Pedro A García Medina

Estrategia para la detección automática de bad smells basada en métricas de software

Tesis de maestría en ingeniería de sistemas y computación

Asesora:
Silvia Takahashi, Ph.D

Universidad de los Andes
Facultad de Ingeniería
Departamento de Sistemas y Computación
Bogotá D.C, Colombia
Diciembre de 2006

UNIVERSIDAD DE LOS ANDES

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

MAESTRÍA EN INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

Autor:	Pedro A García Medina pa.garcia29@egresados.uniandes.edu.co
Título de la tesis:	Estrategia para la detección automática de bad smells basada en métricas de software.
Fecha:	Diciembre de 2006
Grupo de investigación:	Construcción de Software
Asesora de tesis:	Silvia Takahashi, Ph.D
Palabras claves:	bad smell, métricas de software, detección automática, estrategia

Uno de los principales problemas que se tiene a la hora de llevar a cabo procesos de detección automática de bad smells, basados en métricas de software, es que las estrategias diseñadas para ello, aunque sean correctas -o incorrectas-, no presentan una aproximación ordenada ni formal del problema. Esto causa que no exista claridad en lo que se quiere detectar, en las métricas que se van a usar, y en la forma en que se van a usar.

Esta tesis pretende proponer una estrategia que sirva para abordar tales procesos de forma que las falencias mencionadas no se presenten.

Dedicatoria

A mi mamá. Por estar pendiente no solo de las cosas grandes, sino también, y especialmente, de las pequeñas, que muchas veces son las más importantes. Por su amor, comprensión, y respeto.

A todos los que hacen las cosas bien; o, al menos tratan de hacerlo. Con 'bien' me refiero a de forma honesta, limpia; actuando con ética, con ganas, sin miedo; pensando no solo en el propio bienestar; cuidando a la madre naturaleza; y sí, ¡bien hechas!.

A todos los de mente abierta.

Agradecimientos

Al profesor Germán Bravo, por haberme dado la oportunidad de trabajar para él como asistente graduado, sin lo cual tal vez no hubiera emprendido el camino de la maestría.

A la profesora Silvia Takahashi, por brindarme la orientación y apoyo necesarios para el desarrollo y culminación de este trabajo de tesis.

A Carlos Angarita, por su colaboración en cuanto a ideas, y por hacer más fácil mi trabajo de desarrollo de la aplicación de software.

A Pablo Montes, por su colaboración con el arreglo de las 'fallas' en su herramienta XScore.

A aquellos profesores que durante el transcurso de la maestría me transmitieron sus conocimientos, de forma objetiva, imparcial, con entusiasmo y sin reservas; y que ayudaron así a formar las bases necesarias para la consecución de este objetivo. Entre ellos: Milton Quiroga, José Abásolo, Silvia Takahashi, Olga Lucía Giraldo, Francisco Rueda.

A todos los autores consultados y citados en este documento, pues sus investigaciones formaron la base teórica de esta tesis.

Tabla de contenido

ORGANIZACIÓN DEL DOCUMENTO 8

¿CÓMO LEER ESTE DOCUMENTO? 8

1. INTRODUCCIÓN 10

MOTIVACIÓN 11

1.1 MARCO TEÓRICO Y ANTECEDENTES 12

1.1.1 Bad Smells 12

Taxonomía de bad smells 13

1.1.2 Detección de bad smells y refactoring 13

1.1.3 Métricas de software 16

¿Para qué usar métricas? 16

Clasificación de métricas 17

1.2 ANTECEDENTES 18

Trabajos relacionados que utilizan otros enfoques 20

2. DESCRIPCIÓN DEL PROBLEMA 21

2.1 PROBLEMAS CON EL USO DE MÉTRICAS EN LA DETECCIÓN DE BAD SMELLS 21

2.2 PROBLEMA GENERAL 22

2.3 BAD SMELLS ABORDADOS 22

2.3.1 Middle man 22

2.3.2 Inappropriate intimacy 23

2.4 JUSTIFICACIÓN 24

¿Por qué se escogieron éstos bad smells? 24

¿Por qué es valioso este trabajo? 25

2.5 OBJETIVOS DEL TRABAJO 26

Objetivos específicos 27

3. DESCRIPCIÓN DE LA SOLUCIÓN 28

3.1 ESTRATEGIA DE DETECCIÓN 28

3.1.1 Esquema general de solución 28

1. Escogencia o desarrollo de la representación 28

2. Definición de la estrategia de detección 30

3. Aplicación de la estrategia de detección 30

3.1.2 Definición de la estrategia de detección 30

1. Caracterización del bad smell 30

2. Escogencia de las métricas 30

3. Interpretación de las métricas 32

4. APLICACIÓN DE LA SOLUCIÓN A LOS CASOS ABORDADOS 34

4.1 A. INAPPROPRIATE INTIMACY 34

A.2 Definición de la estrategia de detección 34

A.2.1 Caracterización del bad smell 34

A.2.2 Escogencia de las métricas (GQM) 35

A.2.2.1 Definición de objetivos 35

A.2.2.2 Derivación de preguntas 35

A.2.2.3 Selección de métricas a usar 35

A.2.3 Interpretación de las métricas 47

A.2.3.1 Definición de filtros 47

A.2.3.2 Correlación de métricas....49
A.3 Aplicación de la estrategia de detección....49
4.2 B. MIDDLE MAN....49
B.2 Definición de la estrategia de detección....49
B.2.1 Caracterización del bad smell....49
B.2.2 Escogencia de las métricas (GQM)....52
B.2.2.1 Definición de objetivos....52
B.2.2.2 Derivación de preguntas....52
B.2.2.3 Selección de métricas a usar....53
B.2.3 Interpretación de las métricas....60
B.2.3.1 Definición de filtros....60
B.2.3.2 Correlación de métricas....61
B.3 Aplicación de la estrategia de detección....62
4.3 C. FEATURE ENVY....62
C.2 Definición de la estrategia de detección....62
C.2.1 Caracterización del bad smell....62
C.2.2 Escogencia de las métricas (GQM)....63
C.2.2.1 Definición de objetivos....63
C.2.2.2 Derivación de preguntas....63
C.2.2.3 Selección de métricas a usar....64
C.2.3 Interpretación de las métricas....68
C.2.3.1 Definición de filtros....68
C.2.3.2 Correlación de métricas....69
C.3 Aplicación de la estrategia de detección....69

5. HERRAMIENTA DE SOFTWARE PARA LA IMPLEMENTACIÓN DE LA PROPUESTA....70

5.1 DESCRIPCIÓN DE LA HERRAMIENTA....70
Descripción de los paquetes....70
Descripción de las clases....72
5.1.1 ¿Qué hace?....75
5.1.2 ¿Cómo lo hace?....75
5.1.3 Características importantes....76
Cosas para arreglar en futuras versiones....77
5.2 MANUAL DE USO....77
5.2.1 Manual de la herramienta genérica....78
5.2.2 Manual de la herramienta para detección de Inappropriate intimacy y Middle man....78

6. CONCLUSIONES....82

6.1 ANÁLISIS DE RESULTADOS....82
6.1.1 En cuanto al objetivo principal de la tesis....82
6.1.2 En cuanto a las estrategias propuestas para cada uno de los bad smells....82
6.2 ANÁLISIS DE LA SOLUCIÓN....82
6.3 TRABAJO FUTURO....84

REFERENCIAS Y BIBLIOGRAFÍA....85

GLOSARIO....88

ANEXOS....90

A.1 RESUMEN DE NOTACIÓN....90
A.2 PROPIEDADES DE LOS CONJUNTOS SOBRE LOS QUE SE TRABAJA....91
A.3 SINTAXIS....92
A.4 ENCUESTA ENTRE DESARROLLADORES....95
Motivación....95

Objetivos.....*95*
Características.....*95*
Composición de la encuesta.....*97*

Organización del documento

Este documento está organizado en seis grandes partes: introducción (que incluye el marco teórico y los antecedentes), descripción del problema abordado, descripción de la solución propuesta, aplicación de la solución a los casos particulares escogidos, análisis de resultados, y conclusiones.

En la primera parte se hace una introducción general al tema tratado, sin ahondar en los conceptos, con el fin de que el lector se haga una idea rápida de lo que va a encontrar en este documento. También se expone la motivación que se tuvo para emprender este trabajo de tesis. Se presenta el marco teórico en el que se basa el trabajo desarrollado. Allí se dan las definiciones, conceptos y propuestas que han sido el resultado del trabajo de otros autores; y se exponen los antecedentes más relevantes.

En la segunda sección se presenta la descripción del problema que se aborda en este trabajo

En la sección de descripción de la solución se presenta la estrategia propuesta. Se incluyen: la descripción detalla de la propuesta, una justificación de la misma, y la implementación que se llevó a cabo con el fin de lograr los objetivos propuestos.

En la siguiente sección se lleva a cabo la aplicación de la estrategia propuesta a los casos particulares seleccionados.

Luego aparece una sección dedicada al análisis de los resultados obtenidos.

Por último, se listan las conclusiones y los resultados relevantes del trabajo y se insinúa de qué forma se podría continuar esta línea de investigación.

¿Cómo leer este documento?

Para una persona que ya tenga conocimiento del tema de los *bad smells* y de los conceptos relacionados no sería necesario leer la parte del marco teórico; aunque sí debería revisar los antecedentes que se mencionan.

Lo mínimo que el eventual lector debería tener en cuenta son las secciones de descripción de la solución y aplicación de ésta a los casos particulares, ya que estas constituyen el eje central de este trabajo.

A lo largo del documento se hace uso de palabras y frases en inglés, dado que su traducción no es conocida exactamente, no encaja bien en el contexto, o no resulta elegante. En otros casos se hace simplemente porque la documentación original está en inglés, y la forma de buscar referencias al respecto es haciéndolo en el idioma original.

Al final del documento se encuentran dos secciones: el glosario, y el resumen de notación. En el glosario están las definiciones de algunos conceptos importantes que es preciso encontrar con

facilidad. Muchas de éstas están en el contenido del documento, pero se colocan allí para tener un acceso rápido. En el resumen de notación, se hace un compilado de la notación y las fórmulas utilizadas.

1. Introducción

Con la llegada de la programación orientada por objetos también lo hicieron las promesas de que el uso de este paradigma de programación, además de facilitar el desarrollo de las aplicaciones gracias a la facilidad de trasladar conceptos del mundo a conceptos de software y a mecanismos como el encapsulamiento de datos, la herencia y binding[1] dinámico, haría que tales aplicaciones fueran de hecho bien estructuradas, flexibles y fáciles de entender, extender y mantener **[24]**. Pero, como es de suponer, el simple hecho de programar orientado por objetos no es suficiente para lograr tales beneficios.

Aunque se dice que son principalmente las aplicaciones de legado las que presentan problemas como difícil mantenimiento y reutilización, porque se pensaba que simplemente conocer la sintaxis y los conceptos de la tecnología orientada por objetos era suficiente para producir buen software **[24]**, estos mismos problemas se siguen presentando en aplicaciones desarrolladas hoy en día.
Los características benéficas que se espera de la programación orientada por objetos son el resultado de buenos diseños, buenas arquitecturas, y del uso estándares de codificación.
Darse cuenta de esto ha llevado a que prototipos de diseño y arquitectura que han sido probados como exitosos, sean definidos y documentados en la forma de patrones de construcción de software. Así mismo, se han definido antipatrones[2] que tipifican prácticas erróneas comunes en este proceso de construcción.

Otra forma de capturar los conceptos de algunas malas prácticas de programación comunes se dio con la definición de los *bad smells* por parte de Fowler y Beck **[1]**. Estas definiciones muestran los síntomas que dan como resultado ese 'mal olor' en el código fuente de las aplicaciones. Sin embargo, a tales síntomas no se les incluyó una cuantificación que permita en todos los casos decir que algún *bad smell* está presente.
Esto dio pie a que durante algún tiempo se pensara que la detección de los *bad smells* era una tarea que sólo podría llevar a cabo una persona experimentada, con el conocimiento y la intuición suficientes para localizarlos y, dado el caso, corregirlos.

No obstante, tal idea ha sido erradicada gracias al trabajo de varias personas que han demostrado que sí es posible hacer de forma automática la detección y corrección de algunos de los bad smells tipificados. Para hacer posible la detección automática fue necesario poder hallar esa cuantificación faltante en la definición de los *bad smells*. La solución, en principio obvia, se obtuvo mediante el uso de métricas de software y de la correlación entre ellas. Ya se han llevado a cabo varios trabajos usando este enfoque, así como otros totalmente diferentes.
La principal debilidad de este enfoque es la falta de justificación de la selección de las métricas que se van a usar; y con ello la falta de formalización del uso específico dado a éstas. Además, aún no existen valores estándar para esas métricas; es más, ni siquiera existe un consenso sobre qué métricas usar para cada caso, por lo que cada autor define las métricas que va a usar y cómo las va a correlacionar para llevar a cabo la detección.

---

[1] Binding. Es la asociación de un nombre con una clase **[23]**.

[2] Antipatrones. Son, en concepto, similares a los patrones, en el hecho que documentan soluciones recurrentes a problemas recurrentes; sin embargo su uso produce soluciones negativas **[15]**.

El esfuerzo invertido en detectar los bad smells se hace con el fin de que, una vez hecho esto, se pueda proceder a hacer la reestructuración del código necesaria para eliminar el problema. A este proceso se le conoce como Refactoring.
El alcance de esta tesis está en la detección automática, sin llegar a hacer refactoring del código.

El objetivo general del trabajo es presentar una propuesta de una estrategia por seguir en los casos de detección automática de *bad smells*, implementándola para los casos *inappropriate intimacy* y *middle man*. Se espera con esto contribuir a facilitar el proceso completo de *refactoring* de software, lo que a su vez servirá para lograr códigos fuentes que sean más fáciles de entender, mantener, y escalar.

Se debe resaltar que este trabajo está enmarcado dentro de un proyecto más amplio en el cual se estudia el problema del mantenimiento de software ahondando en el problema de la detección de malas prácticas de programación. Es la tercera tesis de maestría en ingeniería de sistemas de la universidad de los Andes que aborda este tema, y hay otras dos que se están desarrollando actualmente (a la fecha de desarrollo de esta).

Motivación

Hay varias causas y factores que hacen del tema de la detección de *bad smells* un tópico interesante de abordar.

+ Este es un tema polémico en el ámbito de la ingeniería de software ya que no se ha llegado a un acuerdo sobre los enfoques apropiados para abordar el problema, por lo cual, toda nueva investigación bien fundamentada es un aporte al objetivo tácito de llegar a un punto en el que se cuente con técnicas depuradas y definiciones y valores precisos que sirvan para llevar a cabo éste proceso.

+ El hecho de poder continuar con una línea de investigación establecida en la Universidad es interesante, porque es una forma de hacer que el trabajo desarrollado no sea perdido, sino que por el contrario, sirva de base para nuevas propuestas como la presentada en este trabajo.

+ El resultado de este trabajo podría ser utilizado para validar otros enfoques de detección de *bad smells*.

+ Investigar y proponer ideas nuevas sobre un tema, al mismo nivel que se está haciendo en el mundo, es una buena motivación para emprender un trabajo de tesis.

1.1 Marco teórico y antecedentes

En esta sección se presentan la teoría que forma la base del tema tratado en este trabajo con el fin de que el lector se familiarice con los conceptos relevantes. También se incluyen menciones a trabajos que han sido desarrollados en el mismo tema, lo cuál servirá para hacerse una idea del estado del arte de la detección automática de *bad smells*.

1.1.1 Bad Smells

Bad Smell es un término coloquial acuñado por Fowler para designar ciertas estructuras en el código fuente que, debido a las malas prácticas de programación con que se produce, sugieren que se debería llevar a cabo un proceso de refactoring[3] **[1]**. Se conocen también con el nombre de fallas de diseño (design flaws) u olores del código (code smells).
Hay que tener en cuenta que un bad smell no es lo mismo que un error en el código. Se podría decir que un código sin errores es un código correcto, mientras que un código sin bad smells es además elegante, y en algunos casos, eficiente. Por ejemplo: una clase en cuyo código falte un ";" al final de una sentencia presenta, obviamente, un error. Si una clase, sin errores en el código, tiene demasiadas líneas de código y reúne conceptos separados del mundo que quiere representar, tiene algo que 'huele mal'.

El grueso del aporte de Fowler consistió en darles nombres a los *bad smells*, crear una taxonomía de éstos, y sugerir los *refactorings* que se deberían aplicar para eliminar cada uno de ellos. Esto sirvió para que la comunidad de la construcción de software tuviera un estándar para referirse a estas fallas en el código. Sin embargo, esta definición es totalmente cualitativa, ya que en ningún caso se determinan rangos o niveles de los criterios evaluados en los que el programador/diseñador se pueda basar para decidir si el código presenta un *bad smell*. Por ejemplo, la definición del *bad smell long method* dice que "si el programador debe cambiar de contexto varias veces para entender lo que hace el método ...,o, siempre que se sienta la necesidad de comentar[4] algo ... Todo lo que hay que hacer para acortar el método es Extraer Método[5]" **[1]**.

Hay varias taxonomías para la clasificación de los *bad smells*, entre ellas la presentada por el mismo Fowler, y otras como las de Wake **[3]** y Mäntylä **[4]**. En este documento se presenta la propuesta por Mika Mäntylä, ya que clasifica los *bad smells* (los mismos propuestos por Fowler) desde una perspectiva que se relaciona más fácilmente con el mundo de la programación orientada por objetos.

A continuación se presenta la taxonomía de bad smells de Mäntyla **[4]**.

[3] Refactoring podría traducirse como reestructuración

[4] Comentar se refiere aquí a adicionar comentarios al código fuente para hacer que se entienda más fácil

[5] Extraer Método es un refactoring que consiste en extraer de un método largo que hace mucho, uno o más métodos que hacen cosas más puntuales.

Taxonomía de bad smells

- **The bloaters**[6]
  Son los que representan a los bad smells que han crecido tanto que no pueden ser fácilmente manejados. Los que pertenecen a esta categoría son: *long method, large class, primitive obsession, long parameter list*, y *data clumps*.

- **The object-orientation abusers**
  En esta categoría se incluyen los casos en los que por no explotar al máximo las posibilidades de la programación orientada por objetos, las soluciones desarrolladas presentan tales 'olores'. En esta categoría están: *switch statements, temporary field*, *refused bequest, alternative classes with different interfaces*, *parallel inheritance hierarchies*.

- **The change preventers**
  La característica de estos es que impiden que el software sea fácilmente modificado y extendido (aunque esta característica aplicaría en general a todos). En esta categoría están: *divergent change y shotgun surgery***.**

- **The dispensables**
  Estos representan algo innecesario y que podría o debería ser eliminado del código. En esta categoría están: *lazy class, data class, duplicated code, speculative generality*.

- **The encapsulators**
  A esta categoría pertenecen dos bad smells que son en cierta forma opuestos, ya que el decremento en uno significa el incremento en el otro; éstos son: *message chains* y *middle man*. Se agrupan en esta categoría porque los dos tienen que ver con el abuso de la propiedad de encapsulamiento propia de la programación orientada por objetos.

- **The couplers**
  Son los que se descubren cuando hay un alto nivel acoplamiento entre clases. A esta categoría pertenecen: *feature envy* e *inappropriate intimacy*.

- **Otros**
  Los que no fueron agrupados en alguna categoría: *incomplete library class, comments*.

1.1.2 Detección de bad smells y refactoring

El proceso de detección es un paso intermedio y necesario en el proceso grande de mejoramiento de la calidad del software. La salida de éste son los sitios del código que deberían ser reestructurados, es decir, sobre las cuales se debería llevar a cabo el proceso de *refactoring*.

[6] Bloater podría traducirse como 'hinchador', es decir que hace que el código se vea más largo de lo que parecería normal.

Refactoring es definido como "una técnica para reestructurar un cuerpo de código existente, alterando su estructura interna sin alterar su comportamiento externo" **[2]**. Esta técnica incluye la aplicación de transformaciones también llamadas *refactorings*, que aplicadas en conjunto forman el proceso completo. En la figura 1 se muestra cómo para un observador el comportamiento externo de un cierto código es el mismo, mientras que su estructura interna fue modificada.
El principal objetivo de este proceso es incrementar la mantenibilidad del código, además de mejorar su legibilidad, estructura y desempeño **[16]**.

La mantenibilidad del código se define como la capacidad que tiene el software de ser modificado; y consiste a la vez de las siguientes características **[7]**:

- Capacidad que tiene el software para ser analizado en busca de deficiencias o fallas (analyzability)
- Capacidad del software para permitir modificaciones (changeability)
- Capacidad del software de minimizar los efectos inesperados a partir de una modificación
- Capacidad del software para ser validado (testability)

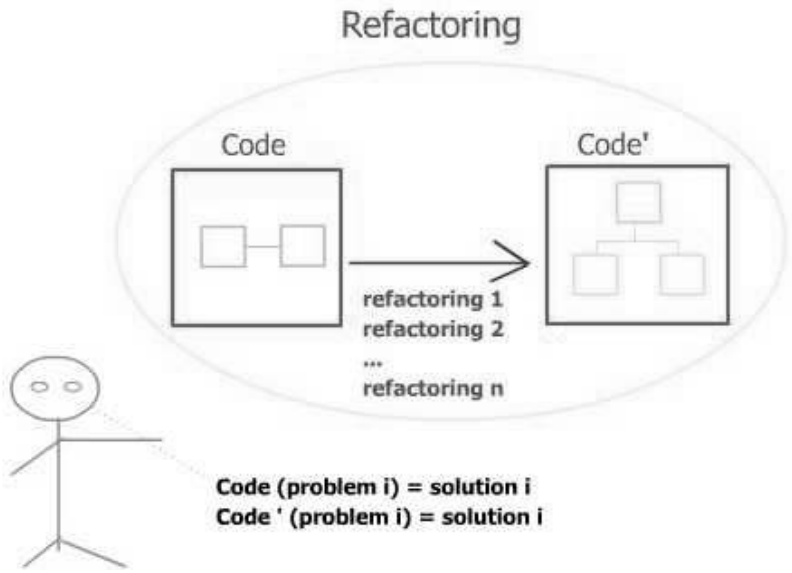


Figura 1. Proceso de refactoring

Otros objetivos del proceso son:

- Hacer que el código sea más consistente, claro y elegante; y por lo tanto más fácil de entender.
- Corregir malas prácticas de programación introducidas en el código.
- Cambiar la estructura y diseño del código.

- Remover código 'muerto'[7]

La figura 2 muestra cómo se involucra la detección de *bad smells* en el proceso de *refactoring* **[8]**.

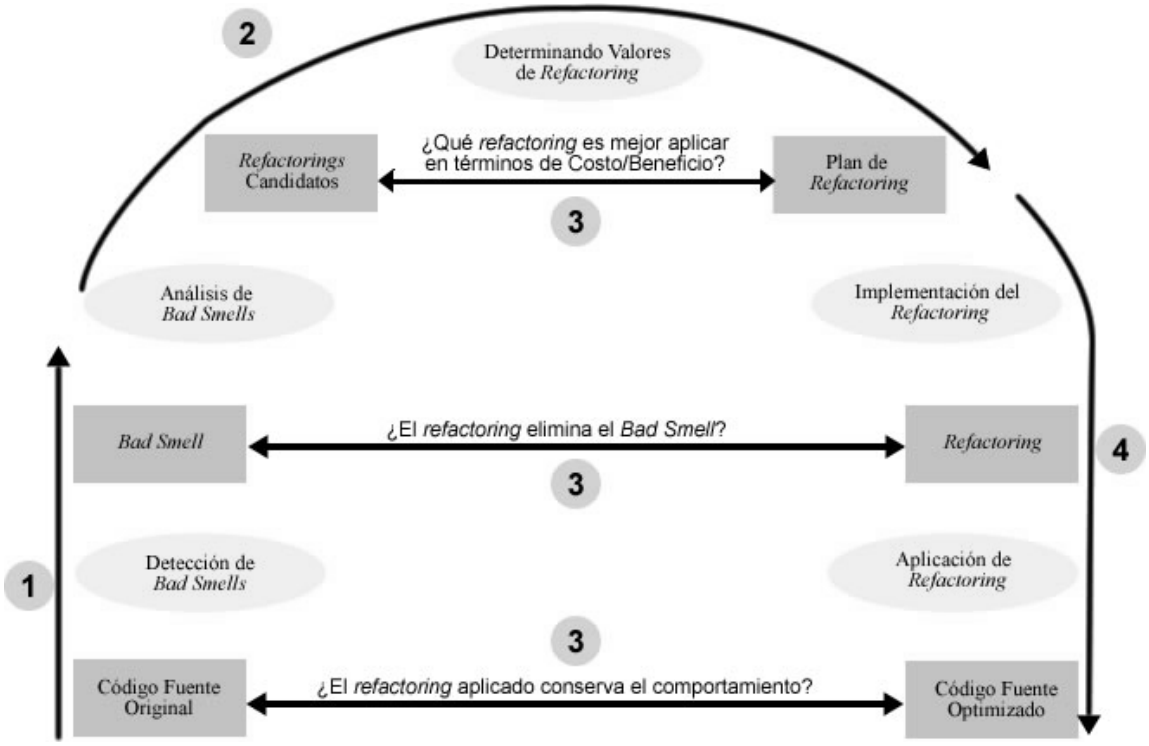


Figura 2. Detección de bad smells dentro del proceso de refactoring

Paso 1. Por medio de la detección de *bad smells* se identifican las posibilidades de hacer *refactoring*.

Paso 2. Dependiendo de los 'olores' detectados en el paso 1 se proponen los posibles *refactorings*.

Paso 3. Análisis del *refactoring* en los términos de cuál es mejor, de si se elimina el *bad smell*, y de si el comportamiento de la aplicación se mantiene.

Paso 4. Aplicación de los *refactorings*.

En la práctica se ha visto que muchas veces no es posible eliminar del todo los malos olores ya que al aplicar *refactoring* para eliminar el *bad smell* X, se puede incrementar el Y. Un ejemplo claro de esta situación se presenta entre los dos *bad smells* que pertenecen a la categoría de los encapsuladores (The encapsulators): al tratar de eliminar el *bad smell message chains* (ver definición en el glosario) en una clase A, es posible que se termine creando una clase intermedia

[7] Código muerto se denomina al que se ha detectado que no es usado en la aplicación.

para gestionar los mensajes entre la clase A y otras clases, lo cual es síntoma del *bad smell middle man* (ver definición en el glosario). Este ejemplo se ilustra en la siguiente figura.

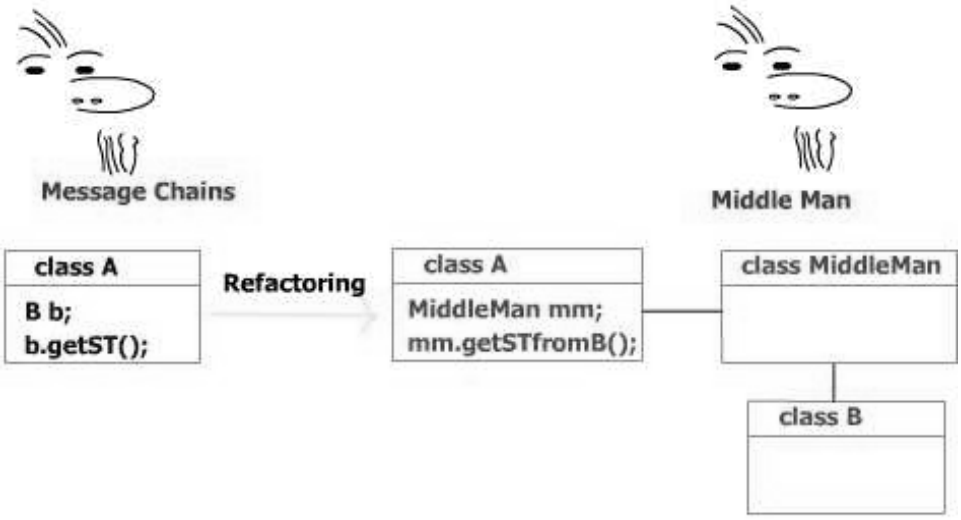


Figura 3. Minimizar un bad smell puede conllevar a que aparezca o se incremente otro

1.1.3 Métricas de software

A lo largo de la investigación en el tema de la detección automática de *bad smells*, se ha concluido que la falta de cuantificación en la definición de éstos se puede resolver con el uso de métricas de software.
Una métrica de software es la medida de alguna propiedad de una parte del software o de sus especificaciones **[5]**. Por lo tanto, es posible obtener información cuantitativa derivada de ellas. Las métricas "actúan como indicadores y proveen información para que la gente pueda tomar decisiones informadas y elecciones inteligentes" **[27]**.

Hay dos grupos principales de clasificación de métricas:

- Las que se usan como parte de la administración de un proyecto.
- Las que reflejan las propiedades del producto en sí mismo.

¿Para qué usar métricas?

Las métricas de software se han vuelto un elemento importante para la buena ingeniería de software. Éstas son usadas también para **[6]** determinar si los requerimientos están completos y

son consistentes, si el diseño es de buena calidad, y si el código es muy complejo; también para determinar la complejidad del mantenimiento y facilitar los análisis del impacto de cambios en el producto.

Según Humphrey, la medición de los productos de software tiene diferentes funciones, según las cuales se podrían clasificar las métricas así **[9]**:

+ **Control**. Métricas que son usadas para monitorear el proceso de construcción de software, y para identificar áreas donde se requieren acciones correctivas.

+ **Evaluación.** Estas son usadas en el proceso de toma de decisiones con el fin de establecer los puntos de partida del proceso y determinar si los estándares, objetivos y criterios de aceptación han sido alcanzados.

+ **Comprensión.** Son las que pueden ser recolectadas con el fin de aprender algo acerca del proceso de construcción y de los productos de software desarrollados

+ **Predicción.** Estas son usadas para crear una base sobre la que se puedan predecir medidas en futuros procesos.

Clasificación de métricas

Al igual que los *bad smells* tienen varias clasificaciones posibles, las métricas de software han sido organizadas en varias taxonomías. En este trabajo se tomó como referencia la taxonomía propuesta por El-Wakil y otros **[10]**:

*** Métricas externas**
Son las que tratan las propiedades visibles a los usuarios de un producto de software, tales como confiabilidad, funcionalidad, desempeño y qué tan usable es.

*** Métricas internas**
Son las que tratan las propiedades que son visibles únicamente para el equipo de desarrollo. Estas a su vez se subdividen en:

Métricas de especificación
Analizan las especificaciones del producto y proveen retroalimentación temprana sobre el producto en desarrollo.

Métricas de diseño
Pueden ayudar a refinar el diseño para evitar futuros problemas incluso antes de que se produzca el código.

Métricas de código
Son las que miden las propiedades del código.

Se tomó ésta clasificación porque presenta la categoría *métricas de código* que es precisamente en la que se agrupan las métricas que se van a manejar en este trabajo como parte de la estrategia propuesta; además, porque es la base de la tesis de su autor, la cual incluye una propuesta interesante: la posibilidad de llevar a cabo la detección de algunos *bad smells* en etapas

tempranas del ciclo de construcción de software basándose en métricas que podrían ser obtenidas en la etapa de diseño (y tal vez en la etapa de especificación), y así evitar que se trasladen al código final.

Hay otros criterios que se pueden usar para clasificar las métricas; por ejemplo: si se calculan en una clase o en varias, si el análisis que se hace es estático o dinámico, si son métricas elementales o compuestas[8], o si dependen del punto de vista del observador (quién las calcula) o no; y todos estos dan origen a otras taxonomías. La que se escoja, al igual que la –clasificación- de *bad smells*, depende del objetivo de su uso.

Las métricas que se van a usar en los casos específicos abordados en este trabajo se pueden clasificar como *métricas internas estáticas de código*.

1.2 Antecedentes

Dada la falta de cuantificación en su definición, la detección de los *bad smells* estaba hasta hace un tiempo condenada a ser una actividad manual y guiada por la necesaria experticia por parte de quien revisaba el código.

Esto se ha visto revaluado por varios trabajos como por ejemplo el de Marinescu[9] y el de otros autores, quienes han logrado dar definiciones formales de algunos *bad smells* basadas en métricas, y han construido herramientas que calculan tales métricas; lo cula sugeriría que la detección automática de los *bad smells* sí es posible.

En al ámbito global se pueden mencionar varios trabajos importantes, algunos de los cuales son tomados como base teórica para el desarrollo de esta tesis. Entre estos están:

***** El libro de Fowler **[1]** en el que se definen y clasifican los *bad smells*, además de sugerir los *refactorings* apropiados en cada caso.

***** Los trabajos de Radu Marinescu **[11]**, **[14]**, en los que se da la definición formal de una estrategia para detección de *bad smells*, se muestra la construcción de un sistema para llevar a cabo el proceso, y se da un importante bagaje teórico en esta línea de investigación.

***** Mika Mäntylä **[4]** desarrolló un trabajo en el que se da una taxonomía de los *bad smells* basada en los principios de la programación orientada por objetos, y se evalúa de forma empírica la relación de las métricas de software con los *bad smells*.

[8] Métricas compuestas se refiere a métricas que son resultado de la correlación -mediante diferentes fórmulas- de métricas simples

[9] Radu Marinescu. Autor de varios trabajos en el tema de métricas de software y detección de bad smells.

***** El trabajo de Matthew Munro **[18]**, que según sus propias palabras, fue un "intento de llevar a cabo la detección automática de bad smells". No se lograron resultados muy exactos (se encontraron muchos falsos positivos) debido a que la interpretación de los bad smells está sujeta a la informalidad de su descripción.

***** Otros:

* Java Quality Assurance by Detecting Code Smells. De Eva van Emden y Leon Moonen
* Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. De Ramanath Subramanyam y M.S. Krishnan

A nivel local, en el ámbito de la Universidad De Los Andes, hay específicamente dos trabajos anteriores, realizados por personas del grupo de construcción de software de la maestría en sistemas, que han establecido una línea de investigación y de trabajo en el tema de la detección y corrección de malas prácticas de programación.
Otro sirvió como apoyo a uno de éstos, y se usa como herramienta complementaria para el desarrollo de la propuesta de detección presentada en este trabajo.

A continuación se da una descripción de estos trabajos.

***** Detección de Bad Smells en aplicaciones JAVA utilizando métricas de software. Tesis de grado de Beatriz Florián **[8]**.
Como resultado de esta tesis se desarrolló una herramienta que se implementa como plug-in de Eclipse, y que permite, en conjunto con otros plug-ins previamente desarrollados (TeamInABox y Metrics) detectar los *bad smells*: *Large Class*, *Data Class*, *Long Method* y *Long Parameter List*.
La conclusión más importante a la que se llegó con este trabajo es que es posible encontrar fórmulas para formalizar técnicas, basadas en métricas de software, para detectar -automáticamente- la presencia de *bad smells* en el código.

***** XSCoRe, Representación de Código Fuente basada en XML como una Herramienta de Asistencia al Proceso de Ingeniería en Reversa. Tesis de Pablo Montes **[13]**.
El resultado del trabajo del señor Montes es una herramienta que representa en XML un código fuente JAVA dado. Esta es usada para el desarrollo de esta tesis, ya que su producto se toma como entrada de la herramienta construida.

***** Detección de *bad smells* en aplicaciones JAVA a partir de una representación *xml* del código fuente. Tesis de Carlos Angarita **[12]**.
Este trabajo dio como resultado una herramienta que de forma automática detecta la presencia de los *bad smells Feature envy* y *Low cohesion*. Esta herramienta toma como entrada la representación del código fuente en XML que es producida por la herramienta XSCoRe.
La principal conclusión de este trabajo es que es posible construir herramientas para detectar de forma automática la presencia de *bad smells* en el código, basadas en métricas de software.

Trabajos relacionados que utilizan otros enfoques

Existen en el área de la detección de bad smells otros enfoques diferentes al que de basa en el uso de métricas de software. Es importante tenerlos en cuenta porque la comparación de resultados utilizando varios enfoques podría eventualmente llevar a un consenso en el tema.

Algunos de estos son:

***** Kataoka y otros **[16]** desarrollaron una herramienta para encontrar lugares en el código propensos a sufrir *refactoring* (bad smells), basada en una aproximación que hace uso de invariantes de programa. Este análisis se hace de manera dinámica, y busca los puntos del código en donde una serie de invariantes definidos se cumple, para así decidir que cierto *refactoring* es aplicable.

***** Tom Tourwé and Tom Mens **[17]** mostraron cómo se pueden identificar las oportunidades de hacer refactoring, llevando a cabo la detección de bad smells por medio de una técnica llamada Meta-programación lógica. Ésta se basa en una fuerte simbiosis entre un lenguaje base orientado por objetos y un meta lenguaje declarativo que expresa la estructura del código en forma de reglas lógicas (al estilo prolog[10]), permitiendo que todas las entidades de éste sean accedidas por medio de predicados.

[10] Prolog: lenguaje de programación lógico.

2. Descripción del problema

En esta sección se dará a conocer el problema específico que se aborda, dando también la definición de los *bad smells* que se escogieron como objetivo, así como la justificación que hay detrás de este trabajo.

El primer problema que se afronta viene como consecuencia del enfoque utilizado para las soluciones previamente propuestas, y que también se va a definir aquí como base de la estrategia: el del uso de métricas de software. A continuación se describen estos problemas.

2.1 Problemas con el uso de métricas en la detección de bad smells

El principal problema del uso de métricas de software en la detección automática de *bad smells* es que no hay medidas estándar, que indiquen que cierto resultado obtenido con una métrica signifique inequívocamente la presencia de un 'mal olor' en el diseño o en el código. Tampoco hay un consenso sobre qué métricas se deban aplicar en los casos en que se requiere de múltiples métricas o de métricas compuestas. Por ejemplo: no hay un número estándar de número máximo de líneas de código que deba tener una clase para que no se considere que ésta presenta el *bad smell large class* (ver definición en el glosario).

Debido a esto, quien desarrolla una herramienta para llevar a cabo la detección debe hacer ciertas suposiciones que, como es normal, pueden llevar a resultados no muy exactos. Por ejemplo: suponer que para detectar el *bad smell* bs es necesario calcular las métricas m1 y m2, y relacionarlas así: "existe bs si m > k, donde m = m1 / m2".

Además de la suposición de las métricas que se deberían usar, algunas veces es necesario inventar métricas cuya validez no puede ser sostenida más allá del contexto en el que se usan; es por esto que las aproximaciones de detección de diferentes autores varían ampliamente.

Otros problemas detectados por Marinescu [11] con el uso de métricas en general son:

- Hay un número muy grande de métricas, pero en muchos casos sus definiciones son imprecisas, confusas e incompletas; lo cual las hace difíciles de usar.
- En la mayoría de los casos no hay modelos de interpretación de las métricas, o estos son muy empíricos, lo cual crea obstáculos en la aplicabilidad y confiabilidad de éstas.
- A menudo, el uso de métricas aisladas no muestra la causa que origina un valor inusual de éstas. Es decir, pueden ayudar a detectar cierto bad smell, pero no lo que lo originó; por lo cual tampoco proveen información suficiente para la(s) trasformación que se debería aplicar.
- Hay una brecha entre los principios de diseño y las medidas del producto; es decir, entre lo que se mide y lo que es importante en el diseño.

× Debido al uso de representaciones incompletas[11], muchas veces se usan solamente métricas simples, dejando de lado valiosa información que se podría obtener con métricas compuestas o con referencias cruzadas de información, cuya obtención depende de la completitud de la representación usada.

2.2 Problema general

El problema de fondo que se busca ayudar a resolver es el de valoración y mejoramiento de la calidad de un diseño de software orientado por objetos. La valoración -en el enfoque utilizado- se hace mediante el uso de métricas de software que cuantifican ciertos aspectos del diseño, y está destinada a 'valorarlo' desde el punto de vista de la presencia o no de los denominados *bad smells* (no bad smells ⇒ buen diseño). El mejoramiento es el objetivo del proceso de *refactoring*, del cual la detección de *bad smells* hace parte, teniendo como objetivo señalar los lugares en que los *refactorings* deberían ser aplicados.

Como se mencionó, la valoración se hace con base en métricas de software; el problema detectado es que cada autor que aborda el tema lo hace de forma intuitiva y sin tener un marco común de referencia sobre el que se pueda evaluar la estrategia que propone para detectar algún *bad smell* en particular. En general, hacen uso de algunas métricas y definen condiciones que deben cumplir tales métricas para decidir si tal *bad smell* está presente o no; pero este enfoque tiene varias falencias:

- La elección de tales métricas no está justificada en forma explícita.
- No se dan definiciones formales de las métricas usadas.
- En muchos casos ni siquiera se define formalmente el *bad smell* que se quiere detectar, por lo que podría parecer que no se sabe bien qué es lo que se quiere encontrar

2.3 Bad smells abordados

Se da la definición de los *bad smells* para los cuales se busca hacer la detección automática. Esta definición, dada por Fowler, es bastante informal; sin embargo, sirve para hacerse una idea de lo que significa que uno de éstos este presente en el código.

En la informalidad de las definiciones se encuentra el primer problema a resolver: el enfoque a usar, y por lo tanto los resultados obtenidos, dependen en cierto grado del buen criterio de quien construye la herramienta de detección.

2.3.1 Middle man

Consiste en el hecho de que una clase delega la mayor parte de sus responsabilidades a otras clases; es decir, que muchos de los métodos de una clase delegan a otra clase para obtener

[11] Estas son representaciones del código (de más alto nivel que el código fuente mismo), a partir de las cuales se calculan las métricas.

alguna información, por lo que ésta clase estaría ejerciendo como un simple intermediario entre clases. **[19]**
Aunque esta forma de delegación de responsabilidades ayuda a mantener el principio de encapsulamiento, usarla en forma muy extendida podría conllevar una especia de burocracia entre los objetos.

Ejemplo:
Suponga que se tiene lo siguiente

```
public class MiddleMan{
	A a;
	B b;
	C c;

	public void foo() {
		a.doSomething();
	}

	public void goo() {
		b.doAnything();
	}

	public void hoo() {
		c.doNothing();
	}
}
```

Dado que la clase `MiddleMan` no implementa ninguno de sus métodos, sino que delega estas responsabilidades, se tendría aquí una clase que sólo está haciendo de intermediaria entre alguna clase que necesite ejecutar los métodos de las clases `A, B, C`, y éstas.

2.3.2 Inappropriate intimacy

Se da en el caso en el que dos clases están muy acopladas; es decir, que los métodos declarados en una de éstas usan los métodos y las variables de instancia de la otra. **[21]**

Muchas veces es necesario cierto nivel de acoplamiento; pero entre más grande sea éste, más difícil resulta de mantener el código, ya que un cambio en alguna de las clases acopladas conlleva cambios en la otra.

Ejemplo:
Suponga que se tiene lo siguiente

```
public class A {

	B b = new B();
	public int cosito;
```

```
	public void foo() {
		b.goo();
	}
}


public class B {

	A a = new A();

	public void goo() {
		...
	}

	public hoo() {
		int st = a.cosito;
	}
}
```

En este ejemplo es muy claro el acoplamiento entre las clases A y B, porque el 100% de los métodos de A usan métodos de B, y el 50% de los métodos de B usan las variables de instancia de A.

2.4 Justificación

En esta sección se pretende mostrar las razones que se tuvieron para el desarrollo de esta tesis, y hacer explícito el aporte que constituye este trabajo en el área de investigación de los *bad smells*.

Los varios trabajos, a la fecha desarrollados, que abordan el tema de la detección automática de *bad smells* basándose en el uso de métricas de software tienen un 'inevitable' componente de informalidad y de falta de estructuración, que no permite seguir esas mismas estrategias en cualquiera de los casos (*bad smells*). Tomar elementos que han sido probados en esos trabajos, agregarle algunos nuevos y formalizar y completar lo que antes existía en forma implícita e incompleta (en algunos casos), de forma que se cuente con una base y guía por seguir en subsecuentes procesos del mismo tipo, fue la principal motivación para llevar a cabo este trabajo.

¿Por qué se escogieron éstos bad smells?

Lo primero que se pretendía era que los bad smell escogidos para aplicar la estrategia propuesta no hubieran sido tomados como tema central de alguna investigación anterior, y que no se hubiera desarrollado una estrategia de detección automática de éstos (por lo menos no usando el mismo enfoque). Esto no significa que si existieran trabajos sobre el mismo tema éste no fuera un aporte

valioso, debido a que los resultados aquí obtenidos servirían como base de comparación para decidir el camino correcto de detección.
Luego de una investigación exhaustiva, se llegó a la conclusión de que los bad smells *middle man* e *inappropriate intimacy* cumplen con estos dos requisitos.

Dado que en nuestra línea de investigación ya había sido abordado el *bad smell low cohesion* (ver definición en el glosario) que, como su nombre lo indica, trata el tema del bajo de la baja cohesión -de una clase-, parecía natural escoger *inappropriate intimacy*, ya que trata el tema del acoplamiento entre clases (este *bad smell* bien podría haberse llamado *high coupling*); así se tendrían cubiertos los dos principales principios de la programación orientada por objetos: cohesión y acoplamiento; específicamente, la forma de mantener alta cohesión y bajo acoplamiento.

El *bad smell middle man* tiene que ver con el exceso de encapsulamiento que se presenta cuando la única función de una clase es ser intermediaria de llamados entre las clases que necesitan métodos de otras clases, y las que se los proveen. Por lo tanto al tratar éste se estaría buscando la forma de mantener niveles aceptables de encapsulamiento, que es otro de los principios base de la programación orientada por objetos.

¿Por qué es valioso este trabajo?

! Porque se va a formalizar una estrategia que pueda ser usada en los procesos de detección de detección automática de *bad smells*.

! Este trabajo se puede considerar novedoso porque, como ya se mencionó, no se encontraron trabajos anteriores que busquen la detección automática de los *bad smells* aquí abordados, manejando el enfoque de uso de métricas de software como criterio de decisión.

! Los *bad smell* escogidos no son de los que se pueden considerar como de fácil detección ya que:

- Su detección requiere de un estudio más profundo del código fuente. Es decir, que el análisis del código fuente no se puede limitar el estudio de una estructura de código por separado (un método o una clase), sino que se requiere recorrer todo el código fuente para hacer el análisis del mismo y poder obtener las métricas necesarias.

- Es necesario hallar varias métricas. La presencia de un *bad smell* no se puede decidir solamente con una métrica como en el caso de *large class*, sino que hay que hallar varias de ellas y correlacionarlas.

- Es necesario validar de alguna forma los resultados obtenidos al aplicar las fórmulas de correlación entre métricas.

! Los resultados de la estrategia propuesta y de la herramienta que la implementa pueden contribuir a:

+ Ayudar a completar el proceso de *refactoring* para eliminar el exceso de encapsulamiento y el alto acoplamiento, y así hacer que el producto final tenga un mayor grado de mantenibilidad.

+ Tener una base de comparación. Los resultados obtenidos en otros trabajos que usen diferentes enfoques (como por ejemplo meta-programación lógica o invariantes de programa), o el mismo enfoque pero que usen diferentes aproximaciones (como por ejemplo la inventada por Marinescu que usa SQL para hallar las métricas a partir de la representación del código en una base de datos **[22]**)

! En la selección de métricas usadas para cada caso abordado se usan métricas que ya han sido definidas y usadas en trabajos anteriores, pero en algunos casos se requiere dar un uso específico, algo diferente al originalmente planteado, a estas métricas. La definición de usos alternativos de métricas que han sido previamente probadas constituye otro aporte de este trabajo.

2.5 Objetivos del trabajo

Los objetivos que se buscan han sido mencionados de forma somera en las secciones anteriores; aquí se hacen explícitos.

La línea de investigación en la detección automática de *bad smells*, aunque ha tenido desarrollos significativos, aún es un campo interesante en el que quedan muchas cosas por hacer, y este trabajo pretende ser un aporte novedoso en el área.

El objetivo general de esta tesis es definir una estrategia general que pueda ser seguida en procesos en los que se quiere llevar a cabo la detección automática de *bad smells*; esto con el fin de apoyar las actividades de valoración de la calidad de un diseño, y posiblemente, el mejoramiento de las aplicaciones.
Al igual que trabajos anteriores, lo que se busca es "reducir la brecha entre afirmaciones cualitativas y cuantitativas" **[11]** acerca del diseño de una aplicación.

Debido a que el tema de la detección de *bad smells* y del uso de diferentes técnicas (como el uso de métricas de software; usada en este trabajo) para hacer esta detección aún carecen de definiciones formales y, por lo tanto, aceptadas en consenso por la comunidad de desarrolladores de software, es necesario plantear, desarrollar y validar tesis que, como en el caso de otras ciencias, permitan establecer por consenso tales definiciones. Este trabajo desarrollado puede servir como referente de comparación con otros trabajos que aborden los mismos *bad smells*, lo cual tal vez alguna vez conlleve a que haya un acuerdo sobre qué métricas usar y los niveles de éstas (rangos, límites) más allá de los cuales se pueda determinar que cierto éstos estás presentes.

Objetivos específicos

- Definir una estrategia de detección automática basada en el uso de métricas de software. Tal estrategia debe incluir métricas con definiciones precisas y modelos de interpretación de éstas que puedan ser aplicados en la evaluación de un diseño, y expresar los principios de diseño que se quieren evaluar como expresiones cuantificadas basadas en esas métricas **[11]**.

- Aplicar la estrategia propuesta para los casos específicos: *Inappropriate intimacy* y *Middle man*.

- Poner a prueba la estrategias propuestas. Para ello se desarrolla un sistema que las implementa y lleva a cabo la detección de los *bad smells* abordados. La detección de éstos se hará sobre código fuente JAVA. Lo que se espera de esta herramienta es que sugiera al desarrollador los sitios del código donde posiblemente están presentes los *bad smells*, dejando a su criterio la decisión final de un posible *refactoring*.

3. Descripción de la solución

En este capítulo se presenta la solución propuesta al problema que ha sido descrito en el capítulo anterior. Se define la estrategia propuesta para abordar el problema y, para cada *bad smell*, se detallan los pasos seguidos según el esquema descrito. En la medida en que es necesario, se incluyen los conceptos teóricos que no lo fueron en la sección que contiene el marco teórico del trabajo.

3.1 Estrategia de detección

En esta sección se describe la estrategia de detección propuesta. Para ello, se enumeran los pasos que la componen, y se detallan cada uno de éstos pasos.
Luego de haber descrito la estrategia, se tomará cada uno de los *bad smell* abordados y se mostrará la aplicación de ésta en cada caso.

La estrategia se enmarca en un esquema general de solución, el cual se describe antes de entrar en el detalle de la definición de la estrategia de detección.

3.1.1 Esquema general de solución

Se denomina **esquema general de solución**, porque describe los pasos que se podrían seguir para definir e implantar cualquier estrategia que se base en el uso de métricas de software como criterio de decisión en la detección de *bad smells*.

Los pasos que componen esta estrategia son:

1. Escogencia o desarrollo de la representación de código que se utiliza
2. Definición de la estrategia de detección
3. Aplicación de la estrategia

Los pasos uno y dos no necesariamente son llevados a cabo en ese orden, puesto que la representación que se utilice debe permitir que todas las métricas escogidas (en el paso 2) sean calculables a partir de ésta. Si esto no es posible, se debería tratar de expresar las métricas escogidas en términos más "sencillos" que sean calculables con la representación que se seleccionó.

1. Escogencia o desarrollo de la representación
Este paso consiste en decidir la representación que se va a usar como base para la extracción de las métricas. Esta es una representación del código fuente de más alto nivel -que el código mismo-, y permite tener una forma más eficiente de recorrerlo en la búsqueda de la información necesaria para la extracción de las métricas.

Aunque este paso podría ser parte de la estrategia misma, no sería práctico escoger una representación diferente para cada *bad smell*.
También sería posible obviar este paso y hacer el análisis para extracción de métricas sobre el código fuente mismo; sin embargo, esta opción ni siquiera se tuvo en cuenta dado que es computacionalmente impráctica, y ya se contaba con una herramienta que produce tal representación.

El objetivo que se busca en este caso con el uso de una representación es permitir un recorrido del código fuente, ya no a nivel de caracteres ni estructuras sintácticas, sino de las estructuras relevantes (métodos, clases) y de sus propiedades, para la extracción de las métricas.

Aunque el uso de una representación no es indispensable sí es muy recomendable porque:

* Disminuye la carga computacional que se requeriría en el caso de tener que hacer tal recorrido cada vez.
* Esta aproximación solo serviría en el caso de métricas simples, que son las que se pueden hallar analizando sólo una estructura del código (por ejemplo la métrica *líneas de código de una clase*). En el caso de métricas que requieran el análisis de varias estructuras simultáneamente, se volvería inmanejable.

Si se cuenta con una buena representación, esta se puede tomar como base para la detección de varios (si no todos) *bad smells*.

La representación escogida en este caso es la desarrollada por el Pablo Montes, como trabajo de tesis de maestría en la universidad de Los Andes **[13]**. Ésta es conocida como *xjava*, y la herramienta que la produce como *XScore*. Entre las principales características que tiene, y por las cuales fue escogida son:

+ Representación en XML del código fuente. Esto facilita el recorrido de la representación porque que ya existen herramientas para manipular y analizar código en XML (por ejemplo JDOM[12], JAXP[13])

+ La información del código fuente que almacena es suficiente para hallar las métricas necesarias.

+ Hay antecedentes de uso exitoso de ésta con el mismo objetivo (obtención de métricas de software)

XScore también tiene algunas debilidades. Entre éstas:

- Está hecha específicamente para código fuente JAVA 1.4. Sin embargo, las métricas que se van a obtener no variarían si la herramienta se mejorara para aceptar código fuente JAVA 1.5.
- Se tiene la información de un objeto de código (clase, método, variable, etc.) en el momento en que es declarado, pero no en el momento en que es usado, lo cual obliga a crear estructuras adicionales especiales que almacenen esta información.

[12] JDOM. Librería de código abierto que provee una manera de representar un documento XML en Java. **[13]**
[13] JAXP. API de Java para Procesamiento XML. **[13]**

2. Definición de la estrategia de detección

El segundo paso en el esquema general de solución es la definición de la estrategia de detección. Debido a que éste es el corazón del trabajo, se dedica una sección aparte para explicarlo.

3. Aplicación de la estrategia de detección

Es el paso en el que la estrategia propuesta en el paso 2 se pone a prueba. Consiste de dos grandes actividades:

- Desarrollo de la aplicación que implementa la estrategia propuesta.
- Ejecución de la aplicación con ejemplos conocidos de código fuente para determinar si los resultados obtenidos son los esperados o no.

En el caso de que los resultados no sean los esperados, es posible que se requiera algún refinamiento de la estrategia y de la aplicación.

A continuación se detalla el paso número dos del esquema general de solución.

3.1.2 Definición de la estrategia de detección

Marinescu denominó como *estrategia de detección* a "una expresión cuantificable de una regla, por medio de la cual, fragmentos del diseño que sean conformes a esa regla pueden ser detectados en el código fuente" **[11]**. Esta definición sirvió de base para denominar lo que es una *estrategia de detección* en este trabajo: es el proceso para hallar expresiones cuantificables basadas en métricas de software, que sirvan como criterio de decisión de la presencia de un *bad smell* en el código fuente. Por lo tanto, la definición de la estrategia de detección para un *bad smell*, es la especificación de cómo va a ser el proceso en ese caso particular.

Este proceso, que se lleva a cabo para cada *bad smell* abordado, consta de tres actividades:

1. Caracterización del bad smell

Se busca presentar una definición más clara de la que se encuentra generalmente, con el fin de precisar qué significa el *bad smell*, y sus características. Para ello se sugiere utilizar el marco e trabajo propuesto por Munro, el cuál se representa en la tabla 1 **[18]**.

2. Escogencia de las métricas

Esta actividad consiste en el análisis que se hace con el fin de decidir qué métricas se van a evaluar para la detección del bad smell. En la medida de lo posible se debe tratar de usar métricas ya existentes y documentadas; sin embargo, puede ser necesaria la inclusión de métricas apropiadas solamente en el contexto tratado.

Nombre del bad smell	*Se citan el nombre y la descripción del bad smell dadas de Fowler y Beck* ***[1]***
Características	*Se exponen las principales características del bad smell deducidas de la descripción inicial y de otra literatura disponible.*
Heurísticas de diseño	*Se identifican las heurísticas y principios de diseño relevantes en la detección del bad smell.*

Tabla 1. Marco para describir formalmente un bad smell

Se encontró que el modelo GQM (goal-question-metric) desarrollado por Basili provee un framework con los pasos que se deben seguir en un proceso de medición de software, para decidir las métricas que se van a usar **[25]**. Se decidió usar este modelo porque ayuda a formalizar el proceso.
GQM dice que en una actividad de medición de software, se deben seguir los pasos:

1. Definición de los objetivos de la medición - *Goal*. En este caso, para cada *bad smell* se tiene un objetivo que se puede expresar como: *se quiere detectar la presencia del bad smell X en el código*. Sin embargo, este gran objetivo se puede expresar de forma más granular a través de sentencias que contengan objetivos más específicos.

2. Derivación de las preguntas - *Question*. Estas preguntas son las que se deben responder para poder determinar si los objetivos (del paso 1) se cumplen. Son deducidas a partir de la definición misma del *bad smell* y de los objetivos planteados en el paso anterior. Según el nivel de dificultad de detección, se tendrán más o menos preguntas: a mayor nivel de dificultad, más preguntas deben ser respondidas.

3. Selección de las métricas a usar - *Metric.* Métricas que se deben calcular para poder contestar las preguntas (del paso 2). Es posible que para responder una sola pregunta haya que obtener varias métricas. Para cada una de las métricas seleccionadas se da información que busca hacer más formal su definición y uso. Esta información se presenta en una tabla cuyo modelo es la tabla 2.

Este es un ejemplo del modelo, guiado por GQM, que se sigue para decidir cuáles métricas se van a usar, en el caso de que se abordara el bad smell *large class*:

Objetivo:
Se quiere detectar si en el código de la aplicación está presente el *bad smell large class*; es decir, se quiere saber si las clases que componen la aplicación tienen demasiadas líneas de código.

Preguntas:
¿Cuántas líneas de código tienen las clases de la aplicación?.

Métrica:
Número de líneas de código por clase.

Nombre	Nombre de la métrica
Definición	Definición encontrada en la literatura o dada por el autor
Uso específico	Uso específico que se le va a dar en el caso tratado. En algunos casos puede ser que éste varíe un poco del dado en la definición original; si no es así, este espacio va en blanco.
Notación	Notación que se usará. En algunos casos no existe tal en la literatura, así que se debe proponer alguna.
Propiedades	Propiedades que presenta la métrica y que permiten predecir su comportamiento y los valores que debería obtener en casos bien definidos. Se da por obvio que la métrica tiene que ser calculable.

Tabla 2. Especificación de métricas seleccionadas

3. Interpretación de las métricas

Es en este paso en dónde se define la forma en que se van a interpretar las métricas seleccionadas una vez hayan sido obtenidas, así como la manera en que éstas se van a correlacionar, es decir, las expresiones cuantificables que se busca obtener. Es aquí en donde el buen criterio de quien define la estrategia entra en juego, ya que es él (o ella) quien define la interpretación y la correlación de las métricas.

Esta actividad se compone de otras dos actividades:

1. Definición de filtros. Es la definición de los valores límites, de los intervalos y de los porcentajes permitidos, a partir de los cuales, o dentro de los cuales, el valor de una métrica se puede considerar como normal (o como problemático según el punto de vista). Por ejemplo, en el caso del *bad smell large class* el filtro aplicado a la métrica CLOC (líneas de código de la clase) podría expresarse de varias formas:
   - Si CLOC es mayor que x, entonces la clase es demasiado larga.
   - Si CLOC está en el intervalo [x, y], entonces la clase no es demasiado larga.
   - Si CLOC supera el x% de líneas de código totales del proyecto, entonces la clase es demasiado larga.

   En este trabajo, se quiere hacer que los filtros puedan ser variables para experimentar con varios valores y tratar de llegar a conclusiones más acertadas; es por esto que, por ejemplo, se define que el filtro de una de las métricas es que el valor de ésta sea mayor que cierta 'Variable'; con lo cual se expresa la intención de dejar abierta la posibilidad de variar tal valor.

2. Correlación de métricas. Es la definición de la forma en que se van a correlacionar las métricas obtenidas, la cual corresponde a las expresiones cuantificables cuya obtención es el resultado de la estrategia de detección. Estas expresiones son

fórmulas aritméticas o lógicas, que denotan un significado -posiblemente- diferente al de las métricas que la componen.
Es posible que sea necesario definir filtros también para los valores que se obtendrán al evaluar las fórmulas. Por ejemplo, suponga que para decidir si en un código fuente está presente el *bad smell large class* se tiene en cuenta, además del número de líneas de una clase (CLOC), el número de métodos de esa clase (NOM). La fórmula para relacionar estas dos métricas, y el filtro definido podrían ser:
`si CLOC / NOM > x, entonces la clase es demasiado larga.`

A las fórmulas obtenidas se les conoce también con el nombre de métricas compuestas, que son "las que se calculan por medio de combinaciones matemáticas de métricas elementales" **[10]**. Esta clasificación de métricas (de elementales y compuestas) corresponde a una de las taxonomías presentadas por Brito y Carapuça **[26]**, en donde el criterio usado es la estructura de las métricas. Esta taxonomía se presenta en la siguiente tabla.

Tipo	Descripción	Ejemplo
Métricas elementales	Cuantificación de un atributo simple	Número de líneas de una clase
Métricas compuestas	Combinación matemática de métricas elementales	Número de líneas por hombre por semana

Tabla 3. Taxonomía de métricas por estructura

Estas fórmulas son también una posible manera de asignar diferentes pesos a las métricas que las componen, para así dar más o menos importancia a alguna métrica en el resultado final. Por ejemplo, si -siguiendo con el ejemplo dado en el paso de correlación- se quisiera dar más importancia al número de métodos que al número de líneas de una clase para decidir si ésta es muy larga, se podría usar la fórmula:
`Si 0.1 * CLOC / 0.9 * NOM > x, entonces la clase es demasiado larga`
Ésta significa que el número de métodos -de la clase- tiene una importancia del 90% en la métrica, mientras que el número de líneas importa el restante 10%. Usando esta nueva fórmula, una clase que podría haber sido considerada como larga ahora no lo es (o lo contrario).

Otra forma de asignación de pesos es posible hacerla desde el paso de definición de filtros. El valor, intervalo o porcentaje que se define para que una métrica sea tenida en cuenta puede ser modificado para hacer que esa métrica se tenga más, o menos en cuenta; es decir, para darle más importancia, no sobre alguna otra en particular, sino sobre el conjunto de éstas. Por ejemplo, si se define que la métrica CLOC va a ser tenida en cuenta para alguna clase X si `CLOC(X) > z`, y luego ese criterio se cambia a si `CLOC(X) > z'`, donde `z' < z`; entonces se le está dando mayor importancia a esa métrica ya que, debido a este cambio, más clases van a ser candidatas a ser clases largas.

Es muy importante tener en cuenta que una vez se ha hallado la fórmula de la métrica compuesta (al menos en el caso de que la expresión resultante sea una fórmula aritmética), se debe verificar que ésta siga siendo una métrica; es decir, que se pueda calcular. Además se deben definir las mismas propiedades que se definen para las métricas simples (rango, valores límite, etc)

4. Aplicación de la solución a los casos abordados

A continuación se describirá el proceso de definición y aplicación (pasos 2 y 3 de la estrategia general de solución) de la estrategia para los *bad smells* abordados. Éstos serán notados con letras en orden alfabético, y los pasos, tanto de la estrategia general, como de la estrategia de detección, serán notados con el mismo orden expuesto anteriormente. Es decir que, por ejemplo, como *inappropriate intimacy* es el primero, le corresponde la letra A; el paso A.1 no se incluye, porque para cualquier letra x, x.1 correspondería al paso de escogencia o desarrollo de la representación adecuada, que como se explicó, ya se hizo y va a ser igual para todos; el paso A.2.1 es la definición de filtros; etc.

4.1 A. Inappropriate intimacy

A.2 Definición de la estrategia de detección

A.2.1 Caracterización del bad smell

Nombre del bad smell	*Inappropriate intimacy*. "Algunas veces las clases se tornan demasiado íntimas y gastan demasiado tiempo hurgando en sus partes privadas" **[1]**
Características	Consiste en el alto acoplamiento de dos clases. Los métodos declarados en una clase, usan mucho los métodos y variables de la otra.
Heurísticas de diseño	En ciertas oportunidades se requiere de un cierto nivel de acoplamiento entre las clases; pero, entre más alto sea éste, más difícil se hace el mantenimiento del código. Además, "El acoplamiento excesivo entre clases va en detrimento del diseño modular y previene la reutilización" **[21]**. El bajo acoplamiento entre clases mejora la modularidad. Según la definición de acoplamiento, éste se da entre dos artefactos de software, y como éste bad smell se define a nivel de clases, se está hablando de una unión bi-direccional de dos clases particulares. Si esta unión fuera unidireccional, (es decir, que los métodos de una clase usan los métodos y variables de la otra, pero no en viceversa) se podría decir que se trata del bad smell *feature envy* (ver definición en el glosario). Aunque éste se define a nivel de métodos, podría hacerse una generalización para todos los métodos de una clase y hacer el análisis a éste nivel.

Tabla 4. Caracterización del bad smell *Inappropriate intimacy*

A.2.2 Escogencia de las métricas (GQM)

A.2.2.1 Definición de objetivos

El objetivo general es detectar la presencia del bad smell *inappropriate intimacy*.
Los objetivos específicos son:

- Para cada par de clases determinar
  - si el número de invocaciones entre ellas es muy alto
  - si los métodos y los atributos de una están muy acoplados con la otra

- Para cada clase determinar
  - si el número de atributos de tipo abstracto de datos[14] es muy alto.
  - si el número de accesos a otras clases es muy alto.

A.2.2.2 Derivación de preguntas

Las siguientes preguntas aplican para un determinado par de clases (X,Y) sobre el que se esté haciendo el análisis.

- **?** ¿Cuál es el número de invocaciones entre las clases?
- **?** ¿Qué tan acoplados están los métodos de una clase con la otra?
- **?** ¿Cuál es el número de atributos en una clase que son del tipo de dato de la otra?
- **?** ¿Qué tan grande es el número de accesos de una clase a los atributos de la otra?

A.2.2.3 Selección de métricas a usar

Nota: se recomienda que antes de leer las descripciones de las métricas se revisen los anexos Resumen de notación y Propiedades de los conjuntos sobre los que se trabaja.

Para este bad smell (*inappropriate intimacy*) como para *middle man* se han escogido métricas previamente definidas en la literatura existente, y se han definido unas nuevas. Algunas de las previamente definidas deben ser usadas de forma un tanto distinta a la definida para que se adapten al problema tratado.

±

Nombre	MPC (message-passing coupling)
Definición	Métrica definida por Li y Henri **[28]**, mide el número de sentencias que envían mensajes[15] en una clase. Ellos encontraron que "el número de mensajes enviados hacia afuera de la clase puede indicar qué tan dependiente de los métodos en otras clases es la implementación de los métodos locales".
	Se calcula MPC para la clase analizada y, además, la métrica se tiene que

[14] Tipo abstracto de datos (abstract data type, ADT). Son los que no son de tipo simple (como: int, float, etc), sino de tipo clase (ejemplo: `String myString`)

[15] Cuando se habla de mensajes enviados de una clase a otra, se refiere a invocaciones de métodos.

Uso específico	calcular entre dos clases específicas y en ambos sentidos; es decir, que no es suficiente saber qué tantos mensajes envía la clase analizada, sino que se debe calcular el número de mensajes enviados desde ésta hacia otra clase y viceversa, para todos los pares de clases del código. No se tiene en cuenta la invocación del método "new" de una clase.
Notación	**MPC(X)** = el número de mensajes enviados desde la clase X a las otras clases. **MPC(X,Y)** = el número de mensajes enviados de la clase X a la clase Y.
Propiedades	**MPC(X)** se calcula para toda clase $X \in Classes_{App}$[16] **MPC(X)** se calcula sobre $Statements_X$[17] Rango: $MPC(X) \in Z$[18] y $0 \leq MPC(X) \leq |Statements_X|$[19] Valor cero: $MPC(X) = 0 \Leftrightarrow$ la clase **X** no hace invocaciones a métodos de las otras clases. Valor máximo: $MPC(X) = |Statements_X| \Leftrightarrow$ todos los statements de la clase X son mensajes hacia métodos en otras clases. (↑)[20]: Suponga que $Statements_{X'} = Statements_X + Message(X)$[21]. Entonces $MPC(X') = MPC(X) + 1$ (↓): Suponga que algún $STi \in Statements_X$ es **Message(X)**, y que $Statements_{X'} = Statements_X - STi$. Entonces $MPC(X') = MPC(X) - 1$ ----------

[16] $Classes_{App}$ denota el conjunto de clases que componen la aplicación analizada.

[17] $Statements_X$ denota el conjunto de statements de la clase X

[18] **Z** denota el conjunto de los números enteros positivos

[19] | **set** | denota la cardinalidad del conjunto **set**

[20] Los símbolos (↑) y (↓) se usarán para denotar las propiedades "cómo crece" y "cómo decrece" de las métricas, respectivamente.

[21] **Message(X)** denota un statement de invocación de método de la clase **X**

	MPC(X,Y) se calcula para todo par de clases **X, Y ∈ Classes$_{App}$**, donde **X ≠ Y.** **MPC(X,Y)** se calcula sobre **Messages$_x$**[22] **MPC(X,X)** no se calcula **¬(MPC(X,Y) = 0 ⇒ MPC(Y,X) = 0)** **MPC(X,Y) > 0 ⇒** que existe algún nivel de acoplamiento entre las clases X y Y Rango: **MPC(X,Y) ∈ Z**, y **0 ≤ MPC(X,Y) ≤ |Messages$_x$|** Valor cero: **MPC(X,Y) = 0 ⇔** la clase X no hace invocaciones a métodos de la clase Y Valor máximo: **MPC(X,Y) = |Messages$_x$| ⇔** todos los mensajes que envía la clase X son hacia métodos en la clase Y (↑): Suponga que **Messages$_{x'}$ = Messages$_x$ + Message(X,Y)**[23]. Entonces **MPC(X',Y) = MPC(X,Y) + 1** (↓): Suponga que algún **Mi ∈ Messages$_x$** es **Message(X,Y)**, y que **Messages$_{x'}$ = Messages$_x$ – Mi**. Entonces **MPC(X',Y) = MPC(X,Y) - 1**

±

Nombre	TATFD (total accesses to foreign data) [24]

[22] **Messages$_x$** es el conjunto de sentencias en la clase X que envían mensajes a otras clases(invocaciones a métodos).

[23] **Message(X,Y)** denota un statement de invocación de método de la clase **X** hacia la clase **Y.**

[24] Los nombres de las métricas definidas en este documento se dan con siglas en inglés por uniformidad con las encontradas en la literatura

Definición	Mide el número total de accesos a atributos de otras clases hechos por parte de la clase analizada.
Uso específico	Esta métrica se usa en forma similar a MPC: se calcula para una clase X, y para X con respecto a otra clase específica Y; así, se puede determinar el porcentaje de accesos a atributos de Y en relación con los accesos a atributos de otras clases Este cálculo se hace para cada par de clases para determinar el nivel de acoplamiento entre éstas respecto al acceso a atributos. Ciertamente, el acceso a métodos *getter*[25] se podría considerar como accesos a atributos; sin embargo, aquí se determina que son contados como accesos a métodos.
Notación	**TATFD(X)** = número total de accesos a atributos foráneos que hace la clase X **TATFD(X,Y)** = número total de accesos a atributos de la clase Y que hace la clase X
Propiedades	**TATFD(X)** se calcula para todas las $\mathbf{X \in Classes_{App}}$ **TATFD(X)** se calcula sobre $\mathbf{Statements_X}$ Rango: $\mathbf{TATFD(X) \in Z}$, y $\mathbf{0 \leq TATFD(X) \leq |Statements_X|}$ Valor cero: $\mathbf{TATFD(X) = 0}$ $\Leftrightarrow$ la clase X no hace accesos a atributos de otras clases Valor máximo: $\mathbf{TATFD(X) = |Statements_X|}$ $\Leftrightarrow$ todos statements de la clase X son accesos a atributos en otras clases. ($\uparrow$): Suponga que $\mathbf{Statements_{X'} = Statements_X + DataAccess(X)}$[26]. Entonces **TATFD(X') = TATFD(X) + 1** ($\downarrow$): Suponga que algún $\mathbf{STi \in Statements_X}$ es **DataAccess(X)**, y que $\mathbf{Statements_{X'} = Statements_X - Sti}$. Entonces **TATFD(X') = TATFD(X) – 1** ----------

[25] Los métodos *getter* son los que simplemente devuelven alguno de los atributos de la clase

[26] **DataAccess(X)** denota un statement de la clase **X** de acceso de datos

	TATFD(X,Y) se calcula para todo par de clases **X, Y** $\in$ **Classes**$_{App}$, donde **X** $\neq$ **Y.** **TATFD(X)** se calcula sobre **DataAccesses**$_X$[27] **TATFD(X,X)** no se calcula Rango: **TATFD(X,Y)** $\in$ **Z**, y **0** $\leq$ **TATFD(X,Y)** $\leq$ |**DataAccesses**$_X$| Valor cero: **TATFD(X,Y) = 0** $\Leftrightarrow$ la clase X no hace accesos a atributos de la clase Y Valor máximo: **TATFD(X) =** |**DataAccesses**$_X$| $\Leftrightarrow$ todos los accesos a datos que hace clase X son accesos a atributos de la clase Y. ($\uparrow$): Suponga que **DataAccesses**$_{X'}$ **= DataAccesses**$_X$ **+ DataAccess(X,Y)**[28]. Entonces **TATFD(X',Y) = TATFD(X,Y) + 1** ($\downarrow$): Suponga que algún **DAi** $\in$ **DataAccesses**$_X$ es **DataAccess(X,Y)**, y que **DataAccesses**$_{X'}$ **= DataAccesses**$_X$ **– DAi**. Entonces **TATFD(X',Y) = TATFD(X,Y) - 1**

±

Nombre	DAC (data abstraction coupling)
Definición	Métrica definida por Li y Henri **[28]**, mide el número de atributos de tipo abstracto de datos (ADT) de una clase. Este es un indicativo de acoplamiento porque "entre más tipos abstractos de datos tenga una clase, es más complejo el acoplamiento entre ésta y las otras clases".
Uso específico	En este caso, esta métrica se usa para medir el número de atributos cuyo tipo es alguna otra clase declarados en la clase analizada, para todos los pares de clases del código. No se cuentan las variables locales que cumplan con la característica expuesta, porque esto sería indicador de acoplamiento a nivel de método,

[27] **DataAccesses**$_x$ es el conjunto de sentencias de la clase **X** que son accesos a atributos.

[28] **DataAccess(X,Y)** denota un statement de acceso de datos desde la clase **X** hacia atributos de la clase **Y**.

	no de clase. Tampoco se cuentan los atributos que tiene como tipo el mismo de la clase donde son declarados, porque esto podría dar falsos negativos (por ejemplo: hay dos clases: X, Y. X declara 97 atributos de tipo X y 3 de tipo Y. Si se contaran los que tienen el mismo tipo de la clase que los declara, se tendría que solo el 3% de los atributos de X son de tipo Y, lo cual, tal vez, no indicaría que hay acoplamiento entre X y Y. Pero, si no se cuentan, daría que el 100% de los atributos de X son de tipo Y, lo cual ciertamente indicaría acoplamiento entre las clases, y estaría más acorde con el resultado esperado).
Notación	**DAC(X)** = número de atributos de tipo abstracto de datos de una clase **DAC(X,Y)** = número de atributos de tipo de la clase Y declarados en la clase X
Propiedades	**DAC(X)** se calcula para todas las $\mathbf{X \in Classes_{App}}$ **DAC(X)** se calcula sobre $\mathbf{Attributes_X}$[29] Rango: $\mathbf{DAC(X) \in Z}$, y $\mathbf{0 \leq DAC(X) \leq |Attributes_x|}$ Valor cero: $\mathbf{DAC(X) = 0}$ $\Leftrightarrow$ la clase X no declara atributos de tipo ADT. Valor máximo: $\mathbf{DAC(X) = |Attributes_x|}$ $\Leftrightarrow$ todos los atributos declarados en la clase X son de tipo ADT de alguna otra clase. ($\uparrow$): Suponga que $\mathbf{Attributes_{X'} = Attributes_X + Attribute(X)}$[30]. Entonces **DAC(X') = DAC(X) + 1** ($\downarrow$): Suponga algún $\mathbf{ATTRBi \in Attributes_X}$, y que **Attributes' = Attributes – ATTRBi.** Entonces **DAC(X') = DAC(X) – 1** **----------** **DAC(X,Y)** se calcula para todo par de clases $\mathbf{X, Y \in Classes_{App}}$, donde $\mathbf{X \neq Y}$. **DAC(X,Y)** se calcula sobre $\mathbf{Attributes_X}$ **DAC(X,X) no se calcula**

[29] **{X, Attributes}** es el conjunto de atributos de la clase **X**

[30] **Attribute(X)** denota la declaración de un atributo en la clase **X**

	Rango: **DAC(X,Y)** $\in$ **Z**, y **0** $\leq$ **DAC(X,Y)** $\leq$ $\lvert$**Attributes**$_X\rvert$ Valor cero: **DAC(X,Y) = 0** $\Leftrightarrow$ la clase **X** no declara atributos de tipo Y. Valor máximo: **DAC(X,Y) =** $\lvert$**Attributes**$_X\rvert$ $\Leftrightarrow$ todos los atributos de la clase X son de tipo Y. ($\uparrow$): Suponga que **Attributes**$_{X'}$ **= Attributes**$_X$ **+ Attribute(X,Y)**[31]. Entonces **DAC(X',Y) = DAC(X,Y) + 1** ($\downarrow$): Suponga algún **ATTRBi** $\in$ **Attributes**$_X$ es **Attribute(X,Y)**, y que **Attributes**$_{X'}$ **= Attributes**$_X$ **- ATTRBi.** Entonces **DAC(X',Y) = DAC(X,Y) - 1**

±

Nombre	Number of accessed methods
Definición	Es el número de métodos de la clase Y que la clase X accede.
Uso específico	La métrica **MPC(X,Y)** da el número total de accesos a métodos de la clase Y que hace la clase X, pero también es necesario saber cuántos de éstos -métodos- en total son accedidos para complementar la información sobre el acoplamiento entre ambas clases, y descartar que se trate de otro bad smell, específicamente, *feature envy* (ver definición en el glosario). Esta situación se podría presentar en el siguiente caso: **MPC(X,Y) = z**, donde **z** se considera un valor alto para esta métrica; pero, si el número de métodos accedidos de la clase **Y** por la clase **X** es uno (algún Method_i), esto quiere decir que la clase **X** tiene 'envidia' de la clase **Y** por su método `Method_i`, pero no necesariamente que se presente *inappropriate intimacy* entre las dos. Sin embargo, sí se debe tener en cuenta el caso en el que el número de métodos accedidos desde **X** hacia **Y** es uno, y el número de métodos de **Y** es uno; en este caso no se podría distinguir, por el criterio de esta métrica, entre *feature envy* e *inappropriate intimacy*. Se calcula **NAM(X,Y)** y **NAM(Y,X)** para todos los pares de clases X, Y que pertenezcan a **Classes**$_{App}$
Notación	**NAM(X,Y)**

[31] **Attribute(X,Y)** denota la declaración de un atributo de tipo **Y** en la clase **X**

Propiedades	Se calcula para todo par de clases $X, Y \in Classes_{App}$, donde $X \neq Y$. **NAM(X,Y)** se calcula sobre $Messages_{X(X,Y)}$[32] **NAM(X,X)** no se calcula. $\neg (NAM(X,Y) = 0 \Rightarrow NAM(Y,X)) = 0$ $NAM(X,Y) > 0 \Rightarrow$ que existe algún nivel de acoplamiento entre las clases X y Y. Rango: $NAM(X,Y) \in Z$, y $0 \leq NAM(X,Y) \leq |Methods_Y|$[33] Valor cero: $NAM(X,Y) = 0 \Leftrightarrow$ la clase X no hace invocaciones a métodos de la clase Y. Valor máximo: $NAM(X,Y) = |Methods_Y| \Leftrightarrow$ la clase X hace por lo menos una invocación a cada uno de los métodos (obviamente, a aquellos a los cuales tenga el nivel de acceso necesario) de la clase Y. Para la definición de las propiedades (↑) y (↓) de esta métrica y de la métrica NAA (la siguiente a esta), se usará la función "Unique" que se define así: **Unique (Set, Criterion)** : es el conjunto que resulta de seleccionar de entre los elementos del conjunto inicial *Set*, un elemento único (cualquiera) de cada uno de los subconjuntos de éste formado por los elementos que cumplen con el criterio *Criterion*. Se definen los atributos de un mensaje: - Clase fuente (SC). La clase desde la que se origina el mensaje. - Método fuente (SM). El método de la clase fuente desde el cual se origina el mensaje. - Clase destino (TC). Clase a la cual está dirigido el mensaje. - Método destino (TM). Método de la clase destino al cual está dirigido el mensaje. (↑): Suponga que $Messages_{X'(X,Y)} = Messages_{X(X,Y)} + Message(X,Y)$. Si $|Unique(Messages_{X'(X,Y)}, TM)| > |Unique(Messages_{X(X,Y)}, TM)|$, entonces $|Unique(Messages_{X'(X,Y)}, TM)| =$

[32] $Messages_{X(X,Y)}$ denota el conjunto de mensajes que envía **X** y son dirigidos a **Y**

[33] $Methods_X$ denota el conjunto de métodos declarados en la clase X

	$\|Unique(Messages_{X(X,Y)}, TM)\| + 1$, y por lo tanto $NAM(X',Y) = NAM(X,Y) + 1$ (↓): Suponga que $Messages_{X'(X,Y)} = Messages_{X(X,Y)} - Message(X,Y)$. Si $\|Unique(Messages_{X'(X,Y)}, TM)\| < \|Unique(Messages_{X(X,Y)}, TM)\|$, entonces $\|Unique(Messages_{X(X,Y)}, TM)\| = \|Unique(Messages_{X(X,Y)}, TM)\| - 1$, y por lo tanto $NAM(X',Y) = NAM(X,Y) - 1$

±

Nombre	Number of accessed attributes
Definición	Es el número de atributos de la clase Y que la clase X accede.
Uso específico	**TATFD(X,Y)** da el número total de accesos a atributos de la clase Y que hace la clase X, pero también es necesario saber cuántos de éstos -atributos- en total son accedidos para complementar la información sobre el acoplamiento entre ambas clases. Si **TATFD(X,Y)** fuera alto, pero por ejemplo se accediera solo un atributo, no necesariamente se estaría dando *inappropriate intimacy* entre las dos clases (en ese caso se podría hablar de un *feature envy* a nivel de atributos). Se calcula **NAA(X,Y)** y **NAA(Y,X)** para todos los pares de clases X, Y que pertenezcan a $Classes_{App}$
Notación	**NAA(X,Y)**
Propiedades	**NAA(X,Y)** se calcula para todo par de clases **X, Y ∈ {Classes}**, donde **X ≠ Y.** **NAA(X,Y)** se calcula sobre $DataAccesses_{X(X,Y)}$[34] **NAA(X,X) no se calcula** Rango: **NAA(X,Y) ∈ Z**, y **0 ≤ NAA(X,Y) ≤** $\|Attributes_Y\|$ Valor cero: **NAA(X,Y) = 0** ⇔ la clase X no hace accesos a atributos de la clase Y. Valor máximo: **NAA(X,Y) =** $\|Attributes_Y\|$ ⇔ la clase X hace por lo menos un acceso a cada uno de los atributos de la clase Y.

[34] $DataAccesses_{X(X,Y)}$ denota el conjunto de accesos a atributos de la clase **Y** que hace la clase **X**

	Se definen los atributos de un acceso de datos: - Clase fuente (SC). La clase desde la que se origina el acceso de datos. - Método fuente (SM). El método de la clase fuente desde el cual se origina el acceso de datos. - Clase destino (TC). Clase a la cual está dirigido el acceso de datos. - Atributo destino (TA). Método de la clase destino al cual está dirigido el acceso de datos. (↑): Suponga que $\mathbf{DataAccesses_{X'(X,Y)} = DataAccesses_{X(X,Y)} + DataAccesses(X,Y)}$. Si $\mathbf{|Unique(DataAccesses_{X'(X,Y)}, TA)| > |Unique(DataAccesses_{X(X,Y)}, TA)|}$, entonces $\mathbf{|Unique(DataAccesses_{X'(X,Y)}, TA)| = |Unique(DataAccesses_{X(X,Y)}, TA)| + 1}$, y por lo tanto **NAA(X',Y) = NAA(X,Y) + 1** (↓): Suponga que $\mathbf{DataAccesses_{X'(X,Y)} = DataAccesses_{X(X,Y)} - DataAccesses(X,Y)}$. Si $\mathbf{|Unique(DataAccesses_{X'(X,Y)}, TA)| < |Unique(DataAccesses_{X(X,Y)}, TA)|}$, entonces $\mathbf{|Unique(DataAccesses_{X'(X,Y)}, TA)| = |Unique(DataAccesses_{X(X,Y)}, TA)| - 1}$, y por lo tanto **NAA(X',Y) = NAA(X,Y) - 1**

±

Nombre	RMC (relative method coupling)
Definición	Definida por Joshi, es la medida de qué tan acoplado está un método de la clase analizada con respecto a otra clase dada; específicamente, "trata de capturar el acoplamiento entre clases a través de un método escogido, en relación con la cohesividad que tiene ese método con su clase propietaria" **[29]**.
Uso específico	Se debe calcular para todos los métodos **m** que pertenecen a $\mathbf{Methods_X}$, y para todos los pares de clases X y Y que pertenecen $\mathbf{Classes_{App}}$ para poder determinar qué tan acoplados están los métodos de X con Y.
Notación	**RMC(Ci,m -> Cj) =** **A(Cj <- m) + M(Cj <- m) / max (1, A(Ci <- m) + M(Ci <- m))** donde **RMC(Ci,m -> Cj)** denota el cálculo de esta métrica para el método **m** de la clase **Ci** con respecto a la clase **Cj**,

	A(Ck <- m) es la cardinalidad del conjunto de atributos de la clase **Ck** que son accedidos por el método **m**. **M(Ck <- m)** es la cardinalidad del conjunto de métodos de la clase **Ck** que son accedidos por el método **m**.
Propiedades	El numerador de la fórmula indica la fuerza del acoplamiento entre el método y la clase acoplada, y el denominador indica la fuerza de la cohesión entre el método y su clase. Por lo tanto RMC es "la razón del acoplamiento de un método dado con una clase externa al acoplamiento entre éste y su clase propietaria" **[29]**. **RMC(X,m -> Y)** se calcula para todo par de clases **X, Y** $\in$ $\mathbf{Classes}_{App}$, y para todo **m** $\in$ $\mathbf{Methods}_{X}$; donde **X** $\neq$ **Y.** **RMC(X,m -> X)** no se calcula **RMC(X,m -> Y)** se calcula sobre $\mathbf{Statements}_{(X,m)}$ [35] Rango: **RMC(X,m -> Y)** $\in$ **Z**, y **0** $\leq$ **RMC(X,m -> Y)** $\leq$ $\lvert \mathbf{Statements}_{(X,m)} \rvert$ Valores característicos[36]:

RMC(X,m -> Y)	=	⇒
	0	m es totalmente cohesivo con su clase propietaria X
	1	no se puede decidir si m está más acoplado con Y que cohesionado con X
	(0,1)	m es cohesivo con su clase propietaria
	> 1, o Máximo valor	m está acoplado con Y

(↑):
Suponga que $\mathbf{Statements}_{(X,m)'}$ **= Statements**$_{(X,m)}$ **- Message(X,m -> Y) - DataAccess(X,m -> Y)** [37]
Entonces **RMC(X',m -> Y) > RMC(X,m -> Y)**

[35] **{X, Statements(m)}** es el conjunto de sentencias del método **m** de la clase **X**

[36] Como este estúpido programa (Microsoft Word) me hizo sufrir mucho por uno de sus estúpidos bugs, pongo este pie de página para recordarlo.

[37] **Message(X,m -> Y)**, **DataAccess(X,m -> Y)** denotan respectivamente: un statement de envío de mensajes desde el método **m** de la clase **X** hacia la clase **Y,** y un statement de acceso de datos desde el método **m** de la clase **X** hacia la clase **Y**

	(↓): Suponga que $\mathbf{Statements_{(X,m)'} = Statements_{(X,m)} + Message(X,m -> X) + DataAccess(X,m -> X)}$ Entonces $\mathbf{RMC(X',m -> Y) < RMC(X,m -> Y)}$

±

Nombre	NOM (number of methods)
Definición	Es el número de métodos declarados en la clase X.
Uso específico	Es necesario tener el número total de métodos declarados en cada clase para poder determinar, por ejemplo, el porcentaje de éstos que están acoplados con alguna otra clase
Notación	$\mathbf{NOM(X) = |Methods_X|}$
Propiedades	$\mathbf{NOM(X)}$ se calcula para toda clase $\mathbf{X \in Classes_{App}}$ $\mathbf{NOM(X)}$ se calcula sobre $\mathbf{Methods_X}$ Rango: $\mathbf{NOM(X) \in Z}$, y $\mathbf{0 \leq NOM(X) < \infty}$ Valor cero: $\mathbf{NOM(X) = 0}$ $\Leftrightarrow$ la clase X no tiene declarado método alguno Valor máximo: Si $\mathbf{NOM(X)}$ tiende a ∞ entonces la clase X declara un número infinito de métodos. (↑): Si $\mathbf{|Methods_{X'}| = |Methods_X| + 1}$, entonces $\mathbf{NOM(X') = NOM(X) + 1}$ (↓): Si $\mathbf{|Methods_{X'}| = |Methods_X| - 1}$, entonces $\mathbf{NOM(X') = NOM(X) - 1}$

±

Nombre	NOA (number of attributes)
Definición	Es el número de atributos declarados en la clase X.

Uso específico	Es necesario tener el número total de atributos declarados en cada clase para poder determinar, por ejemplo, el porcentaje de éstos que son accedidos desde otra clase.
Notación	**NOA(X)**
Propiedades	**NOA(X)** se calcula para toda clase $\mathbf{X} \in \mathbf{Classes}_{App}$ **NOA(X)** se calcula sobre $\mathbf{Attributes}_X$ Rango: $\mathbf{NOA(X)} \in \mathbf{Z}$, y $0 \leq \mathbf{NOA(X)} < \infty$ Valor cero: $\mathbf{NOA(X)} = 0 \Leftrightarrow$ la clase X no tiene declarado atributo alguno. Valor máximo: Si **NOA(X)** tiende a ∞, entonces la clase X declara un número infinito de atributos. ($\uparrow$): Si $\mathbf{\lvert Attributes_{X'} \rvert = \lvert Attributes_X \rvert + 1}$, entonces **NOA(X') = NOA(X) + 1** ($\downarrow$): Si $\mathbf{\lvert Attributes_{X'} \rvert = \lvert Attributes_X \rvert - 1}$, entonces **NOA(X') = NOA(X) - 1**

A.2.3 Interpretación de las métricas

En la sección de definición de filtros se enumera cada una de las métricas definiendo cuándo los valores calculados deberían ser tenidos en cuenta. En el caso de *inappropriate intimacy*, los valores de las métricas son tenidos en cuenta cuando indiquen que puede existir acoplamiento excesivo.

A.2.3.1 Definición de filtros

- MPC
  MPC(X,Y) / MPC(X) = z, donde **z** está entre [0,1] y representa el porcentaje de mensajes que envía la clase **X** que son dirigidos a **Y.** Si **z > 'variable'** se considera alto, y por lo tanto indicativo de acoplamiento.

¦ TATFD

TATFD(X,Y) / TATFD(X) = z, donde **z** está entre [0,1] y representa el porcentaje de los accesos a atributos de otras clases hechos por la clase **X** que son a atributos de la clase **Y.** Si **z > 'variable',** se considera alto, y por lo tanto indicativo de acoplamiento.

¦ DAC

DAC(X,Y) / DAC(X) = z, donde **z** está entre [0,1] y representa el porcentaje de atributos de tipo ADT declarados en la clase **X** que son de tipo **Y.** Si **z > 'variable',** se considera alto, y por lo tanto indicativo de acoplamiento.

¦ NAM

El filtro para esta métrica se define desde dos perspectivas:

- Si **NAM(X,Y) > 1**, entonces se podría descartar el bad smell *feature envy*, y se tiene en cuenta esta métrica. Sin embargo, también puede suceder que la clase tenga un solo método declarado, por lo que la condición completa sería esta: Si **NAM(X,Y) > 1** ∨ (**NAM(X,Y) = 1** ∧ **NOM(Y) = 1)**.
- Si **NAM(X,Y) / NOM(Y) = z,** donde **z** está entre [0,1] y representa el porcentaje de métodos declarados en la clase **Y** que son accedidos por la clase **X.** Si **z > 'variable',** se considera alto, y por lo tanto indicativo de acoplamiento.

¦ NAA

NAA(X,Y) / NOA(Y) = z, donde **z** está entre [0,1] y representa el porcentaje de atributos declarados en la clase **Y** que son accedidos por la clase **X.** Si **z > 'variable',** se considera alto, y por lo tanto indicativo de acoplamiento.

¦ RMC

Si **RMC(X,m -> Y) > 1**, significa que el método **m** está más acoplado con la clase **Y** de lo que está en cohesión con su clase propietaria **X**.

Se definen NOWCM (number of outwards-coupled methods), **NOWCM(X)** como el número de métodos de la clase **X** que están más acoplados con otras clases (de lo que son cohesivos con X), es decir, los métodos **m ∈ {X, Methods}** tal que **RMC(X,m -> C) > 1** para cualquier **C ∈ {Classes}**; y **NOWCM(X,Y)** como el número de métodos de la clase **X** que están más acoplados con la clase **Y** (de lo que son cohesivos con X), es decir, los métodos **m ∈ {X, Methods}** tal que **RMC(X,m -> Y) > 1**. Así, el filtro para esta métrica se define desde dos perspectivas:

- Si **NOWCM(X,Y) >= 1**, entonces se tiene en cuenta esta métrica.
- Si **NOWCM(X,Y) / NOM(X) = z,** donde **z** está entre [0,1] y representa el porcentaje de métodos declarados en la clase **X** que están acoplados con la clase **Y.** Si **z > 'variable',** se considera alto, y por lo tanto indicativo de acoplamiento.

Los pesos de cada métrica son manejados por medio de los valores que para cada una se definen como indicadores de que la métrica va a ser tenida en cuenta.

A.2.3.2 Correlación de métricas

La forma en que se van a correlacionar las métricas va a ser en una expresión lógica en la que se expresa que: si se cumple la condición de que el valor de cada una de las métricas que la componen se puede considerar como alto, entonces se puede afirmar que el bad smell *Inappropriate intimacy* está presente para un determinado par de clases. El principal criterio por el que se decidió hacerlo así es porque no es fácil expresar una fórmula que reúna todas las métricas calculadas, ni encontrar un límite o filtro para una expresión de esa complejidad; además, resulta más intuitivo, sin dejar de ser correcto, decir que "si el valor de la métrica 1 es alto, y, el valor de la métrica 2 es alto, y, … , y, el valor de la métrica n es alto; entonces el bad smell está presente.
Entonces, hay *Inappropriate intimacy* entre las clases X y Y si:

$$
\begin{aligned}
II(X,Y) \Leftrightarrow\ & MPC(X,Y) / MPC(X) > V1 \wedge \\
& TATFD(X,Y) / TATFD(X) > V2 \wedge \\
& DAC(X,Y) / DAC(X) > V3 \wedge \\
& (NAM(X,Y) > 1 \vee (NAM(X,Y) = 1 \wedge NOM(Y) = 1)) \wedge \\
& \qquad NAM(X,Y) / NOM(Y) > V4 \wedge \\
& NAA(X,Y) / NOA(Y) > V5 \wedge \\
& NOWCM(X,Y) / NOM(X) > V6
\end{aligned}
$$

Donde Vi está en el intervalo [0,1], y es la variable que representa el peso dado a cada métrica.

A.3 Aplicación de la estrategia de detección

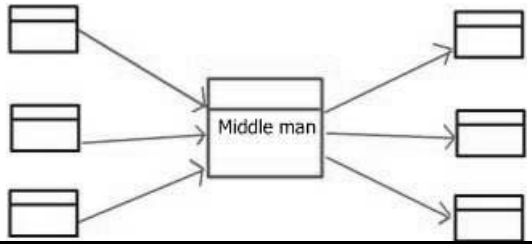
La aplicación de la estrategia se lleva a cabo mediante una aplicación de software construida para hacer el cálculo de las métricas y hacer el análisis de las condiciones definidas para determinar si existe el bad smell. El funcionamiento de esta aplicación se explica en el capítulo cinco: Herramienta de software para la implementación de la propuesta.
Para el caso de detección de *Inappropriate intimacy*, la aplicación permite que el usuario pueda descartar o no los posibles casos de *Feature envy*, con respecto a la métrica NAM.

4.2 B. Middle man

B.2 Definición de la estrategia de detección

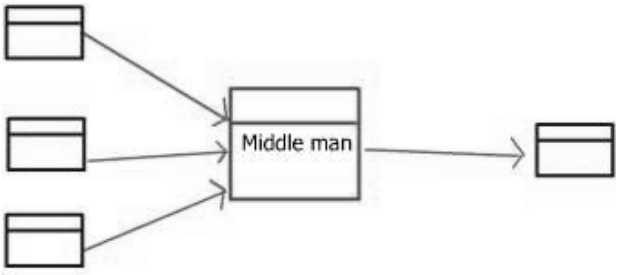
B.2.1 Caracterización del bad smell

Nombre del bad smell	*Middle man*. "El encapsulamiento viene a menudo con delegación... Sin embargo, esto puede ir demasiado lejos. Uno puede ver la interfaz de una clase y encontrar que la mitad de los métodos están delegando a otra clase" **[1]**

Características	Consiste en el hecho de que una clase delega la mayor parte de sus responsabilidades a otras clases. La clase en cuestión se convierte en un intermediario, ya que muchos de sus métodos delegan a métodos de otras clases.
Heurísticas de diseño	La delegación de responsabilidades ayuda a mantener el principio de encapsulamiento; pero, "aunque es un patrón común en la programación OO, puede dificultar el programa" **[4]**. En forma muy extendida puede convertirse en una forma de burocracia entre objetos. El alto nivel de acoplamiento también se hace presente cuando existe este bad smell. El alto acoplamiento se da entre las clases que usan el *middle man* y éste, y, a la vez, entre éste las clases a las que delega. En el análisis, hay que tener en cuenta que la clase que usa el -posible- *middle man* y la clase a la que éste delega, no deben ser la misma; sino, se estaría analizando la presencia del bad smell *inappropriate intimacy*. La característica más importante que se debe tener en cuenta para determinar si una clase es un *middle man*, es el nivel de responsabilidad que tiene en implementar sus propios métodos; es decir, que éstos no sean solo invocadores de métodos en otras clases. Sin embargo, también ayuda al análisis de este *bad smell* ver las cosas desde otros dos puntos de vista: el primero, es el análisis de las clases que delegan hacia la clase analizada; el segundo, es el análisis de las clases a las que la clase analizada delega. Lo que se analiza en estos dos casos es qué tan acopladas están las clases que delegan con la clase analizada, y qué tan acoplada está la clase analizada con las clases a las que delega. Por ello, se identificaron tres tipos de "actores" y cuatro posibles casos : - Los "actores", según su papel, son tres. Las clases que usan al intermediario entre ellas y otras clases: *las delegadoras*; la clase que hace de intermediario: *el middle man*; y las clases que son el destino de los mensajes: *las delagadas*. - Los cuatro posibles casos de situaciones típicas que se pueden detectar como "sospechosas" al hacer el análisis en busca de este bad smell. Caso 1: 

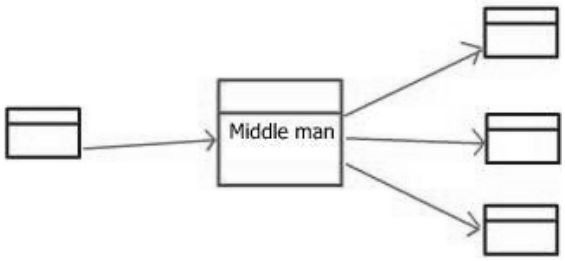
Es el caso en que varias clases delegan al *middle man*, y este a la vez delega a varias clases.

Caso 2:



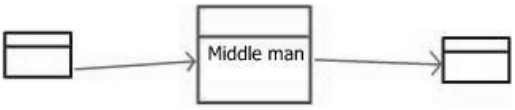
Es el caso en que varias clases delegan al *middle man*, y este a la vez delega a una clase.

Caso 3:



Es el caso en que una clase delega al *middle man*, y este a la vez delega a varias clases.

Caso 4:



Es el caso en que solo una clase delega al *middle man*, y este a la vez delega a una clase.

	Estos casos son usados cuando se definen los filtros de las métricas para poder relacionar los valores sospechosos de éstas con las situaciones que indican.

Tabla 5. Caracterización del bad smell *middle man*

B.2.2 Escogencia de las métricas (GQM)

B.2.2.1 Definición de objetivos

El objetivo general es detectar la presencia del bad smell *middle man*.
Los objetivos específicos son:

- Para cada clase determinar
  - Si el número de clases que están acopladas a la clase que se está analizando como posible *middle man* es alto. Este acoplamiento se toma en forma unidireccional[38], desde las demás clases hacía la analizada; es decir, se analiza si las demás clases usan métodos o atributos de esta clase.
  - Si el número de clases con las que esta clase está acoplada es alto. Al igual que en el objetivo anterior, el acoplamiento es unidireccional, pero se determina desde la clase analizada hacía las demás clases.
  - Si la clase realmente implementa o no la mayoría de sus métodos; es decir, si sus métodos hacen algo más que pasar los mensajes a otras clases, o si su implementación depende mucho de métodos en otras clases.

B.2.2.2 Derivación de preguntas

Las preguntas se derivan teniendo en cuenta que el análisis se hace sobre cada clase, una a la vez.

- ¿Con cuántas clases está acoplada en forma unidireccional está clase?
- ¿Cuántas clases están acopladas en forma unidireccional con esta clase?
- ¿Cuál es el número de clases usadas (clases de las que se acceden métodos o atributos) por esta clase?
- ¿Cuál es el número de clases que usan esta clase?
- ¿Cuál es el número de mensajes enviados desde esta clase?
- ¿Qué tan responsable de sí misma es la clase?; es decir, ¿qué tanto depende de otras clases para implementar sus métodos?

[38] Se usa el término Acoplamiento unidireccional -entre dos clases- para hacer diferencia con el concepto normal de acoplamiento (Ver definición en el glosario). Éste -el unidireccional- se define que va de una clase a la otra, pero no -necesariamente- en viceversa; es decir, acoplamiento unidireccional entre las clases A y B significa que los métodos de la clase A usan los métodos y atributos de la clase B, pero no necesariamente los métodos de B usan los métodos y atributos de A.

B.2.2.3 Selección de métricas a usar

±

Nombre	ATFD (access to foreign data)
Definición	Definida por Marinescu **[11]**, mide "el número de clases externas de las cuales la clase analizada accede a atributos". Entre mayor sea el número de clases de las cuales la clase analizada accede atributos, mayor es el nivel de acoplamiento que, sin referirse a una clase en particular, presenta.
Uso específico	El mismo dado en la definición de la métrica
Notación	**ATFD(X)** = Número de clases de las que la clase X accede atributos
Propiedades	**ATFD(X)** se calcula para todas las $\mathbf{X \in Classes_{App}}$ **ATFD(X)** se calcula sobre $\mathbf{DataAccceses_X}$ Rango: $\mathbf{ATFD(X) \in Z}$, y $\mathbf{0 \leq ATFD(X) \leq |Classes_{App}|}$ Valor cero: $\mathbf{ATFD(X) = 0}$ ⇔ la clase X no hace accesos a atributos de otras clases. Valor máximo: $\mathbf{ATFD(X) = |Classes_{App}|}$ ⇔ la clase X hace accesos a por lo menos un atributo de cada una de las otras clases. (↑): Suponga que $\mathbf{DataAccceses_{X'} = DataAccceses_X + DataAccess(X)}$ Si $\mathbf{|Unique(DataAccceses_{X'}, TC)| > |Unique(DataAccceses_X, TC)|}$, entonces $\mathbf{|Unique(DataAccceses_{X'}, TC)| = |Unique(DataAccceses_X, TC)| + 1}$, y por lo tanto **ATFD(X') = ATFD(X) + 1** (↓): Suponga que $\mathbf{DataAccceses_{X'} = DataAccceses_X - DataAccess(X)}$ Si $\mathbf{|Unique(DataAccceses_{X'}, TC)| < |Unique(DataAccceses_X, TC)|}$, entonces $\mathbf{|Unique(DataAccceses_{X'}, TC)| = |Unique(DataAccceses_X, TC)| - 1}$, y por lo tanto **ATFD(X') = ATFD(X) - 1**

± En la literatura se encuentra la métrica CBO (coupling between objects) definida por Chidamber y Kemerer **[21]** como "un conteo del número de otras clases a las cuales ésta (la clase analizada) está acoplada". Esta métrica diría qué tanto se comunica esta clase con otras clases, lo cual, sumado al hecho de que al mismo tiempo muchas clases se comuniquen con esta, sería un indicativo de que la clase analizada puede ser un *middle man*. Sin embargo, no se da una definición formal de la misma, y dado que lo que interesa en este caso es el nivel de acoplamiento por invocaciones a métodos, se define la métrica ATFM (ver en seguida).

±

Nombre	ATFM (access to foreign methods)
Definición	Se define como el número de clases externas de las cuales la clase analizada accede a métodos.
Uso específico	El acceso a métodos *getters* se toma, efectivamente, como tal (acceso a método), y no como acceso a atributo.
Notación	**ATFM(X)**
Propiedades	**ATFM(X)** se calcula para toda clases $\mathbf{X \in Classes_{App}}$ **ATFM(X)** se calcula sobre $\mathbf{Messages_X}$ Rango: $\mathbf{ATFM(X) \in Z}$, y $\mathbf{0 \leq ATFM(X) \leq |Classes_{App}|}$ Valor cero: $\mathbf{ATFM(X) = 0}$ $\Leftrightarrow$ la clase X no hace accesos a métodos de alguna otra clase Valor máximo: $\mathbf{ATFM(X) = |Classes_{App}|}$ $\Leftrightarrow$ la clase X hace invocaciones a por lo menos un método de cada una de las otras clases. (↑): Suponga que $\mathbf{Messages_{X'} = Messages_X + Message(X)}$ Si $\mathbf{|Unique(Messages_{X'}, TC)| > |Unique(Messages_X, TC)|}$, entonces $\mathbf{|Unique(Messages_{X'}, TC)| = |Unique(Messages_X, TC)| + 1}$, y por lo tanto $\mathbf{ATFM(X') = ATFM(X) + 1}$

	(↓): Suponga que **Messages$_{x'}$ = Messages$_x$ - Message(X)** Si |**Unique(Messages$_{x'}$, TC)**| < |**Unique(Messages$_x$, TC)**|, entonces |**Unique(Messages$_{x'}$, TC)**| = |**Unique(Messages$_x$, TC)**| - **1**, y por lo tanto **ATFD(X') = ATFD(X) - 1**

±

Nombre	nr_afferent-refs (Number of afferent references)
Definición	Es el número de clases que usan una clase. Fue simplemente enunciada por Erni y Lewerentz **[30]**, y no se da una definición formal de ésta.
Uso específico	Esta métrica es el complemento de CBO (la que dice cuántas clases son usadas por una clase) en la decisión de si una determinada clase puede ser un *middle man* Para calcular esta métrica se deben calcular el número de clases que acceden a atributos de esta clase y el número de clases que acceden a métodos de esta clase. Para el cálculo de esta métrica sí se tienen en cuenta los accesos a atributos (los atributos de tipo ADT), porque podrían resultar en llamados a métodos de las clases delegadas.
Notación	**nr_afferent-refs(X)** **nr_afferent-refs(X) = ATFM_back(X) + ATFD_back(X)** donde[39] **ATFM_back(X)** es el número de clases que acceden a métodos de las clase X **ATFD_back(X)** es el número de clases que acceden a atributos de la clase X. (Ver definición de ATFD en el glosario)
Propiedades	Se cumplen para esta métrica las mismas propiedades de las métricas que la componen (ATFM y ATFD)

±

[39] Los sufijos 'back' en los nombres de estas métricas se usan para significar que la métrica mide lo mismo que la métrica original, pero en el sentido contrario; es decir, por ejemplo, **ATFM(X)** mide el número de clases de las que la clase X accede métodos, mientras que **ATFM_back(X)** mide el número de clases que acceden a métodos de X.

Nombre	RMC (relative method coupling)
Definición	Definida anteriormente (Ver definición en la sección de métricas usadas para *inappropriate intimacy*)
Uso específico	Se usa de la misma forma en que fue definida para *inappropriate intimacy*, excepto que no se calcula para un par de clases en particular, sino para la clase analizada con respecto a todas las otras clases; es decir, que se debe calcular **`RMC(X,m -> Y)`** para todos los métodos **`m`** que pertenecen a **`{X, Methods}`** (donde X es la clase analizada), y para todas las clases Y que pertenecen a **`{Classes}`**.
Notación	La misma expuesta anteriormente
Propiedades	La misma expuesta anteriormente

±

Nombre	NOEMA (number of external methods accessors[40]) NOHEMA (number of hidden external methods accessors)
Definición	NOEMA es el número de métodos que hacen invocaciones a métodos externos. NOHEMA es el número de métodos que no hacen invocaciones directas a métodos externos, sino que lo hacen invocando a métodos internos que sí hacen tales invocaciones; es decir, que lo hacen de forma 'oculta'.
Uso específico	Estas dos métricas son necesarias para poder determinar el porcentaje de métodos de una clase que son totalmente implementados por ella misma Esto es algo que tengo que decidir cuando identifique los patrones "parecidos" a este BS. En ese momento también decido si dejo o no NOHEMA (mejor no porque requeriría una iteración adicional, y sería difícil identificar cosas como un EMA que llama a otro EMA y a la vez este llama al HEMA)
Notación	**`NOEMA(X)`** **`NOHEMA(X)`**
Propiedades	A partir de los atributos de un mensaje (Ver definición de los atributos de un mensaje en la tabla de la métrica Number of accessed methods) se definen las funciones: **`- SourceMethod(Mi)`** := Retorna el nombre del método fuente del mensaje Mi.

[40] La palabra del inglés "accessor" no se encuentra en el diccionario, pero su uso es bastante extendido en la literatura existente que se refiere a lo mismo que aquí en el contexto de "accessor methods": métodos que se usan para obtener información de algún objeto.

	- TargetMethod(Mi) := Retorna el nombre del método destino del mensaje Mi. **- SourceClass(Mi)** := Retorna el nombre de la clase fuente del mensaje Mi. **- TargetClass(Mi)** := Retorna el nombre de la clase destino del mensaje Mi. Se da por obvio que **SourceClass(Mi) = X** $\forall$ **Mi** $\in$ **{X, Messages}** Se define $\mathbf{Messages\_out_x}$ como el conjunto de mensajes enviados por la clase **X** que son dirigidos hacia afuera; es decir, hacia otras clases. Entonces $\mathbf{Messages\_out_x = Messages_x - Messages_{x(x,x)}}$. A partir de la definición anterior se definen: * un método como método EMA (external methods accessor), así: **m** $\in$ $\mathbf{Methods_x}$ es **EMA** si $\exists$ **Mi** $\in$ $\mathbf{Messages\_out_x}$ tal que **SourceMethod(Mi) = m**. * un método como método HEMA (hidden external methods accessor), así: **m** $\in$ $\mathbf{Methods_x}$ es **HEMA** si $\exists$ **Mi** $\in$ $\mathbf{Messages_x}$ tal que **SourceMethod(Mi) = m** y **TargetMethod(Mi) = m2**, donde **m2** $\neq$ **m** y **m2** es EMA. Se puede entonces expresar NOEMA(X) como: **NOEMA(X) =** $\mathbf{|Unique(Messages\_out_x, SM)|}$. Donde **SM** = atributo "método fuente". **NOEMA(X) y NOHEMA(X)** se calculan para toda clase **X** $\in$ $\mathbf{Classes_{App}}$ **NOEMA(X) y NOHEMA(X)** se calcula sobre $\mathbf{Messages_x}$ **NOEMA(X) <= NOM(X)** **NOHEMA(X) < NOM(X)** **NOHEMA(X) > 0 => NOEMA(X) > 0** **NOEMA(X) = 0 => NOHEMA(X) = 0** Rango: **NOEMA(X)** $\in$ **Z**, y **0** $\leq$ **NOEMA(X)** $\leq$ $\mathbf{|Methods_x|}$

	NOHEMA(X) $\in$ **Z**, y **0** $\leq$ **NOHEMA(X)** $\leq$ $\lvert$**Methods**$_x$$\rvert$ **- 1** Valor cero: **NOEMA(X) = 0** $\Leftrightarrow$ ningún método de la clase X hace invocaciones a métodos de otras clases. **NOHEMA(X) = 0** $\Leftrightarrow$ la clase X no tiene declarado método alguno que haga llamados a métodos de otras clases a través de algún otro de sus métodos. Valor máximo: **NOEMA(X) =** $\lvert$**Methods**$_x$$\rvert$ $\Leftrightarrow$ todos los métodos de la clase X hacen invocaciones a métodos en otras clases. **NOHEMA(X) =** $\lvert$**Methods**$_x$$\rvert$ **- 1** $\Leftrightarrow$ todos los métodos de la clase X, excepto uno de ellos (el que hace las invocaciones hacia métodos en otras clases), hacen invocaciones de forma 'oculta' a métodos en otras clases. ($\uparrow$): Suponga que **Messages**$_{x'}$ **= Messages**$_x$ **+ Message(X,C)**, donde **C** $\in$ **Classes**$_{App}$ y **C** $\neq$ **X**. Si $\lvert$**Unique(Messages_out**$_{x'}$**, SM)**$\rvert$ > $\lvert$**Unique(Messages_out**$_x$**, SM)**$\rvert$, entonces $\lvert$**Unique(Messages_out**$_{x'}$**, SM)**$\rvert$ = $\lvert$**Unique(Messages_out**$_x$**, SM)**$\rvert$ **+ 1**, y por lo tanto **NOEMA(X') = NOEMA(X) + 1** ($\downarrow$): Suponga que **Messages**$_{x'}$ **= Messages**$_x$ **- Message(X,C)**, donde **C** $\in$ **Classes**$_{App}$ y **C** $\neq$ **X**. Si $\lvert$**Unique(Messages_out**$_{x'}$**, SM)**$\rvert$ < $\lvert$**Unique(Messages_out**$_x$**, SM)**$\rvert$, entonces $\lvert$**Unique(Messages_out**$_{x'}$**, SM)**$\rvert$ = $\lvert$**Unique(Messages_out**$_x$**, SM)**$\rvert$ **- 1**, y por lo tanto **NOEMA(X') = NOEMA(X) - 1** No se encontró una manera formal y no complicada de expresar a NOHEMA(X) como un conjunto para poder expresar formalmente las propiedades "cómo crece" y "cómo decrece", por lo tanto, éstas se expresan en términos de otras definiciones: ($\uparrow$): **NOHEMA(X') > NOHEMA(X)** si **Messages**$_{x'}$ **= Messages**$_x$ **+ Message(X,X)**;

	donde **Message(X,X)** es tal que ¬ ∃ **Mi ∈ Messages$_x$** tal que **SourceMethod(Mi) = SourceMethod(Message(X,X))** y **TargetMethod(Mi) = TargetMethod(Message(X,X)) = m,** donde m es EMA. En este caso, **NOHEMA(X') = NOHEMA(X) + 1** (↓): **NOHEMA(X') < NOHEMA(X)** si **Messages$_{x'}$ = Messages$_x$ - Message(X,X)**; donde **Message(X,X)** cumple las mismas condiciones expresadas en la propiedad anterior. En este caso, **NOHEMA(X') = NOHEMA(X) - 1**

±

Nombre	CSR (class self responsibility)
Definición	Es la medida de qué tan responsable es una clase; es decir, de qué tanto los métodos que ofrece son implementados en su totalidad en la clase sin tener que apoyarse en métodos externos. CSR es una métrica compuesta.
Uso específico	Esta métrica se introduce porque se sabe que una clase que presente el bad smell *middle man*, delegará la mayor parte de su funcionalidad a otra u otras, lo cual se puede expresar como que la clase es poco responsable por sí misma.
Notación	**CSR(X)** **CSR(X) = (NOEMA(X) + NOHEMA(X)) / max (1, NOM(X))** donde **NOM(X)** es el número de métodos declarados en la clase X.
Propiedades	El denominador se expresa como el máximo entre uno y el número de métodos de la clase, para evitar que dé algo indefinido cuando la clase no declara método alguno, y para expresar que esta métrica no tiene sentido si la clase no declara al menos un método. Rango: **CSR(X) ∈ Z**, y **0 ≤ CSR(X) ≤ 1** Valores característicos:

CSR(X)	=	⇒
	0	La clase es totalmente responsable por sí misma
	= 1, o	se podría decir que la clase es en mayor parte 'irresponsable' por la funcionalidad ofrecida

	→ 1	
	→ 0	se puede considerar a esta clase como una clase 'responsable'
	(0,1)	En este caso habría que establecer algún criterio de decisión.

Debido a la forma como está expresada esta métrica (su fórmula y su dependencia de métricas previamente definidas), las propiedades "cómo crece" y "cómo decrece" se pueden expresar así:

(↑):
Si **NOM(X)** se mantiene estable y **NOEMA(X)** o **NOHEMA(X)** crecen

(↓)
Si **NOM(X)** se mantiene estable y **NOEMA(X)** o **NOHEMA(X)** decrecen

B.2.3 Interpretación de las métricas

B.2.3.1 Definición de filtros

A diferencia del caso de *inappropriate intimacy* en donde una métrica es calculada en forma simétrica en dos clases, y una cualquiera de ellas puede servir para indicar la presencia del bad smell; en el caso de middle man, las métricas solo pueden indicar la 'mitad' del bad smell; es decir, o que una o varias clases delegan a la clase analizada, o que la clase analizada delega a una o varias clases. Es por esto que en la definición de filtros se usan los casos descritos en la caracterización del bad smell para señalar lo que estaría sucediendo en el caso de que el valor de la métrica se equipare al del filtro.

Se define la métrica NOC(App) como el número de clases de la aplicación analizada, simplemente para usar los mismos términos de notación en las expresiones usadas para los filtros. Esta métrica ya se ha usado antes en forma indirecta, puesto que **NOC(App) =** |**{Classes}**|.

- ATFM
  ATFM(X) / NOC(App) = z, donde **z** está entre [0,1] y representa el porcentaje de clases de la aplicación de las que la clase X accede a métodos. Si **z > 'variable',** se considera alto, y por lo tanto indicativo de que, se estarían presentando los casos 1 o 3.

- nr_afferent-refs
  nr_afferent-refs(X) / NOC(App) = z, donde **z** está entre [0,1] y representa el porcentaje de clases de la aplicación que usan o hacen referencia a la clase **X**. Si **z > 'variable',** se considera alto, y por lo tanto indicativo de que, se estarían presentando los casos 1 o 2.

¦ RMC

Si **RMC(X,m -> Y) > 1**, significa que el método **m** está más acoplado con la clase **Y** de lo que está en cohesión con su clase propietaria **X**. Esta métrica se puede filtrar desde dos perspectivas dadas por los casos eventualmente presentados:

° Casos 1 o 3. Como indicador de que los casos 1 o 3 se estarían presentando, y se puede tomar en dos sentidos:

- Si **NOWCM(X)** (ver definición en la sección de filtros, donde se define el filtro para RMC) **= 'variable' >= 1**, entonces se tiene en cuenta esta métrica.
- Si **NOWCM(X) / NOM(X) = z,** donde **z** está entre [0,1] y representa el porcentaje de métodos declarados en la clase **X** que están acoplados con otras clases. Si **z > 'variable',** se considera alto.

° Casos 1 o 2. Como indicador de que los casos 1 o 2 se estarían presentando.
Se define Cm-c (classes method-coupled), **Cm-c(X)** como el número de clases que tienen algún método que está más acoplado con X de lo que es cohesivo con ellas mismas, es decir,

Cm-c(X) = sum (NOWCM(C,X) > 1) $\forall$ **C** $\in$ **{Classes})**

Así, se puede tomar en dos sentidos:

- Si **Cm-c(X) = 'variable' >= 1**, entonces se tiene en cuenta esta métrica.
- Si **Cm-c(X) / NOC(App) = z,** donde **z** está entre [0,1] y representa el porcentaje de clases que tienen métodos acoplados a la clase **X.** Si **z > 'variable',** se considera alto.

¦ CSR

Como se explicó en la definición de esta métrica, si **CSR(X) = 1**, es un indicativo de que la clase no es responsable por su funcionalidad, así que se tendría un indicio de que es un *middle man*. También se puede ver CSR(X) como:
CSR(X) = z, donde **z** está entre [0,1] y representa el porcentaje de métodos de la clase X que hacen invocaciones a métodos en otras clases. Si **z > 'variable',** se considera alto, y por lo tanto se considera a X poco "responsable". Por definición de este bad smell, esta métrica es la más importante, por lo tanto debería asignársele mayor peso.

Los pesos de cada métrica son manejados por medio de los valores que para cada una se definen como indicadores de que la métrica va a ser tenida en cuenta.

B.2.3.2 Correlación de métricas

Para determinar si una clase se puede considerar como *Middle man*, se usa una expresión lógica que correlacione las métricas obtenidas. Las razones para usar una expresión tal son las mismas expuestas para el caso *Innappropriate intimacy*. Entonces, se define que la clase X es un *Middle man* si:

MM(X) $\Leftrightarrow$ **ATFM(X) / NOC(App) > V1** $\wedge$
nr_afferent-refs(X) / NOC(App) > V2 $\wedge$
NOWCM(X) / NOM(X) > V3 $\wedge$

```
Cm-c(X) / NOC(App) > V4 ∧
 CSR(X) > V5
```

Donde **Vi** está en el intervalo [0,1], y es la variable que representa el peso dado a cada métrica.

B.3 Aplicación de la estrategia de detección

Como se dijo anteriormente, la aplicación de la estrategia se lleva a cabo mediante la aplicación desarrollada para tal fin. Ver el anexo: Herramienta de software para la implementación de la propuesta.

Aunque no hace parte de los objetivos propuestos en este trabajo, a continuación se presenta la aplicación de la estrategia a uno de los *bad smells* que ya ha sido abordado en trabajos anteriores: *Feature envy*; en el trabajo de Carlos Angarita **[12]**. Esta parte se incluye para mostrar que la estrategia creada puede ser usada como base para cualquier trabajo de detección automática de *bad smells* basado en métricas de software: para formalizar los ya hechos, y para abordar casos no tratados con anterioridad.

Todas las definiciones, métricas y demás, están basadas en lo presentado en trabajo de Angarita. No se pretende ni dar una nueva estrategia para la detección de ese *bad smell*, ni complementar la expuesta allí, sino presentar la de ese autor de la forma en que podría haberlo sido siguiendo la estrategia propuesta aquí. Debido a esto algunas partes pueden estar incompletas (en especial la de las propiedades de las métricas). La mayoría del texto es tomado de forma literal de ese trabajo.

4.3 C. Feature envy

C.2 Definición de la estrategia de detección

C.2.1 Caracterización del bad smell

Nombre del bad smell	*Feature envy*. "un método parece estar más interesado en otra clase que la propia clase a la que pertenece" **[1]**
Características	El *bad smell Feature envy* cuestiona la presencia de un determinado método en una clase, sugiriendo la posibilidad de transportarlo a otra. Se dice que un método incorpora este *bad smell* cuando éste se caracteriza porque para cumplir su función requiere el uso de gran cantidad de información correspondiente a atributos o métodos de objetos de otras clases y relativamente poca información tomada de los propios atributos o métodos de la clase a la cual pertenece el método que se esta analizando. **[12]**

Heurísticas de diseño	La definición de acoplamiento lo define como una relación que se da entre dos artefactos de software a nivel de clases; sin embargo, este *bad smell* se podría definir como un acoplamiento unidireccional a nivel método-clase; es decir, que se da solo en una dirección: de la clase que declara el método "envidioso" hacia otra clase específica; y el análisis se hace a nivel de cada método de una clase con respecto a las demás clases que componen la aplicación analizada. La generalización de este *bad smell* a la mayoría -o todos- los métodos de una clase podría ser un indicativo de que también hay *Inappropriate intimacy* entre dos clases. La característica más importante que se debe tener en cuenta para determinar si un método presenta Feature envy con respecto a otra clase es el porcentaje de atributos y métodos de ésta que accede e invoca. El análisis se hace desde el punto de vista de las clases: cuál de éstas es la que utiliza más un determinado método. Si es la clase propietaria del método, todo está bien; si es otra clase, entonces esa clase tiene envidia por ese método.

Tabla 6. Caracterización del *bad smell Feature envy*

C.2.2 Escogencia de las métricas (GQM)

C.2.2.1 Definición de objetivos

El objetivo general es detectar la presencia del *bad smell Feature envy*.
Los objetivos específicos son:

- Para cada clase determinar
  - El porcentaje de atributos de la clase que son utilizados por algún método para el cual se está haciendo el análisis.
  - El porcentaje de métodos de la clase que son utilizados por algún método para el cual se está haciendo el análisis.

C.2.2.2 Derivación de preguntas

Las preguntas se derivan teniendo en cuenta que el análisis se hace sobre cada clase, una a la vez.

- **?** ¿Estar interesado en otra clase X corresponde a realizar en tiempo de ejecución un volumen alto de invocaciones a atributos o métodos de dicha clase X?
- **?** ¿Estar interesado en otra clase X corresponde a tener muchos objetos instancias de dicha clase X?

- ¿Se considera que hay un acceso a la clase X, cuando se utiliza un objeto Y atributo de esta clase X como puente para llegar a un atributo de la clase Y?
- ¿Se considera que hay un acceso a la clase X, cuando se utiliza un método de X que retorna un objeto Y, para a través de dicho objeto acceder a otro método de la clase Y?
- ¿Cómo se mide el interés en otra clase X, por cantidad de atributos accedidos de ésta o por número de invocaciones a atributos de ésta?
- ¿Cómo se mide el interés en otra clase X, por cantidad de métodos accedidos de ésta o por número de invocaciones a los métodos de ésta?
- ¿Cuando se va a medir el "interés" en otra clase se valoran por igual los accesos a atributos y a métodos de ésta?

C.2.2.3 Selección de métricas a usar

±

Nombre	NAA (Número de Atributos Accesibles)
Definición	Es el número de atributos de la clase que son accesibles desde el método al cual se le esta haciendo el análisis.
Uso específico	Suponga que X es la clase propietaria del método sobre el cual se esta realizando el análisis, y Y es la clase propietaria de los atributos a los que se les esta realizando un análisis para saber si son accesibles desde la clase X. Se asume que un atributo de la clase Y es accesible desde la clase X si cumple alguna de las siguientes condiciones: • Si las clases X y Y son la misma. • Si el atributo tiene modificador de acceso "public". • Si el atributo tiene modificador de acceso "protected" o sin especificar y las clase X e Y pertenecen al mismo paquete. • Si el atributo tiene modificador de acceso "protected" y la clase X hereda de la clase Y.
Notación	**NAA(X, m@Y)**[41] **=** Número de atributos de la clase **X** que son accesibles desde el método **m** de la clase **Y**
Propiedades	**NAA(X, m@Y)** se calcula para todos los métodos $\mathbf{Methods_X}$ para todas las $\mathbf{X \in Classes_{App}}$ Rango: $\mathbf{NAA(X, m@Y) \in Z}$, y $\mathbf{0 \leq NAA(X, m@Y) \leq |Attributes_X|}$ Valor cero: $\mathbf{NAA(X, m@Y) = 0}$ ⇔ ninguno de los atributos de la clase X se puede acceder desde el método m de la clase Y. Valor máximo: $\mathbf{NAA(X, m@Y) = |Attributes_X|}$ ⇔ todos los atributos de la clase X se puede acceder desde el método m de la clase Y.

[41] Notación propuesta por el autor. Igual para las demás métricas de este bad smell

±

Nombre	NMA (Número de Métodos Accesibles)
Definición	Es el número de los métodos de la clase que son accesibles desde el método al cual se le esta haciendo el análisis
Uso específico	Suponga que X es la clase propietaria del método sobre el cual se esta realizando el análisis, y Y es la clase propietaria de los métodos a los que se les esta realizando un análisis para saber si son accesibles desde la clase X. Se asume que un método de la clase Y es accesible desde la clase X si cumple alguna de las siguientes condiciones: • Si las clases X y Y son la misma. • Si el método tiene modificador de acceso "public". • Si el método tiene modificador de acceso "protected" o sin especificar y las clase X e Y pertenecen al mismo paquete. • Si el método tiene modificador de acceso "protected" y la clase X hereda de la clase Y.
Notación	**NMA(X, m@Y) =** Número de métodos de la clase **X** que son accesibles desde el método **m** de la clase **Y**
Propiedades	**NMA(X, m@Y)** se calcula para todos los métodos $\mathbf{Methods}_{X}$ para todas las $\mathbf{X} \in \mathbf{Classes}_{App}$ Rango: $\mathbf{NMA(X, m@Y)} \in \mathbf{Z}$, y $0 \leq \mathbf{NMA(X, m@Y)} \leq |\mathbf{Methods}_{X}|$ Valor cero: $\mathbf{NMA(X, m@Y)} = 0 \Leftrightarrow$ ninguno de los métodos de la clase X se puede acceder desde el método m de la clase Y. Valor máximo: $\mathbf{NAA(X, m@Y)} = |\mathbf{Methods}_{X}| \Leftrightarrow$ todos los métodos de la clase X se puede acceder desde el método m de la clase Y.

±

Nombre	NAU (Número de Atributos utilizados)
Definición	Es el número de atributos utilizados en las instrucciones del método que se esta analizando.
Uso específico	Se contabilizan los accesos ya sean directos o indirectos. A continuación

	se muestra un ejemplo para clarificar a que se llama acceso indirecto a un atributo:
	
	[Instrucción 1]
	`Obj1.nameAttribute1InClassX.nameAttribute1InClassY=5;`
	
	[Instrucción 2]
	`Obj1.nameMethod1InClassX().nameAttribute1InClassY=5;`
	
	En la instrucción 1, se tiene un objeto llamado Obj1 que es instancia de una clase que se llamará X. Dicha clase tiene un atributo llamado nameAttribute1InClassX que es un objeto instancia de una clase que se llamará Y, la cual a su ves tiene un atributo numérico llamado nameAttribute1InClassY. En esta instrucción se contabiliza un acceso a un atributo en la clase X considerado acceso directo y un acceso a un atributo de la clase Y considerado acceso indirecto.
	En la instrucción 2, se tiene un objeto llamado Obj1 que es instancia de una clase que se llamará X. Dicha clase tiene un método llamado nameMethod1InClassX() que retorna un objeto instancia de una clase que se llamará Y, la cual a su ves tiene un atributo numérico llamado nameAttribute1InClassY. En esta instrucción se contabiliza un acceso a un método en la clase X considerado acceso directo y un acceso a un atributo de la clase Y considerado acceso indirecto.
	También son considerados accesos indirectos, los efectuados a atributos o métodos de clases ancestros.
	
	[Instrucción 3]
	`Obj1ClassX.nameAttribute1InClassW=5;`
	
	En la instrucción 3, se tiene un objeto llamado Obj1 que es instancia de una clase que se llamará X. Dicha clase tiene como ancestro una clase llamada W, la cual tiene un atributo llamado nameAttribute1InClassW. En esta instrucción se contabiliza únicamente un acceso a un atributo de la clase W considerado acceso indirecto.

Notación	**NAU(X, m@Y) =** Número de atributos de la clase **X** que son utilizados por el método **m** de la clase **Y**
Propiedades	**NAU(X, m@Y)** se calcula para todos los métodos $\mathbf{Methods_x}$ para todas las $\mathbf{X \in Classes_{App}}$ Rango: $\mathbf{NAU(X, m@Y) \in Z}$, y $0 \leq \mathbf{NAU(X, m@Y)} \leq |\mathbf{Attributes_x}|$ Valor cero: $\mathbf{NAU(X, m@Y) = 0} \Leftrightarrow$ ninguno de los atributos de la clase X es utilizado por el método m de la clase Y. Valor máximo: $\mathbf{NAU(X, m@Y)} = |\mathbf{Attributes_x}| \Leftrightarrow$ todos los atributos de la clase X son utilizados por el método m de la clase Y.

±

Nombre	NMU (Número de métodos utilizados)
Definición	Es el número de métodos utilizados en las instrucciones del método que se esta analizando.
Uso específico	Se contabilizan los accesos ya sean directos o indirectos. Igual que en los casos expuestos en la métrica NAU.
Notación	**NMU(X, m@Y) =** Número de métodos de la clase **X** que son utilizados por el método **m** de la clase **Y**
Propiedades	**NMU(X, m@Y)** se calcula para todos los métodos $\mathbf{Methods}_X$ para todas las $\mathbf{X} \in \mathbf{Classes}_{App}$ Rango: $\mathbf{NMU(X, m@Y)} \in \mathbf{Z}$, y $0 \leq \mathbf{NMU(X, m@Y)} \leq \lvert\mathbf{Methods}_X\rvert$ Valor cero: $\mathbf{NMU(X, m@Y)} = 0 \Leftrightarrow$ ninguno de los métodos de la clase X es utilizado por el método m de la clase Y. Valor máximo: $\mathbf{NAU(X, m@Y)} = \lvert\mathbf{Methods}_X\rvert \Leftrightarrow$ todos los métodos de la clase X son utilizados por el método m de la clase Y.

±

Nombre	FUM (Factor de utilización de Métodos)
Definición	Es el porcentaje de los métodos funcionales que se están utilizando en la clase a la cual se le calcula la métrica.
Uso específico	Esta métrica no se calcula a partir del análisis del código fuente sino como el cociente de NMU entre NMA. Este cociente genera un valor entre 0 y 1. En el caso que NMA tenga un valor de cero, se le asigna un valor de cero a FUM.
Notación	
Propiedades	

±

Nombre	FUA (Factor de utilización de atributos)

Definición	Es el porcentaje de los atributos accesibles que se están accediendo directamente o a través de métodos Getters o Setters en la clase a la cual se le calcula la métrica.
Uso específico	Esta métrica no se calcula a partir del análisis del código fuente sino como el cociente de NAU entre NAA. Este cociente genera un valor entre 0 y 1. En el caso que NAA tenga un valor de cero, se le asigna un valor de cero a FUA.
Notación	
Propiedades	

±

Nombre	FUC (Factor de utilización de clase)
Definición	Es el nivel de pertenencia del método a la clase, en función de la dependencia que tiene dicho método de la información de los atributos y operaciones de los métodos de dicha clase.
Uso específico	Esta es la métrica que finalmente se usa para determinar la presencia o ausencia del *bad smell Feature envy,* y también la que indica como aplicar el Refactoring correspondiente. El valor del FUC indica qué porcentaje de la funcionalidad que ofrece una clase, esta siendo utilizado por algún método.
Notación	Dado que FUA y FUM son métricas cuyos valores están contenidos en el rango de 0 a 1, y que se desea que la tercera métrica FUC, también esté en el rango de 0 a 1, se utiliza una fórmula clásica de asignación de pesos a FUA y a FUM para calcular el FUC. La fórmula para calcular el FUC es la siguiente: FUC = (WeightAttributes * FUA) + (WeightMethods*FUM) Donde: 0 <= WeightAttributes, WeightMethods <=1 WeightAttributes + WeightMethods = 1
Propiedades	

C.2.3 Interpretación de las métricas

C.2.3.1 Definición de filtros

- TA (Total de accesos). Es la sumatoria del total de accesos a la clase -TAC- de todas las clases que son invocadas directa o indirectamente desde las instrucciones del método. Se calcula a nivel de método.

- PP (Porcentaje Participación). Es el cociente entre el TAC de la clase y el TA del método analizado. Si existe una clase que tenga un valor de PP superior al de la clase propietaria del método analizado, se dice que dicho método presenta *Feature envy*.

C.2.3.2 Correlación de métricas

Parece que la correlación de métricas se hace en la métrica FUC.

C.3 Aplicación de la estrategia de detección

Para este *bad smell*, dado que se incluyó solo como demostración, no se hizo la aplicación de la estrategia.

5. Herramienta de software para la implementación de la propuesta

En este capítulo se explica la arquitectura y el funcionamiento de la aplicación desarrollada para implementar las estrategias propuestas.

Aunque no fue planteado como uno de los objetivos de este trabajo, luego de haber desarrollado la herramienta que implementa la aplicación de la estrategia en los casos abordados surgió la idea, como aporte adicional, de organizar los algoritmos creados para hacer el cálculo de las métricas requeridas como un conjunto de herramientas (toolkit). Esto se hizo organizando en un archivo jar las clases que contienen la lógica para el cálculo de cada métrica y las clases complementarias necesarias, de tal forma que cualquier usuario pudiera usar tales métricas para definir diferentes fórmulas para la detección de los bad smells Inappropriate intimacy o Middle man, o cualquier otro en donde la estrategia de detección propuesta incluya alguna de estas métricas.

A continuación se presenta una descripción de la herramienta resultante y un pequeño manual para su uso.

5.1 Descripción de la herramienta

Descripción de los paquetes

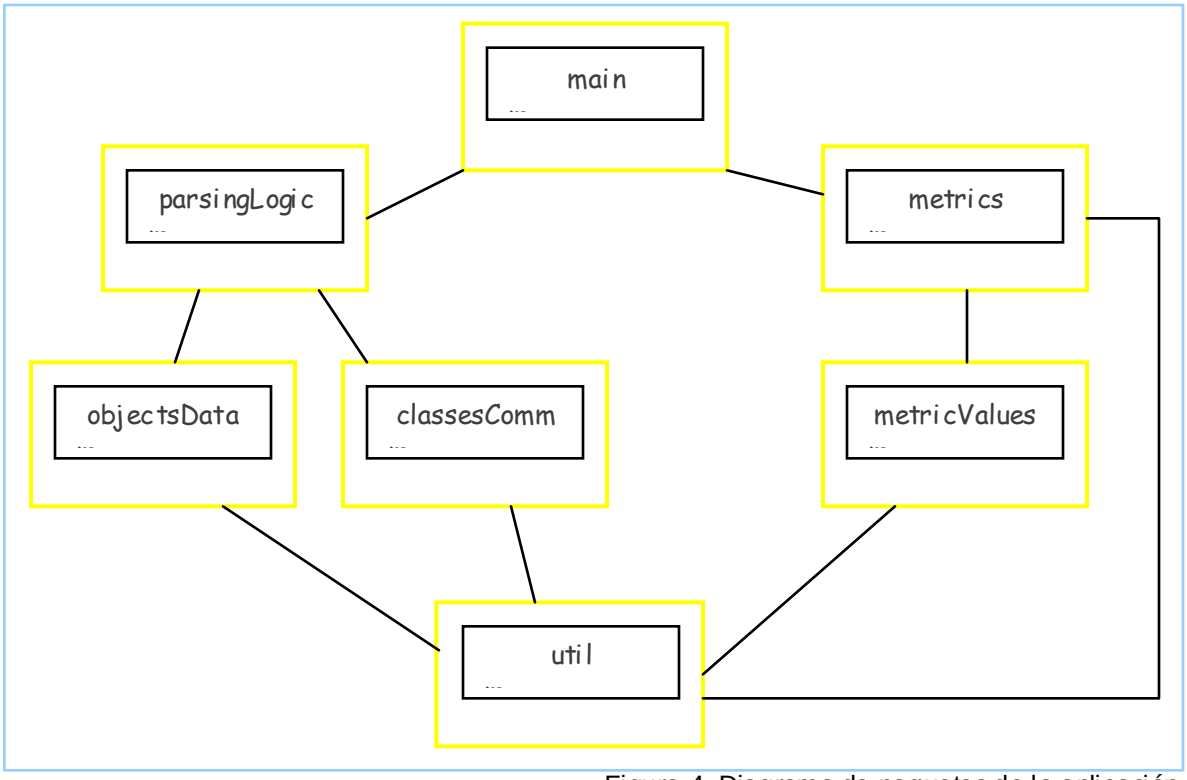


Figura 4. Diagrama de paquetes de la aplicación

La herramienta se compone de los siguientes paquetes que agrupan las clases según sus funcionalidades:

- util. Contiene clases con métodos que son de utilidad general.
- classesComm. Contiene las clases que manejan los tipos de comunicación tenidos en cuenta entre las clases: acceso a atributos (data acesses) e invocaciones de métodos (messages). En el anexo Propiedades de los conjuntos sobre los que se trabaja, se define la sintaxis en donde se especifica qué son accesos a atributos e invocaciones de métodos.
- objectsData. Contiene las clases que manejan las estructuras de datos necesarias para almacenar los datos de las clases, necesarios para el cálculo de algunas métricas.
- metricValues. Contiene las clases que se usan como estructuras para almacenar los diferentes tipos de valores de métricas.
- parsingLogic. Contiene las clases que manejan la lógica del análisis sintáctico (parsing) de los archivos fuente.
- metrics. Contiene las clases que manejan la lógica del cálculo de cada una de las métricas.
- main. Contiene la clase que es utilizada por el usuario.

Descripción de las clases

Para hacer esta descripción, se presentará la aplicación separada por paquetes. Para cada uno se presenta el diagrama de las clases que lo componen, y la descripción de la funcionalidad de éstas.

Paquete metrics

Cada una de las clases xx_metric, es la que contiene la lógica necesaria para el cálculo de la métrica correspondiente. Todas éstas heredan de la clase `Metric`, cuya única finalidad es poder obtener un objeto de ese tipo al ejecutar el método `calculate` en la clase `MT` (la clase principal de la aplicación). `Metric` no es un interfaz ni una clase abstracta porque el método `calculate` (que sería el que las clases que la implementaran o heredaran de ella) recibe diferentes número y tipos de parámetros para el cálculo de cada métrica.

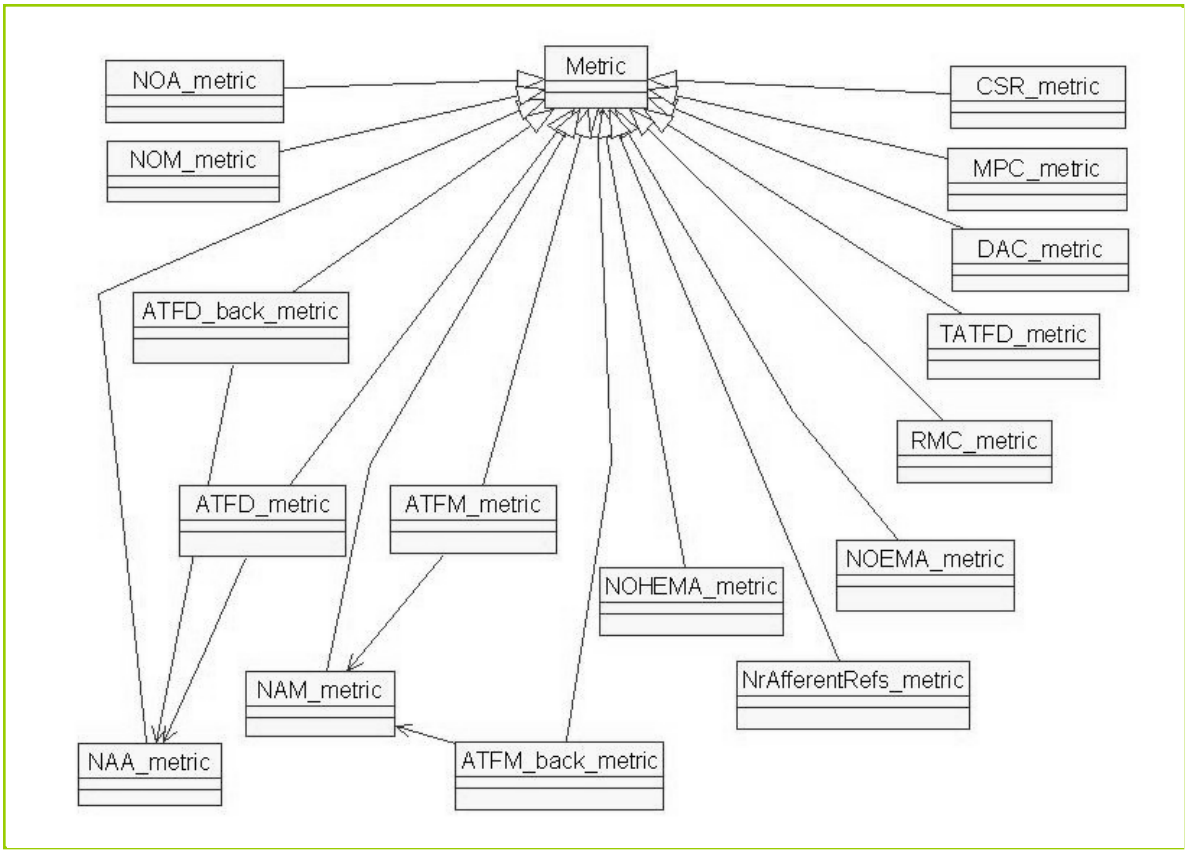


Figura 5. Diagrama de clases del paquete metrics

Paquete parsingLogic

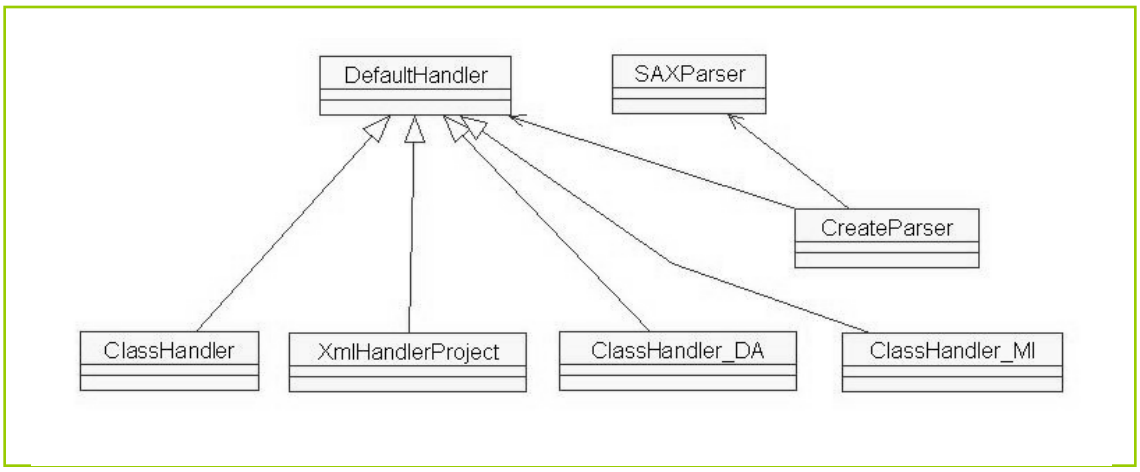


Figura 6. Diagrama de clases del paquete parsingLogic

Las clases handler contienen la lógica del análisis sintáctico de los archivos XML, y son las que se usan como parámetro handler para los parsers SAX; así:

`XmlHandlerProject`: para el análisis del archivo descriptor del proyecto.

`ClassHandler`: para el primer análisis de los archivos de clases.

`ClassHandler_DA`: para el análisis de los statements de accesos a atributos entre las clases.

`ClassHandler_MI`: para el análisis de los statements de invocaciones a métodos entre las clases.

La clase `CreateParser` une objetos handler con un parser SAX.

Paquete classesComm

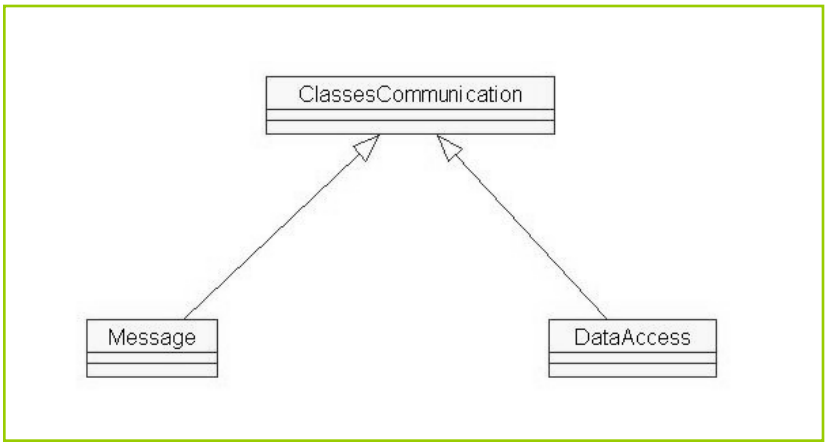


Figura 7. Diagrama de clases del paquete classesComm

Las clases `Message` y `DataAccess` contienen las estructuras para almacenar la información de los mensajes y los accesos a atributos, respectivamente. Cualquier comunicación entre clases (invocación de método o acceso a atributo) tiene como componentes: la clase fuente (la que envía la comunicación), la clase destino (la que recibe la comunicación), y el método fuente (método de

la clase fuente desde donde se origina la comunicación); estos son los atributos de la clase `ClassesCommunication`, de la cual heredan las otras dos.

Paquete objectsData

Las clases `LocalVariableProperties`, `AttributeProperties` y `MethodProperties`, como su nombre lo indica, tienen las estructuras para almacenar la información acerca de las variables locales, atributos y métodos de una clase. Las clases `*List*` manejan las listas de objetos respectivos (ejemplo: la clase `LocalVariablesList` maneja las listas de objetos `LocalVariableProperties`). La clase `StatisticsClass` es la que maneja toda la información pertinente de una clase

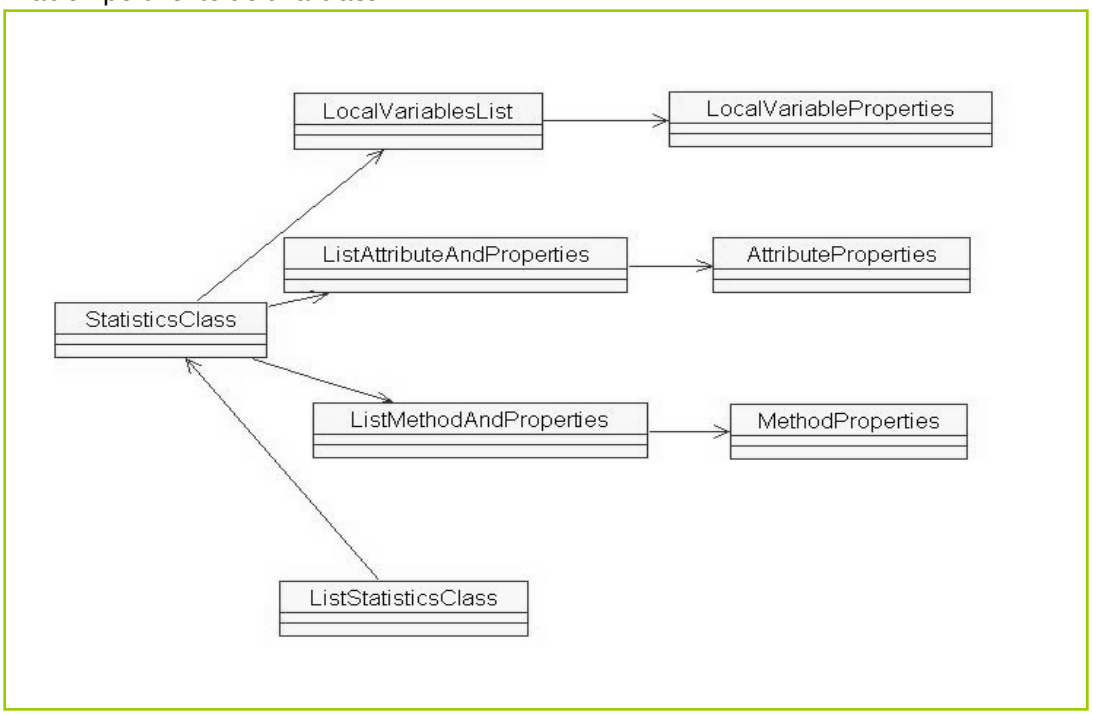


Figura 8. Diagrama de clases del paquete objectsData

.

Paquete metricValues

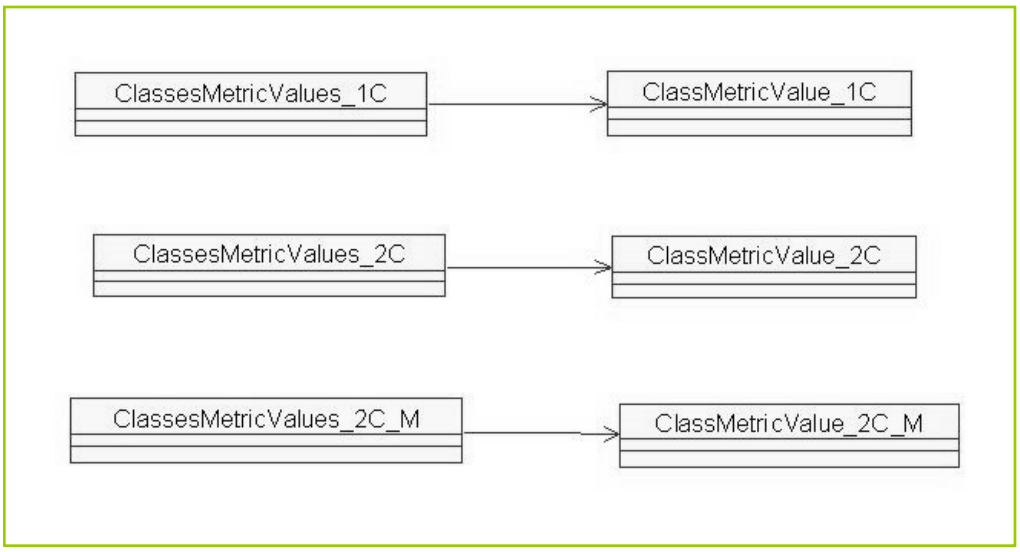


Figura 9. Diagrama de clases del paquete metricValues

Las clases `ClassMetricValue_*` son las que contienen las estructuras para almacenar los valores de los diferentes tipos de métricas:

`ClassMetricValue_1C`: valores de métricas que se obtienen para una clase y con respecto a ella misma; por ejemplo: NOM (number of methods).

`ClassMetricValue_2C`: valores de métricas que se obtienen para una clase -fuente- con respecto a otra clase -destino-; por ejemplo: MPC (message-passing coupling).

`ClassMetricValue_2C_M`: valores para métricas que se obtienen a nivel de -un- método de una clase fuente con respecto a una clase destino.

Las clases `ClassesMetricValues_*` agrupan objetos con valores individuales de métricas del tipo respectivo.

5.1.1 ¿Qué hace?

La herramienta ofrece una clase principal (MT) que es la que se usa de forma directa por medio de sus métodos `calculate` y `getInstance`. El método `calculate` que recibe como parámetro un número que indica la métrica a calcular implementa métodos para el cálculo de cada métrica y retorna un objeto de tipo `Metric` con la respuesta: los valores de la métrica para cada una de las clases o métodos (según la métrica calculada) de la aplicación analizada.
El usuario decide qué hacer con los valores obtenidos.

5.1.2 ¿Cómo lo hace?

La lógica del cálculo de cada métrica está implementada en las clases respectivas de manera independiente, lo que evita que el usuario se preocupe por el cálculo de métricas que son, por llamarlas de alguna manera, prerrequisito (se necesita de éstas para el cálculo de las otras); además facilita que se puedan hacer composiciones de métricas con más facilidad.

El funcionamiento de la aplicación se puede dividir en tres procesos: un primer y segundo recorridos sobre los archivos XML, y el cálculo de las métricas a partir de la información obtenida en estos dos recorridos. Los archivos XML son los que da como resultado la ejecución de la herramienta XScore, que fue la herramienta de producción de representación de código fuente escogida en este trabajo; por lo tanto, el funcionamiento de la herramienta desarrollada es dependiente del formato que XScore da a los archivos XML. Estos archivos son: uno que describe el proyecto (nombre y clases que lo componen), y uno por cada clase que compone la aplicación analizada.

A continuación se especifica qué se hace en cada uno de los procesos:

1. Primer recorrido sobre los archivos XML. En este se evalúa el formato de los archivos, y para cada clase se obtienen datos como su nombre, paquete al que pertenece, atributos y métodos declarados.

2. Segundo recorrido sobre los archivos XML. En este, el más importante, se obtienen los datos que servirán como base para el cálculo de las métricas; éstos son: los mensajes enviados y los atributos accedidos entre las clases; y se obtienen por medio del análisis sintáctico (parsing) de los archivos.

3. Cálculo de las métricas. Éste es el que queda en manos del usuario, quien, como parte de la estrategia diseñada por él, decide los límites permitidos para cada métrica y la composición de las mismas, lo cual servirá como criterio de decisión en la detección de algún bad smell.

Cuando el usuario invoca el método `MT.getInstance` para obtener una –la única- instancia de la xclase `MT` se llevan a cabo –la primera vez- los dos primeros procesos, con lo que los datos necesarios quedan almacenados para permitir el cálculo de las métricas.

5.1.3 Características importantes

A continuación se listan las características más relevantes que el supuesto usuario de la aplicación debería tener en cuenta. Algunas de estas pueden constituir restricciones para su uso.

- Solo se analizan las clases que pertenecen a la aplicación analizada; es decir, los posibles bad smells que haya entre clases de la aplicación analizada y las clases de JAVA o clases de otros APIs no son tenidos en cuenta.

- Los pesos de las métricas son parámetros para la detección. Es el usuario quien debe encargarse de implementar métodos para que el usuario final pueda seleccionarlos.

- No se tienen en cuenta las interfaces.

- No se trata el caso de múltiples clases con el mismo nombre.

- Los métodos y atributos pertenecientes a clases internas son considerados como pertenecientes a la clase de nivel más alto (top level class) que las declara; así, los accesos a atributos e invocaciones a métodos de clases internas se cuentan como tales para la clase que declara tales clases internas.

- Las invocaciones a métodos heredados se toman como invocaciones a métodos de la clase.

- Un acceso a atributos con la forma `DataAccess = anything` no se toma como tal.

- Los llamados a métodos getter, si bien podrían ser tomadas como acceso a datos, son tomadas como invocaciones a métodos.

- No se tienen en cuenta tratan las clases locales (las declaradas dentro de métodos).

- Las variables locales, sin importar que su tipo sea el de una clase de la aplicación, son tratadas como tales.

- En un statement de asignación, se cuentan como accesos a datos los que estén al lado derecho de este. Por ejemplo, en `ClassY.attributeA = ClassX.attributeA` hay un acceso a datos, y es `ClassX.attributeA`.

Cosas para arreglar en futuras versiones

- Como los métodos y atributos de las clases internas son tratados como si fueran de la clase de alto nivel que las contiene, cuando se quiere determinar cuál fue el método más invocado o el atributo más accedido, no se puede hacer correctamente debido a que JAVA permite que las clases internas declaren métodos y atributos con los mismos nombres de métodos y atributos de la clase más alto nivel; por lo tanto, al hacer el conteo de mensajes enviados a algún método mi no se hace diferencia sobre si éste pertenece a la clase de más alto nivel o a alguna de sus clases internas (lo mismo sucede para los atributos), por lo que puede que se obtenga un resultado erróneo. Por ejemplo: si el método foo de la clase X es accedido 10 veces, el método goo –de X- es accedido 6 veces, y el método goo de la clase X_interna (una clase interna a la clase X) es accedido 7 veces; el conteo dirá que el método más accedido de la clase X es goo (13 veces).

- Hacer que sea posible el análisis de aplicaciones con nombres iguales.

5.2 Manual de uso

Esta sección se divide en dos partes: herramienta genérica y herramienta para la detección de los bad smells abordados. En la primera se describe lo que un usuario que desee implementar una herramienta para detección de bad smells debería hacer para usar el conjunto de herramientas para cálculo de métricas; y en la segunda, la forma en que el usuario de la aplicación para la detección de los bad smells abordados interactúa con esta.

5.2.1 Manual de la herramienta genérica

Esta está en forma de un archivo .jar, así que éste debe ser incluido en el classpath del proyecto. La clase con la que el usuario interactúa es MT.
A continuación se presenta un segmento de código en que se estaría haciendo la búsqueda de un supuesto bad smell *excessive number of attributes* (número excesivo de atributos), para ilustrar el uso de la herramienta. Paso a paso se explicará el significado y objetivo del código.

```
10 MT myMT = MT.getInstance("c:\\temp\\project.xml");
11 NOA_metric noa = null;
12 try {
13     noa = (NOA_metric) myMT.calculate(1);
14 } catch (metricException e) {
15     e.printStackTrace();
16 }
17 ListStatisticsClass classList = MT.getAListStatisticsClass();
18 Enumeration classes_SC = classList.keys();
19 String currentSourceClassName;
20 int noaX;
21 while (classes_SC.hasMoreElements()) {
22     currentSourceClassName = (String) classes_SC.nextElement();
23     noaX = noa.noa_values.getMetricValue(currentSourceClassName);
24     if (noaX >= NOAlimit) {
25         //do something;
26     }
27 }
```

Figura 10. Ejemplo de código para el uso de la herramienta

1. Obtención de la única instancia de la clase MT – línea 10-. El parámetro que se pasa a esta función es la ruta del archivo XML descriptor del proyecto a analizar.
2. Cálculo de la métrica –línea 13-. Se invoca el método `calculate` con el parámetro que indica la métrica que se quiere calcular; éstos valores está en la documentación de la herramienta.
3. Análisis del bad smell –líneas 17 a 25-. Línea 17: se obtiene la lista de clases que se va a recorrer en el análisis (todas las pertenecientes al proyecto). Línea 23: se obtiene el valor de la métrica –previamente calculada- para la clase analizada en ese momento. Luego se aplica el criterio de decisión de la presencia del bad smell –línea 24-; en este caso, éste dice simplemente que si el número de atributos declarados en la clase es mayor que cierto límite determinado, se haga algo (ese algo puede ser ir guardando los resultados en alguna estructura para posteriormente mostrar los resultados de análisis).

5.2.2 Manual de la herramienta para detección de Inappropriate intimacy y Middle man

La interacción del usuario con esta aplicación es básicamente para escoger el bad smell que se quiere detectar, el proyecto que se va a analizar, y los límites de cada una de las métricas. El resultado es un archivo XML en el que se muestran, si se detectó el bad smell, para cada clase los valores de cada métrica y las clases con respecto a las cuales se está presentando el bad smell.
Se presentan las pantallas de interacción, y se dice lo que se espera del usuario en cada una.

- Pantalla para escoger el bad smell que se quiere detectar. En ésta se selecciona el bad smell que se quiere detectar.

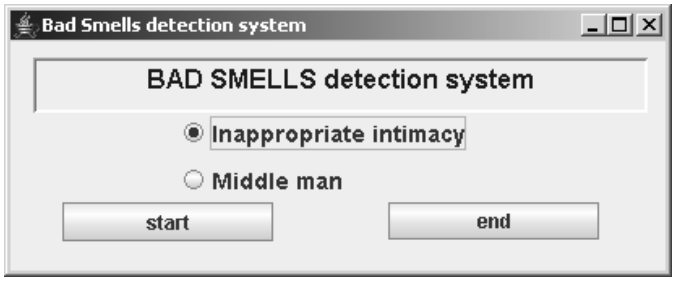


Figura 11. Pantalla de selección del bad smell a analizar

- Pantalla para escoger el archivo descriptor del proyecto. En ésta se define la ruta en donde se encuentra el archivo XML descriptor del proyecto.

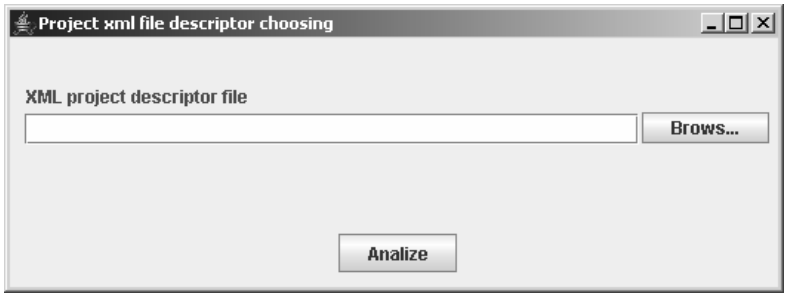


Figura 12. Pantalla de selección del archivo descriptor

Las siguientes dos pantallas se usan para definir los valores límites para cada una de las métricas, dependiendo del bad smell que se va a analizar.

- Pantalla en la que se definen los valores de porcentajes límites permitidos para cada métrica usada en la detección de *inappropriate intimacy* (si se seleccionó este bad smell)

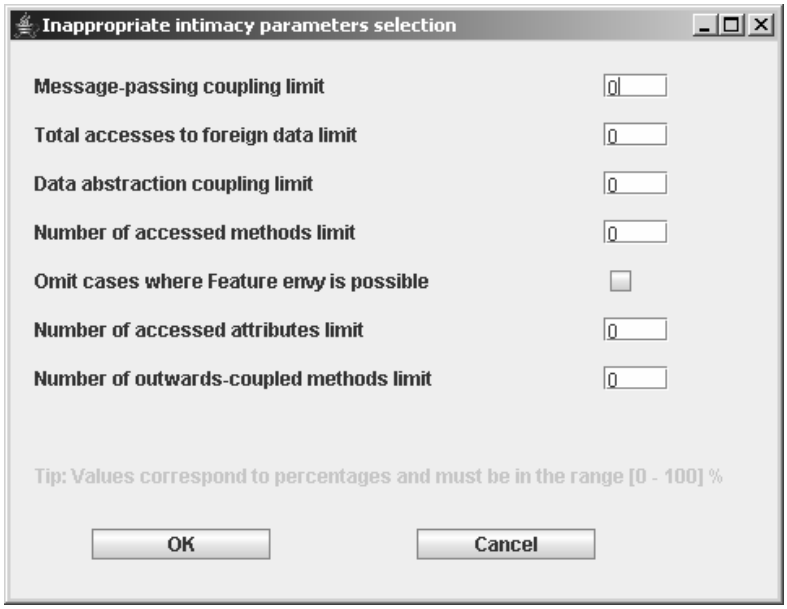


Figura 13. Pantalla para definir los límites de las métricas para Inappropriate intimacy

- Pantalla en la que se escogen los valores de porcentajes límites permitidos para cada métrica usada en la detección de *middle man* (si se seleccionó este bad smell).

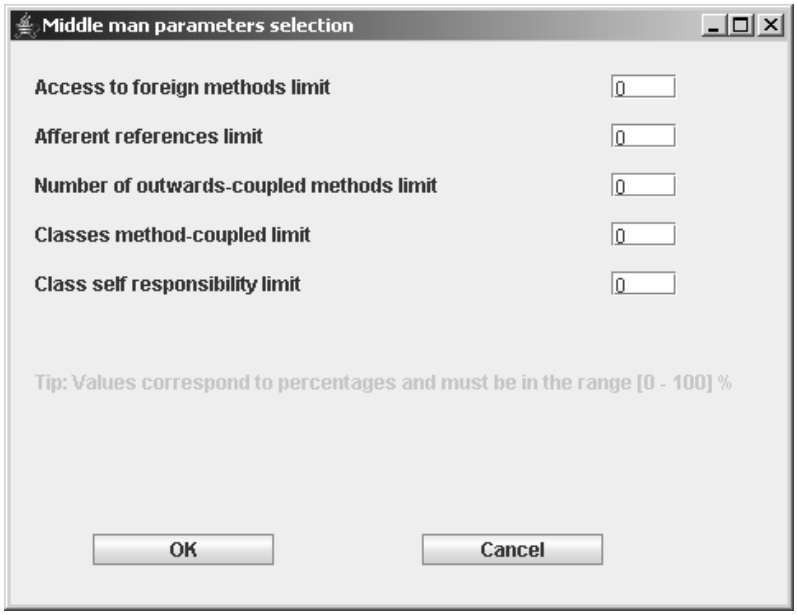


Figura 14. Pantalla para definir los límites de las métricas para Middle man

- Pantalla en la que se muestran los resultados del análisis.

Se muestran las pantallas correspondientes al análisis de ambos bad smells.

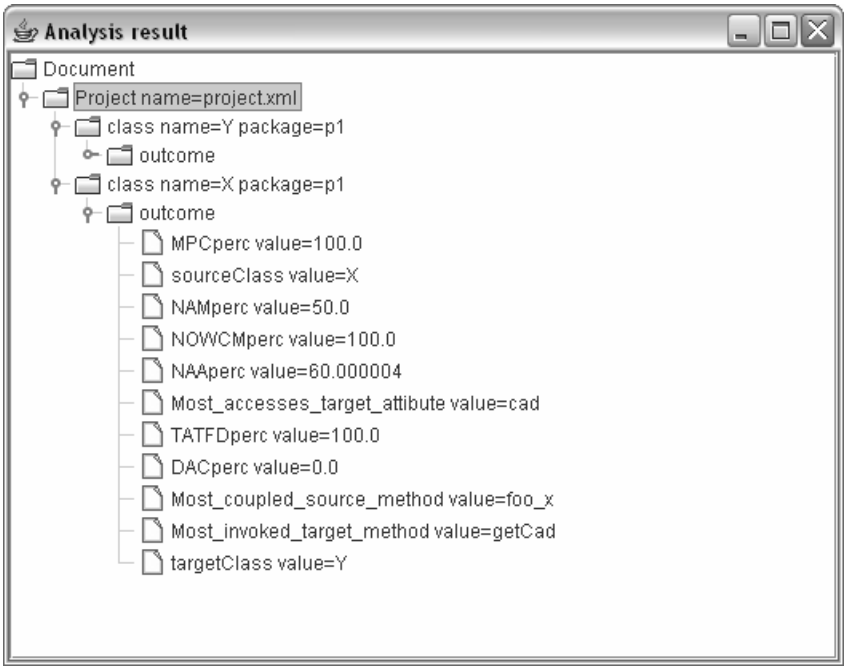


Figura 15. Pantalla de resultados del análisis de i*nappropriate intimacy*

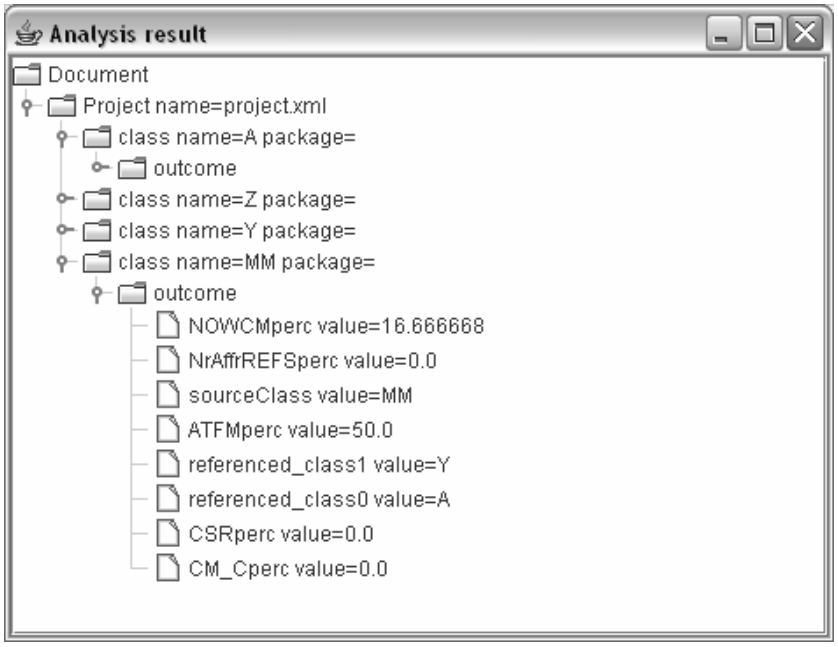


Figura 16. Pantalla de resultados del análisis de *middle man*

6. Conclusiones

En esta sección se hace un resumen de lo que se concluye como resultado del trabajo desarrollado haciendo un paralelo entre lo que se quería lograr y lo que finalmente se consiguió. También se dan algunas pautas de cómo se podría continuar en el futuro con este trabajo u otros que estén en la misma línea de investigación.

6.1 Análisis de resultados

En esta sección se listan los resultados de más relevancia obtenidos con el desarrollo de este trabajo.

6.1.1 En cuanto al objetivo principal de la tesis

El objetivo principal de este trabajo era la definición de una estrategia que pudiera ser aplicada a los casos de detección automática de bad smells para lograr una aproximación más ordenada y formal a éstos. El resultado de aplicar la estrategia a los tres bad smells escogidos se puede considerar como exitoso puesto que se cumplen los objetivos, y además se muestra su efectividad para -de cierta forma- aclarar estrategias previamente desarrolladas.

6.1.2 En cuanto a las estrategias propuestas para cada uno de los bad smells

Aunque no se pretende afirmar que las estrategias desarrolladas para los dos *bad smells* abordados son infalibles ni exactas para todos los casos, sí se puede afirmar que en casos que muestran situaciones típicas de la presencia de éstos, los resultados de la detección, con algo de ajuste en los niveles límites de las métricas, son aceptables.

El *bad smell* que presentó más dificultad para su detección fue *middle man*, ya que sus características hacen que se deban modificar bastante los límites de las métricas dependiendo del número de clases de la aplicación y del caso específico tratado (Ver caracterización).

La herramienta desarrollada se probó, entre otros, con los casos diseñados para la encuesta; para éstos los resultados fueron bastante acertados. También se hicieron pruebas con ejemplos en donde se representaban patrones de diseño (composite, mediator, facade, proxy, entre otros); sin embargo, con éstos no se llegó a resultados concluyentes.

6.2 Análisis de la solución

Los objetivos de esta tesis fueron modificados a lo largo de su desarrollo. La idea inicial sí era seguir con la línea de investigación en el campo de la detección automática de *bad smells*, pero se pensó solo en abordar algunos que no hubieran sido tratados ya en trabajos anteriores, o que por lo menos este hecho no estuviera documentado en la extensa literatura consultada al respecto. Se decidieron los *bad smells* que se iban a atacar, y se decidió usar el enfoque de métricas de

software para hacerlo, ya que se había comprobado su efectividad en trabajos anteriores, y los demás -enfoques- están poco documentados y se desconoce su validez.
Se propuso entonces inicialmente como objetivo abordar el análisis de los *bad smells inappropriate intimacy* y *middle man*: documentarlos, decidir las métricas que se deberían usar para su detección, construir fórmulas para correlacionarlas, y construir una herramienta de software que calculara tales métricas y llevara a cabo la detección.

Durante el desarrollo del trabajo, se detectó la principal falencia de la forma en la que se estaba llevando a cabo, que es la misma presente en trabajos anteriores: la informalidad con que se aproximaban al problema los autores. En algunos casos, ni siquiera se tenía claro qué era lo que se quería detectar, pues no había definiciones completas de los *bad smells* tratados; en otros, las métricas que se usaba no eran definidas correctamente, por lo que no quedaba claro qué es lo que ellas medían del código. Estos hechos se constituyeron el la principal motivación para tomar un rumbo un tanto diferente y plantear nuevos objetivos para esta tesis.

Los nuevos objetivos, centrados en el objetivo principal de proponer una estrategia que sirviera para abordar tareas de detección automática de *bad smells* basándose en el uso de métricas de software, quedaron así sentados.

Al finalizar este trabajo, se establece que se lograron los nuevos objetivos propuestos:

- Se definió una estrategia que pudiera ser seguida como base para formalizar los procesos de detección automática de *bad smells* basados en métricas de software.

- Se probó la utilidad de esta estrategia aplicándola a tres casos particulares; dos de los cuales no habían sido tratados en trabajos anteriores, y otro sí. Esto con el fin de demostrar que la estrategia sería útil para continuar con los *bad smells* faltantes, y para formalizar lo que ya se ha hecho con el fin de -posiblemente- mejorarlo.

- Se dio una caracterización formal de los *bad smells* abordados, y siguiendo la estrategia planteada, se propusieron planes de detección; los cuales incluyen la definición formal de las métricas que se van a usar, y los filtros y correlaciones entre éstas.

- Se desarrolló una aplicación de software que implementa las estrategias definidas, y que permite 'jugar' con los valores límites de las métricas para afinar sus resultados.

- Se desarrolló un API que permite el cálculo de todas las métricas usadas, y que por lo tanto puede ser usado para implementar y probar nuevas estrategias para los mismos *bad smells*, o implementar estrategias para otros en los que se necesite calcular alguna de éstas.

- Se diseño una encuesta que puede ser contestada por desarrolladores de software, con el fin de tratar de hacer un paralelo entre lo que una persona detecta como bad smell, y lo que la aplicación desarrollada detecta como tal; y así validar -o, posiblemente invalidar- las estrategias propuestas (Ver anexo: Encuesta entre desarrolladores).

6.3 Trabajo futuro

Como resultado del trabajo desarrollado surgieron algunas ideas de las formas en que esta línea de investigación podría ser continuada, y de las tareas que podrían llevarse a cabo para arreglar las falencias detectadas.

» Se incluyó en este trabajo la aplicación de la estrategia propuesta a casos previamente abordados por otros autores para demostrar que funciona correctamente; concretamente con el *bad smell feature envy*, del trabajo de Angarita **[12]**. Con el fin de formalizar los trabajos previos en esta línea de investigación, se podría hacer lo mismo para los *bad smells* que se tratan en dichos trabajos; lo cual serviría también para refinar y posiblemente mejorar los enfoques propuestos por sus autores.

» Además de hacerlo para los tratados ya, la aplicación de la estrategia propuesta aquí podría ser aplicada a los faltantes, y así completar una base formalizada de estrategias para la detección automática de *bad smells*.

» Una vez se cuente con esta formalización (la nombrada en el numeral anterior) se podrían probar diferentes combinaciones y composiciones de métricas para lograr detecciones más exactas.

» Aunque el análisis se restringió a código fuente JAVA 1.4, debido a que la representación está limitada a éste, las nuevas características de JAVA 1.5 no alteran los valores de las métricas obtenidas. Sin embargo, para no tener este tipo de restricciones se podría pensar en modificar la representación usada para que acepte código fuente JAVA 1.5, o usar una representación diferente que se acomode a el esquema de solución propuesto.

» En las pruebas realizadas se incluyeron ejemplos de código típico de patrones de software para saber qué pasaba con éstos. Debido a que los valores límites de las métricas pueden ser variados cada vez, los resultados obtenidos en estos casos podrían no ser muy exactos. Una forma de mejorar la herramienta sería haciendo que ésta detecte de alguna forma los patrones típicos de programación y éstos no sean detectados como falsos positivos.

» Las pruebas de detección de *middle man* fallaron en algunos casos: si se ajustaban los valores límites de las métricas para aplicaciones con pocas clases, éstos no servían para aplicaciones con un número mayor de clases, y viceversa. Esto se puede deber a que la fórmula de correlación de métricas se define en términos de porcentajes de clases (dentro del número total de clases) que, por ejemplo, hacen llamados a la clase analizada (**`nr_afferent-refs(X) / NOC(App)`**). Se podrían obtener resultados mas exactos utilizando otra fórmula de correlación que definiera no porcentajes de clases, sino números fijos de clases; por ejemplo, que la regla fuera **`nr_afferent-refs(X) > v,`** donde **`v`** es un número de clases de la aplicación. También se podría implementar éste como método alternativo, y hacer que el usuario pueda escoger el método de detección.

Referencias y Bibliografía

[1]
Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999

[2]
http://www.refactoring.com/. Página principal del sitio

[3]
William C, Wake. Refactoring Workbook. Addison-Wesley. 2003

[4]
Mika Mäntylä. Bad Smells in Software – a Taxonomy and an Empirical Study. May 2003.

[5]
http://en.wikipedia.org/wiki/Software_metrics

[6]
James F. Power y Brian A. Malloy. A metrics suite for grammar-based software. Journal of software maintenance and evolution: research and practice. 2004

[7]
Marcela Genero Bocco, Daniel L. Moody y Mario Piattini. Assessing the capability of internal metrics as early indicators of maintenance effort through experimentation. Journal of software maintenance and evolution: research and practice. 2005

[8]
Beatriz Florián. Detección de Bad Smells en Aplicaciones JAVA utilizando Métricas de Software. Tesis de grado de maestría. Universidad de los Andes. 2005

[9]
Watts S. Humphrey. Managing the Software Process. Addison-Wesley. 1989

[10]
Mohamed El-Wakil, Ali El-Bastawisi, Mokhtar Boshra y Ali Fahmy. Software Metrics - A Taxonomy

[11]
Radu Marinescu. Measurement and Quality in Object-Oriented Design. 2002

[12]
Carlos Angarita. Detección de bad smells en aplicaciones JAVA a partir de una representación xml del código fuente. Tesis de grado de maestría. Universidad de los Andes. 2005

[13]
Pablo Montes. XSCoRe, Representación de Código Fuente basada en XML como una Herramienta de Asistencia al Proceso de Ingeniería en Reversa. Universidad de los Andes. 2004

[14]
Radu Marinescu. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws

[15]
Lloyd G. Williams y Connie U. Smith. PASASM: A Method for the Performance Assessment of Software Architectures.

[16]
Yoshio Kataoka, Michael D. Ernst, William G. Griswold, y David Notkin. Automated Support for Program Refactoring using Invariants.

[17]
Tom Tourwé y Tom Mens. Identifying Refactoring Opportunities Using Logic Meta Programming.

[18]
Matthew James Munro. Product Metrics for Automatic Identification of “Bad Smell” Design Problems in Java Source-Code

[19]
Summarization of Bad Smells. John Neely, Arman Assa. En https://svn.linux.ncsu.edu/svn/slack/trunk/csc326/aassa/Smells.txt

[20]
http://www.orafaq.com/glossary/faqglose.htm

[21]
Shyam R. Chidamber y Chris F. Kemerer. A metrics suite for object oriented design. IEEE transactions on software engineering, vol 20, No 6, june 1994

[22]
Radu Marinescu. Detecting Design Flaws via Metrics in Object-Oriented Systems.

[23]
http://www.microgold.com/version2/stage/tutorial/glossary.html

[24]
Radu Marinescu, Daniel Rat¸iu. Quantifying the Quality of Object-Oriented Design:
the Factor-Strategy Model

[25]
Victor R. Basili y Dieter Rombach. The TAME project: towards improvement-oriented software environments. IEEE Transactions on software engineering, vol 4, No 6, junio 1988.

[26]
Fernando Brito e Abreu y Rogério Carapuça. Candidate Metrics for Object-Oriented Software within a Taxonomy Framework. Proceedings of AQUIS'93 Conference, Venecia, Italia, octubre de 1993

[27]
Linda Westfall. 12 Steps to Useful Software Metrics.

[28]
Object oriented metrics which predict maintainability. Wei Li and Sallie Henry. February, 1993.

[29]
Microscopic Relative Coupling Metrics. Padmaja Joshi.

[30]
Applying Design-Metrics to Object-Oriented Frameworks. Karin Erni y Claus Lewerentz.

Glosario

Acoplamiento

Es el grado en que dos artefactos de software dependen uno de otro. Generalmente se habla de acoplamiento a nivel de clases (A y B), el cual consiste en que los métodos de la clase A usan mucho los métodos y atributos de la clase B; y a la vez, los métodos de la clase B usan mucho los métodos y .atributos de la clase A.
El acoplamiento entre dos clases A y B se nota en este trabajo con **`A <- C -> B`** que se lee "A y B están acopladas".

Acoplamiento unidireccional

Acoplamiento unidireccional entre las clases A y B significa que los métodos de la clase A usan los métodos y atributos de la clase B, pero, no necesariamente, los métodos de B usan los métodos y atributos de A.
La notación dada para esta relación es **`A C -> B`** que se lee "A está acoplada -unidireccionalmente- con B". Por lo tanto, **`A <- C -> B => (A C -> B OR B C -> A)`** pero **`A C -> B ≠> B C -> A`**

Bad smell

Nombre dado por Fowler y Beck a una mala práctica de programación tipificada por ellos en su libro.

Binding dinámico

Es el binding (asociación) en la cuál la asociación nombre-clase no se hace hasta que el objeto designado por el nombre es creado en tiempo de ejecución **[23]**.

Code smell

Nombre con el que se conocen en general las malas prácticas de programación que se han definido. Este concepto es más general que el de bad smell.

Encapsulamiento

Es uno de los principios fundamentales de la programación OO. Describe la habilidad que tiene un objeto para ocultar sus datos y métodos del resto del mundo **[20]**. Por ejemplo **[4]**, si la clase A necesita algún dato de un objeto B, pero la clase A no tiene una referencia a B, A necesita obtener esa información de un tercer objeto C; es decir, A no conoce a B, pero aún así puede obtener los datos que necesita de B por medio de C.

Estrategia de detección

Proceso para hallar expresiones cuantificables basadas en métricas de software, que sirvan como criterio de decisión de la presencia de un bad smell

Feature envy (bad smell)

[12] Se dice que un método presenta *feature envy* cuando éste, en la mayoría de sus instrucciones, accede más a otra clase (a través de sus atributos y métodos) que a la clase propietaria del método que se esta analizando.

Inappropriate intimacy (bad smell)

Se da en el caso en el que dos clases están muy acopladas

Large class (bad smell)

Cuando una clase 'trata de hacer mucho'. Esto implica que tiene muchas variables de instancia, muchos métodos, y por lo tanto muchas líneas de código **[1]**.

Low cohesion (bad smell)

La baja cohesión se refiere a clases que apuntan a la prestación de excesivos servicios, o servicios de naturaleza disímiles **[12]**.

Mal olor -en el código-

Traducción de bad smell o code smell, indistintamente.

Message chains (bad smell)

Cuando un cliente le pide un objeto a otro objeto, al cual el cliente le pide otro objeto, y así sucesivamente **[1]** .

Métrica de software

Propiedad cuantitativa de los productos de software.

Middle man (bad smell)

Consiste en el hecho de que una clase delega la mayor parte de sus responsabilidades a otras clases.

Patrones de construcción de software

Forma definida y documentada de solución a un problema (arquitectura o diseño) particular

XScrore

Herramienta desarrollada por Pablo Montes, que produce una representación en XML de un código fuente JAVA 1.4.

Anexos

A.1 Resumen de notación

(↑) , (↓) : Propiedades “cómo crece” y “cómo decrece” de las métricas

ATFD(X) : (access to foreign data) número de clases externas de las cuales la clase analizada accede a atributos.

ATFM(X) : (access to foreign methods) número de clases externas de las cuales la clase analizada accede a métodos.

$\mathbf{Attributes}_X$: El conjunto de atributos de la clase X.

Attribute(X) : Denota la declaración de un atributo en la clase X.

Attribute(X,Y) : Denota la declaración de un atributo de tipo Y en la clase X.

$\mathbf{Classes}_{App}$: El conjunto de todas las clases del código fuente analizado

Cm-c(X) : Número de clases que tienen algún método que está más acoplado con la clase X de lo que es cohesivo con ellas mismas.

DAC(X) : (data abstraction coupling) número de atributos de tipo no simple declarados en la clase X.

DAC(X,Y) : (data abstraction coupling) número de atributos de tipo de clase Y declarados en la clase X.

DataAccess(X,m -> Y) : Un statement de acceso de datos desde el método m de la clase X hacia la clase Y.

DataAccess(X) : Denota un statement de la clase X de acceso de datos.

DataAccess(X,Y) : Denota un statement de acceso de datos de la clase X hacia la clase Y.

$\mathbf{DataAccesses}_X$: El conjunto de sentencias de la clase X que son accesos a atributos.

$\mathbf{DataAccesses}_{X(X,Y)}$: El conjunto de accesos a atributos de la clase Y que hace la clase X.

$\mathbf{Methods}_X$: El conjunto de métodos de la clase X.

Message(X,m -> Y) : Un statement de envío de mensajes desde el método m de la clase X hacia la clase Y.

Message(X) : Denota un statement de invocación de método de (hecho por) la clase X

Message(X,Y) : Denota un statement de invocación de método de la clase X hacia la clase Y.
$\mathbf{Messages}_X$: El conjunto de sentencias en la clase X que envían mensajes.

Messages$_{X(X,Y)}$: El conjunto de mensajes de la clase X que están dirigidos a la clase Y.

Messages_out$_{X}$: El conjunto de mensajes enviados por la clase **X** que son dirigidos hacia afuera; es decir, hacia otras clases.

MPC(X) : (message-passing coupling) número de mensajes enviados desde la clase X.

MPC(X,Y) : (message-passing coupling) número de mensajes enviados desde la clase X a la clase Y.

NAM(X,Y) : (number of accessed methods) número de métodos de la clase Y accedidos por la clase X .

NAA(X,Y) (number of accessed attributes) : Número de métodos de la clase Y accedidos por la clase X.

NOWCM(X) (number of outwards-coupled methods) : Número de métodos de la clase **X** que están más acoplados con otras clases (de lo que son cohesivos con X).

NOWCM(X,Y) : Número de métodos de la clase **X** que están más acoplados con la clase **Y** (de lo que son cohesivos con X).

RMC(Ci,m -> Cj) : (relative method coupling) denota el cálculo de esta métrica para el método m de la clase Ci con respecto a la clase Cj.

Statements$_{X}$: El conjunto de sentencias de la clase.

Statements$_{X(m)}$: El conjunto de sentencias del método m de la clase X

TATFD(X,Y) : (total accesses to foreign data) número total de accesos a atributos de la clase Y que hace la clase X

Unique (Set, Criterion) : El conjunto que resulta de seleccionar de entre los elementos del conjunto inicial *Set*, un elemento único (cualquiera) de cada uno de los subconjuntos de éste formado por los elementos que cumplen con el criterio *Criterion*.

|**C**| : Cardinalidad del conjunto C

A <- C -> B : Acoplamiento entre A y B

A C -> B : Acoplamiento unidireccional entre A y B ("A está acoplada con B")

A.2 Propiedades de los conjuntos sobre los que se trabaja

Para cualquier aplicación analizada se cumplen las siguientes propiedades:

- $|\mathbf{Classes}_{App}| >= 0$
- Para toda clase $\mathbf{X} \in \mathbf{Classes}_{App}$ se cumplen las siguientes propiedades:
  - $0 <= |\mathbf{Attributes}_{X}| < \infty$
  - $0 <= |\mathbf{Methods}_{X}| < \infty$
  - $\mathbf{Messages}_{X} \subset \mathbf{Statements}_{X}$
  - $\mathbf{DataAccesses}_{X} \subset \mathbf{Statements}_{X}$
  - $\mathbf{Statements}_{X} = \varnothing \Rightarrow \mathbf{M(X)} = 0$, donde $\mathbf{M}$ es cualquiera de las métricas utilizadas en este trabajo.
  - Para todo método $\mathbf{m} \in \mathbf{Methods}_{X}$ se cumplen las siguientes propiedades:
    - $\mathbf{Statements}_{X(m)} \subset \mathbf{Statements}_{X}$
    - $0 <= |\mathbf{Statements}_{X(m)}| < \infty$

A.3 Sintaxis

Se presenta aquí una sintaxis no completa y sencilla de las estructuras de código relevantes para el cálculo de las métricas, con el fin de hacer claridad sobre lo que se está midiendo y sobre algunas de la propiedades de las métricas utilizadas.

```
Statement :: Message
Statement :: DataAccess

Message :: MethodCall
Message :: ClassName “.” MethodCall
Message :: VarName “.” MethodCall

MethodCall :: MethodName “(“ Parameters “)”
MethodCall :: MethodName “(“ Parameters “)” “.” MethodCalls
MethodCall :: MethodName “(“ Parameters “)” “.” DataAccesses
MethodCalls :: MethodCall
MethodCalls :: MethodCall “.” MethodCalls

DataAccess :: FieldAccesses
DataAccess :: ClassName “.” FieldAccesses
DataAccess :: VarName “.” FieldAccesses
FieldAccesses :: FieldAccess
FieldAccesses :: FieldAccess “.” MethodCall
FieldAccesses :: FieldAccess “.” FieldAccesses
FieldAccess :: AttributeName

VarName :: AttributeName
VarName :: LocalVarName
AttributeName :: String (representa el nombre de un atributo de clase)
ClassName :: String (representa el nombre de una clase)
Parameters :: String
LocalVarName :: String (representa el nombre de una variable local de
método)

String :: char
String :: char + String
char :: {a, b, … , z, A, B, … , Z}
```

```
char :: cualquier símbolo permitido por JAVA para nombrar atributos,
        clases y parámetros
```

A.4 Encuesta entre desarrolladores

Como complemento al trabajo de tesis desarrollado, se diseñó una encuesta en la que, por medio de la presentación de varios ejemplos con características seleccionadas, se pretende que quienes la contesten validen -o contradigan- los planteamientos de esta tesis. Este mecanismo -de encuesta- ha sido usado en casos similares y con similares objetivos en trabajos como los de Mäntylä **[4]** y Angarita **[12]**.

Motivación

Como se menciona en este documento, uno de los objetivos generales que se busca con este trabajo es contribuir en el proceso de lograr definiciones precisas de lo que significa la existencia de un determinado *bad smell* en algún código fuente, lo cual se podría lograr eventualmente si existe un consenso acerca de las métricas que se deben usar en la detección, y en los niveles o límites dentro de los cuales deben estar los valores de éstas para poder decir que un código no 'huele mal'. Esta encuesta se presenta como un mecanismo para entrar en contacto con desarrolladores de software para determinar si las caracterizaciones de los bad smells y las métricas propuestas para su detección presentadas en esta tesis coinciden con su intuición y análisis al respecto.

Objetivos

- Tratar de identificar las características que hacen que la presencia del bad smell sea evidente para quien analiza un código fuente.
- Determinar si la definición dada del bad smell corresponde con lo que la gente identifica como tal en el código fuente.
- Determinar si lo que la gente identifica como bad smell es también identificado por la aplicación desarrollada para implementar la estrategia propuesta.
- Conseguir que los parámetros de porcentajes permitidos de las métricas utilizados en la aplicación puedan ser afinados, y tal vez, llegar al punto en que estos sean valores más o menos constantes que se puedan usar en la mayoría de los casos.

Características

- Se crearon ejemplos sencillos para evitar que las personas que contesten se cansen con códigos largos y complicados que harían que la atención se desviara hacia asuntos no fundamentales para el objetivo. El código, en la mayoría, si no en todos los casos, no tiene

alguna funcionalidad o propósito más allá de ejemplificar las situaciones específicas; son, como se suelen denominar, 'ejemplos de juguete'.

- Los ejemplos de código se analizaron con la aplicación desarrollada para verificar que se conociera de antemano el nivel de las métricas.

- La detección de los bad smells tratados en este trabajo se basa en métricas que son calculadas en su mayoría a partir de statements de invocaciones a métodos y de accesos a atributos. La implementación de esta encuesta incluiría la presentación de la sintaxis que se creó en este documento para explicar qué son invocaciones a métodos y accesos a atributos.

- La escalas de evaluación corresponden a los tipos de preguntas incluidas en la encuesta (ver la siguiente sección: Composición) son las siguientes:

    - Para las preguntas de tipo 1
      "Con respecto a la métrica Mx, ¿considera que en el siguiente ejemplo el bad smell BSx está presente?"
      Sí
      No
      No estoy seguro

    - Para las de tipo 2
      "Con respecto a la métrica Mx, ¿en qué porcentaje considera que el bad smell BSx está presente?
      0%
      Alrededor de 50%
      100%

    - Para las de tipo 3
      "¿Considera que en el siguiente ejemplo está presente el bad smell BSx?"
      Sí
      No
      No estoy seguro

    - Para las de tipo 4
      "¿Cuál de las siguientes métricas considera que incide más para que se presente el bad smell BSx en el siguiente ejemplo?"
      Las opciones de respuesta son las métricas correspondientes a cada *bad smell*

    - Para las de tipo 5
      "¿Cuál de los siguientes ejemplos presenta más el bad smell BSx?"
      Las opciones de respuesta son los números que identifican cada uno de los ejemplos dados como parte de la pregunta.

Composición de la encuesta

La encuesta presenta casos que se dividen en grupos según las características especiales con que fueron diseñados. A continuación se enumeran estos grupos, y se especifica qué es lo que se busca con cada uno:

1. Ejemplos con el bad smell presente de forma evidente, y sin éste en absoluto, con respecto a cada una de las métricas de la fórmula final (Ver la sección de Correlación de métricas para cada bad smell).
   Aquí se busca verificar si los encuestados pueden identificar -o no- el bad smell con respecto a cada métrica usada, solamente. Esto servirá para aclarar qué es lo que se está midiendo con cada métrica.

2. Para cada una de las métricas de la fórmula final, dar ejemplos con varios porcentajes conocidos de la métrica.
   Con esto se tratará de determinar qué es lo que intuitivamente parece como "mucho" o "poco"; así como de encontrar un rango de valores más o menos comunes para los parámetros que identifican los pesos dados a cada métrica.

3. Ejemplos en que los valores de las métricas sean altas, pero, visualmente, los statements que causen estos valores estén distribuidos de diferente forma.
   Se usarán éstos ejemplos para determinar qué tanto depende de la parte visual la detección -no automática, obviamente- del bad smell.

4. Ejemplos que presenten los mismos porcentajes de valores de las métricas para ver cuál es el que en apariencia presenta de forma más evidente el bad smell.
   Con estos casos se quiere determinar cuál de las métricas usadas es la que debería tener más importancia en la detección del bad smell.

5. Ejemplos en que los valores de todas las métricas involucradas tengan diferentes niveles.
   Haciendo una mezcla de los casos anteriores se tratará de establecer si los resultados individuales se mantienen en casos que involucran más criterios de decisión.

Se muestran los ejemplos presentados en cada tipo de pregunta. Para ello se utiliza la siguiente tabla:

Bad smell: *nombre del bad smell*	**Caso**: *se reemplaza la pregunta que acompañaría a cada ejemplo por una descripción mínima de éste*
Clases: *código fuente de las clases que componen el ejemplo*	

Tabla 7. Descripción de los casos de la encuesta

Nota:
Se omite el contenido de la encuesta por cuestiones prácticas. Si el lector desea conocerlo, favor escribir a pa.garcia29@egresados.uniandes.edu.co