Exploración de Reglas de Inferencia para Automatizar la Refactorización Aspectual

Sandra Casas[1] and Claudia Marcos[2]

[1]Unidad Académica Río Gallegos, Universidad Nacional de la Patagonia
Lis. De la Torre 1060 – CP 9400 - Río Gallegos - Argentina

[2]Instituto de Sistemas de Tandil, Universidad Nacional del Centro
Pje. Arrojo Seco - CP 7000 - Tandil, Argentina

lis@uarg.unpa.edu.ar, cmarcos@exa.unicem.ar

Resumen. Este trabajo explora la factibilidad de automatizar el catalogo de refactorización aspectual propuesto por Monteiro y Fernández utilizando como estrategia un sistema de reglas de inferencia. En particular se presentan el análisis y diseño de las estructuras de datos y reglas para el grupo del catalogo cuyo objetivo es extraer crosscutting concerns. El prototipo Z-ARIR soporta el proceso planteado.

Abstract. This work explores the feasibility of automating the aspect refactoring catalogue proposed by Monteiro and Fernández using as strategy an inference rules system. In particular the analysis and design of the structures of data and rules are presented to the group whose objective is to extract crosscutting concerns. The prototype Z-ARIR supports the outlined process.

1 Introducción

La Refactorización (Fowler, 1999) (Opdyke, 1993), ha ganado popularidad debido a su valor práctico ya que ayuda a mejorar y evolucionar el código. La Programación Orientada a Aspectos (AOP) (Kiczales et al., 1997) ha recibido creciente atención debido a su poder para encapsular los crosscutting concerns de un sistema a través del uso de nuevas unidades de modularización llamadas aspectos. La Refactorización Orientada a Aspectos (AOR) sinérgicamente combina estas dos técnicas para rafactorizar los elementos crosscutting concerns.

Monteiro y Fernández (Monteiro, 2004) (Monteiro & Fernandez, 2005) proponen un catalogo de refactorización aspectual basado en la identificación de code smells. Este catalogo es específico para Java y AspectJ (Kiczales et al., 2001). En un sistema legado formado por decenas o cientos de clases, la refactorización aspectual requiere analizar cada una en particular. En este contexto la refactorización aspectual manual parece poco probable de ser aplicada en forma exitosa. Esta tarea requiere del soporte y asistencia de herramientas que automaticen en forma parcial o total el proceso, para que la refactorización aspectual sea "posible".

Este trabajo explora la factibilidad de automatizar el catalogo de Monteiro y Fernández utilizando como estrategia una sistema de reglas de inferencia. Una regla es una sentencia del estilo "si-entonces", la parte "si" indica el predicado y la parte "entonces" especifica las acciones. En una aplicación basada en reglas, se escriben sólo las reglas

individuales. Una máquina de inferencia determina qué reglas se aplican (disparan) en cualquier momento dado y las ejecuta de modo apropiado. Este tipo de sistemas 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. Desde este enfoque, 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 catalogo.

A partir de este enfoque, este trabajo explora en particular el grupo de refactorizaciones propuestas para la extracción de crosscutting concerns del catalogo de Monteiro y Fernández. Este articulo se estructura de la siguiente manera: en la Sección dos brevemente se describe el catalogo AOR objeto de implementación, en la Sección 3 se describe la propuesta; en la Sección 4 se presentan algunos trabajos relacionados y finalmente en la Sección 5 se concluye.

2. Catálogo de Refactorización Aspectual

Monteiro y Fernández [12][13] proponen un catalogo de refactorización aspectual basados en la identificación de code smells. Los métodos de refactorización son detallados de una forma muy similar a la utilizada en (Fowler, 1999). Para cada una de las refactorizaciones los autores especifican los siguientes puntos utilizando para ello Java como lenguaje OO y AspectJ para aspectos: Nombre de la refactorización, Situaciones típicas, Acciones recomendadas, Motivación, Mecanismos de reestructuración, Ejemplos de código. El catalogo se resume en 28 refactorizaciones que se clasifican en 4 grupos principales:

- Extracción de crosscutting concerns: el objetivo es extraer distintas características del código OO para encapsularlas en un aspecto.

- Estructura interna de los aspectos: el objetivo es mejorar la estructura interna de aquellos aspectos que surgen de aplicar las refactorizaciones del grupo anterior.

- Generalizaciones de aspectos: el objetivo es aplicar transformaciones que se realizan para las jerarquías de aspectos.

- Código legado: el objetivo es simplificar la extracción de concerns en constructores que son parte de interfaces.

El catalogo resulta ser una valiosa contribución pero de complicada aplicación manual. En un sistema legado, la puesta en práctica real de la refactorización aspectual implica la inspección y análisis de todas las clases que lo componen. Esto requiere de herramientas que de manera automática o semi-automática asistan al desarrollador y faciliten la tarea. Este trabajo propone un estrategia de implementación para el primer grupo del catalogo, es decir para los siguientes métodos de refactorización: Change Abstract Class to Interface (4), Encapsulate Implements with Declare Parents (21), Extract Feature into Aspect(5), Extract Fragment into Advice(9), Extract Inner Class to Standalone(13), Inline Class within Aspect (15), Inline Interface within Aspect (16), Move Field from Class to Inter-type (17), Move Method from Class to Inter-type (19), Split Abstract Class between Aspect and Interface (21).

3. Sistema de Reglas de Inferencias para la Refactorización Aspectual

Los sistemas basados en reglas representan el conocimiento de resolución de problema como reglas *if… then…*que representa el par *condición-acción.* Las premisas de la regla en la parte *if*, corresponden a la condición, y la conclusión en la parte *then*, corresponden a la acción. Cuando la condición es satisfecha, el sistema toma la acción, haciendo la conclusión verdadera. Los datos son guardados en la memoria de trabajo, en unidades denominadas hechos. La máquina de inferencia implementa el ciclo de reconocimiento-acción como en un sistema de producción; este control puede ser dirigido por datos o dirigido por objetivos. El desarrollo de un sistema de reglas es un proceso exploratorio, iterativo e interactivo, para lo cual existen herramientas de programación específicas, como CLIPS (CLIPS, 2008) y JESS (Friedman-Hill, 2003). Esta última se ha utilizado en este caso.

3.1 Los pasos del Proceso

El enfoque completo que da soporte de implementación a la refactorización aspectual se ilustra en la Figura 1, este se compone de 4 pasos básicos.

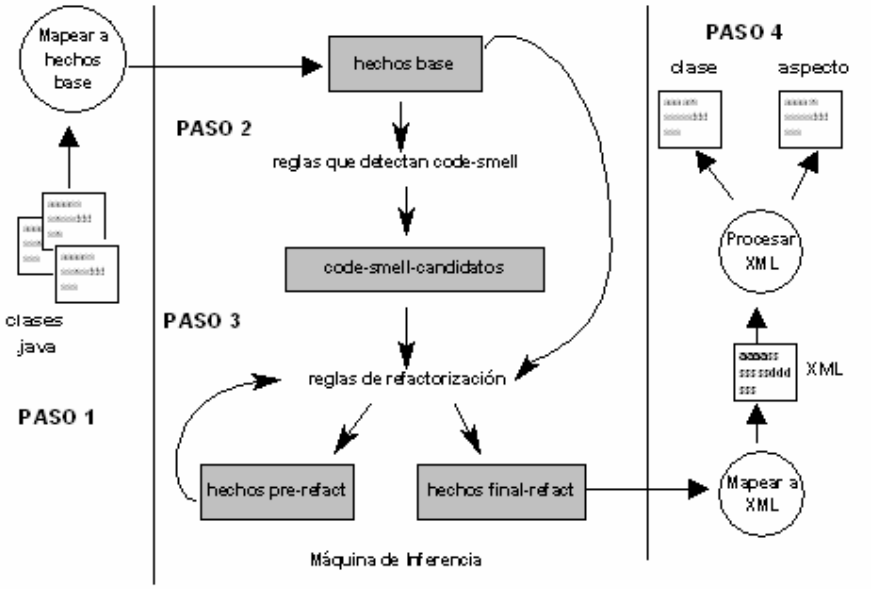

Figura 1: Pasos del Proceso de Refactorización Aspectual

Paso 1: Esta instancia consiste en analizar el texto fuente de las clases codificadas en Java para generar por cada una el conjunto de hechos que representan algunas de sus características. Este proceso es automático. Los hechos obtenidos en esta fase se denominan hechos base.

Paso 2: El conjunto de hechos base que corresponden a una clase se ingresa a la memoria de trabajo de la máquina de inferencia de uno por vez. Automáticamente se dispararán las reglas que detectan code smell candidatos. Identificado un code smell candidato, se informa al desarrollador quien debe confirmar si se continúa con la refactorización de dicho code smell o se ignora. Este paso es semi-automático (requiere intervención mínima del desarrollador). La identificación de un code-smell candidato genera uno o más hechos intermedios, que se han clasificado como hechos "code-smell".

Paso 3: Los hechos code-smell generados por la fase anterior hacen disparar las reglas de refactorización (por lo general más de una). Una regla de refactorización analiza los

hechos base y genera hechos intermedios y/o finales. Estos se han clasificado como hechos pre-refact y final-refact, respectivamente. Los hechos pre-refact requieren posteriores procesamiento, es decir otras reglas de refactorización los procesarán hasta generar hechos final-refact. El proceso de inferencia termina cuando no hay más hechos pre-refact para analizar. Un hecho final-refact indica una acción de refactorización muy concreta sobre elementos específicos de programa, por ejemplo el hecho final-refact *(remove_method (id_method sendMessage) (to_class Account))* indica que se debe eliminar el método *sendMessage* de la clase *Account*.

Paso 4: Los hechos final-refact son extraídos de la memoria de trabajo y mapeados a XML, como acciones. Luego un proceso aplica las refactorizaciones según las acciones XML al código fuente. El ciclo se cierra luego que el desarrollador realiza las pruebas.

3.2 Los Hechos Base

Cada clase es mapeada a un conjunto de hechos base. Este proceso es totalmente automático y consiste en un análisis léxico y sintáctico del código fuente. El mapeo responde estrictamente a las necesidades de refactorización y no a una representación simbólica de la clase. Las plantillas (tipo de hecho y atributos) más usadas que definen el formato son las siguientes:

```
d_class: id_class, class_type, class_access, class_scope
d_method: id_method,  id_class, method_access, slot method_type, method_return
d_field: id_field, id_class, field_type, field_access
d_import: id_class, import_package
args: id_arg, type_arg, id_method, id_class
return: id_method, id_class, slot line
message: id_method, id_class, id_object, message_method, message_class, line
assignation: id_method, id_class, left_exp, right_exp, line
d_extends: id_sub_class, id_super_class
d_implements: id_class, id_interface
throws: id_class_exception, id_method, id_class
d_local: id_var, type_var, id_method, id_class
exception: id_method,  id_class, line_try, line_catch, class_exception
structure_control: id_method, id_class, structure, line
throw: id_method, id_class, exception, line
```

La Figura 2 ejemplifica el mapeo de la clase *Item* a hechos base utilizando el formato de implementación JESS (Friedman-Hill, 2003).

```
                    public class Item {
                      private String id;
                      private Logging logger=new Logging();
                      public String getID() {
                       logger.logp("Item","getID", "Entering");
                       return id;  }
                    }
(d_class (id_class Item)(class_type concrete)(class_access public)
                                                (class_scope standalone))
(d_atributte (id_atributte id) (id_class Item)(atributte_type String)
                                               (attribute_access private))
(d_atributte (id_atributte logger) (id_class Item)(atributte_type
                                        Logging (attribute_access public))
(d_method (id_method getID) (id_class Item) (method_access public)
                           (method_type concrete) (method_return String))
(message (id_method getId)(id_class Item)(id_object logger)(message_class
                                    Logging) (message-method logp) (line 1))
(return (id_method getId) (id_class Item) (line 2))
```

Figura 2: Mapeo a hechos base de la clase Item.

3.3 Reglas que identifican Code-Smell Candidatos

Los code smell relacionados al grupo extraer crosscuting concerns se han simplificado en tres características del código: código diseminado y mezclado, clases abstractas y clases internas. Para el caso del código diseminado y mezclado se ha considerado que el mismo esta representado por los mensajes enviados a instancias de clases que representan crosscutting concerns. Esto implica buscar el mensaje repetido y analizar el tipo de la instancia que despacha el mensaje. La inspección ha requerido tres reglas que se disparan en forma excluyente: *ts1*, *ts2* y *ts3* y cuyas premisas analizan particularmente los hechos base que representan los mensajes. La regla *ts1* identifica cuando el mensaje esta repetido en una misma clase; la regla *ts2* identifica cuando el mensaje esta diseminado en distintas clases y *ts3* identifica cuando una clase contiene un mensaje que ya ha sido identificado como code smell diseminado. Las reglas *ac* e *ic* determinan la presencia de clases abstractas y clases internas respectivamente, éstas analizan en particular el hecho base *d_class*. En la Tabla 1 se especifican las condiciones y acciones principales de las cinco reglas. Las premisas de las mismas analizan los hechos bases, las 5 reglas son excluyentes. En este caso, cada regla genera hechos intermedios denominados *t_s_codesmell*, *a_c_codesmell* e *i_c_codesmell*. Los hechos *t_s_codesmell* describen el método y clase diseminado y en que clase están dispersos. Los hechos *a_c_codesmell* e *i_c_codesmell* solo describen la clase que es interna o abstracta.

Tabla 1: Reglas que identifican code smell candidatos

Descripción	Regla Pseudo-JESS

Si (el mensajes tsm esta esparcido en la clase in_class) **entonces** tsc.tsm es t_s_codesmell en la clase in_class	`Regla ts1`
	`(message(?in_methodx)(?in_class)(?tsc)(?tsm))`
	`(message(?in_methody)(?in_class)(?tsc)(?tsm))`
	`=>`
	`(t_s_codesmell(?tsc)(?tsm)(?in_class))`

Si (el mensaje tsm esta repetido en las clase X e Y) **entonces** tsc.tsm es t_s_codesmell en las clases X e Y	`Regla ts2`
	`(message(?in_methodx)(?in_classX)(?tsc)(?tsm))`
	`(message(?in_methody)(?in_classY)(?tsc)(?tsm))`
	`=>`
	`(t_s_codesmell(?tsc)(?tsm)(?in_class))`
	`(t_s_codesmell(?tsc)(?tsm)(?in_class))`

Si (el mensaje tsc.tsm es un t_s_codesmell) **entonces** tsc.tsm es t_s_codesmell en la clase Y	`Regla ts3`
	`(message(?in_methodx)(?in_classX)(?tsc)(?tsm))`
	`(t_s_codesmell(?tsc)(?tsm)(?in_classY))`
	`=>`
	`(t_s_codesmell(?tsc)(?tsm)(?in_class))`

Si (la clase X es abstracta) **entonces** X es ac_codesmell	`Regla ac`
	`(d_class (?X) (class_type=abstract))`
	`=>`
	`(a_c_codesmell (?X))`

Si (la clase X es interna) **entonces** X es ic_codesmell	`Regla ic`
	`(d_class (?X) (class_scope=inner))`
	`=>`
	`(i_c_codesmell (?X))`

El grupo de reglas especificadas pueden generar hechos code-smell candidatos que no correspondan a crosscutting concerns y crear una explosión redundante de hechos. Esto se puede evitar y así optimizar este proceso estableciendo una lista (conjunto de hechos) con los nombres de clases que a priori se conoce que corresponden a código funcional.

Luego en las reglas descriptas se debe incluir premisas que excluyan del examen a las clases de la lista.

Cuando algún code-smell candidato es identificado, el sistema informa al desarrollador quien decide si el proceso de refactorización continua o se cancela.

3.4 Reglas de Refactorización

Una regla de refactorización implementa una refactorización del catalogo, específicamente se intenta seguir los lineamientos de las secciones "acción recomendada" y "mecanismos". Una regla de refactorización puede provocar que otras reglas de refactorización se disparen ya que en ciertos casos se requiere de varios análisis para determinar todos los pasos de refactorización necesarios. Esto sucede por ejemplo, cuando se aplica la refactorización *Extract Feature into Aspect (5)*, simplificadamente implica los pasos: crear el aspecto vacío y luego aplicar las refactorizaciones *Move Field from Class to Inter-type (17), Extract Fragment into Advice (9) y Move Method from Class to Inter-type (19).*

Siguiendo este caso, la regla de refactorización *efia* (Figura 3) intenta aplicar la refactorización Extract Feature into Aspect (5). Esta se dispara cuando existe un hecho *t_s_codesmell* y genera 4 hechos: el hecho *create_empty_aspect* que es un hecho final-refact, y los tres hechos pre-refact *move_all_field, move_all_method y create_all_advice*.

Si existe t_s_codesmell **Entonces** Crear un aspecto vacio Mover todos los campos Mover todos los métodos Crear todos los avisos	Regla efia (t_s_codesmell(message_class ?tsc)(in_class ?X)) => (create_empty_aspect (aspectCandidate ?tsc)) (move_all_field (from_class ?X) (to_aspect ?tsc)) (move_all_method (from_class ?X) (to_aspect ?tsc)) (create_all_advice (cross_class ?X) (in_aspect ?tsc))

Figura 3: Regla que aplica Extract Feature into Aspect

La refactorización continúa sobre los hechos pre-refact que ha generado la regla efia y hechos bases, siendo analizados por otras reglas de refactorización. Básicamente cada hecho pre-refact será examinado por una regla diferente. Por ejemplo la regla *moveField* (Figura 4) determina los atributos que corresponden al crosscutting concern y deben ser movidos al aspecto y eliminados de la clase en la cual están diseminados. Esta regla se dispara tantas veces como atributos cumplan la condición. Ciertamente esta regla esta aplicando la refactorización *Move Field from Class to Inter-type (17).*

```
    regla moveField
1     (move_all_field (from_class ?X)(to_aspect ?tsc))
2     (d_field (id_field ?f)(id_class ?X)(field_type ?tsc))
    =>
3     (move_field (id_field ?f)(to_aspect ?tsc))
4     (remove_field (id_field ?f)(from_class ?X))
```

1: hecho pre-refact generado por la regla efia que indica que se deben mover todos los atributos de la clase X al aspecto tsc
2: hecho base que determina si el atributo f declarado en la clase X es del tipo tsc
3: mover el atributo f al aspecto tsc
4: eliminar el atributo f de la clase X

Figura 4: Regla que aplica Move Field from Class to Inter-type.

La regla *moveMethod* hace lo propio con los métodos, primero busca el método en el cual existe un mensaje al crosscutting concern y analiza si este es la única lógica para indicar mover el método como inter-type. La regla *createAdvice* (Figura 5) considera

todos los mensajes enviados a métodos que corresponden al crosscuting concern y genera por cada uno la información necesaria para crear el advice en el aspecto. Esta regla indica además que mensajes deben ser eliminados de la clase en la cual están diseminados y mezclados.

```
     Regla createAdvice
1      (create_all_advice (cross_class ?X) (in_aspect ?tsc))
2      (message(id_method ?m)(id_class ?X)(message_class ?tscm)
                                                  (message_method ?tsc))
       =>
3      (add_advice (in_aspect ?tsc) (to_method ?tscm)(j_p_method ?m)(j_p_class ?X))
4      (remove_message (in_method ?m)(in_class ?X)(message_class ?tsc)
                                                  (message_method ?tscm))
```

1: hecho pre-refact generado por la regla efia que indica que se deben crear todos los advises y pointcut que relacionen al aspecto tsc con la clase X.
2: aparea todos los mensajes contenidos en la clase X a métodos del aspecto tsc.
3: agregar un advise en el aspecto tsc al método tscm relacionado al join-point de la clase X.
4: eliminar de la clase X el mensaje tscm

Figura 5: Regla que aplica Extract Feature into Aspect X

De manera similar existen reglas de refactorización para aplicar sobre los hechos code-smell *ic_codesmell* y *ac_codesmell* y las refactorizaciones: Change Abstract Class to Interface (4), Encapsulate Implements with Declare Parents (21), Extract Inner Class to Standalone (13), Inline Class within Aspect (15), Inline Interface within Aspect (16) y Split Abstract Class between Aspect and Interface (21).

Luego que todos los hechos pre-refact han sido analizados se obtiene un conjunto de hechos final-refact. Estos son extraídos de la memoria de trabajo y mapeados a XML para ser empleados en un procesamiento posterior sobre el código real. En la Figura 6 se presenta un breve ejemplo que ilustra como cada hecho final-refact se convierte en una acción que se mapea en XML. Donde las etiquetas indican el tipo acción (crear, mover, eliminar, agregar) y los elementos (aspecto, clase, atributo, método, etc.).

```
<refactoring>
  <action>
   <create_empty_aspect> Logging </create_empty_aspect>
   <move_field>
     <field> logger </field>
     <to_aspect> Logging </to_aspect>
   </move_field>
   <remove_field>
     <field> logger </field>
     <from_class> Item </from_aspect>
   </remove_field>
   <add_advice>
    <in_aspect> Logging </in_aspect>
    <joint_point> Item.getPrice </joint_point>
    <add_message> logp </add_message>
   </add_advice>
   <remove_message>
     <from_class> Logging </from_class>
     <from_method> Item.getPrice </from_method>
     <message> logp </message>
   </remove_message>
  </action>
</refactoring>
```

Figura 6: Mapeo de Hechos final-refact a XML.

3.5 Z-ARIR: Z Aspect Refactoring Inference Rules

Z-ARIR es un prototipo muy simple cuyo objetivo en proporcionar un entorno amigable para realizar las pruebas de la propuesta. Z-ARIR es en esencia un entorno que permite que se apliquen los pasos del método e integrar la interacción ya sea con el desarrollador como con la máquina de inferencia (JESS). En este ultimo caso, Z-ARIR oculta todos los detalles del sistema de reglas, lo cual facilita la tarea del desarrollador ya que no exige que éste conozca la sintaxis y semántica de JESS. En la Figura 7 se visualiza una ejecución de Z-ARIR. La clase Item se ha ingresado y mapeado automáticamente en hechos base (A), las reglas en memoria que se han disparado (B) y la salida que las mismas han producido: se ha identificado el code-smell T&S (Tangled & Scattered) Logging en la clase Item (C), y se indican las siguientes refactorizaciones: crear aspecto vacío para Logging (D), agregar avisos en Logging a los joint-points Item.toString, Item.getPrice e Item.getID y eliminar los mensajes Logging.logp que se encuentran en los métodos Item.toString, Item.getPrice e Item.getID (E) y mover el atributo logger al aspecto Logging, eliminar el atributo logger de la clase Item (F).

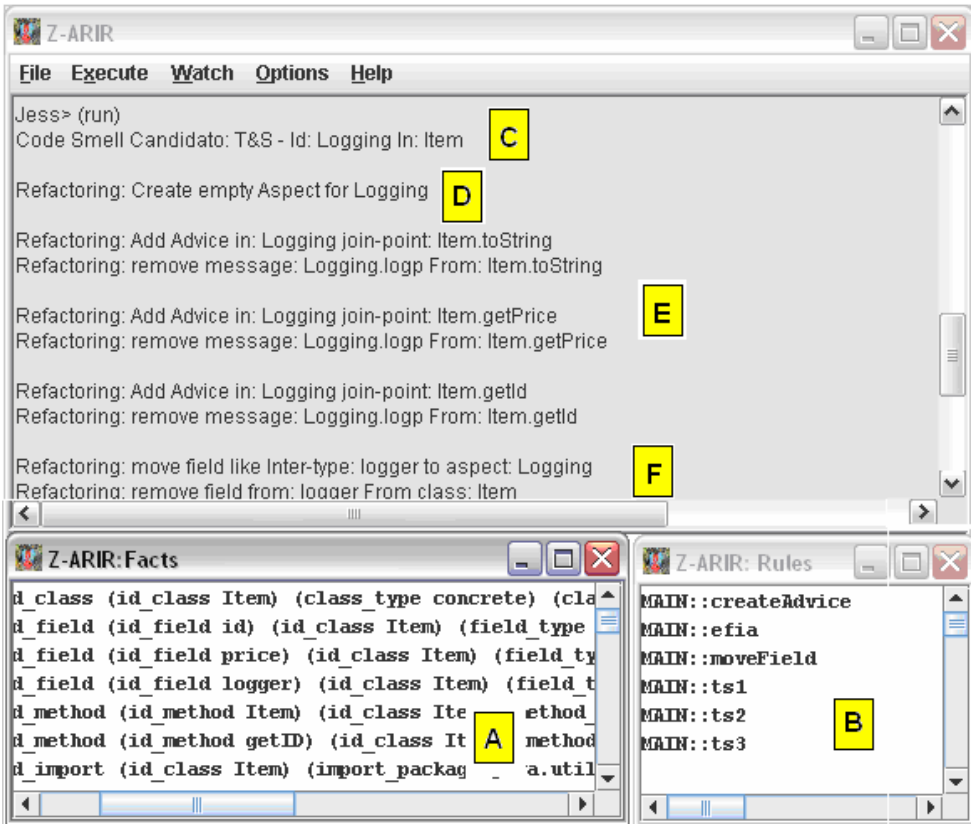

Figura 7: Ejecución de Z-ARIR.

4. Trabajos Relacionados

Diversos trabajos que incluyen propuestas de implementación para la refactorización aspectual se han presentado. Una comparación adecuada sería aquella que mida el nivel de automatización de las herramientas y cantidad de refactorizaciones soportadas, por ejemplo. Pero un estudio de estas características nos resulta poco serio dado el incipiente estado en que se encuentra en general la AOR, que provoca que la mayoría de las herramientas disponibles sean prototipos aún no maduros, consolidados y optimizados.

Sin embargo, interesa plantear alguna diferencia desde la estrategia de implementación usada para dar soporte automático.

AJaTS (Arcoverde et al. 2007) soporta transformación de código en un lenguaje basado en plantillas, similar a AspectJ. Refactorizaciones específicas y transformaciones pueden ser definidas, almacenadas y reusadas. El proceso es automático y proporciona un plug-in para Eclipse. Las clases se mapean a árboles sintáxticos y mediante pattern-matching y el patrón Visitor se aplican las transformaciones. Aunque los autores declaman que se trata de un enfoque declarativo no se hace demasiada especificación al respecto. (Kessler Piveta et al., 2006) plantean algoritmos para detectar automáticamente un conjunto de bad smell code en AspectJ. La propuesta presenta un prototipo como plug-in de Eclipse, este es un enfoque algorítmico mientras que nosotros presentamos un enfoque declarativo. (Binkley et al, 2005) propone la migración de código orientado a objetos a un sistema orientado a aspectos mediante el uso de una herramienta semi-automática de refactorización. Los autores asumen para la realización de esta tarea, que con anterioridad se ha llevado a cabo sobre el código fuente un proceso de minería de aspectos ("aspect mining"), el cual ha identificado y señalado crosscutting concerns que podrían ser "aspectizables". La herramienta itera sobre cuatro pasos fundamentales (descubrimiento, transformación, selección y refactoring). (Hannemann et al., 2005) presenta un método basado en los roles de los crosscutting concerns. Para comprobar que esta técnica funciona los autores han implementado una herramienta semi-automática en la que se experimenta con refactorizaciones basadas en patrones de diseño como los presentados por GoF. AspectJ Refactoring Tool (Iwamoto & Zhao, 2003) es una herramienta que utiliza grafos de dependencia de programa (PDG) y que podría ser utilizada para refactorizar automáticamente los sistemas.

5. Conclusiones

En este trabajo se ha explorado el uso de reglas de inferencia para dar soporte automático al catalogo de refactorización aspectual de Montero y Fernández. En esta primera instancia se han identificado las estructuras de datos (hechos) necesarias para el grupo cuyo objeto es extraer crosscutting concerns de dicho catalogo y se han diseñado e implementado un conjunto de reglas de inferencia que permiten identificar code-smell e indicar la refactorización que se deben aplicar. En este sentido los hechos se han clasificado según su uso en hechos base, hechos code-smell, hechos pre-refact y hechos final-refact. Asimismo se han clasificado las reglas según su propósito en reglas que identifican code-smell candidatos y reglas de refactorización.

Las pruebas preliminares indican que la estrategia adoptada es válida pero requiere de mayor refinamiento y optimizaciones, en las que se esta trabajando. Las pruebas se han realizado sobre conjuntos aislados de clases, por lo que la otra tarea en la que se esta trabajando es la realización de pruebas exhaustivas sobre sistemas legados reales.

Aunque en esta etapa inicial no se han considerado elementos de minería de aspectos ("aspect mining") no se descarta incluir esta estrategia para mejorar la propuesta en lo que refiere a la identificación de crosscutting concerns. En este caso se supone que esta información podría ser mapeada como un conjunto de hechos iniciales que conjuntamente con los hechos base puedan ser utilizados por las reglas que identifican code smell candidatos.

Referencias

Arcoverde R., Lustosa P., Sousa A., Soares S., Borba P. (2007) "AJaTS – AspectJ Transformation System: Tool Support for Aspect-Oriented Development and Refactoring". SBES – TOOLS 07 Brasil

Binkley D., Ceccato M., Harman M., Ricca F., and Tonella P. (2005) "Automated refactoring of object oriented code into aspects". In 21st IEEE International Conference on Software Maintenance (ICSM).

CLIPS (2008) http://www.ghg.net/clips/CLIPS.html

Fowler M. (1999) "Refactoring: Improving the Design of Existing Code". Addison Wesley.

Friedman-Hill E. (2003). "Jess in Action". Manning Publications, ISBN 1-930110-89-8.

Hannemann J., Murphy G., and Kiczales G. (2005) "Role-based refactoring of crosscutting concerns". Aspect-Oriented Software Development (AOSD-2005), pp 135–146. ACM Press.

Iwamoto M. and Zhao J. (2003) "Refactoring aspect-oriented programs" 4th AOSD Modeling With UML Workshop, UML'2003. USA.

Kessler Piveta E., Hecht M., Soares Pimenta M., Price R.T. (2006). "Detecting Bad Smells in AspectJ". JUCS, vol. 12, no. 7, pages 811-827.

Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J. (1997). "Aspect-Oriented Programming". In Proceedings of ECOOP '97.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. (2001) "An Overview of AspectJ". ECOOP.

Monteiro M. (2004) "Catalogue of refactorings for AspectJ". Technical Report UM-DI-GECSD-200401, Universidade do Minho.

Monteiro M and Fernandes J. (2005) "Towards a catalog of aspect oriented refactorings". In Proc. of the 4th International Conference on Aspect-Oriented Software Development (AOSD), pages 111–122. ACM Press.

Opdyke W. (1993). "Refactoring Object-Oriented Framework" Ph.D Thesis. University of Illionois at Urbana-Champaign.

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