Afrontando la Evolución de los Lenguajes de Programación a través de Refactorizaciones

Raúl Marticorena[1], Carlos López[1], y Yania Crespo[2]

[1] Universidad de Burgos, EPS Edificio C, C/Francisco Vitoria s/n, Burgos, España,
rmartico@ubu.es, clopezno@ubu.es,
[2] Universidad de Valladolid, Campus Miguel Delibes, Valladolid, España,
yania@infor.uva.es

Resumen El código fuente juega un papel fundamental en las labores de mantenimiento del software, sufriendo un cambio continuo motivado por la detección de errores, la corrección de defectos en el diseño, cambios en los requisitos, etc (mantenimiento correctivo, perfectivo y evolutivo). Pero se da la particularidad de que a medida que se hacen cambios sobre el código fuente, los mismos fundamentos del lenguaje de programación utilizado, las especificaciones o estándares también cambian. Esto introduce en la práctica nuevos retos en el mantenimiento, dado que surge una nueva causa para modificar el código dada por características del lenguaje que pasan a estar en desuso y por las nuevas características introducidas en el lenguaje, que ofrecen un conjunto de nuevas posibilidades de realización en diseño.
Las operaciones de refactorización de software, definidas como operaciones que ejecutan cambios en la estructura del código que no alteran su comportamiento, y las herramientas que asisten o automatizan en la ejecución de estas operaciones, brindan una forma organizada y controlada de afrontar la transformación del código para adaptarse a la evolución del lenguaje de programación. En el presente trabajo se expone la definición, construcción y ejecución de algunas operaciones de refactorización que permiten transformar el código fuente para adaptarse a la evolución del lenguaje de programación.

Palabras clave: mantenimiento del software, lenguajes de programación, evolución, refactorización

1. Introducción

En las tareas de mantenimiento del software, el código fuente juega un papel primordial y puede ser considerado como una de las piezas básicas. En los procesos y métodos de desarrollo, en particular en los denominados procesos ágiles, el tratamiento adecuado del código se convierte en algo fundamental. Entre las mejores prácticas a aplicar, se cuenta con la refactorización del código como algo inherente al cambio continuo que algunas de estas metodologías defienden [1].

Las refactorizaciones son transformaciones del software. Su objetivo fundamental es mejorar la estructura (diseño), facilitar la comprensión y reducir costes

de mantenimiento del software. Su restricción fundamental es la preservación del comportamiento previo a la transformación [2–4]. Esto se puede conseguir de manera automática o semi-automática, si se provee de las herramientas adecuadas. Pero los cambios en el código no sólo provienen de unos requisitos cambiantes y malas elecciones de diseño al inicio del proyecto, sino que el propio lenguaje de programación, la herramienta básica de trabajo, también está cambiando o evolucionando. En este trabajo se presenta el efecto de la evolución del lenguaje fuente, y cómo una solución de soporte a las refactorizaciones puede utilizarse para afrontar los cambios en las especificaciones del lenguaje y herramientas asociadas (compiladores, *kits* de desarrollo y entornos de desarrollo integrados).

En la Sec. 2, se expone el problema de la evolución de los lenguajes de programación y sus efectos en el mantenimiento del software como una tarea fundamental en la ingeniería del software y se comenta su relación con la refactorización. Posteriormente, en la Sec. 3 se establece la base de nuestro trabajo, describiendo una plataforma basada en *frameworks* para la definición, construcción y ejecución de refactorizaciones sobre distintos lenguajes. En la Sec. 4 se presenta un caso de estudio con Java, los diferentes casos expuestos, y las soluciones dadas con nuestra propuesta. Finalmente, en la Sec. 5 se exponen otros trabajos similares y en la Sec. 6 se presentan las conclusiones y líneas de trabajo futuro.

2. Evolución de los Lenguajes de Programación y su Relación con la Refactorización del Software

La evolución de los lenguajes de programación suele implicar la inclusión de nuevos elementos en el lenguaje o en sus bibliotecas. Dichos cambios llevan a que los patrones, *idioms* y técnicas utilizados anteriormente puedan ser replanteados, afectando a la labor de adaptación del código como parte de las tareas de mantenimiento.

Este cambio en el código se ha denominado en algunos trabajos [5] como "rejuvenecer el código fuente", definido como *"la transformación fuente a fuente que reemplaza propiedades del lenguaje en desuso por nuevas soluciones"*. Entre sus objetivos se vuelve a plantear la reducción de la entropía del software, punto éste en común con la refactorización, como se señaló en [4].

La refactorización marca como objetivo básico la mejora de la estructura del código y su mantenimiento, el rejuvenecimiento del código fuente tiene otros objetivos como la migración del código fuente de versiones previas a actuales del lenguaje elegido, la formación y ayuda a los programadores a través de sugerencias automáticas de cambio o migración en los entornos de desarrollo integrados, y la optimización del código [5].

Si tomamos como ejemplo los cambios en lenguajes de la corriente principal como Java [6] o C# [7], la mayoría de cambios a nivel de lenguaje buscan escribir código más claro como principal objetivo, o más potente al introducir nuevos mecanismos que permitan realizar nuevas acciones, pero en general la optimización queda en un segundo plano.

Es discutible si existe una relación de contención entre el rejuvenecimiento del código y la refactorización (o viceversa). Pero parece obvio que si se dispone de una herramienta que permita realizar los cambios necesarios, ésta puede aplicarse a un gran número de programas escritos en la misma versión del lenguaje [5], y en algunos casos sin mayor necesidad de interacción con el usuario.

En [8], también se propone la utilización de herramientas de refactorización para resolver estas tareas. Desde su punto de vista, la adición continua de nuevas propiedades a los lenguajes hace que aparezcan una diversidad de *"dialectos"*, planteándose un problema muy grave: *"el código no evoluciona con el lenguaje"*. Si las herramientas de refactorización pueden dar una solución al cambio en aplicaciones, bibliotecas y *frameworks*, también podrían reemplazar propiedades del lenguaje caducadas por su equivalentes actuales. Su propuesta va más allá, proponiendo que en las publicaciones de nuevas versiones de los lenguajes, se acompañe del conjunto de refactorizaciones que migran el código de la versión previa a la actual. Desde este punto de vista la herramienta de refactorización (y rejuvenecimiento) pasaría a ser un elemento más del paquete de desarrollo básico a distribuir, junto con compiladores, depuradores, decompiladores, etc.

3. Soporte a la Refactorización con MOON

En esta sección se presenta el metamodelo de representación del código fuente y la arquitectura en los que se basa nuestro *framework* de refactorizaciones, con el principal propósito de potenciar la reutilización y conseguir un cierto grado de independencia del lenguaje.

3.1. Generalización del Lenguaje de Programación

Las herramientas de refactorización necesitan manejar, consultar y modificar la información incluida en el código, que en este caso es el código a refactorizar. Nuestro trabajo comenzó con la definición de MOON[3] [9] del cual, posteriormente, se derivó un metamodelo en el que se basa el almacenamiento, análisis y transformación de código. Por un lado, permite la consulta de la información almacenada y por otro lado, permite cambiar el estado actual del modelo, y obtener el código refactorizado sin pérdida de información. La descripción completa de la versión inicial está disponible en [10], utilizando como base la gramática de MOON, aunque en la actualidad ha sido mejorado con ciertas ampliaciones por una parte, y con la simplificación en cuanto al tratamiento de instrucciones y expresiones.

Los conceptos comunes a una familia de lenguajes de programación orientados a objeto como clases, tipos, métodos, atributos, son manejados a un mayor nivel de abstracción. 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

[3] Acrónimo de Mimimal Object-Oriented Notation

a través de puntos de extensión para características particulares. Por ello, tanto para su diseño como para su implementación, se optó por una solución basada en *frameworks*. El metamodelo MOON constituye el núcleo del *framework*.

Las propiedades particulares de los lenguajes, son ampliadas en instanciaciones concretas del *framework*. Por ejemplo en el caso de JAVA, y en relación con las refactorizacione presentadas en este trabajo, una instanciación del *framework* para JAVA tendría que incluir conceptos como anotaciones, importaciones estáticas, etc. Dicha extensión implementada en la actualidad se denomina JAVAMOON.

Una vez disponibles tanto el metamodelo de MOON como la extensión para el lenguaje concreto, se sigue el proceso de extracción de información del código que se detalla a continuación: se toman las bibliotecas y el código fuente, se analiza y se almacena su información sobre instancias del modelo. La información se almacena en instancias del lenguaje particular *e.g.* JAVAMOON, constituyendo un grafo que modela el estado del código inicial, pero además, al extender las abstracciones de MOON, permiten que se trabaje a un nivel superior, siempre que sea posible.

3.2. Definición y Construcción de Refactorizaciones

Las refactorizaciones son definidas siguiendo el formalismo de pre y postcondiciones, para asegurar la preservación del comportamiento [11]. Para ello, a partir del metamodelo y siguiendo los trabajos de [12], se navega sobre las instancias del metamodelo MOON a partir de una traducción del mismo a una lógica de predicados de primer orden. Los resultados de las acciones que componen la refactorización se deben verificar a través del cumplimiento de las poscondiciones.

La realización práctica de las refactorizaciones, como la suma de consultas y transformaciones sobre el código [13], se llevó a cabo inicialmente desde un planteamiento de construcción programática de manera imperativa [14] y posteriormente se añadió una solución expresada de forma declarativa con XML que permite componer dinámicamente los elementos que forman parte de la refactorización [15]. Los elementos de la definición semiformal de la refactorizacion se traducen uno a uno, a elementos declarados en un fichero tal y como marca la DTD definida.

En esta última solución, las refactorizaciones pueden ser definidas y construidas por un usuario avanzado siguiendo el proceso definido en [16] con ayuda de un asistente. Pasando de ser un elemento estático que no puede cambiar, salvo actualizaciones del propio entorno/herramienta, a algo más configurable.

3.3. Repositorios de Consultas y Transformaciones

Tanto las consultas, las transformaciones, como las propias refactorizaciones, pueden ser vistas como "piezas", realmente implementadas como clases, y almacenadas en lo que denominamos repositorios. Se pueden clasificar aquellos que dependen sólo del metamodelo MOON y por lo tanto, independientes del lenguaje y reutilizables, aquellos cuyo concepto se puede establecer en MOON pero

se difiere su implementación real a la extensión concreta (*e.g.* JavaMOON), y finalmente aquellos que sólo se construyen para un lenguaje particular (*e.g.* JavaMOON).

Las consultas se clasifican en dos tipos: predicados (consultan que un hecho/condición se cumple o no, con un estado verdadero/falso) y funciones que consultan información del código de forma similar a una sentencia SQL en una base de datos relacional. Las transformaciones se denominan acciones y modifican el estado de las instancias concretas del metamodelo, el código almacenado, alterando el estado final del mismo y observable cuando se recupera el código refactorizado. Uno de los objetivos básicos de este planteamiento, es reducir esfuerzos en la construcción de refactorizaciones, pudiendo reutilizar el mayor número de elementos, ya sean predicados, funciones o acciones, de la forma más simple posible en la construcción de nuevas refactorizaciones, y sobre nuevos lenguajes de programación.

3.4. Ensamblaje y Ejecución a través del Motor de Refactorizaciones

Sobre la base de esa necesidad de "poner en movimiento" la refactorización se definió un motor de refactorizaciones. El motor fue diseñado con el fin de lograr una cierta independencia del lenguaje de programación del código fuente sobre el que se ejecuta. En la práctica, ejecuta elementos (predicados y acciones) con independencia de que estos hayan sido construidos para un lenguaje particular, ya sea en común para todos en base a MOON, ya sea particularmente para alguno mediante una extensión *e.g.* JavaMOON.

Una vez definidas las refactorizaciones, se ensamblan en tiempo de ejecución, como se puede ver en la Fig. 1, a partir de un fichero de texto en formato XML siguiendo una DTD. Una descripción más detallada está disponible en [15].

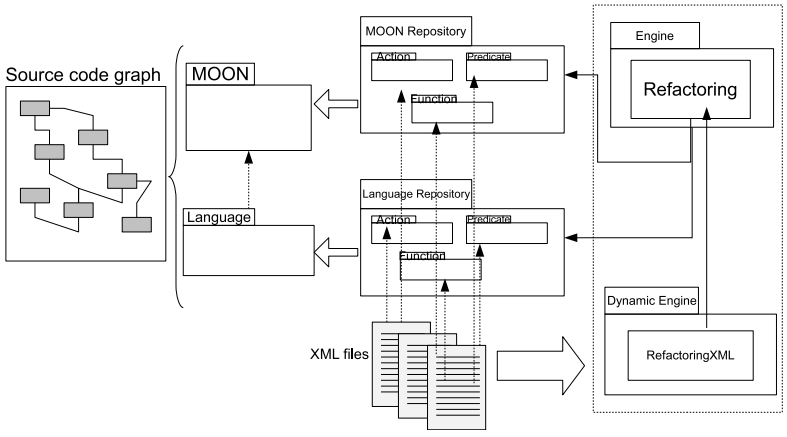

Figura 1: Ensamblaje en tiempo de ejecución de las refactorizaciones.

4. Aplicaciones a la Evolución de los Lenguajes: Caso de Estudio sobre Java

En esta sección se presenta un caso de estudio sobre la evolución del lenguaje de programación Java. En este estudio se muestra cómo se puede dar soporte a las acciones de transformación de código necesarias para hacer evolucionar el código de acuerdo a los cambios en el lenguaje mediante nuestro *framework* de refactorizaciones. En concreto se presentan dos refactorizaciones para la adaptación de código a dos nuevas características del lenguaje.

4.1. Evolución del Lenguaje de Programación Java

Pese a ser un lenguaje relativamente joven, estableciéndose a mediados de los 90 [17], ya ha sufrido diferentes cambios, como la inclusión de las clases internas, locales y anónimas en su versión 1.1 del *kit* de desarrollo, así como la aritmética estricta en punto flotante en la versión 1.2 [18], o la inclusión de los asertos en la versión 1.4 como primera aproximación al diseño por contrato. A partir de la tercera especificación del lenguaje [6], en lo que se denomina Java 5 o también 1.5, el lenguaje incluye una serie de modificaciones en cuanto a su gramática y semántica, que incluyen nuevos conceptos, así como nuevos aditamentos sintácticos que permiten la reescritura del código mejorando la claridad del mismo [19]. La versión 1.6 utilizada en la actualidad no introduce nuevos cambios en el lenguaje.

Entre los cambios que introducen una mayor potencia nos encontramos con:

Anotaciones la inclusión de anotaciones permite incluir ciertas marcas sobre elementos que permiten añadir valor semántico o bien ser procesados posteriormente para realizar acciones.

Tipos enumerados la aparición del concepto de tipo enumerado evita el uso de patrones que utilizan constantes estáticas de tipo entero.

Genéricos se incluye la posibilidad de implementar tipos paramétricos introduciendo el concepto de clases genéricas, y se extiende con los conceptos de genericidad restringida múltiple, acotación por supertipos, métodos genéricos, tipos desconocidos, etc.

Por otro lado, también se incluye otro conjunto de cambios, dentro de la categoría de azúcar sintáctico para la reescritura de código, como son los bucles `for` mejorados, número variable de argumentos en los métodos, importaciones estáticas y mecanismos automáticos de *boxing/unboxing*. En la versión 7 del lenguaje Java se esperan nuevos cambios [20] de menor o mayor calado, como literales numéricos mejorados, `Strings` en instrucciones `switch`, inferencia en instanciaciones de genéricos con el operador diamante, multi captura de excepciones, etc.

Como se puede concluir, incluso en el lenguaje más demandado para su uso en producción del software [21], el cambio es y será continuo, y por lo tanto es necesario el uso de herramientas que faciliten al programador la evolución en su código, de la manera más adecuada y con el menor impacto.

4.2. Redefinición de Métodos: Uso de la Anotación `@Override`

A partir de JAVA 1.5 el programador debe utilizar una anotación `@Override` en los métodos que se redefinen en los descendientes. Sin dicha anotación, el programador puede definir un método en alguno de los descendientes pensando que está redefiniendo un método heredado. Si la signatura de ambos métodos no es coincidente, realmente el método no se redefine sino que se sobrecarga sin ningún aviso por parte del compilador, provocando posteriormente comportamientos no esperados en tiempo de ejecución.

Frente a la solución de otros lenguaje, e.g. C#, en los que la redefinición es explícita en la sintaxis, en JAVA se optó por utilizar el nuevo mecanismo introducido de las anotaciones para que el compilador pueda realizar dicha comprobación. Si no se cumple la condición, el compilador genera un error.

El código desarrollado para la versión anterior de JAVA puede transformarse organizada y controladamente mediante una refactorización. Utilizando nuestro *framework*, presentado en la Sec. 3, para transformar código JAVA que no utiliza la anotación `@Override`, se define la refactorización `UpgradeOverrideAnnotation` que tiene como punto de entrada la clase con los métodos a actualizar con la anotación `@Override`.

El *framework* da diferentes opciones, pero es relativamente fácil navegar a partir del modelo inicial por todas las clases contenidas para aplicar la refactorización, como se puede ver en el fragmento de código del Listado 1.

Listado 1: Añadiendo la anotación en todas las clases del metamodelo

```
Collection<ClassDef> list = JavaModel.getInstance().getClassDef();
for (ClassDef cd : list){
    // si el código fuente de la clase está accesible
     if (cd.isSourceAvailable()) {
                Refactoring ref = new UpgradeOverrideAnnotation(
                        cd, JavaModel.getInstance());
                ref.run(); // ejecutar la refactorización
     }
}
```

Para poder añadir una anotación `@Override` correctamente es necesario analizar si un método redefine a un método de un ancestro. El metamodelo permite la navegación completa por la jerarquía de herencia de clases, almacenado las cláusulas de herencia, los tipos de los que se hereda y las correspondientes clases determinantes del tipo que pasan a ser ancestros, como se muestra en la Fig. 2.

Las signaturas de los métodos se obtienen a partir de su nombre almacenado y de la lista ordenada de las entidades de tipo `FormalArgument` (ver Fig. 3) cada una de ellas con su correspondiente tipo asignado. Las relaciones y reglas de subtipado se extraen de la jerarquía almacenada previamente.

La refactorización se compone de una única acción `AddOverrideAnnotation`, que recibe como argumento de entrada la clase a procesar, añadiendo la anotación en aquellos métodos que no contienen dicha anotación y redefinen a métodos en ancestros. La naturaleza particular de esta refactorizacion, más cercana a la actualización de código ante la evolución del lenguaje, hace que se simplifique su construcción frente a las posibilidades que ofrece nuestra plataforma, sin seguir un esquema complejo de pre, postcondiciones y acciones, y sin necesidad de interacción por parte del usuario.

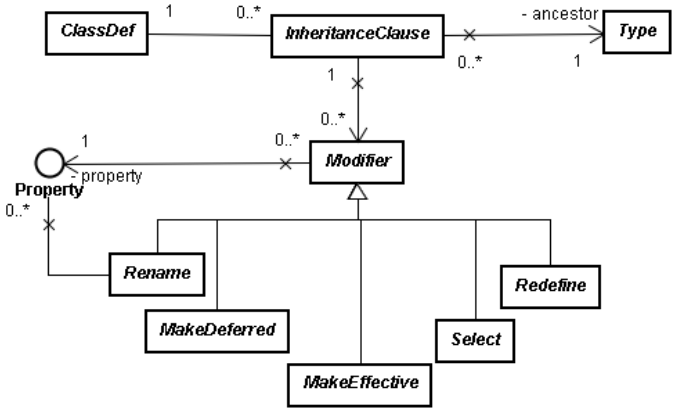

Figura 2: Herencia en el metamodelo MOON

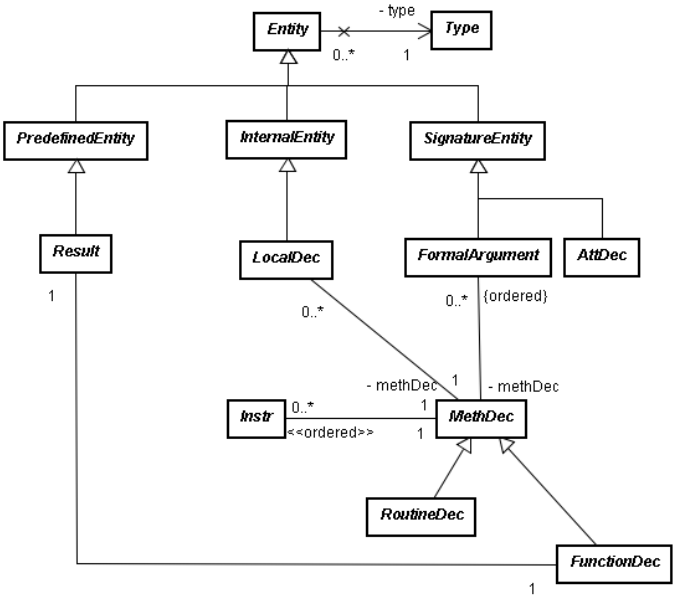

Figura 3: Métodos en el metamodelo MOON

Aplicando la refactorización tal y como se describe en el Listado 1, con las bibliotecas necesarias, sobre distintos ejemplos de código, se obtienen resultados como los expuestos a continuación en la Tabla 1. Se ha añadido información del número de clases y tipos generados. Estos se generan no sólo a partir de

los ficheros fuente, sino a partir de la carga de las bibliotecas necesarias para la compilación, y para un análisis completo de las relaciones existentes.

En el caso de JUnit 4.8.2, se debe comentar el hecho de que inicialmente en el framework se encontraban ya 87 anotaciones `@Override`. Nuestra herramienta detecta y añade 48 anotaciones a métodos, hasta completar un total de 135 métodos redefinidos en el framework. Inicialmente los programadores sí conocían esta posibilidad, pero algunos de los métodos quedan sin marcar, frente a la solución presentada. En el caso de JHotDraw 6.0b1 y JFreeChart 1.0.13, no se encontraba ninguna anotación en la versión procesada, mientras que en JBidWatcher 2.1.4.1 sólo existían 4 anotaciones inicialmente. Como se puede observar en todos estos casos, el número de métodos redefinidos detectados es muy alto, siendo inviable el realizar este proceso en un tiempo razonable, sin una herramienta.

4.3. Incluyendo Genericidad en las Clases

Con la inclusión de la genericidad en Java 1.5, surge el problema de la migración de clases que abusaban del uso del tipo universal `java.lang.Object` y del polimorfismo de inclusión, para simular el polimorfismo paramétrico. Dicha migración se puede realizar manualmente, pero también de forma asistida por herramientas.

Nuestro *framework* introduce el concepto de genericidad, soportando parámetros formales (`FormalPar`) en clases y métodos genéricos (ver Fig. 4) así como variantes de acotación (subtipado o cláusulas *where*).

La extensión JavaMOON, incluye todos estos conceptos para dar un soporte completo, tanto sintáctico como semántico, incluyendo las particularidades de la genericidad en Java no incluidas en MOON.

A partir de la implementación de la refactorización *Parameterize* definida en [9], donde se establecen las entidades dependientes en tipo respecto a una entidad guía, así como las entidades cuyo tipo pasa a ser paramétrico, se obtiene un primer prototipo para la parametrización de una clase.

Para la construcción de la refactorización se han implementado diferentes predicados que chequean precondiciones como: la no existencia de parámetros formales coincidentes en nombre, la entidad guía no puede tener un tipo variable, no introduce recursividad en su definición de tipo, no existen entidades

Tabla 1: Resultados en la aplicación de la refactorización `UpgradeOverrideAnnotation`

Producto	Ficheros .java	Clases	Tipos	LOC	@Override añadidos
JUnit 4.8.2	110	16.914	21.026	7.339	48
JBidWatcher 2.1.4.1	211	23.474	28.371	39.034	566
JHotDraw 6.0b1	484	19.749	23.976	71.917	1.769
JFreeChart 1.0.13	968	18.744	22.847	305.743	2.386

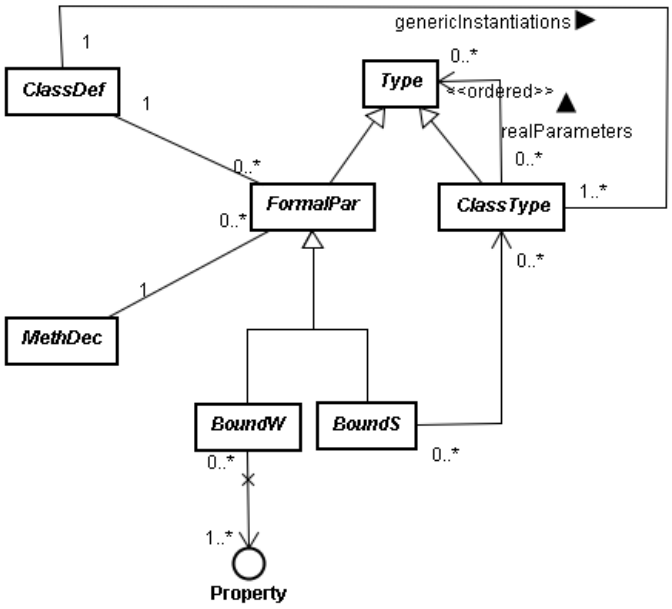

Figura 4: Genericidad en el metamodelo MOON

que tengan que cambiar a la vez a tipo variable y a tipo paramétrico, no hay asignaciones a constantes en las entidades dependientes, y no hay asignación de creación a las entidades que cambian su tipo a variable.

Por otro lado, en el repositorio se añaden un conjunto de acciones que transforman el código añadiendo las cuestiones relativas a la genericidad. En particular se añade una acción particular para transformar las instanciaciones genéricas (e.g. `new Nodo<T>`), cuestión que en MOON no estaba contemplada pero que en JAVA es necesaria.

En la Tabla. 2 se muestra un ejemplo de aplicación de la refactorización *Parameterize* sobre código JAVA. La Tabla 2 muestra la clase objetivo `Lista`, antes y después de aplicar la refactorización definida y construida con nuestro *framework*. Como puede verse en dicho ejemplo, la refactorización no sólo tiene efectos en la clase objetivo sino también en aquellas clases, como la clase `Nodo`, que necesitan ser también parametrizadas. Las entradas de esta refactorización son la clase objetivo, `Lista` en el ejemplo, y una entidad guía, en ejemplo se ha utilizado el argumento formal `elem` del método `insertar`. En este caso, es necesaria la interacción con el usuario para elegir la entidad guía.

Tabla 2: Resultado de la refactorización *Parameterize* con el código antes y después de aplicar la refactorización

```
public class Lista {

  int total = 0;
  Nodo primero = null;

  public void insertar(int elem){
    int i;
    Nodo ultimo;
    Nodo cursor;
    boolean fin;
    boolean seraPrimero;
    i = 1;
    cursor = primero;
    fin = i>=total;
    i = i + 1;
    ultimo = new Nodo();
    ultimo.iniciar(elem);
    seraPrimero = (cursor == null);
    primero = ultimo;
    cursor.ponerSiguiente(ultimo);
    total = total + 1;
  }

  public void eliminar(){
    boolean condicion;
    condicion = vacio();
    primero = primero.siguiente;
  }

  public boolean vacio(){
    return (total == 0);
  }
}
```

```
public class Lista<T> {

  int total = 0;
  Nodo<T> primero = null;

  public void insertar(T elem) {
    int i;
    Nodo<T> ultimo;
    Nodo<T> cursor;
    boolean fin;
    boolean seraPrimero;
    i = 1;
    cursor = primero;
    fin = i >= total;
    i = i + 1;
    ultimo = new Nodo<T>();
    ultimo.iniciar(elem);
    seraPrimero = (cursor == null);
    primero = ultimo;
    cursor.ponerSiguiente(ultimo);
    total = total + 1;
  }

  public void eliminar() {
    boolean condicion;
    condicion = vacio();
    primero = primero.siguiente;
  }

  public boolean vacio() {
    return (total == 0);
  }
}
```

```
public class Nodo {

  int elem;
  Nodo siguiente;

  public void iniciar(int un_elem){
    elem = un_elem;
    siguiente = null;
  }

  public void ponerSiguiente(
                      Nodo otro){
    siguiente = otro;
  }

  public void cambiarElem(
                    int un_elem){
    elem = un_elem;
  }
}
```

```
public class Nodo<T> {

  T elem;
  Nodo<T> siguiente;

  public void iniciar(T un_elem) {
    elem = un_elem;
    siguiente = null;
  }

  public void ponerSiguiente(
                      Nodo<T> otro) {
    siguiente = otro;
  }

  public void cambiarElem(
                     T un_elem) {
    elem = un_elem;
  }
}
```

5. Trabajos Relacionados

La búsqueda de independencia del lenguaje de programación en refactorización, ha sido un punto abierto en la anterior década. El ejemplo más notable es el metamodelo FAMIX [22] que permite el intercambio de información entre herramientas CASE de reingeniería. Una de las herramientas desarrolladas bajo FAMIX fue la herramienta de refactorización MOOSE [23–25]. El metamodelo FAMIX soporta los conceptos de la orientación a objetos como clases, métodos, atributos o herencia. También el acceso a propiedades. Sin embargo, no soporta herencia múltiple o genericidad, motivado por una influencia de lenguajes no estáticamente tipados en el diseño del metamodelo. En la misma línea fueron propuestos, otros metamodelos basados en extensiones de UML 1.4 co-

mo GrammyUML [26]. Estos trabajos no han sido orientados a la renovación o rejuvenecimiento del código en la actualidad.

En la aplicación de refactorizaciones sobre un lenguaje como Java existen multitud de trabajos. Entre las aplicaciones prácticas en entornos de desarrollo integrados, el *plug-in* de refactorización propio de Eclipse [27] da una solución similar en resultados a la mostrada en este trabajo, como la inclusión de anotaciones `@Override` en el código, aunque difícil de extrapolar o reutilizar en otros entornos. Similar situación nos encontramos con otros entornos de desarrollo tan extendidos como Netbeans [28] o IntelliJ IDEA [29] en Java, que o bien no incluyen estas utilidades o no son reutilizables en otros contextos.

Otras trabajos en la inclusión de genericidad, en la línea de la refactorización *Parameterize*, están disponibles en [30, 31], pero centrándose en la migración de la versión de clases de utilidad y estructuras de datos no genéricas a una versión genérica. Estas soluciones no se basan en un modelo ni un motor de refactorizaciones orientados a la reutilización.

La transformación de código con estrategias de reescritura ha sido tratada en Stratego/XT [32], a través de un lenguaje específico de dominio. Por otro lado, existen trabajos actualmente en desarrollo como MoDisco [33] para el rejuvenecimiento o modernización en sistemas legados, por medio de la transformación de modelos. En la medida de nuestro conocimiento, estas herramientas no han sido aplicadas para resolver casos como los aquí planteados.

6. Conclusiones y Líneas de Trabajo Futuro

Los resultados obtenidos al facilitar el mantenimiento ante cambios del lenguaje de programación, tanto en la inclusión de anotaciones como los resultados iniciales introduciendo la genericidad, demuestran la aplicabilidad del *framework* MOON y sus extensiones para realizar labores de mantenimiento ante su evolución.

La aplicación de una manera simple de herramientas que faciliten la transición entre versiones, tiene un alto valor para el programador que realiza tareas de mantenimiento con un menor esfuerzo. En la práctica, es complicado que estas soluciones no acaben tomando un enfoque demasiado restringido a un lenguaje, herramienta y problema concreto, sin posibilidades de extensión y cambio a nuevos problemas que se plantearán en un futuro próximo. Nuestra propuesta, en la medida de lo posible, intenta evitar estos problemas de rigidez. Sin embargo, aunque somos conscientes de que las operaciones de rejuvenecimiento son específicas del lenguaje de programación, la solución aportada permite reutilizar en su construcción ciertos elementos, con un menor esfuerzo.

En la actualidad, la solución descrita en este trabajo se ha integrado como un *plugin* de Eclipse, para el lenguaje Java. Sin embargo es necesario integrar estas herramientas dentro de otros *kits* y entornos de desarrollo integrados.

Dentro de las líneas de trabajo futuro, se deben mejorar los resultados y rendimiento de las refactorizaciones, añadir nuevas refactorizaciones en la línea de

los cambios apuntados, así como estudiar la aplicabilidad en relación a los cambios programados en las futuras versiones JAVA 7 y JAVA 8 [20]. Por otro lado, se debe considerar repetir este planteamiento con otros lenguajes de programación, como en el caso de C# [7] en la familia .NET.

Agradecimientos

Este trabajo ha sido financiado por el *Ministerio de Ciencia e Innovación*, en el proyecto TIN2008-05675.

Referencias

1. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1st edition, October 1999.
2. Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 2000.
3. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.
4. Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1999.
5. Peter Pirkelbauer, Damian Dechev, and Bjarne Stroustrup. Source code rejuvenation is not refactoring. In *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '10, pages 639–650, Berlin, Heidelberg, 2010. Springer-Verlag.
6. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Third Edition*. Addison-Wesley, Boston, Mass., 2005.
7. Microsoft. C# Language Specification Version 4.0, 2010.
8. Jeffrey L. Overbey and Ralph E. Johnson. Regrowing a language: refactoring tools allow programming languages to evolve. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 493–502, New York, NY, USA, 2009. ACM.
9. 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.
10. Carlos López and Yania Crespo. Definición de un Soporte Estructural para Abordar el Problema de la Independencia del Lenguaje en la Definición de Refactorizaciones. Technical Report DI-2003-03, Departamento de Informática. Universidad de Valladolid, septiembre 2003. Available at http://giro.infor.uva.es/docpub/lopeznozal-tr2003-03.pdf.
11. Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.
12. Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. Translating the object constraint language into first-order predicate logic. In *In Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC*, pages 113–123, 2002.
13. Carlos López, Raúl Marticorena, and Yania Crespo. Hacia una solución basada en frameworks para la definición de refactorizaciones con independencia del lenguaje. In Jaime Gómez Ernesto Pimentel, Nieves R. Brisaboa, editor, *Actas JISBD'03, VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante, Spain ISBN: 84-688-3836-5*, pages 251–262, November 2003.

14. Yania Crespo, Carlos López, and Raúl Marticorena. Un Framework para la reutilización de la definición de refactorizaciones. In *Actas JISBD'04, IX Jornadas de Ingeniería del Software y Bases de Datos, Málaga, Spain, ISBN 84-688-89830*, November 2004.
15. 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.
16. Raúl Marticorena, Carlos López, and Yania Crespo. Definición de un Proceso para la Contrucción de Refactorizaciones. In *JISBD'07, XII Jornadas Ingeniería del Software y Bases de Datos, Zaragoza, Spain*, pages 361–367, sep 2007.
17. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Number ISBN 0-201-63451-1. Addison-Wesley, 1996.
18. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.
19. Brett McLaughlin and David Flanagan. *Java 1.5 Tiger. A Developer's Notebook*. O'Reilly.
20. Oracle. JDK 7 Features. http://openjdk.java.net/projects/jdk7/features/, January 2011.
21. TIOBE Company. TIOBE Tiobe software: Tiobe index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2011.
22. Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A Meta-Model for Language-Independent Refactoring. In *Proc. International Workshop on Principles of Software Evolution (IWPSE)*, pages 157–169. IEEE Computer Society Press, 2000.
23. Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proc. Int'l Symp. Constructing Software Engineering Tools (CoSET)*, June 2000.
24. Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. The Moose Reengineering Environment. *Smalltalk Chronicles*, 2001.
25. 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.
26. Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards Automating Source-Consistent UML refactorings. In *UML*, pages 144–158, 2003.
27. Berthold Daum. *Eclipse 3 para desarrolladores Java*. Anaya Multimedia, 1st edition, 2005.
28. Oracle. Welcome to netBeans. http://netbeans.org/, 2011.
29. JetBrains. IntelliJ IDEA :: The Most Intelligent Java IDE. http://www.jetbrains.com/idea/, 2006. Java IDE.
30. Wes Munsil. Case study: Converting to java 1.5 type-safe collections. *Journal of Object Technology*, 3(8):7–14, 2004.
31. Robert Fuhrer, Frank Tip, Adam Kieżun, Julian Dolby, and Markus Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP'05, 19th European Conference Object-Oriented Programming*, Glasgow, Scotland, July 27–29, 2005.
32. Eelco Visser. Stratego program transformation language. http://strategoxt.org/, 2011.
33. Eclipse Foundation. MoDisco. http://www.eclipse.org/MoDisco/.