Utilizando contratos JML para optimizar diseños orientado a objetos siguiendo MDA

Marcelo Uva, Mariana Frutos, Ariel Gonzalez, Ariel Arsaute, Marcela Daniele, Paola Martellotto, Fabio Zorzan.

Departamento de Computación, Facultad de Ciencias Exactas, Físico-Químicas y Naturales,

Universidad Nacional de Río Cuarto.

Ruta 36 Km. 601 –CP 5800 - Río Cuarto – Córdoba - Argentina Tel. (0358) 4676235
{uva, mfrutos, agonzales, aarsaute, marcela,Paola, fzorzan}@dc.exa.unrc.edu.ar

Resumen

Model Driven Architecture (MDA) define un proceso de construcción de software basado en la producción y transformación de modelos. En Ingeniería de Software, refactorización es la técnica que reestructura código de una aplicación, alterando su estructura interna sin modificar su comportamiento externo. Por otro lado, Java Modeling Language (JML) es un lenguaje para especificar programas Java, utiliza precondiciones, postcondiciones e invariantes de la lógica de Hoare. Este trabajo plantea una técnica basada en MDA que posibilita la construcción de una herramienta automática que tomará como entrada código de una aplicación Java y realizará optimizaciones en su diseño basándose en reglas de refactoreo. La técnica requiere contar con los contratos JML de los métodos de las clases involucradas. El principal aporte de este trabajo es la utilización de contratos JML para asegurar que el comportamiento de un módulo se mantiene sin cambios, luego de la aplicación de reglas de refactoreo.

Palabras clave: MDA, Refactoring, JML.

Contexto

La línea de investigación presentada en este trabajo se desarrolla en el marco del proyecto "Ingeniería de Software: Automatización de Procesos de Desarrollo de Software", perteneciente a los Proyectos y Programas de Investigación (PPI) de la secretaría de Ciencia y Técnica de la Universidad Nacional de Río Cuarto.

Introducción

Model Driven Architecture (MDA) surge a partir de las tecnologías estandarizadas por la Object Management Group (OMG)[1]. MDA define un proceso de construcción de software basado en la producción y transformación de modelos. Los principios sobre los cuales se fundamenta MDA son: abstracción, automatización y estandarización. El proceso central de MDA es la transformación de modelos. La idea subyacente es producir nuevos modelos a partir de uno dado, de manera tal de preservar propiedades en un modelo abstracto independiente de los cambios producidos en la tecnología. MDA establece cuatro niveles de abstracción: CIM (Computation Independent Model), PIM (Platform Independent Model), PSM (Platform Specific Model), y la aplicación final.

En Ingeniería de Software se define el término *refactorización* ("Refactoring")[2] a la técnica que posibilita reestructurar el código fuente de una aplicación de software, alterando su estructura interna sin modificar su comportamiento externo. De acuerdo a la jerarquía de abstracción establecida por MDA, la *refactorización* define transformaciones sobre el código fuente de una aplicación, es decir, en el nivel menos abstracto de los cuatro.

La refactorización se realiza a menudo como parte del proceso de desarrollo de software: los desarrolladores alternan la inserción de nuevas funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y claridad. El proceso de

refactorización consiste en la aplicación de una serie de reglas de transformación que preservan comportamiento. Cada transformación puede producir una reestructuración significativa. Posterior a la aplicación de cada regla, el diseñador/programador provee una batería de casos de prueba para "asegurar" que el comportamiento no es alterado. Por otro lado, Java Modeling Language (JML) [3] es un lenguaje de especificación para programas Java, que utiliza precondiciones, postcondiciones e invariantes de la lógica de Hoare, siguiendo el paradigma de diseño por contrato [4]. JML provee una semántica formal para describir el comportamiento de un módulo de Java, dejando de lado todas las posibles ambigüedades con respecto a las intenciones iniciales del diseñador.

Refactorización de modelos

La refactorización ("Refactoring") es una técnica de ingeniería de software que permite reestructurar código fuente, alterando su estructura interna sin modificar su comportamiento externo. En MDA, la refactorización establece las reglas que permiten el mapeo de un modelo en otro dentro de un mismo nivel de abstracción.

Martin Fowler [2] define refactorización al área que se especializa en el análisis y el diseño orientado a objetos, entre otros. La refactorización define cambios estructurales internos en el software para mejorar su comprensión con un costo inferior al de modificar el comportamiento observable del sistema. Este proceso es logrado mediante la aplicación de un conjunto de reglas de refactoreo.

El último punto de las ventajas citadas por Fowler, establece implícitamente al proceso de refactorización como un procedimiento de optimización guiado por heurísticas. Es por ello que para este trabajo se ha acotado el amplio conjunto de reglas definidas en [2] a un subconjunto posible de automatizar. Es decir, la corrección de la aplicación de las reglas ya no dependerá de las habilidades de un desarrollador particular.

La refactorización forma parte del proceso de fabricación de un software: los desarrolladores alternan la inserción de nuevas funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y su claridad. Los casos de prueba deben ser superados por el módulo, antes y después de la aplicación de cada regla de refactorización. Esto provee una "cierta" seguridad acerca de la no alteración del comportamiento del código, aunque "el testing de los programas puede ser muy efectivo para mostrar la presencia de errores, sin embargo resulta inadecuado para mostrar su ausencia" Edsger Dijkstra.

En este trabajo, se propone una alternativa basada en contratos JML para asegurar que los componentes involucrados no modifiquen su comportamiento.

Líneas de investigación y desarrollo

En la literatura existente sobre el tema se pueden encontrar diferentes mecanismos orientados a evolucionar o mejorar diseños aplicando técnicas de refactoreo. Opdyke [5] presentó un conjunto de transformaciones básicas para poder agregar, eliminar y mover elementos entre clases, con la finalidad de optimizar diseños de aplicaciones desarrolladas en C++. Para ello definió una serie de condiciones iniciales para asegurar uno de los principios fundamentales de toda técnica de refactoreo, el mantenimiento del comportamiento del programa.

En este trabajo se plantea una técnica fundada en los principios de la filosofía MDA a partir de la cual se sientan las bases para la construcción de una herramienta automática, que tomará como entrada el código fuente de una aplicación orientada a objetos implementada en Java y realizará una serie de optimizaciones en su diseño aplicando reglas de refactoreo. Las reglas de transformación establecidas en este trabajo son definidas a nivel PSM dentro de la jerarquía MDA, ya que se ha trabajado principalmente con diagramas de clases en UML [6] implementados en el lenguaje Java. La técnica requiere los contratos JML de cada uno de los métodos involucrados. Esto permite contar con la semántica formal de cada componente, eliminando todo tipo de ambigüedad con respecto a las intenciones iniciales del diseñador.

En la figura 1 se presenta la arquitectura genérica de la técnica descripta en este trabajo y sobre la cual los autores están desarrollando un prototipo.

La arquitectura genérica esta compuesta por un conjunto de módulos independientes. Cada módulo posee una interfaz bien definida. El mecanismo toma como entrada un programa Java anotado con sus contratos JML. Para la generación del modelo XMI equivalente, se utiliza la herramienta ArgoUML[7], que permiten realizar ingeniería inversa y luego exportar el diagrama de clases generado a una especificación XMI o XML Metadata Interchange (XML de Intercambio de Metadatos) [8].

Posteriormente, el módulo "Motor de la transformación" en conjunto con el módulo "Reglas de Transformación" interactúan buscando en el modelo XMI algún patrón para la aplicación de una de las reglas de refactorización. Luego, de la aplicación de cada regla, los contratos JML deberán seguir verificándose. Una vez finalizado el proceso de transformación se producirá un nuevo modelo XMI el cual será exportado finalmente a código fuente Java mediante la aplicación de técnicas de ingeniería directa. Este proceso permite asegurar que la semántica del programa original se mantiene sin cambios, ya que todas las modificaciones realizadas no afectarán el cumplimento de los contratos preexistentes.

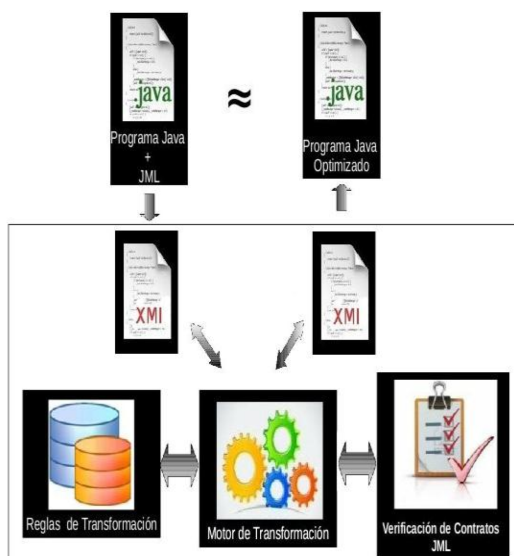

Fig. 1. Arquitectura genérica descripta en la técnica

Este trabajo está basado un conjunto amplio de reglas de refactorización, algunas de ellas, vinculadas a patrones de diseño. Recordemos que un patrón de diseño [9] proporciona una solución efectiva y reusable a un problema recurrente de diseño. Por ejemplo, las reglas de refactorización "Extract Composite" y "Replace Constructor with Factory Method", están relacionados a los patrones "Composite" y "Factory Method". Las mejoras producidas en el diseño luego de la aplicación de estas reglas, están ligadas íntimamente a las ventajas de la aplicación de cada uno de estos patrones.

El módulo "Reglas de transformación" ha sido concebido como un componente que pueda crecer paulatinamente a medida que nuevas reglas se vayan incorporando. Cada una de éstas deberá establecer el patrón de búsqueda a detectar y las condiciones iniciales y finales para su aplicación. Se debe tener en cuenta, que no todas las reglas de refactorización son posibles de automatizar. Un gran número de reglas de refactoreo plantean heurísticas para la mejora de un diseño, cuya aplicación correcta sólo dependerá de las habilidades del diseñador/programador. Es por ello que para la incorporación de nuevas reglas, es necesario un estudio profundo acerca de las condiciones previas a la aplicación de la misma y la factibilidad o no para su automatización.

A modo de ejemplo se presenta la regla *"Extract Composite"* [10].

Regla Extract Composite

La regla *Extract Composite* establece una refactorización combinando la regla anterior y el patrón de diseño Composite.

En aquellos modelos en donde los objetos instanciados de una subclase están compuestos o relacionados con objetos de la misma clase, existe alta probabilidad de que se encuentre duplicación de código. Esta regla plantea un escenario particular en donde se produce la duplicación de métodos (o bien duplicación del "comportamiento" de métodos).

No es necesario que los nombres de los métodos sean iguales, sí los perfiles deberían coincidir (o al menos uno de ellos debería estar incluido en el otro).

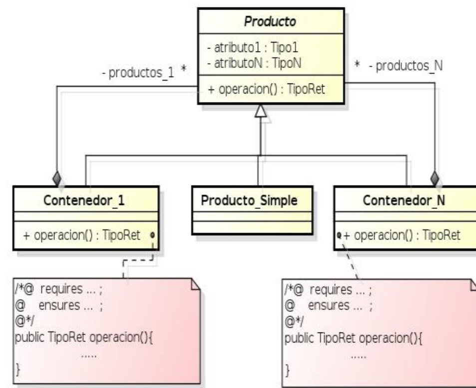

Fig. 2. Diseño genérico previo a la aplicación de la regla Extract Composite

En figura nro. 2 presentamos la estructura genérica de la regla. La aplicación de la regla deberá inicialmente buscar aquellos métodos u operaciones en las subclases "hermanas" que posean la misma signatura, o bien que una incluya a la del otro método. El caso más común es encontrar métodos con el mismo nombre y signatura.

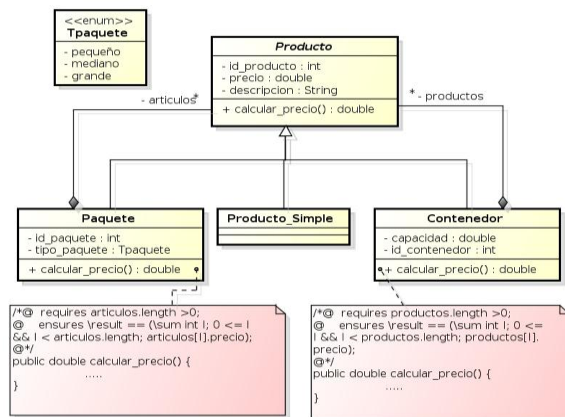

Fig. 3. Diseño genérico posterior a la aplicación de la regla Extract Composite

Luego, se aplica un algoritmo de sustitución para equiparar nombres de variables en ambos contratos JML. En la figura 2, los roles *productos_1* y *productos_N,* se reemplaza en la figura 3 por el rol *productos.* Una vez culminado el proceso de sustitución se verifican si los contratos de las operaciones iniciales se implican mutuamente.

Bastará con que uno de los dos contratos implique al otro, como para reemplazar el contrato genérico por el más débil. Finalmente, se creará una superclase intermedia (en la Figura 3 "*Producto_Compuesto*") que contendrá el método en común con los contratos (previa sustitución) del método con el contrato más débil. En el caso de que los contratos se impliquen mutuamente, se podrá optar por cualquiera de las dos (previas sustituciones).

Resultados y Objetivos

El cambio continuo de los requerimientos, la incorporación de nuevas funcionalidades, la minimización de los tiempos de respuesta junto con la mejora y aprovechamiento de los recursos son características comunes presentes en todo proyecto de desarrollo de software.

Los sistemas actuales requieren de la ejecución de actividades orientadas a elevar los niveles de calidad del software. Dentro de éstas se encuentran las tareas de refactorización que permiten la reorganización y reestructuración de entidades/funcionalidades adaptándolas a los nuevos requerimientos y diseños arquitecturales. Todas estas tareas deben asegurar el mantenimiento del comportamiento original. En este trabajo se ha presentado una técnica para optimizar diseños de aplicaciones Java de manera automática utilizando reglas de refactoreo. La técnica está basada en la filosofía MDA. Se establecen transformaciones a nivel PIM (entre diagramas de clases) y a nivel PSM (entre aplicaciones java concretas). Uno de los aportes de este trabajo es el uso de contratos JML para asegurar que el comportamiento de los módulos se mantiene igual luego del proceso automático de refactorización.

La arquitectura genérica propuesta está conformada por módulos independientes con interfaces bien definidas. El módulo "Reglas de Transformación" almacena las reglas de transformación. Cada una de ellas establece un patrón específico para su correcta aplicación. La incorporación de nuevas reglas se realiza de manera incremental logrando automatizar poco a poco el proceso de optimización. A medida que el módulo

aumente su tamaño (cantidad de reglas) se obtendrán mejores modelos y en consecuencia, mejores aplicaciones.

Actualmente los autores de este trabajo están desarrollando un prototipo de la herramienta que permite la automatización parcial de la propuesta presentada en este trabajo. Para ello se han utilizado las herramientas ArgoUML[7] y ESC/Java2[11].

Formación de Recursos Humanos

Los temas abordados en esta línea de investigación brindan un fuerte aporte al proceso de perfeccionamiento continuo de los autores de este trabajo, que se desempeñan como docentes de las carreras de computación que se dictan en la Universidad Nacional de Río Cuarto y participan en asignaturas relacionadas a dichos temas.

Referencias

[1] Miller, J., Mukerji, J., MDA Guide Version 1.0.1 Document number omg/2003-06-01, http://www.omg.com/mda, 2003.

[2] Fowler, M. Refactoring Imporving The Design of Existing Code. Addison Wesley Longman, Inc. 1999.

[3] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), Behavioral Specifications of Businesses and Systems, chapter 12, pages 175-188. Copyright Kluwer, 1999

[4] Meyer, Bertrand: *Applying "Design by Contract"*, in Computer (IEEE), pp. 40–51. 1992.

[5] Opdyke, W. Refactoring Object-Oriented Frameworks. Tesis doctoral, Universidad de Illinois, Urbana-Champaign.1992.

[6] Grady Booch, Ivar Jacobson & Jim Rumbaugh (2000) OMG Unified Modeling Language Specification. First Edition: March 2000.

[7] ArgoUML. Open Source Software Engineering Tools. argouml.tigris.org Tigris.org.

[8] XML Metadata Interchange Specification OMG Formally Released Versions of XMI. Version 2.0.1 formal/05. http://www.omg.org/spec/XMI/ISO/19503/PDF/

[9] Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. *Design Patterns*: Elements of Reusable Object-Oriented Software. Addison-Wesley, *1995*.

[10] Joshua Kerievsky. Refactoring to Patterns. Addison-Wesley Professional. Part of the Addison-Wesley Signature Series (Fowler) series. 2005.

[11] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik PollBeyond Assertions: Advanced Specification and Verification with JML and ESC/Java2 . http://www.eecs.ucf.edu/~leavens/JML/fmco.pdf