

Refactorizaciones en la Migración del Software

Raúl Marticorena¹, Yania Crespo², y Carlos López¹

¹ Área de Lenguajes y Sistemas Informáticos, Universidad de Burgos
EPS Edif. C. C/Francisco de Vitoria s/n
{`rmartico, clopezno`}@ubu.es

² Depto. de Informática, Universidad de Valladolid
Campus Miguel Delibes
yانيا@infor.uva.es

Resumen. Es habitual que el software evolucione y cambie. Partiendo de las nuevas posibilidades del lenguaje, algunas migraciones entre versiones no implican cambios funcionales pero sí una forma diferente de utilización de la aplicación, biblioteca o *framework*, manteniendo el mismo comportamiento observable. En este trabajo se presenta cómo el concepto de refactorización es una solución que permite automatizar este tipo de migraciones. En particular se mostrará el estudio realizado sobre el uso de anotaciones en la nueva versión de un *framework* ampliamente extendido, como es JUnit. Partiendo de la nueva guía de instanciación del *framework*, se definen, construyen y ejecutan refactorizaciones que permiten facilitar la migración de tests realizados en versiones previas, utilizando los nuevos elementos.

Palabras clave: refactorización, *frameworks*, evolución, migración, anotaciones.

1 Introducción

La evolución del software se ve sujeta a diferentes factores entre los que se incluyen los cambios de los lenguajes de programación, en los que finalmente se implementa la solución de diseño. En la mayoría de los casos, los cambios a los que conducen se pueden clasificar, siguiendo la taxonomía de [1], como reescritura de programas. Normalmente este proceso de reescritura conlleva algún resultado, como mejoras en el diseño, mayor facilidad de comprensión, etc. Si se entiende por refactorización como la mejora del diseño del programa de tal forma que es más fácil de comprender y modificar, preservando su comportamiento [2], y que se pueden clasificar las refactorizaciones como una especialización de la reescritura de software, en el presente trabajo se aplican las refactorizaciones con el fin de asistir y facilitar la migración de aplicaciones.

En la Sec. 2 se plantea el problema de partida, centrado en la evolución de los lenguajes, la migración de versiones de un producto, y el uso de refactorizaciones como solución al problema. En la Sec. 3 se presenta la solución de modelado del código fuente utilizada en el resto del trabajo, para en la Sec. 4 presentar algunas refactorizaciones planteadas. A partir de esta base, se presenta cómo se pueden definir y ejecutar refactorizaciones que permitan migrar código, entre diferentes versiones de bibliotecas o *frameworks*. La Sec. 5 presenta los trabajos relacionados y se finaliza en la Sec. 6, con las conclusiones obtenidas y las líneas de trabajo futuro que pueden explorarse a partir de este trabajo.

2 Planteamiento del Problema

2.1 Evolución del Lenguaje

Es difícil que los lenguajes de programación no sufran alguna lógica evolución a lo largo del tiempo. De la experiencia adquirida por los programadores, siempre se encuentran elementos

necesarios de los que carecen, que en otros lenguajes similares están disponibles, y que podrían facilitar la labor de desarrollo si fuesen incluidos. Como ejemplo, en la especificación 3.0 de Java [3], se incorporan una nueva serie de características, con el fin de mejorar el lenguaje como: genericidad, enumeraciones, anotaciones, bucles `for` mejorados, número de argumentos variables, *boxing* y *unboxing* automático, importaciones estáticas, etc. El trabajo actual se centrará en dos nuevos elementos, las anotaciones y las importaciones estáticas.

2.2 Evolución del *Framework*

El concepto de *framework* que se emplea en este trabajo es el de *framework* orientado a objeto [4]. Un ejemplo típico de *framework* de caja blanca es JUnit [5] en su versión 3 y subversiones, permitiendo automatizar la ejecución de pruebas. El funcionamiento en estas versiones se basaba en mecanismos de herencia y convención de nombres. Posteriormente el *framework* resolvía a través de mecanismos de reflexión la recolección de los tests a ejecutar. A continuación se muestra el código parcial de un test de acuerdo a las reglas marcadas en la versión 3 del *framework*, donde se pueden observar las reglas de herencia a seguir, y las convenciones de nombres obligatorias para la inicialización, ejecución y liberación de recursos en los test³:

```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
public class ListTest extends TestCase {
    public void setUp() {...}
    public void testCapacity() {...}
    public void testElementAt() {
        Integer i= (Integer)fFull.elementAt(0);
        assertTrue(i.intValue() == 1);
        try {
            fFull.elementAt(fFull.size());
        } catch (ArrayIndexOutOfBoundsException e) { return; }
        fail("Should raise an ArrayIndexOutOfBoundsException");
    }
    public void tearDown() {...}
}
```

En las siguientes versiones (versión 4 en adelante) se aprovecharon las nuevas mejoras del lenguaje, utilizando anotaciones e importaciones estáticas:

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
import org.junit.After;
public class ListTest {
    @Before public void setUp() {...}
    @Test public void testCapacity() {...}
    @Test(expected = ArrayIndexOutOfBoundsException.class)
    public void testElementAt() {
        Integer i = (Integer) fFull.elementAt(0);
        assertTrue(i.intValue() == 1);
        fFull.elementAt(fFull.size());
        return;
    }
    @After public void tearDown() {...}
}
```

2.3 Herramientas de Refactorización como Solución

Tomando la definición de refactorización en [2] como: *un cambio hecho a la estructura interna del software para facilitar su comprensión y reducir el coste de mantenimiento, sin*

³ Por motivos de brevedad se omite el cuerpo de los métodos no relevantes en el ejemplo

cambiar su comportamiento observable, se puede observar que el problema objeto de nuestro estudio entra en esta categoría. Se entiende que cambiar una clase construida bajo las directrices del *framework* JUnit (versión 3) con el objetivo de ser ejecutada en el *framework* JUnit (versión 4) no debe modificar su comportamiento observable. Así pues los tests tienen que poder ser ejecutados de diferente forma, pero con el mismo resultado. Por lo tanto, el problema de la migración de las instanciaciones puede ser resuelto mediante la aplicación de refactorizaciones.

3 Representación del Código Fuente

3.1 Generalización del Lenguaje

Las herramientas de refactorización necesitan manejar, consultar y modificar la información incluida en el código. El problema es buscar una representación adecuada. Nuestra línea de trabajo se basa en el empleo del lenguaje modelo MOON [6] y el metamodelo derivado del mismo como soporte al código. Por un lado, permite la consulta de la información almacenada, y por el otro cambiar el estado actual del metamodelo, obteniendo el código refactorizado sin pérdida de información.

La generalización conseguida con MOON permite que los conceptos comunes como clases, tipos, métodos, atributos, sean manejados a un mayor nivel de abstracción, sin tener que particularizar para un lenguaje concreto. Sin embargo, aunque el lenguaje modelo MOON puede representar puntos comunes y variantes generales, no incluye todas las características de los lenguajes de programación. Es necesario dar soporte a la variabilidad y puntos de extensión para características particulares. Por ello, tanto para su diseño como implementación, se ha optado por una solución basada en *frameworks*. Las propiedades y variantes generales se definen en el núcleo del *framework*. Las propiedades particulares de los lenguajes son ampliadas en instanciaciones concretas.

3.2 Particularizando para la Inclusión de Anotaciones

Para poder almacenar la información, es necesario que el modelo utilizado almacene de una manera natural los nuevos conceptos. La extensión para Java en su primera versión, no incluía el soporte necesario para las anotaciones incluidas finalmente en la versión 1.5, exigiendo una cierta evolución. Mientras que la adición de las importaciones estáticas tiene un escaso impacto, puesto que las diferencias son mínimas respecto al modelado de las importaciones ya incluido en versiones previas, la inclusión de las anotaciones implica una revisión y ampliación más profunda.

El concepto de anotación en nuestro modelo, se representa como una extensión del concepto de clase en MOON (`ClassDef`) y más concretamente, como extensión de una clase en Java (`JavaClassDef`), activando el flag que indica que se trata de una *interface* (ver Fig. 1). La clase utilizada para modelar la declaración de las anotaciones es `JavaAnnotation` y añade los métodos necesarios para almacenar valores por defecto asociados a los retornos de las funciones.

Desde el punto de vista del uso de las anotaciones, se modela mediante el concepto de `JavaAnnotationReference`. La referencia a una anotación está ligada a un tipo, con la restricción de que el tipo haya sido generado por una clase `JavaAnnotation` (ver Fig. 1). Finalmente el uso de la anotación implica poder dar valores particulares. Estos valores se modelan como expresiones que mantienen una referencia al método que retorna el valor, y por otro lado, al valor concreto.

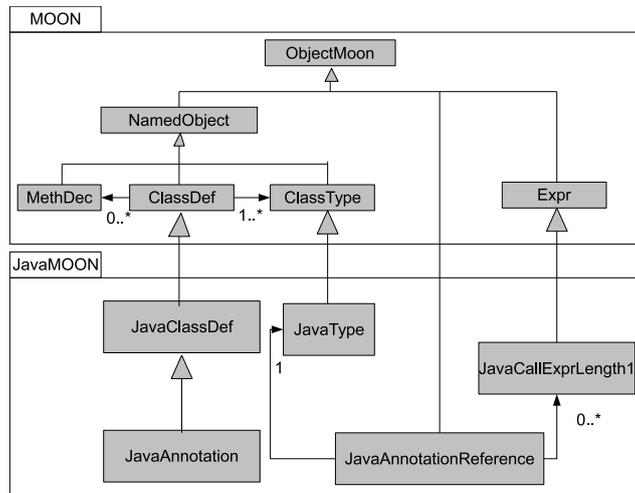


Fig. 1. Diagrama de clases simplificado de los paquetes de modelado de código

4 Refactorizaciones para la Migración entre Versiones de JUnit

4.1 Refactorización *Migrate Class From JUnit3 To JUnit4*

Se migra una clase con tests, construida en la versión 3, a una versión 4 utilizando anotaciones (ver ejemplo en la Subsec. 2.2).

Entradas: la entrada principal de la refactorización viene determinada por la **clase** que contiene los tests.

Precondiciones: la clase debe extender de la clase `junit.framework.TestCase`, tal y como marca la guía de desarrollo del *framework* en su versión 3.

Acciones:

- (1) eliminar las importaciones de la versión 3.
- (2) añadir las importaciones de la versión 4.
- (3) eliminar la cláusula de herencia de `junit.framework.TestCase`.
- (4) añadir las anotaciones correspondientes en cada método:
 - (4.1) si el nombre del método es `setUp` añadir `@Before`.
 - (4.2) si el nombre del método comienza por `test` añadir `@Test`.
 - (4.3) si el nombre del método es `tearDown` añadir `@After`.

Postcondiciones: la clase NO debe extender de la clase `junit.framework.TestCase`, tal y como marca la guía de desarrollo de la nueva versión del *framework*.

4.2 Refactorización *Migrate Exception Method*

La refactorización toma un test ya en la versión 4, que incluye bloques `try - catch - finally` para probar el lanzamiento de excepciones, y se sustituye por el valor `expected` de la anotación `@Test`, asignando el nombre de la excepción esperada.

Entrada: determinada por el **método** (test) que comprueba el correcto lanzamiento y captura de excepciones.

Precondiciones: el método es un test, tal y como marca la guía de desarrollo del *framework* en su versión 4: contiene una anotación `@Test`, es público y su tipo de retorno es `void`.

Acciones:

(1) añadir valor `expected` a la anotación `@Test` y excepciones a la cláusula `throws`, si fuera necesario, en los siguientes pasos:

(1.1) añadir el valor `expected` con las excepciones que figuran en un bloque `catch` sin instrucción `fail`.

(1.2) añadir en la cláusula `throws` aquellas excepciones que figuran en bloques `catch` conteniendo una instrucción `fail`.

(2) eliminar las instrucciones `try - catch - finally` del cuerpo del método.

(3) eliminar cualquier instrucción `fail` del cuerpo del método. Aquellos `fail` que no están en bloques `catch` serán también eliminados.

Postcondiciones: el método es un test, tal y como marca la guía de desarrollo del *framework* en su versión 4.

4.3 Ejecución de las Refactorizaciones

En nuestro caso particular, las entradas se corresponden con elementos de los modelos de MOON o JavaMoon, los predicados (pre y postcondiciones) y las acciones se implementan como clases concretas almacenadas en repositorios, distinguiendo aquellos definidos sobre MOON o Java. En particular, el uso de nuestro *framework* de ejecución, o motor de refactorizaciones [7], permite trabajar en una doble vertiente:

Codificación manual: la refactorización se puede codificar literalmente como el ensamblaje de los elementos mencionados, directamente en el constructor de la clase en Java, con los mecanismos de instanciación del lenguaje, reutilizando los elementos de los repositorios.

Ensamblaje en modo declarativo: la refactorización se compone mediante un asistente que guía en los pasos necesarios (entradas, precondiciones, etc.) Como resultado final de la ejecución del asistente se genera un fichero en XML que posteriormente es procesado por el motor de refactorizaciones para ejecutar la refactorización, utilizando reflexión [8].

5 Trabajos Relacionados

La migración de *frameworks* orientados a metadatos, ha sido planteada en [9], aplicando las refactorizaciones con un lenguaje específico de dominio, en particular sobre *frameworks* de pruebas y de persistencia. El *framework* de pruebas TestNG [10] también dispone de una herramienta específica para la migración de test realizados en JUnit. Otras variantes de migraciones también han sido planteadas en otros trabajos como [11] [12], pero centrándose básicamente en la migración de la versión de clases de utilidad y estructuras de datos no genéricas a una versión genérica. Estas soluciones son *ad-hoc*, sin un modelo ni motor de refactorizaciones orientados a la reutilización sobre una familia de lenguajes.

En relación con el soporte a la ejecución de las refactorizaciones, en [13] se expone el entorno de reingeniería denominado Moose. Las modificaciones sobre el código se realizan de manera específica, dependiendo del lenguaje analizado. La parte operacional es difícilmente reutilizable, y tampoco se plantea su uso para resolver problemas de migración.

La independencia del modelo de código, del lenguaje de programación, y la manipulación del lenguaje, usando XML como única herramienta, están descritos en [14]. Aunque el uso de XML da una gran independencia, las consultas y actualizaciones son reescritas para cada nuevo esquema (o lenguaje de programación), y la construcción de refactorizaciones requiere trabajar con XML y tecnologías derivadas, sin asistencia automática, y con menor grado de reutilización.

6 Conclusiones y Líneas de Trabajo Futuro

El trabajo aquí planteado ha mostrado cómo las refactorizaciones pueden ser utilizadas en el proceso de migración entre versiones, siempre bajo la condición de que la funcionalidad no varíe entre las mismas. La base para la definición de refactorizaciones de forma declarativa se presentó en [8], mediante un prototipo en Java, que ha sido migrado en la actualidad a un *plug-in* para Eclipse. Como línea de trabajo futuro, se plantea la ampliación del catálogo de refactorizaciones que ayuden a realizar estas migraciones. Por otra parte, parece interesante resolver el problema de la evolución de los *frameworks* y las instancias concretas (aplicaciones clientes) a través de la solución utilizada en este trabajo. Aunque el trabajo se ha centrado en un lenguaje concreto, Java, y un framework particular como JUnit, la solución mostrada en este trabajo, se puede aplicar en otros contextos: la última evolución de la especificación de los *Enterprise JavaBeans* se basa en el empleo de anotaciones, en una forma similar a la evolución seguida en JUnit, por otro lado el uso de enumeraciones y anotaciones en la plataforma .NET (denominados atributos) es una parte natural de la plataforma, por lo que refactorizaciones como la aquí planteada pueden ser también reutilizadas.

Referencias

1. E. Visser. A survey of rewriting strategies in program transformation systems. In *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57. B. Gramlich and S. Lucas Alba, 2001.
2. Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 2000.
3. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Third Edition*. Addison-Wesley, Boston, Mass., 2005.
4. Mohamed Fayad, Goug Schmidt, and Ralph Johnson. *Building Applications Frameworks: Object-oriented Foundations of Framework Design*. Wiley Computer Publishing, 1999.
5. Kent Beck and Erich Gamma. Test-infected: programmers love writing tests. pages 357–376, 2000.
6. Yania Crespo. *Incremento del potencial de reutilización del software mediante refactorizaciones*. PhD thesis, Universidad de Valladolid, 2000. Available at <http://giro.infor.uva.es/docpub/crespo-phd.ps>.
7. Yania Crespo, Carlos López, and Raúl Marticorena. Un framework para la reutilización de la definición de refactorizaciones. In *Actas JISBD'04, Málaga, Spain, ISBN 84-688-89830*, November 2004.
8. Raúl Marticorena and Yania Crespo. Dynamism in Refactoring Construction and Evolution. A Solution Based on XML and Reflection. In *3rd International Conference on Software and Data Technologies (ICSOFT)*, pages 214 – 219, July 2008.
9. Wesley Tansey and Eli Tilevich. Refactoring Object-Oriented Applications for Metadata-Based Frameworks. Technical report, Virginia Tech, Blacksburg, 2008.
10. Cédric Beust and Hani Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley Professional, 2007.
11. Robert Fuhrer, Frank Tip, Adam Kiezun, Julian Dolby, and Markus Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Glasgow, Scotland, July 27–29, 2005.
12. Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting java programs to use generic libraries. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA*, pages 15–34. ACM, 2004.
13. Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The Story of Moose: An Agile Reengineering Environment. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 1–10. ACM, 2005.
14. Nabor C. Mendoça, Paulo Henrique M. Maia, Leonardo A. Fonseca, and Rossana M. C. Andrade. RefaX: A Refactoring Framework Based on XML. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 147 – 156, 2004.