Revista Tecnología y Ciencia
ISSN 1666-6933 DOI: https://doi.org/10.33414/rtyc.33.1-22.2018

Universidad Tecnológica Nacional
Septiembre-Diciembre 2018 / Año 16 - Nº 33

Análisis de Dependencias entre Refactorings para Solucionar *Code Smells*

Claudia Marcos
ISISTAN-UNICEN, Tandil, Argentina - CIC
cmarcos@exa.unicen.edu.ar

Santiago Vidal
CONICET
svidal@exa.unicen.edu.ar

J. Andrés Diaz Pace
CONICET
adiaz@exa.unicen.edu.ar

Presentación: 11/2016.
Aprobación: 07/2017.

Resumen

Los *code smells* son síntomas en el código fuente que pueden revelar problemas de diseño. Para poder solucionar un smell deben aplicarse un conjunto de refactorings que permitan restructurar el sistema. Sin embargo, al aplicar un conjunto de refactorings en un orden determinado, pueden surgir problemas que impiden que éstos se apliquen. Por ejemplo, porque un refactoring que depende de una reestructuración realizada por otro refactoring que aún no fue aplicado, o porque un refactoring referencia un artefacto del sistema que fue modificado por un refactoring aplicado anteriormente. Por estos motivos, para aplicar un conjunto de refactorings, se deben analizar las dependencias que existen entre estos para poder establecer el orden de aplicación. En esta línea, este trabajo presenta una herramienta que identifica y soluciona los conflictos originados por dependencias entre refactorings para luego aplicar automáticamente los mismos. Los resultados, si bien son preliminares, indican que este enfoque permite identificar y solucionar un alto porcentaje de conflictos.

Palabras clave: *code smells*, refactoring, evolución de software

Revista Tecnología y Ciencia
ISSN 1666-6933

Abstract

Code smells are symptoms in the source code that can reveal design problems. To fix a smell, a set of refactorings must be applied that allow the restructure of the system. However, by applying a set of refactorings in a given order, problems can arise that prevent them from being applied. For example, a refactoring could depend on a restructuring made by another refactoring that was not yet applied, or a refactoring could reference a system artifact that was modified by a previously applied refactoring. For these reasons, to apply a set of refactorings, the developer must analyze the dependencies that exist between them to be able to establish the order of application. In this line, this work presents a tool that identifies and solves the conflicts originated by dependencies between refactorings and then automatically apply them. The results, although preliminary, indicate that this approach allows identifying and solving a high percentage of conflicts.

Keywords: *code smells*, refactoring, software evolution

Introducción

Para facilitar la evolución de un sistema de software se debe realizar un proceso de mantenimiento que prevenga su "envejecimiento" [1]. Este proceso, generalmente, implica la reestructuración del sistema para solucionar problemas en el diseño o errores. La reestructuración del sistema se realiza mediante la técnica de refactoring [2], la cual permite modificar la estructura interna del mismo sin alterar el comportamiento externo.

El proceso de refactorización de un sistema consta de tres etapas [3]. En primer lugar, se deben identificar las partes del sistema a reestructurar. Para esto se emplean los *code smells* [2, 4], los cuales son indicios de posibles problemas en el código. Una vez identificados los *code smells* existentes en el sistema, se deben proponer los refactorings necesarios para cada uno. Mediante los refactorings propuestos, se realizan las reestructuraciones que resuelven los problemas indicados por los *code smells*. La última etapa del proceso, comprende aplicar al código el conjunto de refactorings propuestos para los *code smells*.

Uno de los problemas que surge al aplicar un conjunto de refactorings, es el orden en que se deben aplicar. Esto se debe a que, un refactoring puede depender de la reestructuración que realiza otro refactoring. Entonces, se debe aplicar primero el refactoring que realiza la reestructuración de la cual dependen otros refactorings. Esto es lo que se conoce como refactorización en cascada [5]. Otro problema que puede surgir al aplicar un conjunto de refactorings, es que un refactoring interfiera con la aplicación de otros refactorings. Esto puede suceder debido a que un refactoring hace referencia a un artefacto del sistema que fue modificado previamente por otro refactoring, y quedó mal referenciado [6].

Si bien existe un conjunto de herramientas [7, 8, 9, 10, 11] que buscan refactorizar *code smells*, la mayoría de ellas solo se centran en la identificación de smells y en la sugerencia de refactorings, pero no en la aplicación automática de los mismos.

En este trabajo se presenta RAtool (Refactoring Analysis tool), la cual realiza un análisis de dependencia entre los refactorings recomendados para solucionar los problemas identificados por los *code smells*. El objetivo del enfoque es encontrar un correcto orden de aplicación para los refactorings y resolver los conflictos que existen entre ellos. De esta manera, se permite aplicar un conjunto de refactorings al código del sistema en un orden que no produzca inconsistencias.

La herramienta recibe como entrada el código del sistema, junto con los *code smells* y los refactorings asociados. Como salida, se obtiene el código del sistema refactorizado.

Para demostrar los beneficios del enfoque se realizó un caso de estudio sobre el sistema SportsTracker. Para este sistema, se llevó a cabo el proceso de refactorización, se detectaron los *code smells* presentes en el sistema y se seleccionaron los refactorings para solucionar un subconjunto de estos *code smells*. Éstos fueron tomados como entrada de RAtool, a través de la cual se realizó el análisis de dependencia entre refactorings, se resolvieron los conflictos y se aplicaron al código. Como resultado del uso de la herramienta, se aplicó un alto porcentaje de los refactorings propuestos. De esta forma, se eliminaron en gran medida los *code smells* tomados sobre el sistema SportsTracker, generando de esta manera, clases mejor modularizadas y reduciendo la complejidad de las mismas.

El resto de este trabajo está organizado de la siguiente manera. La Sección 2 presentan conceptos de evolución del software y mantenimiento de los sistemas. La Sección 3 describe los trabajos relacionados. La Sección 4 discute los principales problemas para lograr automatizar el refactoring de *code smells*. La Sección 5 presenta nuestro enfoque, RAtool, para de análisis y resolución de dependencias entre refactorings. La Sección 6 presenta los resultados de aplicar RAtool sobre el sistema SportsTracker. Finalmente, la Sección 7 describe las conclusiones y trabajos futuros.

Evolución del software

Los sistemas de software inevitablemente sufrirán cambios, una vez entregados, motivados por nuevos requerimientos funcionales, cambios en el negocio, cambios en el ambiente, correcciones a fallos encontrados en el funcionamiento, actualizaciones de los requerimientos existentes, etc. [1]. Una de las soluciones que se han propuesto tradicionalmente es diseñar para el cambio, con lo cual se tratan de prevenir los eventuales problemas que pueden surgir con la evolución del software. Con este propósito algunas de las técnicas que se han utilizado son [12]: interfaces, ocultamiento de información, separación de concerns, capas, herencia, polimorfismo, patrones de diseño, documentación actualizada.

Para que el sistema no quede obsoleto, éste debe evolucionar junto con sus requerimientos. Con este fin, se debe realizar un proceso de mantenimiento. Este proceso, generalmente, implica la reestructuración del sistema, para mejorar el diseño del sistema o solucionar errores. La reestructuración del sistema se realiza mediante la técnica de refactoring, la cual permite mejorar la estructura interna del sistema, sin modificar su funcionalidad, antes de introducir los cambios necesarios. El objetivo es lograr un balance entre una buena codificación y un buen diseño, permitiendo mantener el sistema evolucionable [2].

Revista Tecnología y Ciencia
ISSN 1666-6933

El proceso de refactorización de un sistema consta de tres etapas [3]. En primer lugar, se deben identificar las partes del sistema a reestructurar. Para esto se emplean los *code smells* [4], los cuales en el código que sugieren posibles problemas o errores en el diseño, los cuales pueden generar un impacto negativo en la evolución del sistema.

Una vez identificados los *code smells* existentes en el sistema, se deben proponer los refactorings necesarios para cada uno de éstos. Mediante los refactorings propuestos, se realizan las reestructuraciones que resuelven los problemas indicados por los *code smells*. En general, un solo refactoring no es suficiente para resolver un code smell sino que deben aplicarse un conjunto de refactorings. Uno de los problemas que surge al aplicar un conjunto de refactorings, es el orden en que se deben aplicar. Esto se debe a que, un refactoring puede depender de la reestructuración que realiza otro refactoring. Entonces, se debe aplicar primero el refactoring que realiza la reestructuración de la cual dependen los otros refactorings. Este proceso se conoce como refactorización en cascada [5], es decir, aquellas oportunidades de aplicar un refactoring que surgen de las reestructuraciones realizadas por otros refactorings previos.

Trabajos relacionados

Existen varias herramientas que se enfocan en las distintas etapas del proceso de refactorización. Algunas identifican *code smells*, otras sugieren refactorings y varias permiten aplicarlos al código.

Por ejemplo, JSpIRIT [7] es una herramienta que permite identificar los *code smells* en un sistema Java. El enfoque permite priorizar los *code smells* encontrados en base a diferentes criterios, tales como, historia de la aplicación, impacto de los *code smells* en escenarios de modificabilidad o la relevancia de los *code smells*. Sin embargo, la tarea de refactorizar los smells encontrados es delegada al desarrollador. Tsantalis et al. [8] proponen JDeodorant, la cual identifica *code smells* y sugiere los refactorings apropiados. Sin embargo, no permite aplicar automáticamente estos refactorings.

Otros enfoques se centran, particular- mente, en aplicar los refactorings de acuerdo a los *code smells* encontrados. Herbold et al. [9] proponen la herramienta AddFix, la cual relaciona los resultados obtenidos por las herramientas que identifican los problemas en el código, con las de aplicación de refactorings. Refactoring Browser [10] automatiza los pasos atómicos de los refactorings en Smalltalk. Esta herramienta se integra al browser de Smalltalk, el cual es la herramienta base de desarrollo. Refactoring browser permite realizar correctamente los refactorings descriptos en [10] en cualquier sistema de Smalltalk. La herramienta JRefactory [11] permite aplicar refactorings al código de manera automatizada. Adicionalmente, provee funcionalidad para insertar comentarios javadoc apropiados, generar e imprimir los diagramas UML del código, generar métricas sobre el código, entre otras. JRefactory soporta la integración con los IDEs jEdit, NetBeans, JBuilder, entre otros.

Mens y Tourwé [3][5] analizan la posibilidad de refactorizar automáticamente *code smells*. Los autores demuestran que la aplicación de un conjunto de refactorings

de manera automática está limitada principalmente por el concepto de refactoring en cascada. Específicamente, no es posible aplicar los refactorings en cascada antes que las reestructuraciones que los desencadenan. Además, la aplicación de un refactoring puede generar conflictos en los demás, ya que puede modificar o eliminar un artefacto requerido por otro refactoring.

Similarmente, Liu et al. [6] proponen un enfoque para el análisis de refactorings con el objetivo de resolver los problemas del refactoring en cascada. Este enfoque propone realizar un análisis entre refactorings para planificar el orden en que se deben aplicar en el código. Para esto, se propone realizar una matriz de conflictos entre refactorings que posteriormente se traduce a un grafo direccional. Mediante algoritmos heurísticos que recorren el grafo, se busca un correcto orden de aplicación para los refactorings. El enfoque propuesto en [6] destaca la importancia del análisis de refactorings y su validez, pero no presenta herramienta que lo materialice.

En este contexto, sería interesante proveer un enfoque que permita materializar el análisis de dependencia entre refactorings. Mediante este análisis, sería posible encontrar un correcto orden de aplicación de los refactorings, respetando el principio de refactorización en cascada, y resolver los conflictos existentes. De esta manera, se permitiría aplicar el conjunto de refactorings, necesarios para solucionar los problemas indicados por los *code smells*, al código del sistema.

Problemas en la automatización del refactoring

Una forma de identificar los refactorings a utilizar, es a través de los *code smells*. Los catálogos de *code smells* existentes, sugieren por cada uno de ellos un conjunto de refactorings para solucionarlos. Estos refactorings permiten reestructurar el sistema, mejorando su estructura interna, sin alterar el comportamiento externo del mismo.

Dado que un refactoring puede depender de la reestructuración que realiza otro refactoring, pueden surgir problemas al momento de aplicar refactorings en cascada. La Figura 1 muestra un ejemplo de este concepto. El método getArea() de la clase Cylinder está afectado por el code smell Feature Envy. Este smell consiste de métodos que utilizan más datos de otras clases que los de la propia. Por esta razón, se aconseja mover el método a la clase en la cual se encuentran los datos que utiliza. Adicionalmente, parte del método getArea() debería ser extraído para que éste resulte más legible. En este contexto, considere que es necesario utilizar el refactorings Move Method [2] para mover el método getAreaBase() a la clase CylinderData y el refactoring Extract Method [2] para extraer las 2 primeras líneas del método getArea() a uno nuevo denominado getAreaBase(). Si primero se intenta aplicar el Move Method se genera un conflicto, debido a que el método getAreaBase() no existe inicialmente en la clase Cylinder. El método referenciado es generado por Extract Method. Entonces, el Move Method depende del Extract Method, y por lo tanto se debe realiza primero la reestructuración indicada por éste último para poder aplicar el Move Method.

Revista Tecnología y Ciencia
ISSN 1666-6933

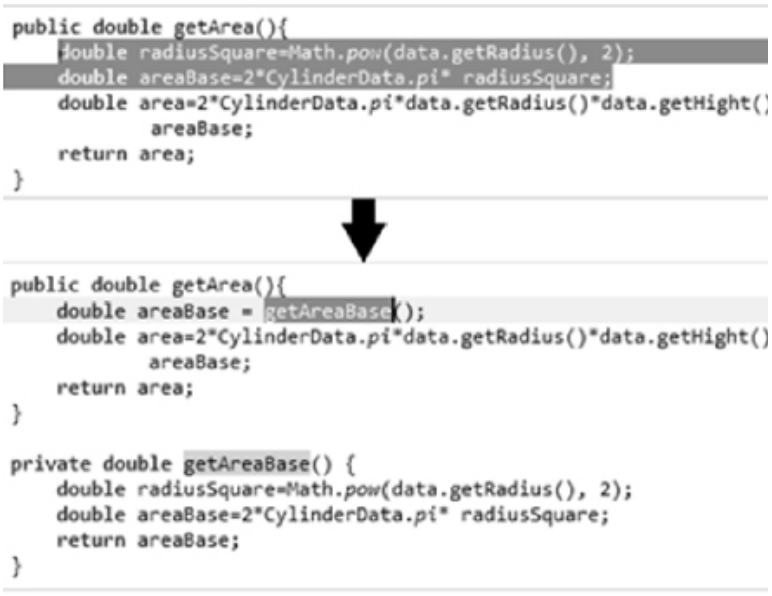

Figura 1: Ejemplo de refactoring de code smell

Otro problema que puede surgir al aplicar un conjunto de refactorings, es que un refactoring puede hacer referencia a un artefacto del sistema que fue modificado previamente por otro refactoring, y quedar mal referenciado [6]. Por ejemplo, esto ocurriría si en la clase Cylinder, se aplica el Extract Method descripto anteriormente, y luego un Rename para renombrar la variable local radiusSquare, dentro del método getArea(). En este caso, el conflicto se genera porque el artefacto referenciado por el Rename fue modificado por el Extract Method.

A partir de los ejemplos presentados anteriormente, se puede inferir que para aplicar un conjunto de refactorings se debe llevar a cabo un análisis de dependencias. Esto se debe a que las dependencias entre refactorings pueden generar conflictos que impidan que se apliquen refactorings en cascada.

Enfoque ra (refactoring analysis)

En este trabajo, se propone un enfoque llamado Refactoring Analisys (RA) que permite analizar las dependencias entre refactorings para solucionar los *code smells* de un sistema. El objetivo del enfoque es encontrar un orden correcto de aplicación de los refactorings y resolver los conflictos que existen entre ellos. El enfoque recibe como entrada el código del sistema, junto con los *code smells* y los refactorings asociados para solucionarlos. Como salida, se obtiene el código del sistema refactorizado (Figura 2).

[1]El plugin puede ser descargado en https://db.tt/zm5Q6oJ6

[2] http://www.saring.de/sportstracker/index.html

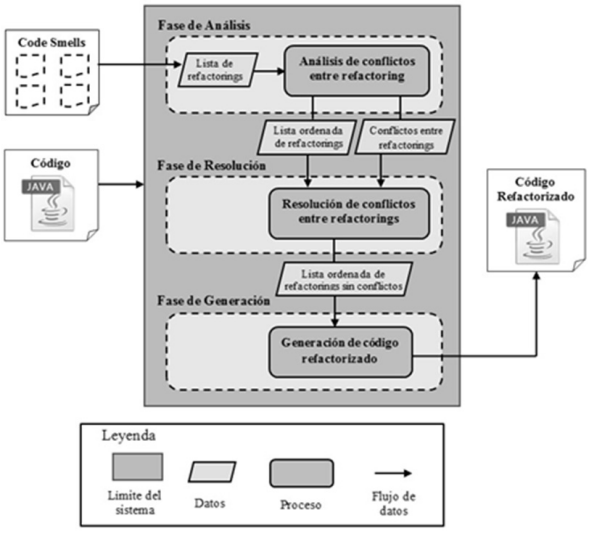

Figura 2: Esquema conceptual del enfoque

El enfoque se divide en tres fases: fase de análisis, fase de resolución de conflictos y fase de generación de código. La fase de análisis, recibe la lista de refactorings e identifica los conflictos que se generan por la aplicación en cascada de los mismos. Una vez concluida la fase de análisis, se obtiene una lista ordenada de refactorings y los conflictos que existen entre ellos. Los conflictos, se toman como parámetro de entrada de la segunda fase, denominada fase de resolución de conflictos. Durante esta fase, se utiliza un conjunto de reglas para corregir los conflictos identificados en la fase anterior. Esta fase genera como salida una lista ordenada sin conflictos de refactorings. En la última fase, la de generación de código, se aplican los refactorings sobre el código, generando como salida el código refactorizado. El enfoque ha sido materializado en la herramienta RAtool la cual es un plugin para Eclipse.

A continuación se detallan cada una de dichas fases. El enfoque es ilustrado con la refactorización de los *code smells* del sistema SportsTracker2. SportsTracker es una aplicación para realizar un seguimiento de actividades deportivas, su código es libre y se encuentra disponible en la web.

Se utiliza JSpIRIT [7] para identificar los *code smells* en el sistema, y proponer una lista de refactorings para solucionarlos. Esta herramienta identifica los *code smells* del catálogo de Marinescu [4], además prioriza los *code smells* encontrados asignándole un número de ranking a cada uno. JSpIRIT encontró un total de 191 *code smells* en el sistema SportsTracker (Tabla 1).

Revista Tecnología y Ciencia
ISSN 1666-6933

CodeSmell	Cantidad
BrainClass	4
BainMethod	8
Data Class	1
DispersedCoupling	42
FeatureEnvy	62
GodClass	10
IntensiveCoupling	23
RefusedParentBequest	16
ShotgunSurgery	25
TOTAL	191

Tabla 1. *Code smells* de Sports Tracker

En base a los *code smells* encontrados en el sistema, el desarrollador selecciona los que considera más importante resolver y JSpIRIT identifica los refactorings necesarios para aplicarlos (Figura 3). Para este caso de estudio, el desarrollador selecciona aquellos *code smells* detectados en las clases OverviewDialogController y StackedRenderer. RAtool recibe los refactorings propuestos agrupados por *code smells* y lleva a cabo las 3 fases.

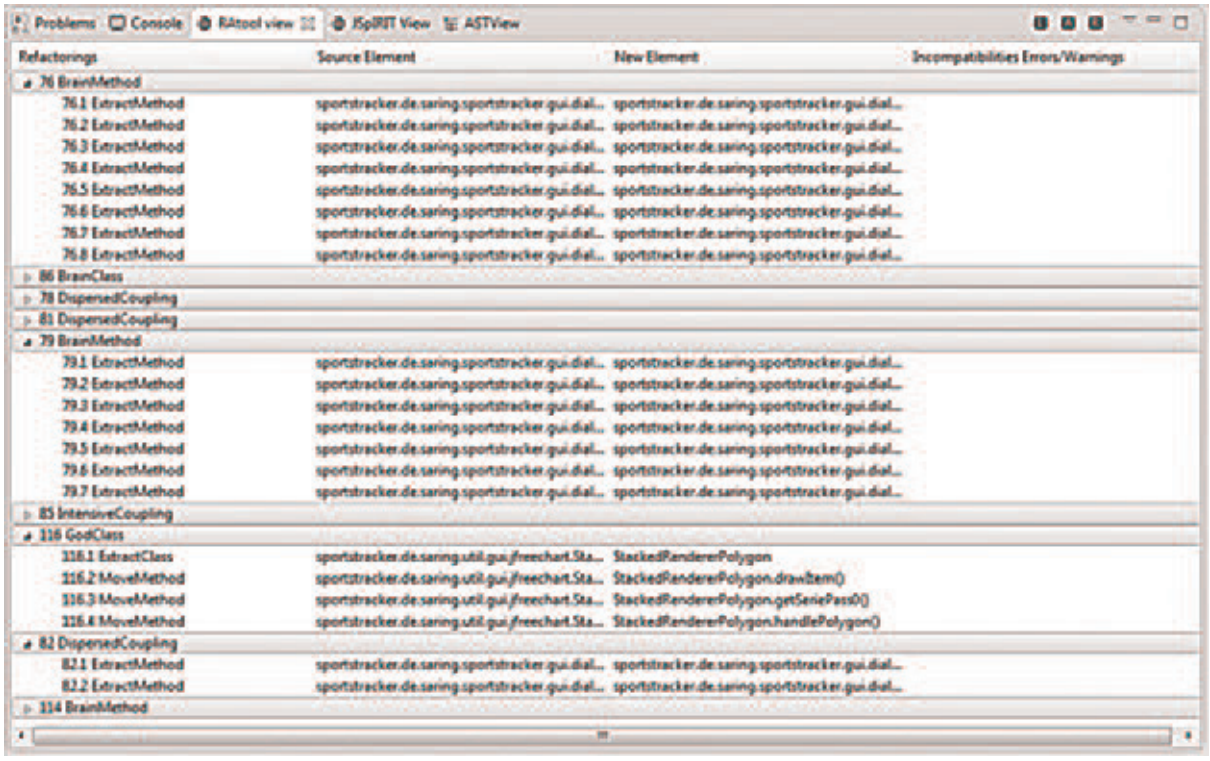

Figura 3: Lista de *Code smells* y refactorings propuestos.

Fase de análisis

La fase de análisis tiene como objetivo analizar las dependencias entre los refactorings de los *code smells* de la aplicación que generan conflictos al momento de aplicarlos. Como resultado del análisis se obtiene una lista de refactorings con las incompatibilidades de orden de aplicación resueltas e indicando aquellos refactorings que presentan incompatibilidades por parámetros mal definidos. Adicionalmente, se obtiene otra lista de refactorings cuyas incompatibilidades no tienen solución.

Análisis de conflictos entre refactorings

Los conflictos entre refactorings, se pueden producir por el orden de refactorización o porque un refactoring referencia un artefacto del sistema que fue modificado por otro refactoring, aplicado anteriormente, y quedó mal referenciado [6]. Si el conflicto se produjo porque el artefacto fue modificado previamente o porque no existe, pero es creado por otro refactoring que no se aplicó, se lo considera una incompatibilidad que tiene solución. Por el contrario, si el artefacto nunca existió en el código y ningún otro refactoring lo genera, es una incompatibilidad que no tiene solución posible. Por esta razón, se distinguen los siguientes tipos de incompatibilidades:

- Incompatibilidades que tienen solución: son los conflictos detectados durante el análisis que pueden ser resueltos por la herramienta. Dentro de éstos existen dos tipos:
- Parámetros mal definidos: El refactoring referencia un artefacto del sistema que fue modificado previamente y quedó mal referenciado. También denominado warning.
- Orden de aplicación incorrecto: Se trata del principio de refactorización en cascada. Un refactoring trata de modificar un artefacto que no existe en el sistema, pero éste es creado por otro refactoring.
- Incompatibilidades que no tienen solución: Son aquellos conflictos que no pueden ser resueltos por la herramienta ya que hacen referencia a un artefacto que no existe en el sistema y no es generado por ningún refactoring. También denominadas error.

En la Figura 4 se muestra el pseudocódigo del algoritmo de análisis, a través del cual se analizan los refactorings para determinar si son aplicables (Actividad 1).

Si es aplicable, es decir, no presenta conflicto, se agrega a una lista de refactorings ordenados (Actividad 2). En caso contrario, se lo agrega a una lista de refactorings con conflictos (Actividad 4).

En este punto es importante destacar que, se considera aplicable al primer refactoring que se pueda aplicar al código fuente. Esto se debe a que, se considera importante reducir los tiempos de análisis, debido a que la actividad de mantenimiento para facilitar la evolución de un sistema, generalmente no es una actividad solicitada por el cliente.

Revista Tecnología y Ciencia
ISSN 1666-6933

La lista de refactorings con conflictos almacena los refactorings cuyas incompatibilidades no tienen solución o pueden solucionarse por un orden de aplicación correcto. Entonces, por cada refactoring aplicable se chequea si, con el nuevo artefacto generado, se pueden resolver los conflictos de los refactorings incluidos en dicha lista.

La Figura 5 muestra un pseudocódigo que detalla el proceso utilizado para determinar si un refactoring es aplicable (Actividad 1). Para determinar si un refactoring es aplicable, se identifica si el artefacto referenciado se encuentra en la estructura de historial de cambios (Actividad 1.1). Si el artefacto es encontrado significa que ya fue modificado por un refactoring anterior y en este caso se registra el nuevo cambio, indicando si existen incompatibilidades de parámetros mal definidos (Actividad 1.2). Si existen incompatibilidades de este tipo, el refactoring igualmente es considerado aplicable, ya que éstas se pueden solucionar en la siguiente fase.

Si no existe el artefacto en el historial de cambios, se chequea que exista en el código fuente (Actividad 1.3). Si se encuentra en el código, se crea el cambio en el historial de cambios y se toma el refactoring como aplicable (Actividad 1.4). En caso que el artefacto no exista en la estructura de cambios ni en el código, se ha identificado una incompatibilidad por orden de aplicación incorrecto o que no tiene solución, y el refactoring es considerado no aplicable.

```
1  private void doAnalysis(AbstractChangeHistory project,ArrayList<SimpleRefactoring> refactorings){
2          while (refactorings.size() > 0){
3              //Refactoring sin incompatibilidad de orden de ejecución?
4  Actividad 1 if (doRefactoring(project,refactorings.get(0))){
5  Actividad 2     orderedRefactoringList.add(refactorings.get(0));
6              //Analizar refactorings con incompatibilidades
7              if (conflictedRefactoringList.size() > 0){
8  Actividad 3     analyzeConflictedRefactorings(project,visitedConflictedRefactoring,0);
9              }
10             }
11             else{
12 Actividad 4     conflictedRefactoringList.add(refactorings.get(0));
13             visitedConflictedRefactoring.add(false);
14             }
15             refactorings.remove(refactorings.get(0));
16         }
17     }
```

Figura 4:Pseudocódigo del algoritmo de análisis.

```
3      //Element in Historical Changes?
4      if (changedElement==null){
5          //Element in Source Code?
6          if (refactoring.sourceProperlyDefined(refactoring.getSource())){ Actividad 1.3
7              return registerChange(project,changedElement,source,refactoring); Actividad 1.4
8          }
9          //Element is not in HistoricalChanges && is not in SourceCode
10         else{ //Error Incompatibility
11             return false;
12         }
13     }
14     //Element in Historical Changes
15     else{
16         if (refactoring.getTypeOfRefactoring().equals("Rename")){
17             return registerRenameChange(project,changedElement, source, refactoring);
18         }
19         if (refactoring.getTypeOfRefactoring().equals("ExtractMethod")){
20             return registerExtractMethodChange(changedElement, source, refactoring);
21         }
Actividad 1.2
22         if (refactoring.getTypeOfRefactoring().equals("ExtractClass")){
23             return registerExtractClassChange(changedElement, source,(ExtractClassRefactoringUser) refactoring);
24         }
25         if (refactoring.getTypeOfRefactoring().equals("MoveMethod")){
26             return registerMoveMethodChange(changedElement, source,(MoveMethodRefactoring) refactoring);
27         }
28         return false;
29     }
30 }
31         }
32     }
33     else{
34         return false;
35     }
36 }
```

Figura 5: Pseudocódigo de la actividad de análisis.

Las incompatibilidades generadas por un orden de aplicación incorrecto se generan dependiendo del orden en que se aplican los refactorings de entrada. Por lo tanto, no se puede anticipar este tipo de incompatibilidades. Por el contrario, las incompatibilidades de parámetros mal definidos son provocadas por un conjunto fijo de conflictos, que pueden ser descriptos independientemente de los refactorings de entrada. Estos conflictos surgen por dependencias entre los refactorings, las cuales pueden ser preestablecidas. Por ejemplo, si se realiza un Rename-Package, todos los refactorings posteriores que afectan elementos dentro del paquete renombrado, deben tener en cuenta que el nombre de éste cambió.

Además, existen incompatibilidades que no tienen solución, que también surgen por las dependencias preestablecidas entre refactorings. Por ejemplo, dos refactorings Extract Method intentan extraer un conjunto de líneas que se sobreponen, por lo que uno de los dos no se puede aplicar. En este caso se aplica el primer refactoring tomado como aplicable.

Catálogo de dependencias entre refactorings

Para analizar las dependencias entre refactorings y su posible solución se ha definido un catálogo de incompatibilidades [13].

En la Figura 6 se muestran las dependencias que existen entre los refactoring incluidos en la herramienta RAtool.

En la primera columna se listan los refactorings precedentes y en la primera fila los que se aplican luego de los precedentes. El contenido de la matriz indica si la aplicación, en conjunto, del par de refactorings correspondiente, puede generar incompatibilidades.

Es decir, si los refactorings pueden tener una dependencia preestablecida, por modificar un mismo artefacto, y por lo tanto requieren análisis.

Precedente \ Posterior		Rename					Extract Method	Extract Class	Move Method
		Package	Class	Field	Method	Local Variable			
Rename	Package	!	!	!	!	!	!	!	!
	Class	✓	!	!	!	!	!	!	!
	Field	✓	✓	!	✓	✓	✓	!	!
	Method	✓	✓	✓	!	!	!	✓	!
	Local Variable	✓	✓	✓	✓	!	✓	✓	✓
Extract Method		✓	✓	✓	✓	!	!	✓	✓
Extract Class		✓	✓	!	✓	✓	✓	!	!
Move Method		✓	✓	✓	!	!	!	✓	!

Leyenda
✓ No generan incompatibilidades
! Requieren análisis

Figura 6: Dependencias preestablecidas entre refactorings.

Revista Tecnología y Ciencia
ISSN 1666-6933

Sportstracker: fase de análisis

El resultado de la fase de análisis de SportsTracker es mostrado en la Figura 7, donde se puede observar la lista ordenada de refactorings (Ordered Refactoring List) y la lista de refactorings con incompatibilidades que no pudieron solucionarse (Conflicting Refactoring List).

La lista de refactorings ordenados contiene los refactorings que se aplicarán al código. Dentro de la lista ordenada, se distinguen con un tick aquellos refactorings que no presentan conflictos, o presentaban incompatibilidades por orden de aplicación incorrecto, pero ya fueron resueltas. Además, se identifican los warning, que se corresponden con incompatibilidades de parámetros mal definidos. Estos últimos conflictos se resuelven antes de aplicar los refactorings al código.

Figura 7: Resultado del análisis de dependencia entre refactorings (vista parcial)

Incompatibilidades de orden de aplicación

Para resolver los conflictos generados por un orden de aplicación incorrecto, en primer lugar se agrupan todos los Extract Methods.

Esto se realiza así, debido a que el refactoring Extract Method recibe como parámetro el start position (sp) y el length de las selecciones de statements a extraer.

Estos valores no son relativos al método desde donde se extraen los statements, sino que corresponden a la posición que ocupan estos statements dentro de la clase.

Ante esta situación, cualquier modificación que se realice en la clase, antes de un Extract Method, cambia estos valores y puede impedir que el refactoring se aplique.

Por este motivo, también se los ordena por mayor valor de sp.

De este modo, un Extract Method no interfiere con los demás, ya que se empieza por el fin de la clase, extrayendo los nuevos métodos debajo.

De esta manera, no se ve afectado el sp y el length de los demás Extract Methods de la clase.

Luego, se realizan los Extract Class necesarios para realizar los Move Methods, y posteriormente se aplican los Move Methods correspondientes.

Aquellos refactorings que no tienen dependencias con los demás, se aplican en cualquier orden luego de los Extract Methods.

Incompatibilidades de parámetros mal definidos

Los conflictos detectados por parámetros mal definidos, para el caso presentado, se generan debido a los refactorings de tipo Rename.

Por ejemplo, el refactoring 116.3 Rename (Figura 7) intenta renombrar el método drawItem de la clase StackedRenderer.

Para esto, en primer lugar, se chequea en la estructura de historial de cambios si existe el método a renombrar. En este caso, el método es encontrado en dicha estructura, debido a que fue movido a otra clase en un refactoring previo.

Ante esta situación, se detecta en el Rename una incompatibilidad de parámetro mal definido. Ésta se identifica mediante el código correspondiente (ChangeW04).

Este código indica que el método fue movido (Tabla 2).

De manera similar, se detectan las incompatibilidades de parámetros mal definidos en el resto de los refactorings (Tabla 2).

Refactoring	Código de warning	Mensaje de warning	Refactoring causante de la incompatibilidad
116.3 Rename	*Change W04*	Method has been moved	*116.2 MoveMethod*
114.2 Rename	*Change E02*	Field has been extracted	*116.1 ExtractClass*
114.3 Rename	*Change E02*	Field has been extracted	*116.1 ExtractClass*

Tabla 2. Incompatibilidades de parámetros

Incompatibilidades sin solución

Las incompatibilidades que no pudieron ser resueltas son listadas en Conflicting Refactoring List. Por ejemplo, el refactoring 86.15 MoveMethod (Figura 7) intenta mover el método convertUnitToEnglish de la clase OverviewDialogController.

Para esto, en primer lugar, se chequea en la estructura de historial de cambios si existe el método a mover.

En este caso, el método es encontrado en dicha estructura, y debido a que fue movido a otra clase en un refactoring previo, no puede moverse por el refactoring 86.15 MoveMethod. Ante esta situación, se detecta una incompatibilidad que no puede ser resuelta.

Revista Tecnología y Ciencia
ISSN 1666-6933

Ésta se identifica mediante el código correspondiente (ChangeE04). Este código indica que el método fue movido.

De manera similar, se detectaron otras 8 incompatibilidades que no pudieron ser resueltas por RAtool.

Fase de resolución

Esta fase tiene como objetivo resolver las incompatibilidades de parámetros mal definidos detectadas durante la fase anterior y los conflictos generados por los cambios (Figura 8).

Se recibe el refactoring con las incompatibilidades de parámetros mal definidos y el código que identifica cada incompatibilidad.

En base al refactoring y al código recibido, se emplea la regla correspondiente.

La regla lee los datos de la estructura de historiales de cambios y modifica los parámetros mal definidos del refactoring, con los datos obtenidos.

De esta manera, se obtiene el refactoring sin incompatibilidades.

Reglas de incompatibilidades

En la Tabla 3 se listan todas las posibles incompatibilidades analizadas por nuestro enfoque. Por cada una se detalla el código al que se asocia la regla de incompatibilidad, una descripción del problema que indica y el tipo de incompatibilidad que describe, si tiene solución (warning) o si no tiene solución (error).

Las reglas redefinen los parámetros del refactoring en conflicto, en base al problema que indica su código. Los códigos que indican incompatibilidades de parámetros mal definidos son las del tipo warning. Por ejemplo, se tiene como entrada un Rename, que renombra el nombre de un método, y un conflicto representado por el código ChangeW01, que indica que el nombre del método referenciado cambió. En este caso, lo que hace la regla es cambiar el atributo del refactoring que hace referencia al nombre del método origen, con el último nombre asignado al método. Para esto, se utiliza la estructura de cambios, en donde se tiene registro de todos los cambios realizados a los artefactos del sistema. De igual manera se definen las reglas para todos los refactorings, en relación a los códigos que pueden aparecer en cada tipo.

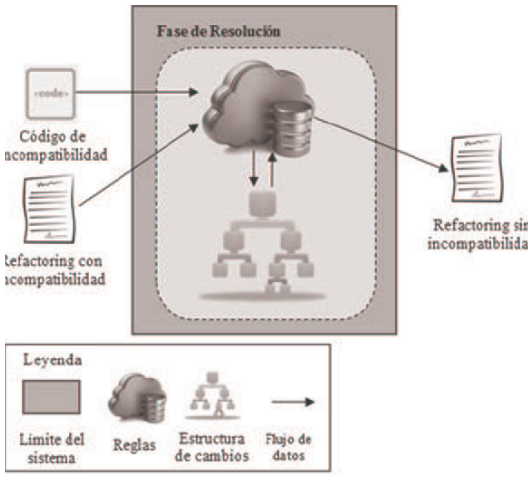

Figura 8: Resolución de conflictos entre refactorings

Las reglas que resuelven los conflictos introducidos por los cambios generados por los refactorings, renombran el artefacto generado. Estas reglas, agregan al nuevo nombre asignado, el identificador del refactoring que generó el conflicto y el del *code smells* al que pertenece. Así por ejemplo, si el refactoring que genera el conflicto es el cinco (5) del code smell dos (2), el nuevo nombre estará determinado por: "nuevoNombreC2R5". De esta manera, se soluciona el conflicto y se sabe qué refactoring lo generó. Los códigos que identifican estos tipos de conflictos son ChangeW02, ChangeW07 y CodeW02.

Luego de aplicar todas las reglas a la lista de refactorings, se obtiene una lista ordenada de refactorings libres de conflictos. Esta lista es utilizada en la siguiente fase de generación de código refactorizado.

Fase de resolución

Solo se encontraron tres incompatibi-lidades de parámetros mal definidos en SportsTracker (Tabla 2). En los primeros dos casos, se genera el conflicto debido a que se intenta renombrar un field que fue extraído. En estos casos, RAtool recibe el código de warning correspondiente (ChangeE02), junto con el refactoring. Luego, empleando las reglas definidas para el refactoring Rename y la estructura de cambios, que almacena todos los cambios realizados a los artefactos del sistema, modifica el parámetro que hace referencia a la clase en la que estaban contenidos estos field, con el de la clase a la que fueron movidos.

En el caso del 116.3 Rename (Figura 7), el conflicto aparece debido que se intenta renombrar un método que fue movido a otra clase por un refactoring previo, y quedó mal definido. En este caso, RAtool recibe el refactoring y el código ChangeW04. Luego, haciendo uso de las reglas modifica el parámetro que hace referencia a la clase en la que se encuentra el método, con el de la clase a la que fue movido.

Revista Tecnología y Ciencia
ISSN 1666-6933

Fase de generación

Esta fase tiene como objetivo aplicar los refactorings al código de manera automatizada. Para llevar a cabo su funcionalidad recibe como parámetro el código fuente del sistema y la lista de refactorings ordenados.

Código	Descripción	Tipo
Incompatibilidades que tienen solución		
ChangeW01	" name has changed"	*warning*
ChangeW02	" has same name as "	*warning(Change)*
ChangeW03	"Local Variable has been extracted"	*warning*
ChangeW04	"Method has been moved"	*warning*
ChangeW05	"Receiver Field has been renamed"	*warning*
ChangeW06	"Receiver Class has been renamed"	*warning*
ChangeW07	" has same name as moved "	*warning(Change)*
ChangeW08	"Field to be Extracted has been renamed"	*warning*
CodeW02	"Already exists an element with the same name"	*warning(Change)*
ChangeE02	"Field has been extracted"	*warning*
Incompatibilidades que no tienen solución		
CodeE01	"Source Code element not found"	*error*
CodeE03	"The selection does not cover a set of statements"	*error*
CodeE04	"Element has been extracted"	*error*
CodeE05	"Variable has not been found"	*error*
CodeE06	"Field has not been found"	*error*
CodeE07	"Lines are not enclosed in method"	*error*
CodeE08	"Receiver Class not exists"	*error*
ChangeE01	"Lines has been extracted"	*error*
ChangeE02	"Field has been extracted"	*error*
ChangeE03	"Receiver Field has been extracted"	*error*
ChangeE04	"Method has been moved"	*error*

Tabla 3: Incompatibilidades detectadas por el enfoque.

La herramienta utilizada para aplicar los refactorings en el plugin de Eclipse LTK [14]. Esta herramienta permite aplicar los pasos atómicos de un refactoring en el código de manera automática. Los refactorings en Eclipse siguen un procedimiento predefinido:

1. El refactoring es iniciado.
2. Se realiza un chequeo inicial para determinar si el refactoring es aplicable en el contexto deseado (checkInitialConditions()).
3. Se provee la información adicional de ser necesaria.
4. Luego de tener toda la información necesaria para aplicar el refactoring, se hace un nuevo chequeo (checkFinalConditions()) y se calculan los cambios individuales.
5. Se crean los cambios y se aplican en el código.

Para hacer los chequeos iniciales no se requiere que el refactoring este provisto de todos los parámetros, pero sí aquellos que hacen referencia al contexto en el que se aplicará el mismo.

En el caso de este trabajo, los refactorings son provistos de todos sus parámetros en primer lugar y luego se hacen los chequeos de las condiciones iniciales y finales. Por último, se crea el cambio y se aplica en el código.

En la Figura 9 se muestra un ejemplo de la creación del refactoring Rename. En primer lugar, se proveen los parámetros necesarios por el refactoring y luego se lleva a cabo el ciclo de vida del mismo.

Como resultado de aplicar todos los refactorings, de la lista de entrada de esta fase, se obtiene el código del sistema refactorizado.

```
1  private void doRename(RefactoringContribution contribution, IJavaElement javaElement) {
2      if (javaElement != null){
3          RenameJavaElementDescriptor descriptor = (RenameJavaElementDescriptor) contribution.createDescriptor();
4          //Element to Rename
5          descriptor.setJavaElement(javaElement);
6          //Rename getters & Setters
7          descriptor.setRenameGetters(renameGetters);
8          descriptor.setRenameSetters(renameSetters);
9          //Update references
10         descriptor.setUpdateReferences(true);
11         //New element's name
12         descriptor.setNewName(newName);
13         try {
14             Refactoring refactoring = descriptor.createRefactoring();
15             refactoring.checkInitialConditions();
16             refactoring.checkFinalConditions();
17             Change change = refactoring.createChange();
18             change.perform(monitor);
19         }
20     }
21     else{
22         IncompatibilityType containerChanged=new IncompatibilityType(Constants.CodeE01, null);
23         this.addIncompatibility(containerChanged);
24     }
25 }
```

Parámetros del refactoring

(1) Creación del refactoring

(2) CheckInitialConditions

(4) CheckFinalConditions

(5) Creación y aplicación del cambio

Figura 9: Refactoring Rename automatizado

Análisis de los resultados de sportstracker

Con el objetivo de realizar una validación inicial de nuestro enfoque, en esta

Revista Tecnología y Ciencia
ISSN 1666-6933

sección se presentan los resultados de aplicar RATool a SportsTracker (los detalles del proceso de aplicación fueron presentados en la Sección 5).

En la Tabla 4 se listan los *code smells* para los que se seleccionaron refactorings, junto con la cantidad de refactorings seleccionados para cada uno.

Además, se muestra la cantidad de refactorings aplicados para cada code smell, la cantidad de warnings detectados y la cantidad de incompatibilidades que no pudieron ser resueltas.

De esta tabla, se puede deducir lo compleja que resulta la tarea de refactorización de un sistema.

Para dos clases tomadas del sistema SportsTracker, fueron necesarios 73 refactorings para solucionar los problemas detectados.

Además, se puede observar que mediante el análisis de dependencias entre refactorings, el número de refactorings que se pueden aplicar son 63.

Dentro de los refactoring que se pueden aplicar, se detectaron 3 warnings, es decir, incompatibilidades de parámetros mal definidos.

Por otro lado RAtool identificó 10 incompatibilidades que no pudo resolver.

Teniendo en cuenta la totalidad de refactorings propuestos (para las dos clases tomadas del sistema SportsTracker) y los aplicados, se puede decir que RAtool realizó un 86% de las refactorizaciones.

Mediante los refactorings propuestos se logró reducir la cantidad de líneas de la clase OverviewDialogController de 1061 a 746, y de la clase StackedRenderer de 554 a 390, lo que significa que se redujo su complejidad.

Además, su funcionalidad quedó mejor encapsulada y se resolvieron la mayoria de los *code smells* seleccionados.

Code smells	Cantidad de refactoring seleccionados	Cantidad de refactorings aplicados	Cantidad de warnings detectados	Cantidad de incompatibilidades no resueltas
Clase Overview Dialog Controller				
76. Brain Method	8	7	0	1
78. Dispersed Coupling	1	1	0	0
79. Brain Method	7	7	0	0
81. Dispersed Coupling	3	3	0	0
82. Dispersed Coupling	2	2	0	0
85. Intensive Coupling	10	5	0	5
86. Brain Class	30	26	0	4
SUBTOTAL	61	51	0	10
Clase Stacked Renderer				
114. Brain Method	6	6	2	0
116. God Class	6	6	1	0
SUBTOTAL	12	12	3	0
TOTAL	73	63	3	10

Tabla 4. *Code smells* y refactorings seleccionados

En la Tabla 5 se listan los refactorings seleccionados en base al tipo de refactoring.

Adicionalmente, se indica la cantidad propuesta de cada tipo y la cantidad aplicada.

Se puede inferir que, para el caso de estudio, los refactorings que mayor inconveniente presentan son los Move Methods.

Para este tipo de refactoring, RAtool aplica el 76% de los seleccionados.

Refactorings	Cantidadpropuesta	Cantidadaplicada
Extract Method	30	29
Extract Class	7	6
Move Method	33	25
Rename	3	3
TOTAL	73	63

Tabla 5. Refactorings utilizados para la refactorización.

Luego los Extract Method y Extract Class son los que menor inconvenientes presentan, aplicando el 97% y el 86% respectivamente.

Por último, los Renames se aplican en el 100% de los casos dado que no generan incompatibilidades que no puedan ser resueltas.

Clase	*Code smells* seleccionados	*Code smells* con refactorings	*Code smells* resueltos completamente	*Code smells* resueltos parcialmente	*Code smells* generados
Overview Dialog Controller	13	7	9	1	8
Stacked Renderer	3	2	2	0	0

Tabla 6. *Code smells* por clases.

En la Tabla 6 se presentan los *code smells* seleccionados por clase.

Además, se presenta la cantidad de *code smells* para los cuales se propusieron refactorings, los que se pudieron resolver completamente, parcialmente y los *code smells* generados.

Se observa que, para la clase OverviewDialogController se seleccionaron un total de 13 *code smells* y se propusieron refactorings para 7 de ellos.

Como resultado de aplicar los refactorings, se resolvieron completamente 9 de los 13 *code smells* seleccionados, 1 parcialmente y se generaron 8 nuevos *code smells*.

De los *code smells* generados, 3 se introdujeron debido a los refactorings aplicados y los otros 5 *code smells*, fueron generados debido a los 10 refactorings que no se pudieron aplicar en esta clase.

Con respecto a la clase StackedRenderer, se seleccionaron 3 *code smells* y se ingresaron refactorings para 2 de éstos.

Como resultado de aplicar los refactoring se solucionaron los 2 *code smells* y no se generaron nuevos *code smells*.

Revista Tecnología y Ciencia
ISSN 1666-6933

Por otro lado, utilizando la métrica LoC (Lines of Code) [15] se puede determinar qué tan complejo y propenso a errores es un sistema, en termino de cantidad de líneas.

Generalmente, cuanto más grande sea el tamaño del código de una clase, más complejo y propenso a errores será.

En el caso de estudio, para ambas clases analizadas, se redujo el número de líneas en un 30%, lo cual da indicios de que la complejidad de estas clases disminuyó.

Además, al dividir la funcionalidades de los Brain Methods en métodos con responsabilidad más limitada, y separar la funcionalidades de las God Class y BrainClass en nuevas clases, se mejoró el encapsulamiento de datos.

Al mejorarse el encapsulamiento de datos, se mejora la cohesión de las clases, ya que ahora las clases refactorizadas ofrecen servicios más relacionados a su propósito.

Otra métrica que puede utilizarse es CBO (Coupling Between Objects) [15]. Esta métrica permite conocer el número de clases acopladas a una clase.

Cuanto más alto es este valor más dependencias existen entre las clases, ya que utilizan más métodos o variables de otra clase.

Esto afecta negativamente al diseño modularizado del sistema. Para el caso de estudio, si bien se mejoró la modularidad dividiendo funcionalidad en distintas clases, el acoplamiento aumentó.

Esto se debe a que, ahora las clases deben utilizar variables de instancias de otras clases y los métodos que fueron encapsulados, para realizar la funcionalidad que antes era propia.

Conclusiones

Al aplicar un conjunto de refactorings para solucionar un code smell pueden surgir problemas que impiden que éstos se apliquen. Estos problemas pueden surgir porque el orden de refactorización no es el correcto.

Es decir, se intenta aplicar un refactoring que depende de una reestructuración realizada por otro refactoring, que todavía no se aplicó.

Además, se puede presentar un problema cuando un refactoring referencia un artefacto del sistema que fue modificado por otro refactoring, aplicado anteriormente, y quedó mal referenciado.

Por estos motivos, para aplicar un conjunto de refactorings, se deben analizar las dependencias que existen entre estos.

Este trabajo presenta un enfoque que aborda estas problemáticas.

El enfoque se soporta en una herramienta denominada RAtool, la cual es un plugin para Eclipse.

Esta herramienta toma como entrada los *code smells* identificados en el sistema y los refactorings propuestos para cada uno.

Luego, tomando los refactorings de entrada, se realiza un análisis de dependencia entre refactorings, con el objetivo de identificar los conflictos existentes.

De esta manera, se encuentra un correcto orden de aplicación de los refactorings y se resuelven estos conflictos.

El caso de estudio presentado, si bien es acotado, permite demostrar la

factibilidad del enfoque para resolver incompatibilidades y automatizar la aplicación de conjuntos de refactorings.

Como limitación puede mencionarse el hecho de que actualmente el enfoque soporta un conjunto de 8 refactorings por lo que no es posible resolver cualquier tipo de code smell.

Por esta razón, como trabajo futuro se espera extender el enfoque con nuevos tipos de refactorings.

Adicionalmente, se pretende validar la efectividad del enfoque con un mayor número de sistemas.

Revista Tecnología y Ciencia
ISSN 1666-6933

Referencias

Parnas, D. L. (1994). *Software aging. Proceedings of the 16th international conference on Software engineering.* IEEE Computer Society Press.

Fowler, M. (2009). *Refactoring: improving the design of existing code*. Pearson Education India.

Mens, T., Tourwé T. and Munoz F. (2003). *Beyond the Refactoring Browser: Advanced Tool Support for Software Refactoring,* Proc. IWPSE\'03: 39-44.

Lanza, M., and Radu M. (2007). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media.

Tourwé, T., and Tom M. (2003). *Identifying refactoring opportunities using logic meta programming. Proceedings.* Seventh European Conference on. IEEE.

Liu, H., Li, G., Ma, Z., y Shao, W. (2007). *Scheduling of conflicting refactorings to promote quality improvement.* Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering. ACM,

Vidal, S., Marcos C., y Díaz-Pace, J. A. (2014). *An approach to prioritize code smells for refactoring.* Automated Software Engineering.

Tsantalis, N. (2010). E*valuation and Improvement of Software Architecture: Identification of Design Problems in Object-Oriented Systems and Resolution through Refactorings.* Diss. Ph. D. dissertation, Univ. of Macedonia.

Herbold, S., Jens, G., y Helmut, N. (2009). *Automated Refactoring Suggestions Using the Results of Code Analysis Tools.* Advances in System Testing and Validation Lifecycle.

Roberts, D., Brant J., y Johnson, R. (1997). *A refactoring tool for Smalltalk.* Urbana 51.

Seguin, C. y Atkinson, M. J. Refactory. http://jrefactory.sourceforge.net/

Sommerville, I. (2004). *Software Engineering. International computer science series.* Addison Wesley,

Ditz, L. M., Vidal, S., y Marcos C. (2016). *Análisis de dependencia entre refactorings.* Trabajo final de grado, UNICEN.

Frenzel, L. (2006). *The Language Toolkit: an API for automated refactorings in Eclipse-based IDEs.* Eclipse Magazin 5.

Chidamber, S. R., y Kemerer, C. F. (1994). *A metrics suite for object oriented design*. IEEE Transactions on software engineering, 20(6), 476-493.