Estrategias de Resolución del Code Smell Feature Envy

Claudia Marcos[1,2], Juan Pablo Antivero[2] , Lucas Arias[2], Santiago Vidal[3,2]

[1]ISISTAN-CIC, Campus Universitario, Tandil, Buenos Aires, 7000, Argentina
[2]Fac. Cs. Exactas-UNICEN, Campus Universitario, Tandil, Buenos Aires, 7000, Argentina
[3]ISISTAN-CONICET, Campus Universitario, Tandil, Buenos Aires, 7000, Argentina

Resumen. Los code smells son síntomas útiles para la identificación de problemas estructurales de un sistema que se relacionan con problemas de modificabilidad. Surgen por la utilización de malas prácticas al desarrollar un sistema. Para poder solucionar los code smells es necesario aplicar el refactoring que permitan mejorar aspectos de calidad como mantenibilidad, comprensibilidad y reusabilidad. El code smell Feature Envy puede ser considerado el síntoma más común relacionado con problemas de acoplamiento y cohesión. Es un método que parece más interesado en los datos de otra clase que en los de su propia clase. Este problema puede ser solucionado aplicando los refactorings Extract Method y Move Method. Sin embargo, la identificación de la mejor estrategia de resolución no siempre es sencilla dado que requiere de un análisis detallado de las diferentes alternativas. Por esta razón, en este trabajo se propone una estrategia de resolución del code smell Feature Envy la cuál propone al desarrollador diferentes alternativas de solución utilizando un algoritmo heurístico de manera tal que pueda analizar dichas posibilidades y utilizar la que considere más adecuada al proyecto.

Palabras claves. Code Smell, Feature Envy, Refactoring

1. Introducción

En los sistemas de software es muy importante centrar el desarrollo en su calidad [5,9]. En el contexto del software, calidad es la totalidad de aspectos y características de un producto o servicio que tienen que ver con su habilidad de satisfacer las necesidades explícitas o implícitas [10]. Para asegurarlo existen distintas prácticas a nivel proceso como ciclo de vida del software, metodologías de desarrollo, modelos de mejora continua de procesos y distintos aspectos relacionados al área de SQA (Sofware Quality Assurance) [5]. Sin embargo, la calidad y la estructura de los sistemas se ven deterioradas a menudo como resultado de los cambios que introducen los programadores para resolver objetivos a corto plazo, sin conocer completamente el diseño del código [8]. Estos problemas de diseño normalmente son conocidos como code smells. Un code smell, se refiere a cualquier síntoma que puede indicar que el código fuente presenta problemas [8], como lo es el código duplicado, métodos o clases muy largos, mal encapsulamiento de clases o extensas listas de parámetros, entre otros.

La calidad del software no se refiere únicamente a obtener un producto sin errores, sino también a que su código fuente se pueda extender, modificar de manera simple y entender sin que ésto resulte un problema con el paso del tiempo. La existencia y aumento de code smells generan una degradación de la calidad del software, impactando fuertemente sobre la comprensión, mantenimiento y evolución del código fuente [2].

La estrategia de solución de los code smells es mediante la refactorización [8] del código fuente. Refactorizar, es el proceso de realizar un cambio en la estructura interna del software para que este sea más fácil de entender y más simple de modificar sin cambiar su comportamiento. Este proceso, además, permite mejorar el diseño del software mejorando su modularización, legibilidad y mantenibilidad.

A pesar de la importancia del proceso de refactoring como medio para mantener la calidad del software, éste no siempre es realizado con la frecuencia ni con el tiempo que requiere. La falta de refactorings, en general, se debe a lo complejo y costoso que puede resultar el proceso de detección de code smells, la forma de solucionarlos, y el riesgo de generar errores durante el proceso.

Feature Envy (FE) es un code smell que impacta principalmente en la modificabilidad del sistema. Lanza y Marinescu [11] lo definen de la siguiente manera: “La desharmonía de diseño Feature Envy se refiere a los métodos que parecen más interesados en los datos de otras clases que en los de su propia clase. Estos métodos acceden o una gran cantidad de datos de otras clases directamente o mediante métodos de acceso”. La presencia de FE afecta negativamente el mantenimiento y la extensibilidad, minimizando la cohesión de una clase y generando el riesgo de sufrir un efecto en cascada. La FE es un code smell sobre el cual se posee poca información que permita inferir un método automatizado e intuitivo para eliminarla. Martin Fowler [8] plantea que la solución consiste en mover el método a la clase de la cual envidia mediante el refactoring Move Method. En caso de que únicamente una parte del método sufra esta anomalía, previamente hay que extraerla mediante Extract Method para luego realizar el Move Method únicamente de esa parte.

Debido a la complejidad que conlleva solucionar la FE y asegurar que no se ha modificado la funcionalidad del sistema ni se han introducido errores de compilación, es importante encontrar la forma de automatizar su refactorización. Pocos estudios intentan solucionar code smells con una refactorización automatizada [7] [4] [3]. En el caso particular del FE, existen herramientas que intentan lograrlo. Sin embargo, ninguna solución combina la aplicación de Extract Method con Move Method.

En este trabajo se presenta una extensión de la herramienta Bandago [6] para el code smell Feature Envy. Bandago es una herramienta que tiene como objetivo la solución de code smells y luego el análisis del costo beneficio de la solución mediante la presentación de métricas pertinentes. Originalmente, dicha herramienta poseía la funcionalidad para solucionar el code smell Brain Method, y fue extendida para soportar la resolución de Feature Envy.

El resto de este artítulo está organizado de la siguietne manera. La Sección 2, describe las características principales del code smell Feature Envy y cómo soluciona. La Sección 3, presenta diferentes trabajos que proponen la identificación y/o solución de las Feature Envy. La Sección 4, describe en detalle la propuesta presentada en este trabajo para solucionar las FE. En la Sección 5, tres preguntas de investigación son

desarrolladas con el objetivo de validar la propuesta. Por último, en la Sección 6 las conclusiones son presentadas.

2 Code Smell Feature Envy (FE)

Es común que con el paso del tiempo se degrade la calidad de un sistema, debido a los cambios paulatinos que los desarrolladores deben introducir durante la evolución del software, ya que muchas veces se realizan sin el total conocimiento de la arquitectura del sistema y de sus requerimientos iniciales. El uso de malas prácticas, denominadas code smells, ha sido establecido como un concepto para identificar patrones o aspectos del diseño de software que pueden causar problemas para el desarrollo o el mantenimiento del sistema [17].

Uno de los 21 code smells catalogados por Lanza y Marinescu [11] es Featury Envy (FE). Este code smell indica la existencia de métodos que parecen más interesados en los datos de otras clases que en los de su propia clase. Estos métodos acceden directamente o vía métodos accesores a una gran cantidad de datos de otras clases [11]. Esto puede ser una señal de que el método está ubicado en un lugar incorrecto y debe ser movido a otra clase. Para poder identificar si un método es FE se calculan tres métricas: Access to Foreign Data (ATFD), indica el uso directo de atributos de otras clases; Locality of Attribute Accesses (LAA), indica la relación entre el acceso a atributos locales y el acceso a atributos extranjeros; y Foreign Data Providers (FDP), indica la cantidad de clases que proveen atributos extranjeros al método.

(ATFD > Few) AND (LAA < One Third) AND (FDP <= Few)

Es necesario refactorizar las FE para mejorar la cohesión y evitar los "efectos cascada" [11]. La solución de este code smell consiste en mover el método a la clase de la cual envidia mediante el refactoring Move Method [8]. En caso de que únicamente una parte del método sufra esta anomalía, previamente hay que extraerla mediante el refactoring Extract Method para luego realizar el Move Method únicamente de esa parte. Adicionalmente, se deben tener en cuenta cuatro condiciones al realizar el refactoring Extract Method:

1. El código seleccionado para extraerse tiene que ser una lista de sentencias.
2. Dentro del fragmento de código no pueden existir asignaciones a variables que sean utilizadas luego en el flujo del método.
3. No pueden existir sentencias del tipo return.
4. El bloque de código no puede contener ramas de ejecución que vayan por fuera del código seleccionado.

En cuanto al uso de Move Method, adicionalmente se deben tener en cuenta las condiciones que imposibilitan la realización del mismo:

- El método no debe estar declarado en superclases o subclases.
- Deben modificarse todos los llamados que referencien a ese método.
- En la clase candidata a la cual se moverá no debe existir un método del mismo nombre.
- El método no debe sobreescribir un método abstracto.

- El método debe referenciar a la clase candidata a través de sus parámetros o de los campos de la clase original.

Adicionalmente, la existencia de FE está relacionada directamente con la existencia de otros code smells como God Class o Data Class (Fig. 1), por lo que se podría eliminar otros code smells al solucionar las FE de un sistema.

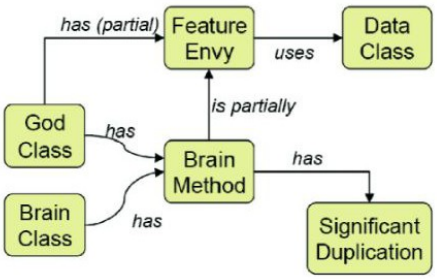


Figura 1. Relación entre Code Smells

3 Trabajos Relacionados

Para realizar el refactoring de una FE, se deben realizar dos pasos fundamentales: identificar la oportunidad de refactoring, y luego aplicarla. La mayoría de las herramientas sólo aplican el refactoring Move Method sin utilizar el Extract Method. En cuanto a las posibles acciones realizadas por las herramientas, se pueden clasificar en tres enfoques: recomendación, aplicación y, recomendación y aplicación.

Las herramientas de recomendación sólo encuentran métodos candidatos y recomiendan la aplicación del refactoring. No distinguen si el refactoring es posible o si realmente soluciona la FE. En este grupo se encuentran algunas herramientas. MethodBook [4] está basado en la técnica probabilística Relational Topic Models (RTM). La herramienta MORE [3] es un recomendador automatizado de refactorings. Posee tres objetivos: Mejorar la calidad de diseño, resolver code smells, introducir patrones de diseño. La herramienta c-JRecRef [1] tiene como principal objetivo tener en cuenta la semántica del código, además del análisis de métricas para medir la efectividad de las recomendaciones. Adicionalmente, se encarga de detectar las oportunidades de refactoring a medida que el programador compila su código. La herramienta JMove [14] tiene la particularidad de ser más simple que las otras herramientas, y de no analizar en ningún momento el impacto de sus recomendaciones sobre el code smell FE. JMove utiliza un coeficiente de similitud para identificar un método más similar a una clase que a la clase en la que se encuentra.

Por otro lado, las herramientas de aplicación realizan el refactoring propiamente dicho, es decir, mueven el método de una clase a otra. No realizan ningún tipo de análisis o recomendación en cuanto al método que se desea mover, únicamente realizan el Move Method siempre y cuando se cumpla la regla de no modificar el comportamiento del sistema ni generar errores de sintaxis. Un ejemplo es la API nativa de Eclipse. Esta API posee la capacidad de realizar el refactoring Move Method.

Por último, están las herramientas de recomendación y aplicación. Son más complejas que las anteriores, ya que realizan ambos procedimientos e intentan obtener un resultado válido. JDeodorant [7] es una herramienta que se encarga de identificar, recomendar y aplicar el refactoring Move Method para la resolución de la FE. Este plugin de Eclipse no realiza un complejo análisis para la recomendación, simplemente utiliza el ASTParser del Eclipse Java Development Tool (JDT) para analizar las relaciones entre entidades y aplicar el refactoring.

4 Estrategia para Refactorizar el Code Smell Feature Envy

En este trabajo se propone un enfoque para la eliminación de las FE de un sistema que facilite su evolución y mantenimiento. El enfoque consiste en el análisis del código de un método identificado como FE para definir si es posible mover el mismo o un fragmento del mismo a una clase candidata, y luego la aplicación automática de dichas acciones. Además, se permite la visualización de las diferentes alternativas de solución generadas para que el desarrollador pueda analizarlas, observando las métricas propuestas, y así encontrar la más adecuada para luego aplicar al proyecto. En la resolución de la FE intervienen 2 refactorings diferentes: Extract Method para obtener el fragmento de código y Move Method para moverlo a la clase a la que debería pertenecer. Estos refactorings no son triviales debido al conjunto de precondiciones a tener en consideración para no modificar la funcionalidad del sistema (explicado en la Sección 2 Code Smell Feature Envy).

El enfoque propuesto sigue un ciclo iterativo donde la entrada es una FE y la salida es un conjunto de soluciones a la misma, en caso de existir. Estas soluciones son el conjunto de refactorings realizados junto con distintas métricas las cuales son presentadas al adesarrollador para que selecciona la que mejor se adapta al proyecto. Este enfoque consta de 6 actividades (Fig. 2), aunque dependiendo las características del método, puede requerir actividades adicionales para realizar extracciones: 1. Obtener características del método, 2. Elegir las clases candidata, 3. Seleccionar el fragmento de código a mover, 4. Calcular métricas de la solución generada, 5. Aplicar el refactoring, 6. Corregir errores.

El proceso itera en las actividades 3 y 4 hasta encontrar soluciones o hasta agotar las clases que pueden ser consideradas como candidatas para realizar el refactoring. Se itera por cada clase candidata seleccionando el fragmento de código a mover recomendada, acompañando cada solución con sus métricas. En este punto se pueden dar 3 escenarios: 1. No se encontró solución a la FE; 2. El programador no desea aplicar ninguna solución propuesta; 3. El programador elige una solución. Los primeros dos escenarios conllevan a la finalización del flujo de ejecución, no refactorizando la FE. El tercer caso permite continuar el flujo con las últimas dos actividades.

En cada iteración, se deben priorizar de manera lo más precisa posible las clases candidatas, y seleccionar el mejor statement a extraer (o el método completo) para iterar la menor cantidad de veces posible. La selección de clase se prioriza utilizando la métrica ATFD (Access to Foreing Data, cantidad de atributos de otras clases que uti-

liza el método), mientras que la selección de statement es dependiente de la heurística utilizada. De esta forma, se itera hasta agotar las clases que pueden ser candidatas.

Para la aplicación de este enfoque, se optó por extender la herramienta Bandago [6] perteneciente al plugin JSpIRIT [16] de Eclipse. Por lo tanto, la solución presentada es aplicada para el lenguaje Java. JSpIRIT analiza código estáticamente y provee el método catalogado como FE que se utiliza de entrada. Bandago provee la estructura general para poder cumplir el ciclo de ejecución y presentar los resultados obtenidos de una forma sencilla.

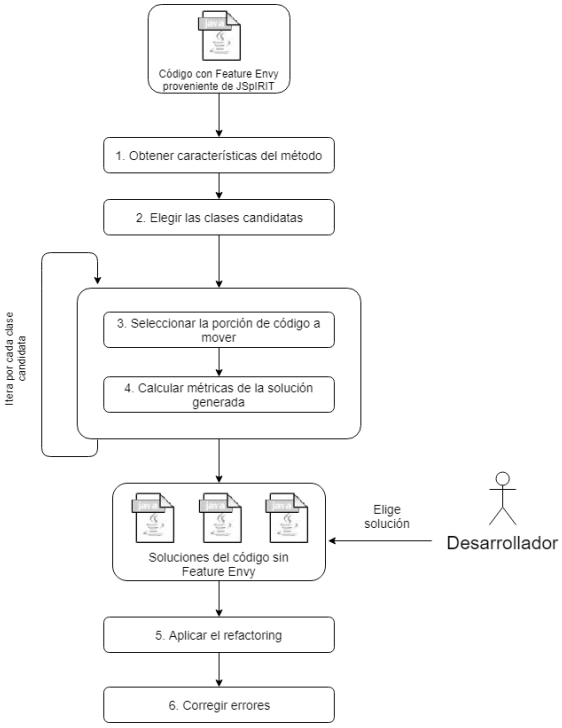


Figura 2. Actividades para solucionar la Feature Envy

A continuación se detalla cada actividad involucrada en el enfoque junto con un ejemplo guía. Para describir las actividades realizadas para refactorizar una FE se tomará como ejemplo el método mapEntry perteneciente a la clase OptionsService tomado de la aplicación jMoney, uno de los casos de estudios analizados.

4.1 Actividad 1: Obtener características del método

Esta actividad recibe como entrada un método catalogado como FE seleccionado desde la herramienta JSpIRIT. El objetivo de esta actividad es recuperar las características específicas del método que ayudarán a decidir cuál será la mejor estrategia de refactorización. En esta actividad, se identifican los llamados a métodos propios y uso de variables de instancia de la clase origen (Fig. 3).

```
private void mapEntry(Entry e, Account acc, Entry splitEntry, net.sf.jmoney.model.Entry oldEntry) {
    e.setAccount(acc);
    e.setSplitEntry(splitEntry);
    e.setAmount(oldEntry.getAmount());
    e.setCreation(oldEntry.getCreation());
    e.setDate(oldEntry.getDate());
    e.setDescription(oldEntry.getDescription());
    e.setMemo(oldEntry.getMemo());
    e.setStatus(this.entryStates[oldEntry.getStatus()]); } Uso de atributo local
    e.setValuta(oldEntry.getValuta());

    // Category might not exist yet, so this is done in a second pass.
    net.sf.jmoney.model.Category oldCat = oldEntry.getCategory();
    if (oldCat != null) {
        this.entryToOldCategoryMap.put(e, oldCat); } Uso de atributo local
    }

    this.em.persist(e); } Uso de atributo local
}
```

Figura 3. Identificación llamados a métodos propios y uso de variables de instancia

Al observar el método se puede identificar que utiliza variables de instancia. Las variables utilizadas son “entryStates”, “entryToOldCategoryMap” y “em”. Esta detección es realizada mediante la interfaz provista por JSpIRIT. Al analizar la métrica Locality of Attribute Accesses (LAA), la cual establece el porcentaje de uso de atributos locales, se define que se están utilizando variables de instancia cuando dicha métrica es mayor a cero.

Para el ejemplo presentado, el número de accesos a variables de instancia es 3. En cuanto al número total de accesos, se deben sumar los 3 accesos a variables de instancia y todos los accesos a parámetros y variables locales del método. En este caso, se deben sumar todos los accesos a los parámetros “e” y “oldEntry”, dando como resultado un total de 19 accesos. De esta forma, se puede utilizar la ecuación observando que la métrica LAA posee un valor mayor a 0:

$$LAA = \frac{\#accesos\ a\ variables\ de\ instancia}{\#accesos\ a\ variables} = \frac{3}{19} = 0{,}15 > 0$$

4.2 Actividad 2: Elegir la clase candidata

Una vez identificadas las características del método, se procede a identificar la clase candidata a la cual se debe mover el método. Con este objetivo, se utiliza la métrica Access to Foreign Data (ATFD, cantidad de atributos de otras clases que utiliza el método). La misma se obtiene calculando el número de llamados directos o indirectos a campos de clases externas. De esta forma, al identificar individualmente los accesos a atributos o métodos de otras clases, se puede definir cuál es la clase que posee más accesos utilizados por el método. Para obtener el valor ATFD se debe contabilizar el total de dichos llamados, para el ejemplo presentado es 17 (Fig. 4). Se puede observar que se debió detallar el path completo de las clases externas que intervienen debido a que poseen el mismo nombre. Para definir las clases que pueden ser consideradas candidatas, se calcula el valor de la métrica ATFD para cada una de las clases extranjeras involucradas.

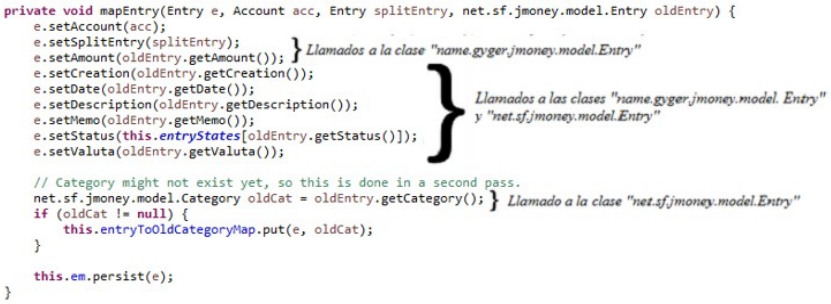


Figura 4. Llamado a clases externas

En la Tabla 1, la clase name.gyger.jmoney.model.Entry es la más envidiada al ser la que recibe la mayor cantidad de llamados por parte del método FE, con un valor muy similar está la clase net.sf.jmoney.model.Entry.

Ranking	Nombre de la clase	Nombre del parámetro	Cantidad de accesos
1	name.gyger.jmoney.model.Entry	e	9
2	net.sf.jmoney.model.Entry	oldEntry	8
3	name.gyger.jmoney.model.Entry	splitEntry	1

Tabla 1. Cálculo de la métrica ATFD

4.3 Actividad 3: Seleccionar el fragmento de código a mover

En esta actividad se utiliza la información obtenida en la actividad 1 para definir cuál es el fragmento del método FE. El primer aspecto a tener en cuenta es si el método analizado utiliza variables de instancia de su propia clase, para chequear que al mover dicho método no genere una FE en la clase candidata. En caso de no utilizar ninguna variable de instancia, el método podría ser movido completamente a la clase candidata. Si existiera algún acceso a variables de instancia de la clase, se debe seleccionar el fragmento de código a ser movido. Para lograrlo, se utiliza una heurística que: 1. Identifica la primera y última ocurrencia de un llamado a un método o atributo de la clase candidata. 2. Selecciona el statement o conjunto de statements más pequeño que contiene ambos llamados. Este paso evita la rotura de flujos de control, por ejemplo, cortar una sentencia IF a la mitad.

Estos pasos se repiten para cada clase candidata identificada. Para el ejemplo, esta actividad genera dos resultados, el primer resultado se observa en la Figura 5.

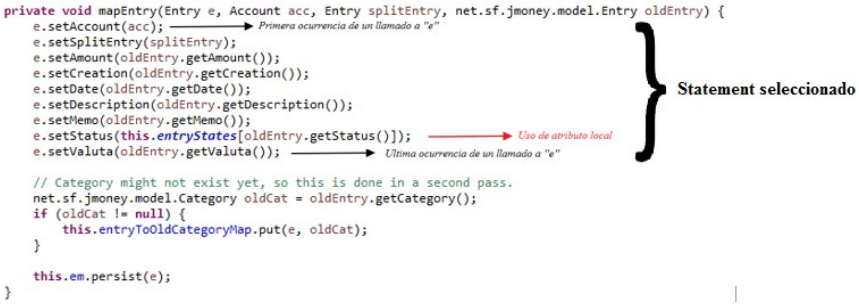


Figura 5. Selección de statement candidata

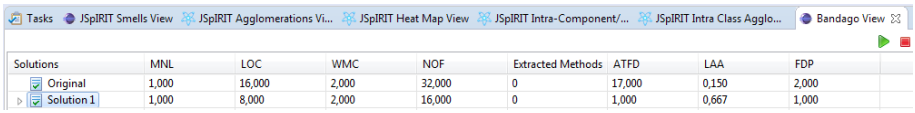
Para el ejemplo, la primera y última ocurrencia no se encuentran en ningún flujo de control, por lo cual es posible seleccionar un statement que comience en la primera ocurrencia y termine en la última. Otro factor importante es la existencia de una variable de instancia de la clase original en dicho statement, pero se puede corroborar analizando la métrica LAA (Locality of Atribute Acceses). Para el statement del ejemplo, el número de accesos a variables de instancia es 1 (un acceso a la variable de instancia entryStates). El número total de accesos es 16 (el acceso anteriormente mencionado y los llamados a métodos de los parámetros e y oldEntry):

$$LAA = \frac{\#accesos\ a\ variables\ de\ instancia}{\#accesos\ a\ \ variables} = \frac{1}{16} = 0{,}0625 < 0{,}33$$

De esta forma, se visualiza que el statement seleccionado se sigue encontrando en los límites establecidos por Lanza y Marinescu [11] (LAA < ONE THIRD), y se considera candidato a ser extraído.

4.4 Actividad 4: Calcular las métricas de la solución generada

El objetivo de esta actividad es analizar el fragmento seleccionado para el cálculo de métricas. Para lograrlo, se simula cómo quedaría la clase objetivo y se calculan las métricas correspondientes a la FE (ATFD, LAA, FDP). Las métricas permiten al desarrollador evaluar la calidad de cada solución.



Solutions	MNL	LOC	WMC	NOF	Extracted Methods	ATFD	LAA	FDP
Original	1,000	16,000	2,000	32,000	0	17,000	0,150	2,000
Solution 1	1,000	8,000	2,000	16,000	0	1,000	0,667	1,000

Figura 6. Métricas para las soluciones candidatas

Al comparar los valores del método original con los de la solución propuesta (Fig. 6), se puede observar que ATDF disminuyó su valor considerablemente debido a que una gran parte de los accesos a clases extranjeras pertenecen a la clase a la cual se moverá el método, por lo que dichos llamados dejarán de ser extranjeros. Por esta misma razón, la relación entre el número de accesos locales y número total de accesos

aumenta, observado por el elevado LAA. El valor de FDP disminuye al reducirse el número de clases extranjeras utilizadas por el método.

4.5 Actividad 5: Aplicar el refactoring

Esta actividad consiste en la aplicación efectiva de la solución seleccionada. La herramienta se encarga de mostrar a la izquierda el método original y a la derecha el fragmento de código que se extraerá (Fig. 7). De esta forma, el desarrollador puede ver aplicada la solución propuesta, en la cual se extrae el statement o conjunto de statements identificados por la actividad 3 y seleccionar la más adecuada.

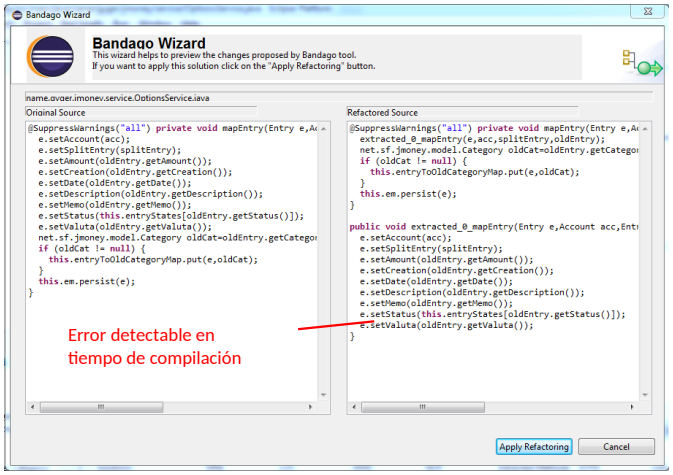


Figura 7. Visualización del código resultante

4.6 Actividad 6: Corregir errores

Esta actividad es la única que queda a cargo del desarrollador. Se debe verificar que el refactoring Move Method no haya introducido errores de código en tiempo de compilación. En el ejemplo (Fig. 7) se puede ver que el refactoring ha introducido un error a causa del llamado a una variable de instancia de la clase original.

5 Casos de Estudio

Para validar el enfoque se respondieron tres preguntas de investigación (RQ):

- RQ#1: ¿Qué porcentaje de FE son solucionadas automáticamente o con intervención del desarrollador?
- RQ#2: ¿Solucionar las FE permite solucionar otros Code Smells?
- RQ#3: ¿Son complementarias las soluciones identificadas por el enfoque propuesto y por JDeodorant en la refactorización de FE?

Para responder estas preguntas se tomaron 10 proyectos Java (Tabla 2) para ser analizados por Bandago, los cuales fueron obtenidos de Qualitas Corpus [13], una

colección de sistemas open source utilizados para fines de investigación. Para cada proyecto se calculó las FE, un total de 345.

Nombre del sistema	Cant. de clases	Cant. de Feature Envy
Apache-jspf-resolver	140	28
Fitjava	95	31
Jmoney	47	9
Jparse	75	50
Jpf	139	43
Junit	1077	46
MobileMedia	51	10
Oscache	115	42
Quickserver	199	45
Squirrel	73	41

Tabla 2. Aplicaciones seleccionadas y FE identificadas

Se calcularon también, las métricas asociadas a la existencia de FE (ATFD, LAA y FDP). La métrica ATFD tiende a ser muy alta (5,295), que significa que los métodos utilizan muchos atributos/métodos pertenecientes a otras clases. Los valores bajos de la métrica FDP, identifican clases candidatas que recibían una gran cantidad de accesos a sus atributos desde el método identificado como FE. Para LAA, el valor promedio dio un resultado bajo (0,02), indicando que la mayoría de los métodos identificados no utilizan ninguna variable de instancia o método propio, con lo cual toma aún más sentido la necesidad de mover dicho método.

A continuación las preguntas de investigación son desarrolladas.

5.1 RQ#1: ¿Qué porcentaje de Feature Envy son solucionadas automáticamente o con intervención del desarrollador?

La RQ#1 tiene como objetivo analizar el porcentaje de FE solucionadas, se analiza el conjunto de code smells identificados por JSpIRIT de cada proyecto (Tabla 3). Luego se compara el conjunto de code smells identificados luego de refactorizar todas las FE posibles. Además, se tiene en cuenta si dicha refactorización fue resuelta automáticamente (autom.) o necesitó la intervención del desarrollador (con intervención). Analizando los datos de la tabla se puede concluir que de 345 FE detectadas, el 4,06% fueron solucionadas sin la intervención del desarrollador y un 33,33% con la intervención del desarrollador. Los peores porcentajes obtenidos, 0% automáticamente para 6 proyectos y 4,34% con la intervención del desarrollador para junit. El 0% de resolución automática se debe a que el enfoque no está centrado en identificar el conjunto de precondiciones y postcondiciones previos a la aplicación del refactoring Move Method. Esto conlleva a que la gran mayoría de las soluciones identificadas requieran intervención del desarrollador.

Realizando un análisis integral de porcentaje de resolución (total) se obtiene que el 37,39% fueron resueltas. Se detectó un alto porcentaje de métodos en los cuales fue

necesario extraer completamente el mismo y no solo una porción (89,15%). Esto se debe a dos factores. El más importante es que, un gran porcentaje de las mismas no posee accesos a atributos (LAA = 0), por lo cual, de acuerdo a la estrategia del enfoque, se decide extraer completamente el método. En segundo lugar, varios métodos en un sistema poseen el último acceso a una clase extranjera dentro de un statement indivisible y por lo tanto se debe mover la totalidad del método.

Sistema	Cant FE	Autom.	% autom.	Con intervención	% con intervención	Cant. total	% Total
Apache-jspf-resolver	28	6	21,43%	7	25%	13	46,43%
Fitjava	31	3	9,67%	14	45,16%	17	54,84%
Jmoney	9	0	0%	4	44,44%	4	44,44%
Jparse	50	2	4%	12	24%	14	28%
Jpf	43	2	4,76%	15	35,71%	17	40,47%
Junit	46	0	0%	2	4,34%	2	4,34%
MobileMedia	10	0	0%	2	20%	2	20%
Oscache	42	0	0%	13	30,23%	13	30,23%
Quickserver	45	0	0%	16	35,55%	16	35,55%
Squirrel	41	1	2,44%	30	73,17%	31	75,61%
Totales:	345	14	4,06%	115	31,30%	129	37,39%

Tabla 3. Resolución de FE

5.2 RQ#2: ¿Solucionar las Feature Envy permite solucionar otros code smells?

Para responder la research question se realiza una comparación entre el listado de code smells generado por JSpIRIT a partir del código original de cada proyecto con respecto al generado luego de realizar el refactoring. Se analizan únicamente los code smells detectados por JSpIRIT que estén relacionados con la FE que son God Class, Intensive Coupling, Shotgun Surgery, Tradition Breaker y Data Class.

Los smells solucionados al menos una vez fueron God Class, Shotgun Surgery e Intensive Coupling. Para el caso de God Class, se comprende el bajo porcentaje obtenido (4,35%) debido a que la mayoría de las Feature Envy, como se vio en la RQ#1, fueron movidas completamente, por lo que, en caso de ser una God Class, simplemente se movió este code smell a una nueva clase, sin solucionarlo. En cuanto a Intensive Coupling, el bajo porcentaje (2,70%) cobra sentido al comprender que es necesario que la FE refactorizada sea a su vez una Intensive Coupling y la clase a la que es movida sea la misma que genera Intensive Coupling. En cuanto a Shotgun Surgery, el bajo porcentaje (3,54%) debido a que el número de proveedores para que este smell se genere debe ser alto (CC > MANY), con lo cual es poco probable que un método afectado por Shotgun Surgery sea también considerado FE.

En segundo lugar, se analizaron los smells creados luego de refactorizar todas las FE posibles de un sistema. La cantidad de code smells adicionales afectados negativamente también es muy baja. Esto se debe principalmente al bajo porcentaje de FE

solucionadas para cada sistema. Los smells creados al menos una vez fueron Tradition Breaker, Data Class y God Class.

Dado el análisis realizado es posible concluir que el enfoque no posee un alto impacto en la solución de otros code smells como consecuencia de la solución de FE.

5.3 RQ#3: ¿Son complementarias las soluciones identificadas por el enfoque propuesto y por JDeodorant en la refactorización de Feature Envy?

El objetivo de esta evaluación es comparar el enfoque propuesto con la herramienta JDeodorant. Para responder esta pregunta se analizan los resultados teniendo en cuenta dos universos diferentes de soluciones: Las Feature Envy resueltas por ambas herramientas simultáneamente; y Cantidad total de Feature Envy resueltas.

Se utilizaron los 10 proyectos presentados inicialmente en el caso de estudio. Es importante aclarar que JDeodorant solo identifica las Feature Envy que soluciona. En cambio, JSpIRIT genera todas las FE de un proyecto, y luego se le aplica el enfoque para definir si existe solución. Por lo tanto, la intersección de FE identificadas resulta en un conjunto de métodos mucho más reducido, debido a que JDeodorant identifica una cantidad mucho menor (47) de FE que JSpIRIT (345).

Por esta razón, JDeodorant tiene un 100% de efectividad en la identificación de un refactoring para la solución del code smell. De todas formas, el enfoque propuesto consiguió obtener una solución en el 90% de los casos.

Posteriormente, se analizó el universo completo de FE (identificadas por JDeodorant y/o por JSpIRIT). JDeodorant identifica 47 FE y resuelve la totalidad. Por su lado, JSpIRIT identifica 345 y resuelve 129. Analizando, el porcentaje de resolución del universo total considerado, JDeodorant resuelve un 12,30%, mientras que JSpIRIT u 33,77%. Éste análisis permite concluir que ambas herramientas son complementarias para lograr mejores resultados en la resolución de FE de un sistema. Los resultados confirman que la intersección de ambos universos de resoluciones es pequeña en comparación al universo total, y que ambas herramientas resuelven diferentes Feature Envy por su estrategia de identificación, por lo que el porcentaje de resolución total se ve significativamente mejorado al aplicar ambas herramientas simultáneamente.

6 Conclusiones

Los code smells son síntomas útiles para la identificación de problemas estructurales de un sistema que se relacionan con problemas de modificabilidad. El code smell Feature Envy es considerado el síntoma relacionado con problemas de acoplamiento y cohesión. Es un método que parece más interesado en los datos de otra clase que en los de su propia clase. Este problema puede ser solucionado aplicando el refactoring Move Method únicamente, o aplicando Extract Method y Move Method en conjunto.

En este trabajo se presentó un enfoque basado en heurísticas que ofrece al desarrollador más de una solución en caso de ser posible, y analizando oportunidades de realizar una extracción de código del método previo a ser movido a otra clase. Analiza la necesidad de mover todo el método o una porción del mismo y ofrece diferentes al-

ternativas de solución acompañada con las métricas relacionadas a la Feature Envy para que el desarrollador seleccione la más adecuada a su sistema. Para poder validar los beneficios del enfoque se realizó un caso de estudio con tres research questions que permitieron validar la propuesta presentada.

7 References

1. Ali, O., Katsuro I., Takashi I., Naoya U. c-JRefRec: Change-Based Identification of Move Method Refactoring Opportunities. Department of Computer Science and Software Engineering, CIT, UAE University, (2016).
2. Steidl, Daniela, and Sebastian Eder. Prioritizing maintainability defects based on refactoring recommendations. Proceedings of the 22nd International Conference on Program Comprehension. ACM, (2014).
3. Ali O., Marouane K., Mel O., Houari S., Kalyanmoy D., Katsuro I., MORE: A Multi-Objective Refactoring Recommendation Approach to Introducing Design Patterns and Fixing Code Smells. Journal of Software: Evolution and Process (2017).
4. Bavota G., Oliveto R., Gethers M., Poshyvanyk D., De Lucia A. MethodBook: Recommending Move Method Refactorings via Rational Topic Models IEEE Transactions on Software Engineering, (2014).
5. Ben-Menachem, Mordechai, and Marliss. Software Quality: Producing Practical, Consistent Software, Slaying the Software Dragon Series. Boston, MA: International Thomson Computer Press, (1997).
6. Berra, Iñaki and Zuliani Held, Santiago. Refactorización Automatizada para la Eliminación de Brain Methods Tesis de grado. Facultad de Ciencias Exactas, UNICEN (2015).
7. Fokaefs, M. and Tsantalis, N. "JDeodorant: Identification and Removal of Feature Envy Bad Smells" Department of Applied Informatics, University of Macedonia (2007).
8. Fowler, Martin. Refactoring: improving the design of existing code. Pearson Education India, (2002).
9. Garcia, Rubio, and Mario Piattini. Calidad en el desarrollo y mantenimiento del software. RA-MA (2003).
10. International Organization for Standardization. ISO 8402: 1994: Quality Management and Quality Assurance-Vocabulary. International Organization for Standardization, (1994). [12]
11. Lanza, Michele, and Marinescu, Radu. Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer Science & Business Media, (2007).
12. Murphy-Hill, Emerson, and Andrew P. Black. Breaking the barriers to successful refactoring. Software Engineering. ICSE'08. ACM/IEEE 30th International Conference on. IEEE, (2008).
13. Qualitas web page: http://qualitascorpus.com.
14. Sales V., Terra R., Miranda L. F., Valente M., JMove: Seus Métodos em Classes Apropiadas. Departamento de Ciencia da Computaçao, Universidade General de Minas Gerais, (2013).
15. Senthil Murugan, C. and Prakasam, S. Ph.D. A literal Review of Software Quality Assurance. International Journal of Computer Applications Volume 78. No.8 (2013).
16. Vidal, Santiago A., Marcos, Claudia, and Díaz-Pace, J. Andrés. An approach to prioritize code smells for refactoring. Automated Software Engineering : 1-32 (2014).
17. Yamashita, A. Code Smells as system-level indicators of maintainability: An empirical study. The Journal of Systems and Software, Elsevier, (2013).