

Un Framework para la Reutilización de la Definición de Refactorizaciones*

Yania Crespo¹, Carlos López² y Raúl Marticorena²

¹ Universidad de Valladolid, Dpto. de Informática
{`yania`}@infor.uva.es

² Universidad de Burgos, Área de Lenguajes y Sistemas Informáticos
{`clopezno`, `rmartico`}@ubu.es

Resumen Actualmente la inclusión de refactorizaciones en los entornos de desarrollo orientados a un lenguaje concreto de programación está ampliamente extendida. Sin embargo se detecta una pérdida de reutilización del esfuerzo realizado cuando se implementan esas mismas refactorizaciones para otros lenguajes de programación.

El presente trabajo expone el diseño de los distintos elementos de un motor de refactorizaciones que permite definir y aplicar refactorizaciones. Su diseño basado en frameworks abre la posibilidad de extensión sobre otros lenguajes y otras refactorizaciones, de cara al desarrollo para y con reutilización de herramientas de soporte a la refactorización de código. Por otro lado también se discuten diferentes alternativas de diseño frente al problema de regeneración de código una vez realizadas las refactorizaciones.

Palabras clave: refactorización, motor de refactorizaciones, frameworks, lenguaje modelo, programación orientada a objetos, regeneración de código.

1. Introducción

Como se señala en [1], el creciente auge de los trabajos realizados en el área de la refactorización de software está marcando una serie de tendencias: identificar cuándo debe ser refactorizado el código, identificar refactorizaciones a aplicar, ejecutar las refactorizaciones, ver los efectos sobre la calidad, etc. Entre las actividades en auge se encuentra como línea abierta de investigación, la búsqueda de una cierta independencia del lenguaje a la hora de definir, analizar y desarrollar herramientas que asistan al proceso de refactorización.

En este artículo se muestra en la Sección 2 los trabajos que se han publicado con un propósito similar a éste: avanzar en la definición y la construcción de un motor de refactorizaciones con cierta independencia del lenguaje y con soporte para la reutilización. En la Sección 3 se describe una panorámica de nuestro trabajo desde donde parte el actual. En la Sección 4 se analiza la definición de

* Este trabajo ha sido parcialmente financiado por la Junta de Castilla y León a través del proyecto de investigación VA093/03.

refactorizaciones sobre un framework, como solución al propósito que se plantea en el artículo y en la Sección 5 se muestra un ejemplo explicando las posibilidades de reutilización. En la Sección 6 se propone una solución al problema de la regeneración del código refactorizado a partir de la información almacenada. Por último en la Sección 7 se concluye y se muestran las líneas de trabajo futuras.

2. Trabajos Relacionados

En trabajos previos en la definición de refactorizaciones independientes del lenguaje [2] se define el modelo FAMIX como un metamodelo para almacenar información con el objetivo de integrar varios entornos de desarrollo, junto con una herramienta para asistir a las refactorizaciones, denominada MOOSE. Como punto de partida para la definición del modelo se centran en el estudio de dos lenguajes de características diferentes como SMALLTALK y JAVA, no tomando en cuenta características complejas en los lenguajes fuertemente tipados, objeto de nuestros estudios previos [3,4], como herencia avanzada y genericidad. En MOOSE se adopta como técnica el uso de aserciones [1] utilizando la información del metamodelo para verificar las precondiciones de la refactorización. Basándose en la comprobación de las mismas, se garantiza la preservación del comportamiento.

Sin embargo, las transformaciones asociadas a la refactorización se realizan directamente sobre el código fuente original. Esta alternativa obliga a realizar transformadores de código específicos para cada lenguaje apoyados en expresiones regulares. En estos trabajos se propone como línea futura incluir mejoras utilizando árboles de sintaxis abstracta (AST).

Otra línea de investigación que busca un soporte de la información del código orientado a objeto como [5], utiliza lenguajes lógicos de programación sobre meta lenguajes. Para la definición de las refactorizaciones se propone en realizar una serie de transformaciones sobre los elementos que forman el código. Para asegurar la corrección se verifica el cumplimiento de precondiciones, similar a lo propuesto en [6,7]. Sus actuales trabajos y prototipos se centran en SMALLTALK.

Finalmente, existe una propuesta todavía más ambiciosa en el trabajo de [8] para definir refactorizaciones genéricas con independencia del paradigma de programación.

3. Contexto Inicial

El enfoque seguido en el diseño del framework respecto a la independencia del lenguaje es utilizar un lenguaje modelo intermedio MOON (módulo MOON CORE en Fig. 1), que recoge las características comunes de distintas familias de lenguajes orientados a objeto (LOO). Cada lenguaje en particular tiene su propia extensión definida a partir de MOON, donde se recogen sus características específicas (módulos JAVA EXTENSION, EIFFEL EXTENSION, etc en Fig. 1).

Respecto a la técnica utilizada para garantizar parcialmente la preservación del comportamiento se utiliza el enfoque basado en aserciones a través de pre y postcondiciones definidas semiformalmente sobre los elementos del lenguaje

modelo [3, 4] y que se plasman en un conjunto de funciones y predicados de consulta sobre las instancias de MOON CORE. Las acciones asociadas se definen como operaciones de transformación sobre dichas instancias.

El sistema debe quedar abierto a la incorporación de nuevas definiciones de refactorizaciones a partir de los predicados, funciones y acciones existentes en los repositorios (ENGINE REPOSITORY).

El grado de automatización de la herramienta propuesta es semiautomático, en el sentido que el desarrollador debe indicar a la herramienta qué parte de software necesita ser refactorizado y seleccionar la refactorización concreta a realizar.

Actualmente el trabajo se centra en la aplicación de la refactorización sobre el código fuente sin afectar al resto de artefactos asociados (especificación de requisitos, arquitecturas, modelos de diseño, etc.)

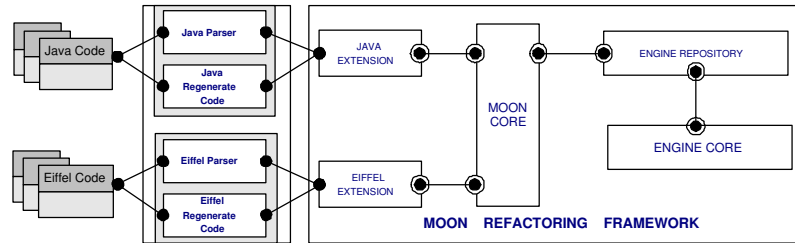


Figura 1. Arquitectura global

Bajo estas premisas, en la Fig. 1 se muestra la arquitectura diseñada en la que cabe destacar:

- El **motor de refactorizaciones**, formado por un núcleo y repositorio (ENGINE CORE y ENGINE REPOSITORY), actúa sobre las instancias de las abstracciones del lenguaje modelo, tanto para la verificación parcial de preservación del comportamiento, como para realizar las propias acciones de la refactorización.
- Los **regeneradores de código** actúan sobre una extensión del lenguaje modelo (JAVA EXTENSION, EIFFEL EXTENSION, etc.) utilizando la funcionalidad común definida sobre sus abstracciones.

Los módulos MOON CORE y las distintas extensiones han sido presentadas en trabajos previos [3, 9] siendo el objetivo de este trabajo en sus secciones posteriores presentar la estructura del motor y repositorio de refactorizaciones junto con una propuesta de diseño para solucionar la regeneración del código refactorizado.

4. Motor de Refactorizaciones

Las refactorizaciones deben ser definidas sobre un soporte que facilite por un lado su ejecución y por otro, la reutilización y extensión en la construcción de nuevas refactorizaciones. En esta sección se muestra la definición y diseño del núcleo del motor de refactorizaciones así como una extensión de los elementos, operando sobre MOON.

4.1. Núcleo del motor

Partiendo de los estudios previos [3, 4], se ha identificado una plantilla con una serie de elementos descriptivos (nombre, descripción, motivación), argumentos de entrada y pasos que forman un algoritmo común a todas las refactorizaciones. En concreto, los elementos que participan en la ejecución de una refactorización son:

Precondiciones se identifica una fase de verificación para poder llevar a cabo la refactorización.

Acciones operan directamente sobre las instancias del código minimal obtenido.

Postcondiciones se identifica una fase de validación de predicados que deben verificarse a la finalización de las acciones.

El diseño de las clases que forman el núcleo del motor (paquete `engine.core`) se representa en la Fig. 2. El algoritmo de ejecución de refactorizaciones es común a todas ellas dejando como puntos de enganche la extensión de elementos que participan en una refactorización particular. Tomando como base el patrón de diseño **Template Method** [10] se establece en la clase abstracta **Refactoring** el método `run` que cumple el papel de *método plantilla*, que define el algoritmo que controla la interacción entre los elementos. Estos elementos se agrupan para formar las refactorizaciones concretas a partir de las clases en los repositorios (paquetes `engine.repository.*`): predicados y funciones formando pre y postcondiciones, y acciones.

Para implementar la validación de precondiciones, y postcondiciones y la realización de acciones en cada una de las refactorizaciones se sigue el patrón de diseño **Command** [10] tomando dicho papel las clases **Predicate** y **Action**. En el caso concreto de las acciones, se permite la posibilidad de deshacer y registrar dichas operaciones, en caso de producirse alguna excepción en el proceso de refactorización.

4.2. Extensiones del motor

El motor de refactorizaciones opera sobre las abstracciones del framework MOON CORE. Es necesario definir extensiones concretas a los elementos del núcleo. Para ello se definen cada una de las refactorizaciones concretas como por ejemplo **Add**

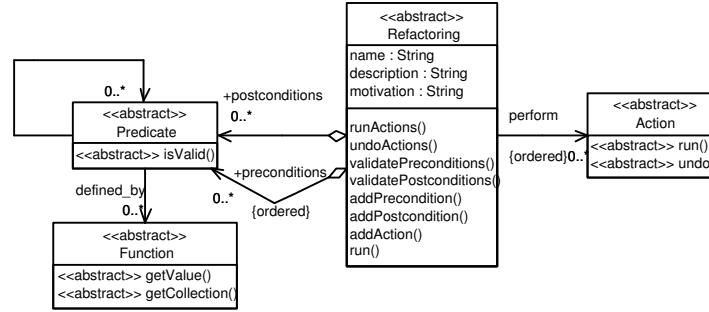


Figura 2. Diseño del framework para el motor de refactorizaciones

Class, Remove Method, ReplaceFormalParameterWithType, etc. [4,11], como extensiones concretas de la clase **Refactoring**.

En nuestro diseño, las clases abstractas **Predicate** y **Action** definen predicados y acciones generales mientras que las subclases en los paquetes **engine.repository.concretepredicate** y **engine.repository.concreteaction** implementan las clases concretas. En el diseño del framework, el *método plantilla* toma el rol de *invocador* mientras que las clases concretas del framework MOON CORE son los *receptores* de las operaciones implicadas en los métodos **isValid** (en **Predicate**) y **runAction** (en **Action**). Los constructores de las refactorizaciones concretas son los responsables de instanciar y asociar los predicados (como pre o postcondiciones) y acciones en el orden oportuno, para ser ejecutadas.

5. Reutilización en la Definición de Refactorizaciones

A continuación se presenta un caso de estudio de una refactorización concreta, a partir de clases definidas en los paquetes **engine.repository.***. En este apartado se trabaja con la refactorización **ReplaceFormalParameterWithType**, descrita en [4]. En dicha refactorización se elimina uno de los parámetros formales de una clase genérica sustituyendo su presencia por un tipo completamente instanciado.

La refactorización **ReplaceFormalParameterWithType** (dentro del paquete **engine.repository.concreterefactoring**), hereda de **Refactoring**. En su constructor (Listado. 1.1) se realiza la instanciación de elementos, colaborando en el algoritmo del *método plantilla* (**run**) descrito previamente en la Sec. 4.

Como resultado de este proceso continuo de definición de refactorizaciones se obtiene una posibilidad de reutilización en la definición de nuevas refactorizaciones. El número de elementos en **engine.repository.*** a definir desde cero decrece gradualmente. Tomando como ejemplo otra refactorización, como **SpecializeBoundS**, definida en [4], el predicado *IsFormalPar* ya estaría desarrollado y almacenado en el repositorio como consecuencia del desarrollo previo

Listado 1.1. Constructor en ReplaceFormalParameterWithType.java

```
public ReplaceFormalParameterWithType( ClassDef classDef ,
    FormalPar formalPar , ClassType classType , Collection types ) {

    super( NAME );

    this.addPrecondition( new IsFormalPar( formalPar , classDef ) );
    this.addPrecondition( new IsSingleGenericInstance( classDef ,
        formalPar , classType ) );

    this.addAction( new DeleteAllRealParameter( classDef , formalPar ,
        types ) );

    this.addAction( new DeleteFormalParameter( classDef , formalPar ) );

    this.addAction( new SubstituteFormalParameter( classDef ,
        formalPar , classType ) );

    this.addPostcondition( new IsNotFormalPar( formalPar , classDef ) );
}
}
```

de la refactorización `ReplaceFormalParameterWithType`. Progresivamente esta situación se reproduce, de tal forma que se llega a un repositorio suficientemente completo como para desarrollar nuevas refactorizaciones a partir de elementos ya definidos en un tiempo mínimo.

Por otro lado, el esfuerzo para llevar a cabo estas refactorizaciones ya definidas sobre un código escrito en otro LOO se reduce a la elaboración de un parser hacia su correspondiente extensión del framework `MOON CORE`. La refactorización es ejecutada por el motor, quedando pendiente la regeneración del código. Este sistema permite realizar las mismas refactorizaciones contenidas en el repositorio del motor, siempre y cuando las características básicas del lenguaje estén soportadas sobre el lenguaje modelo `MOON`, por lo tanto soportadas sobre el framework `MOON CORE` y se cuente con su extensión particular.

6. Regeneración del Código Refactorizado

La regeneración de código refactorizado pasa por obtener la información contenida en las instancias de las extensiones de `MOON CORE`. Este apartado se ha desglosado en dos secciones, la primera discute la problemática asociada a la regeneración de código a partir de los elementos del lenguaje modelo y la segunda, describe una solución de diseño para un módulo de regeneración de código.

6.1. Definición del problema

`MOON` es una representación intermedia. Para obtener dicha representación se utilizan parsers específicos (patrón de diseño **Builder** [10]) de cada lenguaje, que capturan la información de los distintos códigos fuentes. Al no ser `MOON` un lenguaje completo respecto a todas las construcciones sintácticas de las distintas familias de lenguajes, la transformación hacia `MOON` puede llevar asociada una pérdida de información sintáctica.

Bajo esta premisa y en un contexto de refactorización se puede categorizar la información contenida en el código en tres tipos: información común en todos los

LOO, información específica de cada lenguaje e información no relevante en el conjunto de refactorizaciones. El problema planteado, es decidir cómo almacenar la información no relevante respecto al conjunto de refactorizaciones necesarias para poder obtener el código en su lenguaje original. Las soluciones propuestas indican dónde y cómo debe ser almacenada sobre las extensiones:

- Almacenar de forma textual el cuerpo completo de los métodos en su lenguaje original dentro de un campo de las extensiones de la clase `METHOD_DEC`, como por ejemplo la clase `JMETHOD_DEC` definida en `JAVA_EXTENSION`. Como ventaja, esta alternativa no incrementa la jerarquía de clases en las distintas extensiones pero necesita de algún proceso adicional que se encargue de modificar el cuerpo de los métodos.
- Definir sobre las distintas extensiones de `MOON CORE` (`EIFFEL_EXTENSION`, `JAVA_EXTENSION` en Fig. 1) el AST correspondiente a la definición del cuerpo de los métodos. Se debe proporcionar una asociación desde la clase específica que extiende de `METHOD_DEC` a la nuevas clases creadas a partir del AST. No se necesita de procesos adicionales para la modificación del cuerpo de los métodos pero incrementa en exceso la jerarquía de clases en las distintas extensiones.

6.2. Módulo de regeneración de código

Independientemente de la alternativa seleccionada para almacenar la información referente al cuerpo de los métodos, se puede definir una solución para los módulos de regeneración de código mostrados en la Fig. 1.

Un regenerador de código obtendrá la información a partir de las extensiones específicas del lenguaje. El regenerador recorre la estructura de objetos asociada a la definición de una clase pidiendo a cada elemento la obtención de su código asociado. Se implementa la operación `getCode()` en cada uno de los elementos de la gramática que representan las extensiones concretas. Para dar soporte a esta operación se define el método abstracto en el nodo raíz de la jerarquía obligando a hacerla efectiva en alguna clase correspondiente a una extensión de un lenguaje concreto.

En previsión de poder añadir funcionalidad para la cuantificación en términos de calidad, y con el objetivo de mejorar el mantenimiento, comprensión y capacidad de cambio de las clases que conforman el framework `MOON CORE`, se propone para futuras versiones encapsular cada operación que implique recorridos sobre el árbol sintáctico en una clase independiente aplicando el patrón de diseño **Visitor** [10].

7. Conclusiones y Líneas de Trabajo Futuro

El presente trabajo ha presentado la definición de un framework para el soporte de un motor de refactorizaciones buscando una cierta independencia en la ejecución de refactorizaciones, abriendo posibilidades de reutilización tanto en la

ejecución de refactorizaciones como en su definición. Por otro lado, se han planteado distintas soluciones de diseño en la regeneración del código refactorizado.

En la actualidad, se está trabajando sobre un parser de código JAVA a instancias de `JAVA EXTENSION`, y por lo tanto extensiones del framework `MOON CORE`. Sobre `MOON CORE` se han implementado un conjunto de refactorizaciones concretas, implementando sus elementos (predicados, funciones y acciones) dentro de `ENGINE REPOSITORY`. Las refactorizaciones concretas son ejecutadas por el motor, contenido en el framework `ENGINE CORE`.

A partir de este punto se marcan como líneas de trabajo a seguir: estudiar la disminución progresiva de tiempo en la definición de nuevas refactorizaciones y por otro lado, aumentar el número de extensiones concretas al framework que soporta el motor, profundizando en la incorporación de genericidad en lenguajes como `JAVA` y `C#` y dejar abierto el camino a la inclusión del soporte del Microsoft Intermediate Language (IL), dentro de la plataforma `.NET`.

Como objetivo final, está la integración de todos estos elementos en herramientas de desarrollo multilenguaje que incluyan la ejecución de refactorizaciones, buscando esa cierta independencia del lenguaje empleado.

Referencias

1. Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.
2. Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, 2001.
3. Carlos López, Raúl Marticorena, and Yania Crespo. Hacia una solución basada en frameworks para la definición de refactorizaciones con independencia del lenguaje. In *Actas JISBD'03, Alicante, España*, November 2003.
4. Raúl Marticorena, Carlos López, and Yania Crespo. Refactorizaciones de especialización en cuanto a genericidad. Definición para una familia de lenguajes y soporte basado en frameworks. In *Actas PROLE'03, Alicante, España*, November 2003.
5. Tom Tourwé and Tom Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *Proc. 7th European Conf. on Software Maintenance and Reengineering*, pages 91 – 100, Benvento, Italy, 2003. IEEE Computer Society.
6. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.
7. Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1999.
8. Ralf Lämmel. Towards Generic Refactoring. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, October5 2002. ACM Press. 14 pages.
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. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
11. Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison Wesley, 2000.