Propuestas de Refactorización POA

Sandra Casas[1], Angélica Zuñiga[1], Claudia Marcos[2] y Eugenia Marquez[1]

[1]Universidad Nacional de la Patagonia Austral.
Lisandro de la Torre 1070. CP 9400. Río Gallegos, Santa Cruz, Argentina
Tel/Fax: +54-2966-442313/17.
E-mail: {lis, azuniga}@uarg.unpa.edu.ar

[2]ISISTAN Research Institute. Facultad de Ciencias Exactas. UNICEN
Paraje Arroyo Seco. CP 7000. Tandil. Buenos Aires. Argentina
Tel/Fax: + 54–2293–440362/3.
E-mail: cmarcos@exa.unicen.edu.ar

Resumen
La refactorización aspectual propone extraer del código funcional el código diseminado y duplicado para transformarlo en código orientado a aspectos. De esta forma el código refactorizado resulta más reutilizable, mantenible y evolucionable. El catálogo de refactorización de Monteiro y Fernández ha sido propuesto para refactorizar código OO-Java a código OA-AspectJ. Este catálogo de manera sencilla identifica situaciones potenciales de mejorar y un método (refactorización) asociado para hacerlo. El presente artículo presenta las dos líneas de trabajo (proyecto 29/A200) que al respecto se están desarrollando en forma conjunta entre investigadoras de la UNPA y el ISISTAN.

1. Introducción

La Programación Orientada a Aspectos (POA) [1] es un nuevo paradigma para el desarrollo de software que apunta a incrementar la compresibilidad, adaptabilidad y reusabilidad mediante la introducción de una nueva unidad modular, llamada "aspecto", para la especificación de "crosscutting concerns". De esta forma, el código relacionado a requisitos no funcionales transversales se especifica en forma separada y aislada, evitando que se mezcle y duplique por las restantes unidades modulares funcionales. Los lenguajes POA extienden lenguajes convencionales e incorporan nuevos mecanismos y abstracciones para dar soporte al enfoque. Así un aspecto se compone de construcciones sintácticas y semánticas específicas como los puntos de cortes, puntos de unión, avisos, introducciones, etc. Un proceso denominado "weaver" compone los aspectos con las unidades funcionales, en tiempo de compilación o ejecución. Una referencia clásica suele ser el lenguaje POA AspectJ [2]. Esta herramienta extiende Java, y actualmente se considera la herramienta más popular y difundida en la comunidad POA.

La refactorización [3][4][5] es una técnica de Ingeniería de Software para reestructurar el código fuente, alterando su estructura interna sin cambiar su comportamiento externo. La refactorización permite mejorar el diseño del código, para hacerlo más reusable y flexible para subsecuentes modificaciones semánticas. Inicialmente la refactorización fue una técnica propuesta para el paradigma OO, sobre los lenguajes Smalltalk y Java. Sus precursores impulsaron un proceso disciplinado mediante un catálogo de métodos habituales de refactorización [3]. Un método de refactorización consiste en una descripción de cómo aplicar el método e indicaciones sobre cuándo debería (o no debería) aplicarse. Un ejemplo típico de refactorización es el renombramiento de una variable. Un ejemplo complejo de refactorización es la aplicación de un patrón de diseño. Más tarde aparecieron catálogos de refactorización para otros lenguajes como C [6], Prolog [7], Haskell [8], PHP [9], Lisp [10], Perl [11], etc. En este sentido el proceso de refactorización trasciende y resulta conveniente independientemente del lenguaje de programación y paradigma subyacente. Por ello la refactorización se considera un concepto tan importante que ha sido identificado como una de las innovaciones más importantes en el campo del software.

Muy recientemente las ideas de POA y refactorización han confluido en lo que se conoce como *refactorización aspectual* ("aspect refactoring"). En principio la refactorización aspectual pretende extraer del código convencional (por ejemplo código OO) el código diseminado y duplicado que

corresponde a los crosscutting concerns y transformarlo en aspectos. En este sentido, los objetivos de la refactorización se mantienen, ya que se busca mejorar la estructura y diseño del código alterando su estructura interna sin modificar su comportamiento. Pero existe una diferencia de mapeo considerable, ya que el código origen pertenece a un paradigma y lenguaje diferente al código destino. Por ejemplo, en la refactorización aspectual, el código OO-Java se refactoriza a código OA-AspectJ

Este trabajo presenta las líneas de investigación y trabajo abordadas en el proyecto de investigación referente a POA que desarrollan en forma conjunta investigadoras de la Universidad Nacional de la Patagonia Austral (UARG) y la Universidad Nacional del Centro (ISISTAN).

2. Catálogos de Refactorización

Inicialmente Beck y Fowler generaron un catálogo de 72 refactorizaciones basándose en la identificación de 22 "code smells". Un code smell es un síntoma que sugiere que algo en el código no esta bien. Por ejemplo, un método largo es un clásico code smell, ya que es más difícil de entender, mantener y modificar, las refactorizaciones propuestas para solucionar este problema son: (110) Extract Method, (120) Replace Temp with Query, (295) Introduce Parameter Object, (135) Replace method to method object y (238) Decompose Conditional.

De manera similar, Monteiro y Fernández [12][13] proponen un catálogo de refactorización aspectual basados en la identificación de code smells. En este sentido los autores primero realizan una revisión de los smells orientados a objetos para corroborar si son aplicables para refactorizar aspectos. Concluyen que el smell Divergent Change ocasiona código mezclado y los smells Shotgun Surgery y Solution Sprawl generan código duplicado. La refactorización que recomiendan para estos casos es Extract Feature into Aspect [13].

También se analizan smells que son propios del paradigma orientado a aspectos. El primero de estos casos es Double Personality el cual aborda el problema de aquellas clases que cumplen múltiples roles. Los autores proponen distintas refactorizaciones que deben ser adoptadas según el caso particular de cada implementación, algunos de ellos son: Replace Implements with Declare Parents, Split Abstract Class into Aspect and Interface y Extract Feature into Aspect todos ellos presentados en [12] y [13].

El segundo smell específico para aspectos es Abstract Classes, el cual propone transformar clases abstractas en interfaces valiéndose de mecanismos que presenta AspectJ como las composiciones. Para realizar esta tarea se propone utilizar los refactorings Split Abstract Class into Aspect and Interface y Change Abstract Class to Interface (ambos de [12]).

Finalmente, el tercer y último smell es Aspect Laziness, el cual se refiere a aquellos aspectos que desligan parte de sus responsabilidades a las clases utilizando mecanismos como inter-type declarations. Ante esto se propone utilizar dos de las refactorizaciones propuestas en [12] Replace Inter-type Field with Aspect Map y Replace Inter-type Method with Aspect Method.

En resumen en [12] y [13] se define un conjunto de 28 refactorizaciones y code smells. Estos son detallados de una forma muy similar a la utilizada por Fowler en [3]. Para cada uno de los refactorizaciones los autores especifican los siguientes puntos utilizando para ello Java como lenguaje orientado a objetos y AspectJ para aspectos: Nombre del refactorización, Situaciones típicas, Acciones recomendadas, Motivación, Mecanismos de reestructuración, Ejemplos de código.

3. Objetivos de Investigación

En principio el estudio toma como punto de partida el catálogo de Monteiro y Fernandez [13], con dos propósitos: (i) diseñar e implementar una herramienta que automatice el proceso de refactorización y (ii) adaptar el catálogo a otros contextos lingüísticos diferentes a Java y AspectJ. A continuación se definen con mayor precisión estas líneas de trabajo.

3.1 Automatización de refactoring basado en mecanismos inferenciales de reglas

Un sistema basado en reglas es un sistema que usa reglas para derivar conclusiones de un conjunto de premisas. Una regla es un tipo de instrucción u orden que se aplican en cierta situación. Una regla es una sentencia del estilo "si-entonces" de los lenguajes de programación tradicionales. La parte "si" indica el predicado, o premisas; y la parte "entonces" especifica las acciones, o conclusiones. El dominio de una regla es el juego de toda la información almacenada en la memoria de trabajo, estructurada en forma de hechos. En un programa basado en reglas, se escriben sólo las reglas individuales. La máquina inferencial determina qué reglas se aplican (disparan) en cualquier momento dado y las ejecutan de modo apropiado. Los sistemas basados en reglas resultan ser soluciones declarativas naturales para problemas que involucran diagnóstico, predicción, clasificación, reconocimiento de patrones, configuración, etc., para los cuales las soluciones algorítmicas son menos claras y plausibles.

Entonces usando como estrategia de implementación para la automatización del catálogo de refactorización de Monteiro y Fernández un sistema basado en reglas, el proceso de refactorización se analizaría y descompondría en reglas de la siguiente manera:

Si (en alguna/s clase/s existe el smell X) *Entonces (aplicar la refactorización Y)*

Donde *X* e *Y* están definidas en el catálogo.

Este tipo de esquema declarativo permite centrarse en el problema y en la solución de una manera más fácil y manejable. Los pasos que preliminarmente se definen para el diseño e implementación de la herramienta son:

-*Definición de la Base de Hechos.* En esta etapa se diseñan y representan las estructuras de datos que procesará la máquina de inferencia a través de las reglas. Tres tipos de hechos se identifican a priori: (i) hechos iniciales: surgen del mapeo del código fuente OO mediante pre-procesamiento; (ii) smells candidatos: estos hechos representan los posibles síntomas identificados en el código fuente OO; (iii) refactorizaciones: estos hechos representan la aplicación de las refactorizaciones.

-*Diseño de Base de Reglas.* En este paso se diseñan e implementan el conjunto de reglas del sistema. Las reglas se clasifican en 2 grupos principales: reglas que identifican "smell candidatos" y las reglas que aplican las refactorizaciones.

- *Salida XML:* el conjunto de refactorizaciones aplicadas deben ser representadas en un formato que permita posteriores procesamientos. Para tal propósito se escoge la representación en formato XML.

- *Transformación:* la salida del sistema de reglas representada en XML se utiliza para generar en forma automática o semiautomática el código fuente refactorizado.

3.2 Refactorización aspectual de código PHP

PHP es un lenguaje de script utilizado para el desarrollo de aplicaciones Web de pequeña y mediana complejidad. En lo últimos años PHP ha alcanzando importantes niveles de uso (más de 1 millón de Webs lo utilizan). Las primeras 2 versiones de PHP, PHP 3 y PHP 4, lograron constituir una plataforma potente y estable para la programación de páginas dinámicas en la parte servidor, que es interpretado los clientes navegadores. Estas versiones han servido para instalar a PHP en la comunidad Web, de manera contundente. La versión PHP 5 da soporte más adecuado y completo a la programación orientada a objetos (POO).

phpAspect [14] es una extensión de PHP 5 que da soporte de implementación a la POA. El compilador phpaspect teje los aspectos transversales de la aplicación en el código fuente de PHP. El proceso de tejido es estático y se basa en análisis de Lex & Yacc – XML y XSLT. El código fuente PHP puede ser ejecutado con cualquier intérprete de PHP 5.

En la Figura 1, se presenta gráficamente el proceso de tejido de phpAspect. La lógica de negocios y la funcionalidad transversal se codifican por separado en clases y aspectos. Estas unidades son analizadas y transformadas en árboles sintácticos en XML. Por último una serie de procesamientos XSLT realiza la composición (tejido) de los aspectos y clases y genera la aplicación final.

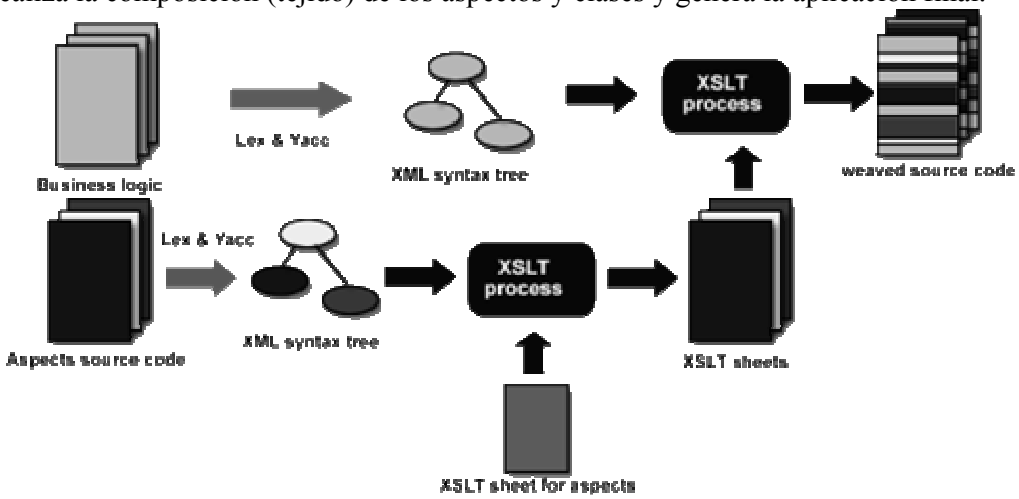

Figura 1. Proceso de Tejido de phpAspect

phpAspect proporciona una semántica y sintaxis muy similar a AspectJ. Soporta todos los join-points (llamadas o ejecución de métodos y constructores, acceso o modificación de campos, excepciones, etc.), cortes (call, execution, set, get, etc.) y avisos (before, after, around) tradicionales. A modo de ejemplo, en la Figura 2 se presenta el aspecto *TraceOrder*. Este aspecto define los pointcuts *logAddItem* y *logTotalAmount*. El primero establece una acción que se ejecutará inmediatamente después que se ejecute el método *addItem* de la clase *Order*; y el segundo establece una acción que se ejecutará después que se invoque al mismo método.

```
<?php
aspect TraceOrder{
  pointcut logAddItem:exec(public Order::addItem(2));
  pointcut logTotalAmount:call(Order->addItem(2));
  after($quantity, $reference): logAddItem{
    printf("%d %s added to the cart\n", $quantity, $reference;)
  }
  after(): logTotalAmount{
    printf("Total amount of the cart : %.2f euros\n",
    $thisJoinPoint->getObject()->getAmount());
  }
}
?>
```

Figura 2. Aspecto TraceOrder en phpAspect

En función del masivo uso de PHP se considera importante definir mecanismos de refactorización aspectual para convertir el código PHP correspondiente a requerimientos transversales en código phpAspect. El propósito de este trabajo se centra en la definición de un catálogo de refactorización de aspectos de código PHP 5 a phpAspect. Se toma como punto de referencia el catálogo de Monteiro. Los pasos que inicialmente se trazan son:

1. Definir diferencias entre Java y PHP: analizar y comparar sintáctica y semánticamente los lenguajes bases.

2. Definir diferencias entre AspectJ y phpAspect: analizar y comparar sintáctica y semánticamente los lenguajes POA.

3. Identificar code smells comunes entre Java y PHP: utilizando el catálogo de Monteiro y teniendo en cuenta las diferencias y similitudes halladas en los pasos previos se perfila un paralelismo entre los code smells.
4. Redefinir las refactorización de aspectos PHP - phpAspect: de manera similar que en el paso anterior se traza un paralelismo entre refactorizaciones.
5. Identificar code smells propios de PHP: se analizan las características particulares de PHP que sugieran la presencia de code smells.
6. Definir refactorizaciones específicas para PHP y phpAspect: se diseñan las refactorizaciones necesarias para extraer los aspectos sugeridos de los code smells propios de PHP.

4. Conclusiones.

En este artículo se han presentado en forma concreta dos líneas de trabajo e investigación referentes a refactorización aspectual. Ambos trabajos toman como punto de partida el catálogo de refactorización de aspectos de Monteiro y Fernández. El primero de estos trabajos se refiere concretamente al diseño e implementación de un sistema basado en reglas e inferencia que automatice el proceso de refactorización. El segundo trabajo, de índole más teórico, se basa en la redefinición del catálogo para PHP 5 y phpAspect.

El presente trabajo fue parcialmente financiado por la Universidad Nacional de la Patagonia Austral, Santa Cruz, Argentina y el proyecto PICT 32079 (ANPCYT).

Referencias

[1] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J. "Aspect-Oriented Programming". In Proceedings of ECOOP. 1997.
[2] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. "An Overview of AspectJ". ECOOP 2001.
[3] Fowler M. "Refactoring: Improving the Design of Existing Code". Addison Wesley. 1999.
[4] Beck K. "Extreme Programming Explained: Embrace Change". Addison-Wesley 2000.
[5] Opdyke W. "Refactoring Object-Oriented Framework" Ph.D Thesis. University of Illionois at Urbana-Champaign 1993.
[6] Garrido A. "Software Refactoring Applied to C Programming Language". Tesis de Master. Univ. of Illinois at Urbana-Champaign. 2000.
[7] Schrijvers T. y Serebrenik A. "Improving Prolog programs: refactoring for Prolog". 20th International Conference, ICLP 2004, Proc. vol 3132, LNCS, pp. 58-72, 2004
[8] Ul Huinqing Li. "Refactoring Haskell Programs" – PhD thesis The University of Kent - 2006
[9] Refactoring-PHP-Code http://devzone.zend.com/article/2514-
[10] Leitdo A.M. "A formal pattern language for refactoring of Lisp programs" Software Maintenance and Reengineering, 2002. Proc. Sixth European Conference 2002 pp186 – 192
[11] Nagler R. "Extreme Programming in Perl". 2005, http://www.extremeperl.org/bk/home
[12] Monteiro M.P. "Catalogue of refactorings for AspectJ". Technical Report UM-DI-GECSD-200401, Universidade do Minho, 2004.
[13] Monteiro M. P. y Fernandes J. "Towards a catalog of aspect oriented refactorings". In Proc. of the 4th International Conference on Aspect-Oriented Software Development (AOSD), pp 111–122. ACM Press, March 2005.
[14] Homepage of phpAspect: http://phpaspect.org/wiki/doku.php