejecución. El ejemplo que se dio fue el refactoring *Extract Method* que necesita el nombre del método a crear. Una de las precondiciones que tiene este refactoring es que en la clase que realice, no puede existir un método con el mismo nombre que el método nuevo. Sin embargo, como este dato es pedido durante la ejecución en vez de pedirse en la creación del refactoring, no existe una precondición que haga este análisis.

Para poder re-ejecutar este tipo de refactorings usando los datos de la ejecución original, se creó otra versión de los mismos que no requiere la intervención del usuario dado que dispone de todos los datos necesarios al momento de su instanciación. Como esta versión ya tiene la definición completa del refactoring, se le deben agregar aquellas precondiciones que le faltan a la versión original. Siguiendo con el ejemplo del *Extract Method*, su versión re-ejecutable ya conoce el nombre del método que se creará, por lo tanto, además de las precondiciones que define la versión original, también tiene que chequear que no exista un método con ese nombre. Esta condición no solo es necesaria para la detección de posibles dependencias, sino que también tiene que estar para poder validar el refactoring.

6.3.3 Comparación de las condiciones

El método para encontrar las dependencias entre refactorings se basa en la comparación de las condiciones. No solamente se tiene comparar el tipo de análisis que hacen dos condiciones, sino que también se tienen que comparar los argumentos de las mismas, de modo que dos condiciones sean *equivalentes* cuando realizan la misma función con los mismos argumentos. Para poder establecer esta comparación, las condiciones atómicas tienen un atributo *type* que contiene un arreglo con toda la información de la condición que incluye el chequeo que realiza y el valor de sus argumentos.

La Figura 6.1 muestra el valor de *type* de siguiente condición:

```
RBCondition definesSelector: #foo in: Monolith.
```

#definesSelector	Monolith	#foo

Figura 6.1. Ejemplo de *type*.

Mediante el mensaje = en Smalltalk se puede saber rápidamente si dos arreglos son equivalentes o no. Por lo tanto, comparando el *type* de dos condiciones con el método = se determina si son exactamente iguales o no.

Si bien el *type* de una condición facilita la comparación, este atributo permite comparar dos condiciones atómicas, es decir, instancias de `RBCondition`. La mayoría de los refactorings poseen varias precondiciones y postcondiciones, por lo tanto, lo más probable es que estén definidas mediante condiciones compuestas (`RBConjunctiveCondition`). Entonces, para poder identificar una dependencia entre dos refactorings, es necesario desarrollar un algoritmo que, dadas dos condiciones, calcule si comparten al menos una condición atómica sin importar si las condiciones son atómicas o son compuestas.

El algoritmo se implementó a través del método `containsAny:aCondition` definido en la clase abstracta de todas las condiciones (`RBAbstractCondition`). A grandes rasgos, el método analiza la condición receptora del mensaje y la condición enviada como argumento: si ambas condiciones son atómicas directamente retorna el resultado de la comparación de sus atributos *type*, mientras que, si alguna de ellas es compuesta, se obtienen sus partes y se vuelve a enviar el mensaje `containsAny:` a cada una de ellas. El resultado final del método es un booleano que indica si las condiciones poseen alguna condición elemental en común. Para variar su comportamiento dependiendo el tipo de la condición, el método es redefinido en cada una de las subclases de `RBAbstractCondition`.

Habiendo definido la comparación entre dos condiciones, es muy simple desarrollar un método que para un par de refactorings cualesquiera, calcule si hay una dependencia entre ambos. El método implementado se denomina `dependsOn: aRefactoring` y retorna un booleano que establece si el refactoring receptor depende o no del refactoring `aRefactoring`. El código se muestra a continuación:

```
dependsOn: aRefactoring
    ^ self preconditions &
    self implicitPreconditions containsAny: aRefactoring postconditions
```

El método está definido en la clase `RBRefactoring`, superclase de todos los refactorings, lo que permite analizar las dependencias de cualquier refactoring. La ventaja de esta solución es que, si se quiere agregar un nuevo refactoring al sistema, es suficiente con definir sus precondiciones y postcondiciones para identificar las dependencias con los demás refactorings.

El punto clave que permitió implementar esta forma genérica de encontrar las dependencias es la posibilidad de comparar las condiciones y de poder analizar cada una de las partes de una condición compuesta. Si las precondiciones y postcondiciones estarían definidas de forma estática, sería mucho más difícil poder comparar cada una de ellas.

Para que `dependsOn:` proporcione el resultado correcto, es necesario que previamente se realice al menos la primera parte de la ejecución de los dos refactorings, tanto del receptor del mensaje como del argumento. Suponiendo que existen los refactorings *R1* y *R2*, si *R2* depende de *R1, R2* deberá ser instanciado después de aplicar o simular *R1* ya que, si se crea antes de esto, alguno de los argumentos que necesita todavía no estarán disponibles y esto hará que la comparación de sus precondiciones con las postcondiciones de *R1* no arroje el resultado esperado. Por lo tanto, no es posible instanciar dos refactorings y saber si hay una dependencia entre ellos antes de ejecutarlos.

6.4 Análisis de dependencias en la herramienta

Con el mecanismo desarrollado antes el objetivo es chequear las dependencias de los refactorings en el momento en el que el usuario elige cuáles refactorings quiere aplicar, para asegurarse que aquellos que selecciona se ejecuten correctamente. Este análisis se hace en una colección de refactorings que ya se sabe que son válidos y que cumple con las siguientes propiedades:

- ✓ Los refactorings están ordenados siguiendo el orden en el que se realizaron originalmente, por lo tanto, las dependencias de cada refactoring siempre estarán anteriormente a este en la secuencia.
- ✓ No existen dependencias circulares. En la imagen origen, el usuario realiza los refactorings de a uno por vez y estos se exportan respetando su orden, de manera que todos los refactorings de la secuencia en algún momento podrán ser ejecutados.

Para mostrar cómo trabaja la herramienta, se continuará con el ejemplo del capítulo anterior, en el cual se cuenta con la versión inicial de un framework de recomendación de objetos. El objetivo de la siguiente versión es que los usuarios puedan definir sus propios mecanismos de recomendación y que puedan elegir fácilmente cuál usar. Para esto, se decidió realizar los siguientes refactorings:

1. *addClass* (`RecommendationStrategy`, `Object`, []). Este refactoring crea la clase `RecommendationStrategy` como subclase de `Object`. [] indica que `RecommendationStrategy` no posee subclases.
2. *addClass* (`NumberOfFriends`, `RecommendationStrategy`, []): Incorpora la clase `NumberOfFriends` como subclase de `RecommendationStrategy`.
3. *addInstanceVariable* (`User`, *recommendationStrategy*): agrega una variable de instancia en la clase `User` denominada *recommendationStrategy*.
4. *moveMethod*(`User`,`recommend:`,*recommendationStrategy*,`user:recommend:`, `NumberOfFriends`): mueve el método `recommend:` definido en la clase `User` a un nuevo método denominado `user:recommend:` en `NumberOfFriends`. El método `recommend:` no se elimina, sino que su contenido se reemplaza por el envío del mensaje `user:recommend:` al objeto referenciado por la variable *recommendationStrategy*.

Suponiendo que todos los refactorings son válidos en una imagen determinada, el usuario puede ejecutar cualquiera de ellos. La Figura 6.2 muestra la secuencia de refactorings en la que se debe analizar las dependencias. Los refactorings marcados con gris están seleccionados para aplicarlos.

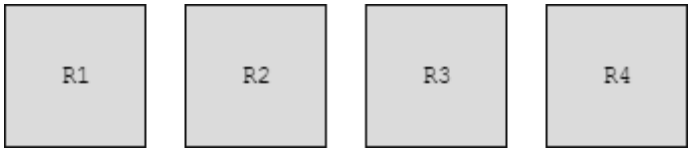

Figura 6.2. Refactorings ejecutables.

Si el usuario decide no realizar *R2*, tampoco se deben hacer aquellos que necesitan de la ejecución de éste, por lo que se deben encontrar todos los refactorings que dependan de *R2*. Como la secuencia está ordenada, estos refactorings tienen que estar después de *R2*, de manera que solo se analizan *R3* y *R4*. La Figura 6.3 muestra el estado de la secuencia después de quitar a *R2* de la lista de refactorings a ejecutar.

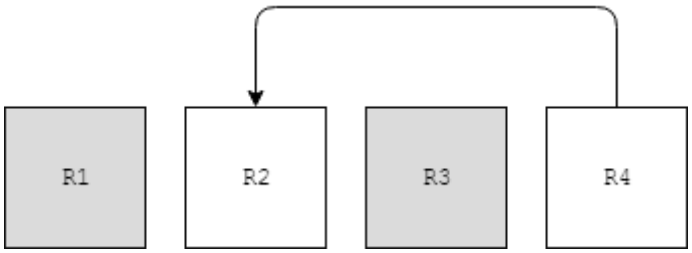

Figura 6.3. Estado de la secuencia luego de excluir *R2*.

La flecha de *R4* a *R2* que se ve en la imagen indica que existe una dependencia entre ambos, por lo tanto, al quitar *R2* también se quita *R4*. Al no realizar *R4*, debería repetirse el proceso anterior hecho para *R2*, sin embargo, al no haber ningún refactoring después de *R4*, el estado de la secuencia no se modifica.

Siempre que se excluye un refactoring, los refactorings que dependan de este se buscan en la lista de los seleccionados. Al sacar de la lista a *R1*, el único refactoring queda por analizar es *R3*. Si bien *R2* depende de *R1,* como *R2* ya fue quitado antes no se analiza; lo mismo ocurre con *R4. R3* no depende de *R1* por lo tanto el usuario puede no ejecutar *R1* y si aplicar *R3.*

Anteriormente se describió como es el proceso de descartar los refactorings que no son de interés. Para ver cómo funciona la selección de refactorings, se asumirá que todos los refactorings anteriores inicialmente están excluidos. La Figura 6.4 muestra el estado inicial de la secuencia.

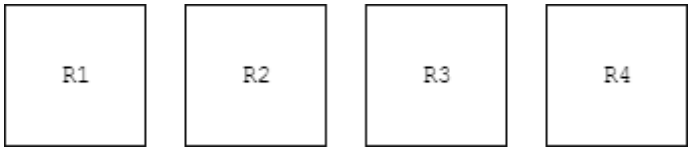

Figura 6.4. Refactorings no seleccionados.

Suponiendo que se selecciona *R4*, se deben buscar los refactorings de los cuales este dependa. Nuevamente como la secuencia está ordenada, estos refactorings tienen que estar antes de *R4*, no pueden estar después. Así como cuando se descartan refactorings se analizan únicamente los seleccionados, cuando se eligen las dependencias se buscan en la lista de refactorings excluidos. En este caso se analizan *R1*, *R2* y *R3*.

La Figura 6.5 muestra el estado de la secuencia luego de elegir el refactoring *R4*. *R4* depende de *R3* y de *R2*, por lo tanto, estos dos refactorings también tienen que ejecutarse. El proceso hecho para *R4* se repite para *R3* y *R2*, quedando solamente *R1* entre los refactorings excluidos; *R2* depende de *R1* y por eso *R1* también debe ejecutarse.

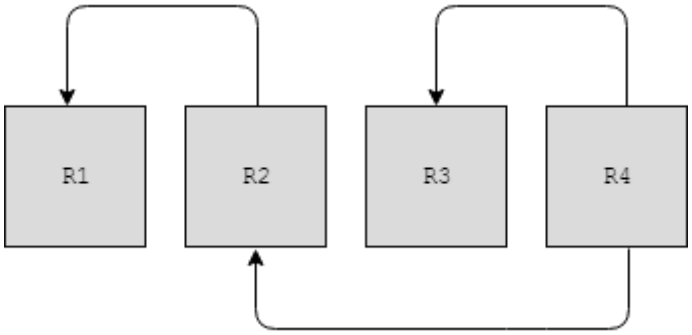

Figura 6.5. Estado de la secuencia luego de seleccionar *R4*.

Una vez que el usuario eligió todos los refactorings que le interesan, los puede aplicar sabiendo que no ocurrirá ningún error de ejecución. Considerando que los refactorings re-ejecutables ya fueron validados, mediante la identificación de las dependencias se asegura que cualquier conjunto de refactorings que se elija se ejecutará correctamente.

6.5 Resumen

Se presentó un mecanismo para identificar dependencias entre refactorings que se basa en extender la definición de un refactoring agregándole sus postcondiciones que describen los cambios que realiza mediante propiedades que el programa debe cumplir. Teniendo las postcondiciones de cada refactoring, dado cualquier par de refactorings, si las precondiciones de uno y las postcondiciones del otro comparten alguna condición en común significa que hay una dependencia entre ambos y por lo tanto no pueden ejecutarse en cualquier orden.

Tomando como base la arquitectura del Refactoring Browser, se analizó que es muy simple desarrollar la estrategia anterior ya que las precondiciones de un refactoring están implementadas mediante objetos que pueden reusarse para definir sus postcondiciones y así poder establecer la comparación.

Por último, pudiendo determinar si hay una dependencia entre dos refactorings del Refactoring Browser, se describió cómo se realiza el análisis de las dependencias en la herramienta creada cuando el usuario decide re-ejecutar o descartar un determinado refactoring: si decide realizarlo se deben encontrar y realizar también aquellos refactorings de los cuales depende, mientras que si lo descarta también se deben descartar todos los que dependen de éste.

7 Uso de la herramienta

En el siguiente capítulo se explicará el uso típico de la herramienta desarrollada. Para esto se instanciarán en una imagen los refactorings planteados en el capítulo anterior y luego serán reproducidos en otra imagen. Durante el proceso de grabado y re-ejecución, se irán mencionando cada una de las funcionalidades de la herramienta detalladas a lo largo del trabajo.

La herramienta es un plugin para Pharo que puede ser iniciada desde el Menú World de este. La Figura 7.1 muestra la ventana principal al abrir este plugin.

Figura 7.1. Ventana principal.

La vista principal ofrece dos opciones: por un lado, realizar una captura de refactorings (*Record Refactorings*) y por el otro aplicar refactorings contenidos en un archivo (*Replay Refactorings*). Asumiendo que se está en la imagen del desarrollador de un componente software, se elegirá la primera opción.

7.1 Grabado de refactorings

Al seleccionar la operación de capturar refactorings, se muestra una ventana con los controles necesarios para gestionar la captura. Esta ventana fue realizada lo más pequeña posible para no obstaculizar el trabajo del desarrollador. La Figura 7.2 muestra su diseño:

Figura 7.2. Record Refactorings.

Hasta que no se inicie una sesión de grabado con la opción *Start*, cualquier refactoring que se realice no será guardado, una vez que se empezó con la captura, se habilitan las demás opciones. La Figura 7.3 presenta la realización del refactoring *R2* del capítulo anterior.

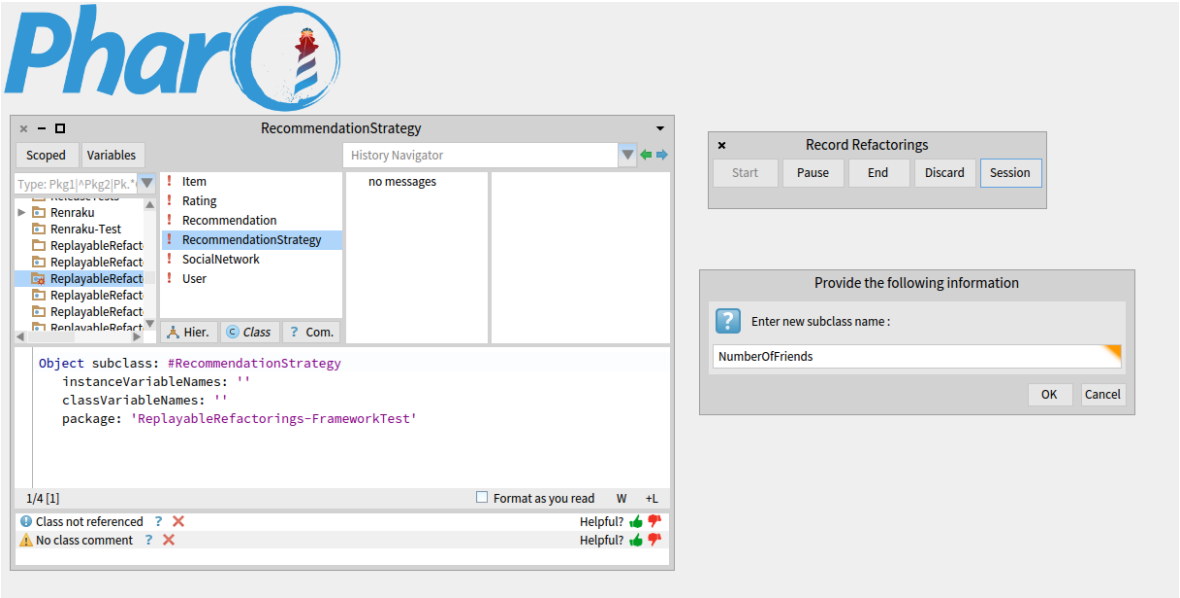

Figura 7.3. Realización de *R2*.

Session permite ver los refactorings capturados hasta el momento. El programador mediante esta opción puede corroborar que cada refactoring se guarda correctamente. La Figura 7.4 muestra el estado de la sesión luego de realizar *R2*.

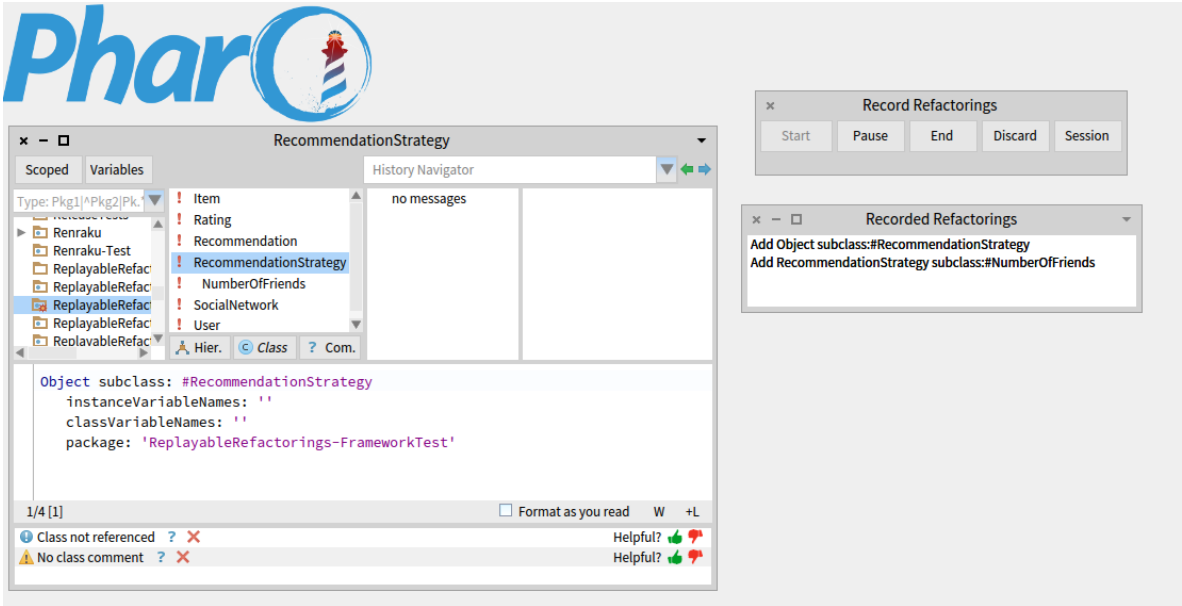

Figura 7.4. Refactorings capturados.

Antes de *R2* se ejecutó *R1*, por eso también aparece este refactoring en la sesión. *Pause* permite detener la captura de refactorings temporalmente para que el usuario pueda realizar modificaciones no quiere exportar. Si alguna refactoring fue hecho de forma errónea, la opción

Discard elimina todos los refactorings guardados. En caso de elegir esta opción, se deberán deshacer los refactorings hechos e iniciar una nueva sesión de captura.

De la misma manera que se aplicaron *R1* y *R2*, también se hacen *R3* y *R4*. *End* finaliza la sesión activa; al elegir esta opción, automáticamente se solicita el nombre del archivo al que se exportarán los refactorings guardados. La Figura 7.5 permite ver cómo se realiza la exportación a un archivo denominado *framework-v2*.

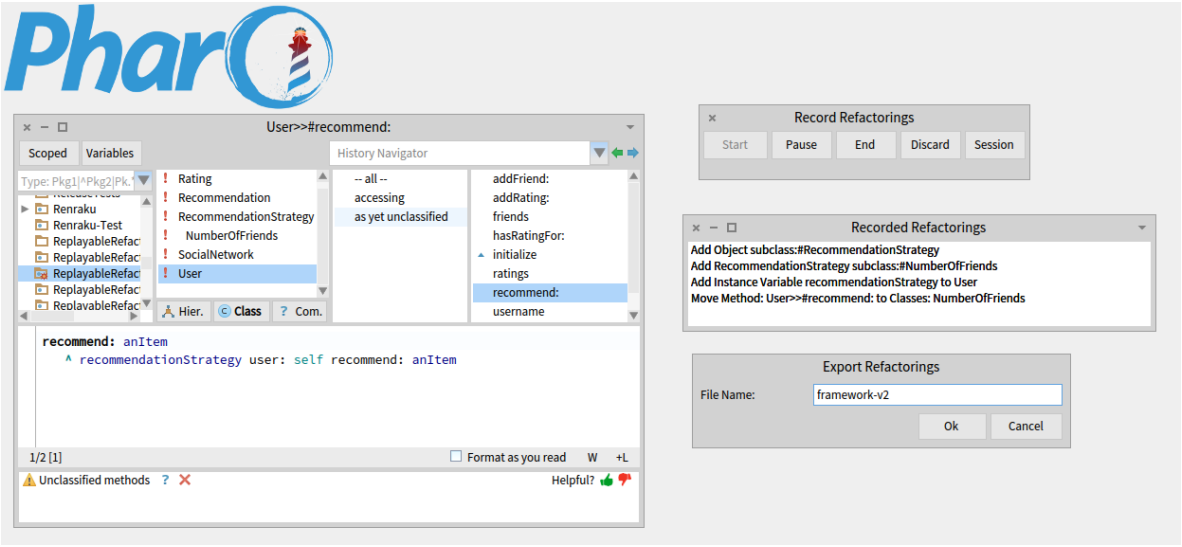

Figura 7.5. Exportación de refactorings.

7.2 Re-ejecución de refactorings

Para re-ejecutar los refactorings creados anteriormente, es necesario importar el archivo *framework-v2* en otra imagen que posea la versión inicial del framework de ejemplo. Al elegir la opción *Replay Refactorings* en la vista principal, se solicita el nombre del archivo que contiene los refactorings. La Figura 7.6 muestra cómo se realiza esta acción. Esta Figura también permite ver se está trabajando sobre la versión inicial del framework dado que el método `recommend:` de la clase `User` es diferente al obtenido en la Figura 7.5. Para que la importación sea exitosa, el archivo debe residir en el mismo directorio que la imagen.

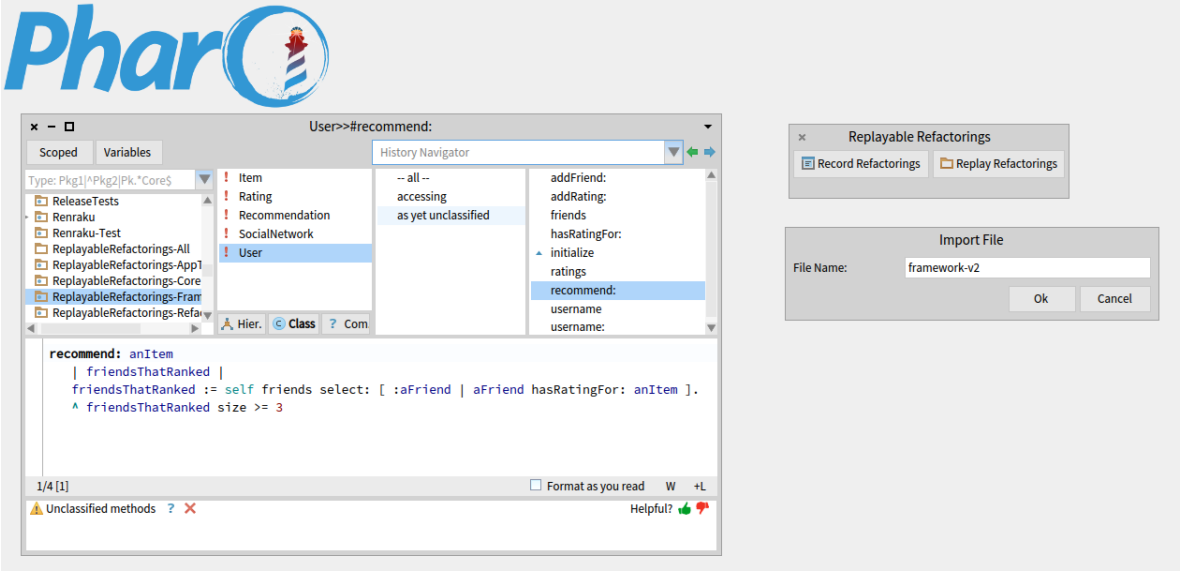

Figura 7.6. Importación de refactorings.

Luego de confirmar la importación, se muestra un listado con todos los refactorings contenidos en el archivo, indicando por cada uno de ellos si es válido en la imagen o no. Suponiendo que el usuario ya tiene definida en esa imagen una clase llamada `RecommendationStrategy`, la Figura 7.7 expone cómo se ven los refactorings.

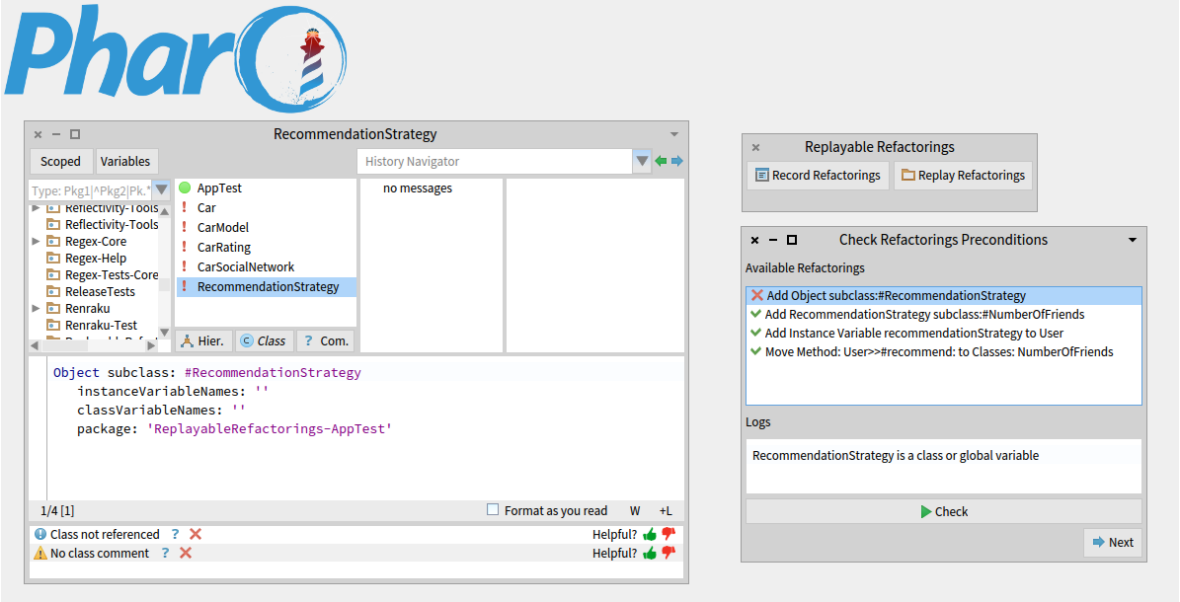

Figura 7.7. Refactorings disponibles.

Por cada refactoring inválido, la sección *Logs* presenta una descripción que explica por qué fallo la validación. Para el caso de *R1*, indica que ya existe en el sistema una clase con ese nombre. El usuario puede corregir el error cambiándole el nombre a `RecommendationStrategy`.

Posteriormente, mediante *Check* puede realizar la validación de los refactorings nuevamente. La Figura 7.8 muestra el resultado del chequeo.

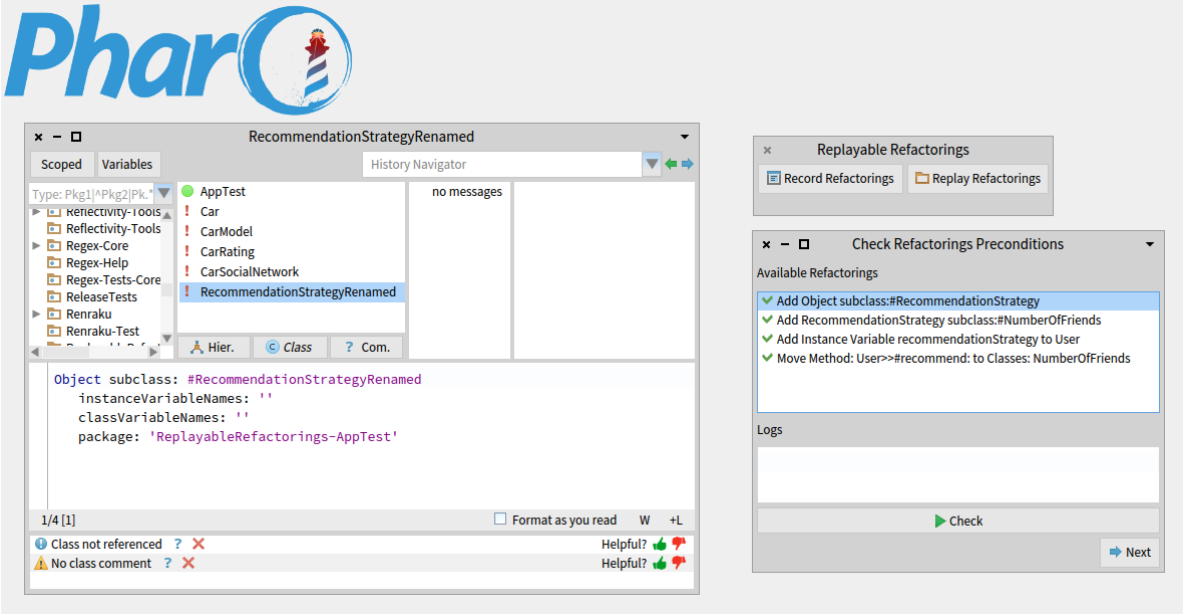

Figura 7.8. Corrección de un refactoring.

Los refactorings son todos válidos por lo tanto se procede a elegir cuales de ellos se aplicarán. Inicialmente todos están seleccionados, pero el usuario puede quitar aquellos que no sean de su interés. La Figura 7.9 presenta cómo se visualizan los refactorings elegidos y los descartados.

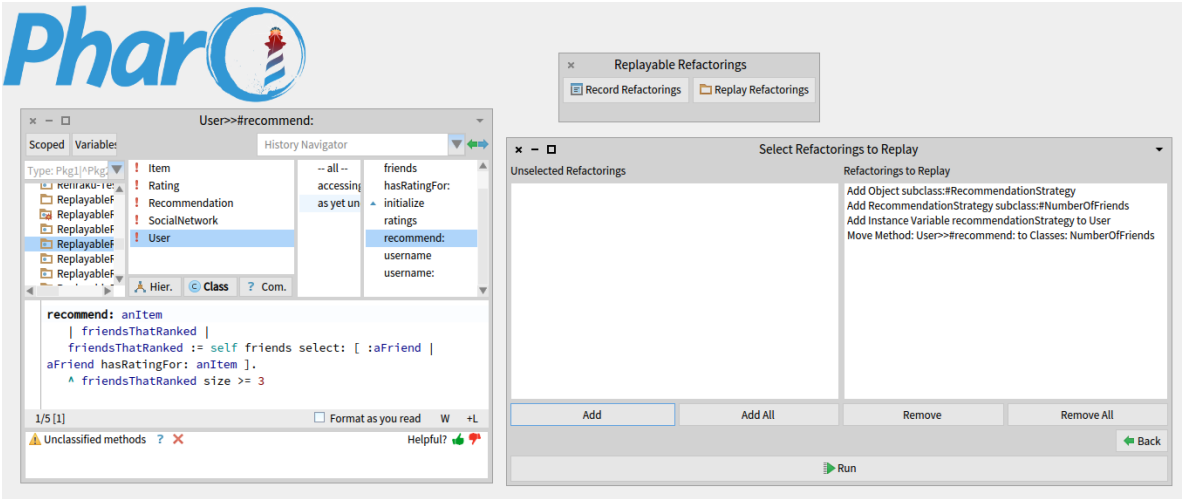

Figura 7.9. Selección de refactorings.

Si se descarta el primer refactoring, como todos los siguientes dependen de este directa o indirectamente, también son movidos a la lista *Unselected Refactorings*. Algo similar ocurre si todos los refactorings están descartados y se selecciona solamente el último; como este depende de todos los anteriores (aunque no sea directamente), todos son llevados a la lista *Selected*

Refactorings. En las dos listas siempre se conserva el orden de la ejecución original de cada refactoring, para poder analizar los refactorings anteriores o posteriores según sea necesario.

Suponiendo que se quieren ejecutar todos los refactorings, lo único que queda por hacer es elegir la opción *Run*. Esto hará que los refactorings se ejecuten de a uno por vez en forma ordenada y sin posibilidad de fallar. La Figura 7.10 muestra el código del método `recommend:` de la clase `User`, se puede ver que es exactamente el mismo que el de la Figura 7.5.

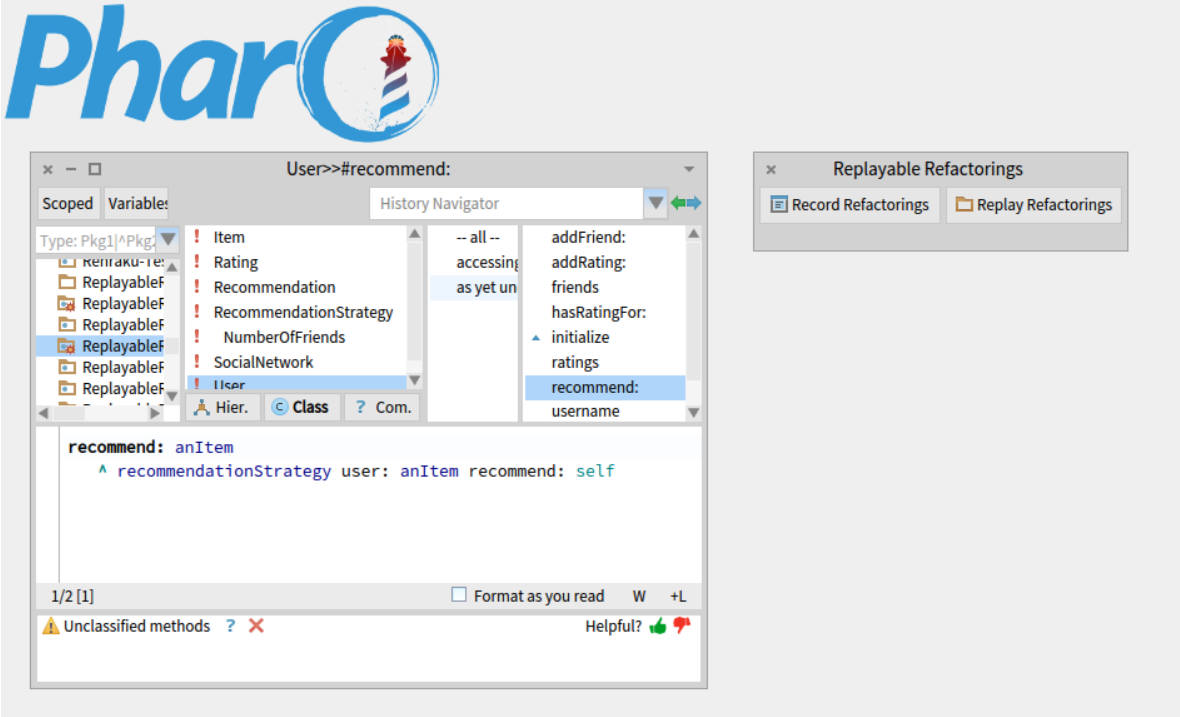

Figura 7.10. Resultado de la re-ejecución.

8 Conclusiones y trabajos futuros

El presente capítulo pretende dar un pequeño resumen de todo el trabajo, presentando la conclusión final, las contribuciones y las limitaciones encontradas. Por último, se definen los trabajos futuros que pueden derivarse.

8.1 Conclusiones

Cada vez es más común reusar componentes de software para acelerar el proceso de desarrollo de un sistema. Si bien esto reduce considerablemente el tiempo de trabajo, las aplicaciones dependen de los componentes que reutilizan, lo que significa que, si se realiza un cambio sobre estos componentes, las aplicaciones también deberán modificarse para continuar funcionando correctamente. Esta dependencia, hace que los desarrolladores de componentes cada vez que tengan que realizar una modificación en respuesta a cambios en los requerimientos, tengan que elegir entre dos opciones: implementar una solución eficiente asumiendo el riesgo de invalidar las aplicaciones clientes, o mantener la compatibilidad lo que implica una mayor complejidad en el diseño y por lo tanto un aumento en el costo de mantenimiento. Generalmente, se termina eligiendo el segundo camino.

Un caso de estudio realizado en [Dig06] muestra que gran parte de los cambios en un componente que causan que las aplicaciones clientes dejen de funcionar, se deben a refactorings. Es por eso que muchas veces los desarrolladores dejan de realizar refactorings que podrían facilitarle notablemente la tarea de desarrollo.

El objetivo de este trabajo fue desarrollar una herramienta para darle soporte a los programadores en el proceso de actualización de un componente, de forma que los desarrolladores de este puedan aplicar todos los cambios que consideren necesarios sin preocuparse por el impacto que tengan en las aplicaciones clientes, y que los programadores de estas puedan replicar esos cambios automáticamente.

Tomando como base la posibilidad de realizar refactorings automáticamente, la solución diseñada se basa en el grabado y exportación de refactorings: el programador de un componente hace todos los cambios requeridos por medio de refactorings y luego los exporta para que los usuarios de este componente puedan recrearlos en sus aplicaciones. El ambiente de trabajo elegido fue Pharo Smalltalk principalmente porque provee una herramienta para automatizar refactorings que puede ser fácilmente extensible y que además proporciona una definición casi completa del catálogo de refactorings propuesto en [Fowler99], lo que permite expresar cualquier modificación de código a través de un refactoring.

En la solución desarrollada, cada actualización de un componente consiste de una colección de refactorings. Los usuarios de los componentes participan en el proceso de actualización eligiendo cuáles de esos refactorings quieren reproducir. Dado que los refactorings describen los cambios que realizan, esto les permite a los usuarios descartar aquellas modificaciones que no les interesan. El hecho que el usuario pueda seleccionar qué refactorings aplicar exige analizar las dependencias entre los refactorings de la actualización, de manera que si se elige un refactoring también se ejecuten aquellos que este necesita. Por esta razón, como parte del objetivo principal del trabajo, también se planteó poder determinar cuándo existe una dependencia entre dos refactorings.

Para identificar las dependencias entre refactorings se encontró un método práctico que se basa en extender la definición de cada refactoring para incorporarle sus postcondiciones y así poder comparar los cambios que efectúa un refactoring con las condiciones que otro necesita para aplicarse correctamente. La gran ventaja que tiene este método es que podría ser implementado en cualquier framework de refactorings.

La herramienta permite capturar y re-ejecutar un subconjunto de refactorings del Refactoring Browser, e identificar sus posibles dependencias al momento de la re-ejecución. Los refactorings soportados son aquellos que se consideran los más utilizados:

- ✓ *Add Class.*
- ✓ *Rename Class.*
- ✓ *Remove Class*
- ✓ *Add Method*
- ✓ *Rename Method*
- ✓ *Remove Method.*
- ✓ *Add Parameter to Method*
- ✓ *Remove Parameter to Method*
- ✓ *Extract Method*
- ✓ *Move Method.*
- ✓ *Add Instance Variable.*

Fácilmente se podrían incorporar nuevos refactorings.

8.2 Contribuciones

8.2.1.1 Descripción detallada de la arquitectura del Refactoring Browser.

En el Capítulo 3 se describieron los componentes principales del Refactoring Browser. Se hizo hincapié en aquellos que necesitaron extenderse para poder implementar la captura y re-ejecución de refactorings y la identificación de las dependencias entre estos.

8.2.1.2 Diseño e implementación de una herramienta para guardar y re-ejecutar un subconjunto de los refactorings implementados en Pharo Smalltalk.

Se implementó la herramienta para guardar y re-ejecutar refactorings como una extensión del Refactoring Browser. En el Capítulo 4 se describieron las decisiones de diseño más importantes para capturar y re-ejecutar refactorings sin tener en cuenta posibles errores. El Capítulo 5 expone las razones por las cuales los refactorings grabados pueden ser inválidos en distintos entornos de desarrollo, y explica un método para validarlos. Por último, el Capítulo 6 describe cómo identificar las dependencias entre refactorings para asegurar que la re-ejecución no pueda fallar.

8.2.1.3 Análisis del concepto de dependencias entre refactorings introducido inicialmente por Don Roberts [Roberts99].

El concepto de dependencias entre refactorings es mencionado en varias oportunidades a lo largo del trabajo y finalmente se explica en forma detallada en el Capítulo 6. En este capítulo se describe también cómo determinar si existe una dependencia entre dos refactorings cualesquiera.

8.2.1.4 Incorporación de Extensiones a los refactorings soportados por la herramienta.

La primera extensión que se incorporó a los refactorings fue la posibilidad de exportarlos, en el Capítulo 4 se describió cómo funciona el método `serialize`. Las otras extensiones tienen que ver con la detección de dependencias: se agregaron precondiciones a algunos refactorings y se definieron las postcondiciones para todos. En el Capítulo 6 se mencionan estas extensiones.

Los refactorings implementados en Refactoring Browser no están preparados para fallar. Esto significa que hay validaciones que no se realizan y que frente a cualquier conflicto que no permita la ejecución de un refactoring, se levanta una excepción que se le muestra al usuario. Dado que en la solución diseñada es bastante común que un refactoring sea inválido, se necesitó extender algunas validaciones y mostrar los errores de una forma amigable para el usuario, es decir, ocultar las excepciones generadas y brindar una descripción general de estos errores.

8.2.1.5 Se añadió soporte para tratar de manera uniforme todos los refactorings, incluido el que permite agregar un nuevo método (Add Method).

El refactoring *Add Method* está definido en Refactoring Browser, pero no se utiliza: no existe una opción para que el usuario pueda agregar un nuevo método usando este refactoring. En la herramienta se agregó soporte a este refactoring para poder exportar los métodos que se incorporan. De esta manera, todos los refactorings se tratan de manera uniforme.

8.3 Limitaciones

La herramienta desarrollada no contempla todos los refactorings disponibles en el IDE, sólo un subconjunto. De todas maneras, se eligieron aquellos más utilizados de manera de poder demostrar la factibilidad de la herramienta.

Para comprobar la correctitud de los refactorings al momento de su re-ejecución en la imagen destino, se desarrollaron pruebas de unidad que comprueban, por un lado, la validez de los refactorings re-ejecutados por la clase `RefactoringData`, y, por otro lado, la validez de las dependencias que se calculan entre refactorings. A pesar de esto, no se llegaron a realizar pruebas de usuario para comprobar la usabilidad de la herramienta y sus interfaces gráficas.

Si bien la herramienta puede capturar todos los refactorings que se ejecutan a través del Refactoring Browser, hay refactorings que un programador puede realizar en el ambiente de Smalltalk de forma programática e implícita. Es el caso de *Add Class* que puede realizarse directamente escribiendo la definición de la nueva clase. Sería importante en el futuro capturar también estos refactorings implícitos.

8.4 Trabajos Futuros

- ✓ Introducir más refactorings en la herramienta. Si bien los que están soportados actualmente son los más usados, extender este conjunto haría que los desarrolladores puedan exportar mayor cantidad de modificaciones de código.
- ✓ Desarrollar pruebas de usuario para comprobar la usabilidad de la herramienta.
- ✓ Poder exportar los refactorings a distintos formatos de archivos. Haciendo la exportación a archivos de texto, los usuarios podrían ver cada uno de los refactorings de una actualización antes hacer la importación. De esta manera, si algún refactoring no puede ser re-ejecutado automáticamente, el usuario podría aplicarlo manualmente leyendo la información desde el archivo de actualización.

- ✓ Por cada refactoring de una actualización, poder ver cada una de las modificaciones que realiza en el código, para que el usuario tenga mayor información al momento de decidir si quiere aplicar ese refactoring.
- ✓ Si bien la herramienta fue diseñada para no obstruir el trabajo de los programadores, integrarla con el browser de clases de Pharo (System Browser), haría más fácil de proceso de actualización de un componente.
- ✓ Considerando que en Smalltalk todo el código es accesible, se podría extender la herramienta para contemplar la posibilidad de que los usuarios de los componentes hagan sus propios cambios sobre estos. De esta manera, el proceso de actualización de un componente en una imagen determinada, consistiría en realizar una especie de *merging* de los refactorings contenidos en el archivo de actualización y de aquellos que hizo el usuario.

9 Bibliografía

- ✓ [Roberts99] D.Roberts. Practical analysis for refactoring. Tesis Doctoral. University of Illinois at Urbana Campaign,1999.
- ✓ [Dig07] D. Dig. Automated upgrading of component-based applications. Tesis Doctoral. University of Illinois at Urbana Campaign,2007.
- ✓ [Henkel05] J. Henkel, A. Diwan. CatchUp! Capturing and Replaying Refactorings to Support API Evolution. En: Proceedings of the 27th international conference on Software engineering, páginas 274-283, 2005.
- ✓ [Foote89] Brian Foote y Ralph Johnson. "Reflective Facilities in Smalltalk-80". Proceedings of OOPSLA'89. New Orleans, 1989.
- ✓ [Kotter12] John Kotter. Leading Change. 2012.
- ✓ [Lehman78] Manny Lehman, "Laws of program evolution - rules and tools for programming management," in Infotech State of the Art Conference, Why Software Projects Fail, pp. 11/1 - 11/25, 1978.
- ✓ [Dig06] D. Dig y Ralph Johnson. "How do APIs evolve? A Story of refactoring. Journal of Software Maintenance and Evolution: Research and Practice. University of Illinois at Urbana Campaign, 2006.
- ✓ [Fowler99] M.Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley. 1999.
- ✓ Pharo – the collaborActive Book. http://pharo.gemtalksystems.com/book/table-of-contents . Consultado el 02/11/2017.
- ✓ Pharo by Example 5. http://files.pharo.org/books-pdfs/updated-pharo-by-example/2017-01-14-UpdatedPharoByExample.pdf. Consultado el 02/11/2017.