

MANTENIMIENTO DE ASERCIONES EN LA REFACTORIZACIÓN DEL CÓDIGO EN EL DISEÑO POR CONTRATO

J. Coves¹, D. Rodríguez², M. Satpathy³

1: UPC (Universitat Politècnica de Catalunya)
C/Jordi Girona Salgado 1-3, 08034, Barcelona, España
e-mail: josepcoves@gmail.com

2: Universidad de Reading
Reading, RG6 6AY, Reino Unido
e-mail: d.rodriguezgarcia@rdg.ac.uk

3: Åbo Akademi University
Lemminkisenkatu 14 A FIN-20520, Turku, Finland
e-mail: msatpath@abo.fi

Palabras clave: Ingeniería de Software, Aserciones, Diseño-por-contrato, Herramientas de programación

Resumen. *Las aserciones son restricciones formales sobre el estado de las variables del código de un programa, las cuales están insertadas como anotaciones en el código fuente. Las aserciones se usan ampliamente en la industria del software para detectar, diagnosticar y clasificar los errores de programación durante la etapa de test. Cuando un trozo de código ha sido anotado con aserciones y ha estado sujeto a refactorización, las aserciones originales pueden no ser consistentes con el código original. El principal objetivo de este artículo es especificar cómo mantener la consistencia de las aserciones a través del proceso de refactorización teniendo en cuenta el esquema de diseño-por-contrato. Este artículo también analiza y clasifica las aserciones desde el punto de vista de la refactorización del código: hay aserciones que se mantienen automáticamente, otras que pueden ser adaptadas después de aplicar unas reglas de transformación, y finalmente, hay una última categoría que requiere la intervención del usuario dado que no es posible automatizar el proceso. Además, detallamos y demostramos los pasos requeridos para mantener las aserciones a través del proceso de refactorización.*

1 INTRODUCCIÓN

La etapa de mantenimiento es la parte más costosa en el ciclo de vida del desarrollo de software. Como resultado, los lenguajes de programación y los IDEs (Integrated De-

velopment Environments) han evolucionado para facilitar este proceso. Las herramientas de esta evolución incluyen aserciones, diseño-por-contrato y refactorización.

Las aserciones [2] son restricciones formales sobre el estado de las variables del código de un programa, las cuales están insertadas como anotaciones en el código fuente. Las aserciones son expresiones booleanas evaluadas a *true* cuando el estado del programa satisface las restricciones deseadas. Si una aserción es evaluada a *false* significa que el programa ha entrado en un estado inconsistente y, por lo tanto, no podemos confiar en el comportamiento del programa aún y cuando la salida sea correcta. Las aserciones se usan ampliamente en la industria del software, primeramente para detectar, diagnosticar y clasificar los errores de programación durante la etapa de test.

La refactorización, generalmente descrita como patrones, ayuda a los desarrolladores a reconstruir una estructura de un código fuente para mejorarlo, para hacer su mantenimiento más fácil y veloz. En esta labor, nosotros definimos un conjunto de pasos y reglas para adaptar las aserciones durante el proceso de refactorización. Dado que no hay una completa definición de todas las posibles refactorizaciones existentes (y esto es imposible, ya que nuevos tipos de refactorización se están desarrollando continuamente para solucionar nuevos problemas) el objetivo de este proyecto no es tanto definir un conjunto de reglas para cada una de las refactorizaciones existentes, sino detallar algunos de los más importantes y hacer una clasificación general para dar un patrón de cómo actuar para adaptar las aserciones en las nuevas refactorizaciones que aparezcan en el tiempo.

Meyer [4] creó la noción de Diseño-por-Contrato (*Design by Contract*) en el contexto del la construcción de software orientado a objetos (OO). Cada método tiene una precondition y una poscondition que esperan ser cumplidas respectivamente en la entrada y salida de la función. Las aserciones puede ser usadas para chequear dichas condiciones.

En este artículo, discutimos la refactorización de aserciones en el contexto de software OO; especialmente, cuando seguimos un Diseño-por-Contrato. Extendemos los pasos de refactorización teniendo en cuenta las expresiones booleanas de las aserciones.

La organización del resto del artículo se divide en las siguientes secciones. La sección 2 es una recapitulación de trabajos relacionados. La sección 3 se centra en cómo las aserciones de un sistema pueden ser mantenidas para afrontar los cambios de diseño. Después se dan ejemplos de refactorizaciones en la sección 4. Finalmente, la sección 5 concluye el artículo y presenta direcciones futuras.

2 TRABAJOS RELACIONADOS

2.1 Aserciones

Las Aserciones fueron creadas por Hoare como un sistema de axiomas para demostrar la correctitud de programas Algol [2]. Rosenblum [6] las define como: *“Las aserciones son restricciones formales en el comportamiento de un sistema software que comúnmente están escritas como anotaciones en el código fuente. El objetivo primordial de escribir aserciones es especificar qué se supone que un sistema debe hacer en lugar de cómo debe*

hacerlo".

Dado que podemos expresar suposiciones del estado del programa durante la ejecución, podemos usar las aserciones como un modo de documentación (en lugar de escribir los típicos comentarios) con la ventaja de que seremos notificados en el caso que nuestras suposiciones sobre el código no fueran ciertas. Siguiendo esta idea, podemos definir las especificaciones de un software OO, incluso antes de tener el código implementado, expresando sus precondiciones y poscondiciones de métodos. Además, como las aserciones son un modo de verificar nuestras suposiciones en cualquier punto del código, también podemos detectar errores de implementación.

Podemos ahorrarnos una gran parte del tiempo de *debug* que gastaríamos buscando un error, ya que un fallo en la aserción nos devuelve información detallada sobre dicho error. A pesar de que Rosenblum observó que no existía una clara correlación entre la localización de un error y el lugar donde estaba situada la aserción que lo detectaba [6], Satpathy et al [8] (durante la realización de un proyecto software) demostraron que poner las aserciones en puntos clave era muy frecuentemente de gran ayuda para localizar un error dado. A consecuencia de ello, programar con aserciones puede llevarnos a tener nuestros programas ejecutándose más rápidamente y con un mayor grado de confianza sobre la correctitud del programa.

Rosenblum [6] clasificó las aserciones en el contexto de mantenimiento y testeo de software en dos categorías principales: (a) La especificación de interfaces de función; (b) La especificación de cuerpos de función. La primera categoría de aserciones, describe el comportamiento de una función, independientemente de su implementación (*visión de caja-negra*), mientras la segunda procura asegurar su correcta implementación (*visión de caja-blanca*).

2.2 Diseño-por-Contrato

El principio del diseño-por-contrato [4] representa las relaciones entre una clase y sus clientes como un acuerdo formal. Es decir, podemos expresar el contrato como: "*Si tú, el cliente, me aseguras ciertas precondiciones, entonces yo, el proveedor, obtendré unos ciertos resultados para ti. De lo contrario, si tú violas las precondiciones, no te prometo nada.*" Siguiendo este concepto, las aserciones son una herramienta perfecta para expresar todos los derechos y obligaciones de cada una de las partes, porque, por ejemplo, con ellas podemos expresar las obligaciones del cliente con precondiciones, y las obligaciones del proveedor con poscondiciones. Además, inconscientemente, estaremos determinando cuando una precondición es correcta.

Meyer definió el diseño-por-contrato con la pretensión de ser la mejor metodología para construir software fiable. Meyer define fiabilidad como correctitud y robustez. Meyer afirma que antes del diseño-por-contrato la única forma para asegurar esta propiedad era *confiando que sus autores habían aplicado todo el cuidado necesario, incluyendo una etapa de técnicas de validación y test extensivas* [4]. Si usamos diseño-por-contrato tenemos una definición formal de correctitud de una clase, y una método para asegurarse que una clase

es correcta: *Una clase como cualquier otro elemento de software, es correcta o incorrecta no por sí misma, sino con respecto a una especificación. Introduciendo precondiciones, poscondiciones e invariantes incluimos una parte de la especificación dentro de la clase misma. Esto nos da una base que tasa la correctitud: la clase es correcta si y solamente si su implementación, dada por los cuerpos de rutina, es consistente con las precondiciones, poscondiciones e invariantes [...]. Si una clase está equipada con aserciones, es posible definir formalmente qué significa para la clase ser correcta [4].*

2.3 Refactorización

Fowler [1] define: *La refactorización es el proceso de cambio en un sistema software de tal modo que no altera el comportamiento externo del código, mientras que su estructura interna. Es un método disciplinado para limpiar el código y minimizar las posibilidades de introducir nuevos bugs. En esencia, cuando aplicamos una refactorización, estamos mejorando el diseño del código después de que éste ya haya sido escrito.* La razones que Fowler [1] cita para usar técnicas de refactorización incluyen: (i) mejora el diseo del software; (ii) hace que el software sea más fácil de entender; (iii) nos ayuda a encontrar errores, y (iv) nos ayuda a programar más rápido. Fowler usa la terminología *Bad Smells* (*maloliente*) en un código para decidir cuándo una refactorización debería ser aplicado.

La principal propiedad de la refactorización es la preservación en el comportamiento del código, por lo que primeramente, deberíamos tener una clara idea de lo que significa. En su tesis, Opdyke [5] definió y demostró la preservación del comportamiento de un código a través del proceso de refactorización. Siguiendo la definición de Opdyke [5], el comportamiento es preservado cuando después de la refactorización: (a) Hay una única superclase (b) Las clases tienen nombres distintos (c) Las funciones y miembros dentro de una misma clase deben tener nombres diferentes (d) En la redefinición de funciones, las firmas deben ser compatibles (para preservar el principio de substitución de Liskov (LSP) [3]) (e) Las asignaciones deben conservar el tipo (f) Las referencias y operaciones deben ser *semánticamente equivalentes*¹

A pesar de que la refactorización sea una herramienta muy poderosa, no podemos aplicarla bajo cualquier circunstancia. Opdyke [5] definió una serie de precondiciones para cada tipo de refactorización, las cuales deben ser cumplidas para preservar el comportamiento del programa a través del proceso de refactorización. Opdyke las dividió en dos grandes categorías: (i) *low-level refactorings* (*refactorizaciones de bajo-nivel*), la preservación del comportamiento de los cuales es fácil de demostrar si sus precondiciones son respetadas y (ii) *composite refactorings* (*refactorizaciones compuestas*) los cuales se construyen con las de bajo-nivel. Dado que las compuestas están construidos usando refactorizaciones que preservan el comportamiento, el comportamiento estará preservado

¹Opdyke definió la equivalencia semántica de este modo: supongamos que la interfaz externa del programa sea vía la función main. Si la función main es llamada dos veces (una antes y la otra después de la refactorización) con el mismo conjunto de entradas, hemos de obtener el mismo conjunto de valores de salida.

también con ellos si las precondiciones no se violan.

3 Mantenimiento de Aserciones en el Proceso de Refactorización

En esta sección describiremos algunas refactorizaciones al usar aserciones y diseño-por-contrato asumiendo: (i) Java como lenguaje de programación (ii) El método `Inv()` es el método de *Invariante de Clase* (CI) (en cada clase en la que aparezca), siguiendo la nomenclatura de Java (iii) las aserciones escritas justo al inicio del método se considerarán la *precondición* de dicho método; y (iv) las aserciones escritas justo antes del final de un método se consideran la *poscondición* de dicho método. Una sencilla clasificación dependiendo en cómo afectan las aserciones podría ser:

Refactorizaciones con mantenimiento trivial de las aserciones Dado que en muchos lenguajes las aserciones son expresiones como cualquier otra, deben ser tratadas como tales durante el proceso de refactorización. Siguiendo este criterio, muchas refactorizaciones descritas por Fowler [1] mantienen la consistencia de sus aserciones sin hacer nada. Algunas refactorizaciones en esta categoría (usando la nomenclatura que les asignó Fowler) incluyen: *Inline Temp*, *Replace Temp with Query*, *Introduce Explaining Variable* y *Remove Assignments to Parameters*. Por ejemplo, el *Rename Method* (método de renombrar) reemplaza el nombre antiguo de una variable por otro nuevo nombre en todos los sitios que dicha variable aparezca, tanto si está dentro de una aserción como fuera. Después de ejecutarlo, y si tratamos las aserciones como cualquier otra expresión, las aserciones serán consistentes.

Refactorizaciones automatizables Tenemos refactorizaciones como *Encapsulate Field* (*Encapsular un campo*) en los que las aserciones no son inconsistentes después de la refactorización pero no podemos garantizar que sean correctas. Deberíamos garantizar que los nuevos cambios sean consistentes con la idea del diseño-por-contrato para poder demostrar su correctitud. Por ejemplo, cuando añadimos nuevas funciones públicas es fácil añadir la comprobación del CI antes del final del método. O bien, cuando creamos una nueva sub-clase debemos heredar y comprobar el CI de la clase base en cada método público y en el constructor de la nueva clase.

El *Encapsulate Field Refactoring* consiste en crear los métodos de *get* y *set* para un campo dado. Todas las expresiones que antes llamaban al campo directamente, ahora están obligadas a hacerlo a través de esos nuevos métodos. Dado que todas las aserciones son tratadas como expresiones, después de la refactorización son consistentes todavía. Pero, a pesar de que no sean inconsistentes, deberíamos preguntarnos si estamos manteniendo el diseño-por-contrato, o bien, si no lo usábamos, como mínimo deberíamos tener algún procedimiento para asegurarnos que los nuevos métodos son correctos. Para demostrarlo, la forma propuesta es garantizar que éstos preservan las reglas del diseño-por-contrato. Un estudio más profundo sobre esta refactorización es llevado a cabo más adelante.

Refactorizaciones Manuales Algunas refactorizaciones no nos permiten mantener las aserciones automáticamente. A veces porqué es necesario saber el significado de lo

que el programador está intentando hacer con una aserción dada para poderla modificar adecuadamente, y a veces, porqué la funcionalidad de alguna aserción ha cambiado y no podemos derivarla automáticamente y solamente podemos informar al programador. Una posible solución sería enviar un tipo de aviso (como los avisos de compilación) para que el programador decida qué hacer con aquella aserción. Algunos ejemplos podrían ser *Remove Parameter* (*eliminar un parámetro*) que consiste en la supresión de un parámetro que no ha sido usado en un método. Por otro lado, *Add Parameter* (*aadir un parámetro*) realiza la acción opuesta. En ambos casos, necesitamos tener información semántica para modificar correctamente la precondition y poscondición del método dado. Por lo tanto, o bien eliminamos la aserción o preguntamos al programador cómo deberíamos tratar la aserción.

4 EJEMPLOS DE REFACTORIZACIONES AUTOMATIZABLES

Para lograr mantener las aserciones a través del proceso de refactorización necesitamos añadir unos cuantos pasos extra a los marcados por Fowler en su libro, a los que hemos llamado (*ARS*) *Augmented Refactoring Steps* (Pasos de Refactorización Aumentados).

4.1 Refactorización: *Pull Up Method*

El objetivo de *Pull Up Method* consiste en mover un método de una o más subclases hacia su superclase (en el segundo caso la función debe ser la misma en cada subclase). Siguiendo los pasos de Fowler [1] para mantener las aserciones, añadimos los pasos en negrita:

1. Examinar los métodos para asegurarnos que son idénticos.
 - Si los métodos hacen lo mismo pero no son idénticos, deberemos usar la refactorización de sustitución de algoritmos para hacerlos idénticos.
2. Si los métodos tienen signatures diferentes, cambiarlas a la que queramos usar en la superclase.
3. Crear un nuevo método en la superclase copiando el cuerpo de uno de los métodos a él, adaptarlo y compilar.
 - (a) Si estamos usando un lenguaje de programación fuertemente tipificado y el método llama a otro que está presente en las dos subclases pero no en la superclase, debemos declarar un método abstracto en la superclase.
 - i Si estamos en un lenguaje que permite la definición de **precondiciones y poscondiciones de métodos abstractos (como por ejemplo, Eiffel)**, deberíamos definir la **precondición de dicho método como la AND de la precondition de los métodos originales, y la**

poscondición como la OR de la poscondición de los métodos originales.

- (b) Si el método usa un campo de la subclase, usaremos *Pull Up Field* o *Self Encapsulate Field* y declararemos y usaremos unos métodos de *get* y *set* abstractos para poder usar dicho campo.
- (c) **Asegurarnos que el CI (Invariante de clase) llamado es el de la superclase, de otro modo se tendrán que hacer los cambios respectivos.**

4. Eliminar el método de una subclase.
5. Compilar y verificar.
6. Continuar eliminando el método del resto de subclases y verificando que el código funciona hasta que solamente quede el método en la superclase.
7. Echarle una ojeada a los que llaman al método para ver si es posible cambiar el tipo antiguo requerido al de la superclase.

4.2 Refactorización: *Self Encapsulate Field*

Esta refactorización consiste en encapsular un campo, normalmente creando sus métodos *get* y *set*. Todas las expresiones que antes llamaban o usaban el campo directamente están obligadas a ser modificadas y acceder al campo a través de estos nuevos métodos. Siguiendo los pasos de Fowler [1], añadimos los que están en negrita:

1. Creamos los nuevos métodos de *get* y *set* para acceder al campo
 - (a) **Añadir una precondición al método de *set* que verifique si el valor que vamos a asignarle al campo está dentro del rango de valores permitidos para el campo según esté indicado en el CI. Esta precondición puede ser extraída del CI.**
 - (b) **Añadir una poscondición al método de *set* que verifique si el valor del campo ha sido cambiado por el que se le ha pasado como parámetro.**
 - (c) **Añadir una precondición y una poscondición al método de *get* para verificar que el valor que vamos a retornar es el del campo correspondiente.**
 - (d) **Añadir la comprobación del CI antes del final de cada método**
2. Buscar todas las clases (que no sean la propia) que referencien al campo. Si una clase usa el valor del campo, reemplazar la referencia actual con una llamada al método de *get*. Si una clase modifica el valor del campo, reemplazar la referencia con una llamada al método de *set*.

3. Si el campo es un objeto y el que lo llama invoca un modificador del objeto eso se considera una consulta. Solamente se debe usar el método de *set* para reemplazar una asignación.
4. Compilar y testear después de cada cambio.
5. Una vez se han modificado todas las llamadas al campo por sus correspondientes métodos *get* y *set*, declarar el campo como privado.
6. Compilar y testear.

4.2.1 Pasos del procedimiento de extracción

Un CI (Invariante de Clase) es un predicado lógico que puede ser expresado con ANDs, ORs y NOTs. Una vez hayamos reducido el predicado, podemos crear un árbol binario tal que cada nodo es una operación lógica AND o OR. Vamos a detallar algunos pasos y reglas para poder extraer la precondition de un CI.

1. **Crear un árbol binario.** Construir un árbol binario a partir del CI, tal que cada nodo sea una operación de AND o OR lógica.
2. **Reemplazar el nombre de la variable.** Sustituir el nombre de la variable que representa el campo a encapsular por el nombre del parámetro del método de *set* en cada hoja del árbol en donde aparezca el campo. Marcar todas las hojas dónde se ha realizado esta sustitución.
3. **Aplicar las reglas de sustitución ASR y OSR.** Sustituir por TRUE todo lo que cumpla cualquiera de las siguientes reglas:
 - **AND Substitution Rule (ASR).** En un nodo AND podemos sustituir **una rama** por TRUE si se cumplen las siguientes propiedades:
 - 1 Ninguna de las sub-ramas contiene una hoja marcada
 - 2 Ninguno de sus predecesores es un nodo OR
 - **OR Substitution Rule (OSR).** En un nodo OR podemos sustituir **el nodo entero** por TRUE si se cumplen:
 - 1 Ninguna de sus ramas contiene una hoja marcada
 - 2 Ninguno de sus predecesores es un nodo OR
4. **Simplificación lógica del árbol resultante.** Simplificar el árbol usando operaciones de lógica de predicados.
5. **Reconstrucción de la precondition desde el árbol.** Reconstruir el predicado desde el árbol que hayamos obtenido. Este predicado será la precondition del método de *set*.

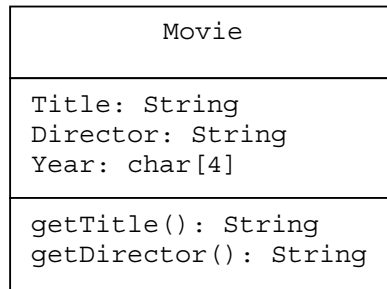


Figura 1: Clase Movie

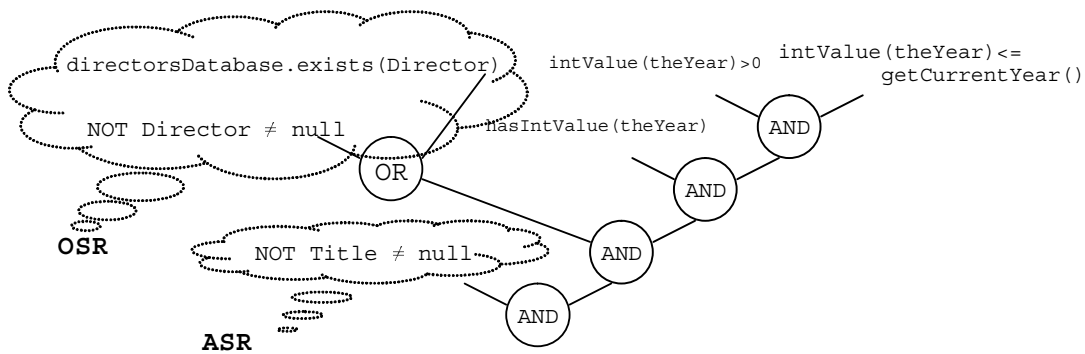


Figura 2: Árbol Binario — clase Movie

4.2.2 Ejemplo: *Encapsulate Field*

Supongamos que tenemos una clase `Movie` diseñada como muestra la Figura 1 con el siguiente CI:

```
CI=(Title != null) AND ((NOT Director != null) OR
directorsDB.exists(Director)) AND (hasIntValue(Year) AND
intValue(Year)>0 AND intValue(Year) <= getCurrentYear() )
```

Queremos aplicar el refactoring de *Encapsulate Field* al atributo `year`. Por ejemplo, supongamos que el método de *set* para dicho campo sea: `void setYear(char[4] theYear)`. La transformación del CI a árbol binario es directa. La figura 2 muestra el árbol y todos los subárboles que pueden ser simplificados aplicando las reglas ASR y OSR. Al final obtenemos la siguiente precondition:

```
hasIntValue(theYear) AND intValue(theYear)>0 AND
intValue(theYear)<=getCurrentYear()
```

Este ejemplo muestra cómo la derivación de la precondition nos ayuda a mantener el diseño-por-contrato. Si hubiésemos creado el nuevo método de *set* sin esta precondition, el cliente podría haber introducido cualquier string de 4 caracteres, violando fácilmente el CI.

Añadir postcondición al método de *set* Puede ser automatizado con una plantilla que será diferente para cada lenguaje de programación. Por ejemplo, en Java podemos usar la siguiente: `Post = {this.field == theField}`. Por lo tanto, para automatizar este paso, solamente necesitamos reemplazar *field* por el nombre del campo que vamos a modificar y *theField* por el nombre de la variable parámetro del método de *set*.

Añadir una precondición y una postcondición al método de *get* Dado que no hay parámetros de entrada en un método de *get*, cualquier llamada será aceptada. Por lo tanto, podemos escribir `TRUE` como la precondición del método *get*. Como en el paso anterior, esto puede ser fácilmente automatizado relleno una plantilla que será diferente para cada lenguaje de programación. Por ejemplo, en Java podemos usar: `Post={result == this.field}` Para automatizar este paso, solamente tenemos que reemplazar *field* por el nombre del campo que vamos a consultar y *result* por el nombre de la variable que contenga el valor que vamos a devolver con el método *get*.

Validar el CI en ambos métodos Dado que los nuevos métodos de *get* y *set* son ambos públicos, Siguiendo la regla *Class Correctness Rule*[4] de Meyer, el CI debe ser comprobado antes de la finalización de ambos métodos. Si hemos definido reglas para escribir los CIs es posible automatizar este paso.

5 Conclusions and Future Work

Las aserciones deberían ser mantenidas durante el proceso de refactorización. Después de analizar una serie refactorizaciones, concluimos que necesitamos una herramienta para ser capaces de juzgar cuando las aserciones son útiles y correctas. La solución fue encontrada en el diseño-por-contrato, que nos ayuda a juzgar si las aserciones son útiles y correctas con respecto al diseño del programa. En este artículo clasificamos las refactorizaciones basándonos en su mantenimiento con respecto a las aserciones. Descubrimos que hay algunos refactoring que mantienen automáticamente las aserciones, algunos que requieren cambios que se pueden realizar automáticamente, y finalmente, algunos que no pueden ser automatizados y que requieren de la intervención del programador (por ejemplo, preguntándole información o empleando un nuevo tipo de avisos como warnings de compilación. A pesar de que no se pueden adaptar todas las refactorizaciones automáticamente, hay bastantes que sí que lo permiten. Aquí hemos descrito los *Augmented Refactoring Steps*(ARS) para mantener las aserciones en los refactorings de *Pull Up Method* y *Self Encapsulate Field*. Para mantener *Self Encapsulate Field* describimos y demostramos un procedimiento para la extracción de la precondición de un método de *set* desde el CI. Este procedimiento puede ser usado o extendido en otros refactorings que requieran la extracción de información desde el invariante de la clase (CI).

Como trabajo futuro, sería necesario una implementación de esos ARS en una herramienta de refactoring existente. Planeamos la realización de este trabajo modificando

algunos de los entornos IDE Open Source, como, por ejemplo, Net Beans o Eclipse, para extender los refactorings que ya están implementados en dichas plataformas. Un trabajo futuro, incluirá un estudio más profundo sobre la simplificación de predicados lógicos para obtener mejores predicados. A pesar de que demostramos la extracción de la precondition desde el CI solamente dimos un primer esbozo. Este procedimiento puede ser mejorado para obtener una mejor simplificación.

Agradecimientos

Este trabajo ha sido posible gracias a la ayuda CICYT TIN2004-06689-C03.

REFERENCIAS

- [1] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [2] C.A.R. Hoare, An Axiomatic basis for computer programming, *Communications of the ACM*, Vol. 12 (10), October 1969, pp. 576–580,583.
- [3] B.H. Liskov and J.M. Wing, A Behavioural Notion of Subtyping, *ACM TOPLAS*, 16(6): 1811–1841, 1994.
- [4] B. Meyer, *Object Oriented Software Construction*, Prentice Hall, 1997.
- [5] Opdyke, W.F., *Refactoring Object Oriented Frameworks*, Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.
- [6] Rosenblum, D.S., A Practical Approach to Programming with Assertions, *IEEE Transactions on Software Engineering*, Vol 21(1), 1995, pp. 19-31.
- [7] Roberts, D.B. *Practical Analysis for Refactoring*, Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.
- [8] Satpathy, M., N.T. Siebel, and D. Rodríguez, Assertions in Object Oriented Software Maintenance: Analysis and a Case Study. 20th IEEE ICSM'04pp. 124-135, 2004