Automatización de un Proceso de Refactorización para la Separación de Concerns

Santiago A. Vidal[1, 2], Claudia A. Marcos[1]

[1]ISISTAN Facultad de Ciencias Exactas, UNICEN
Campus Universitario, Pje. Arroyo Seco (B7001BBO)
Tandil, Buenos Aires, Argentina
[2]CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas), Argentina
{svidal, cmarcos}@exa.unicen.edu.ar

Resumen La separación de concerns es una problemática importante de la ingeniería de software que influye en la modificabilidad de los sistemas. Si bien se han presentado prácticas arquitecturales que buscan solucionar esta problemática, existen concerns que son ortogonales a los módulos de un sistema (llamados crosscutting concerns) y no pueden ser modularizados por estas prácticas o por paradigmas de programación como el orientado a objetos. Con el objetivo de proveer mejores mecanismos de separación de concerns ha surgido la Programación Orientada a Aspectos (POA) la cual encapsula en un nuevo componente, llamado aspecto, los crosscutting concerns. Por esta razón, con el objetivo de aprovechar los beneficios de la POA ha surgido la necesidad de migrar los sistemas legados orientados a objetos a la orientación a aspectos. En este trabajo se propone la utilización de modelos de Markov con el fin de determinar el orden en el cual el código orientado a objetos debe ser migrado y para identificar las reestructuraciones a ser aplicadas durante el proceso de migración.

1. Introducción

La presencia de ciertas características arquitecturales como extensibilidad, reusabilidad y adaptabilidad en un sistema de software es un factor muy importante para determinar que tan evolucionable [8] será ese sistema. Uno de los puntos más importantes a tener en cuenta para alcanzar este objetivo es una correcta separación de concerns [10]. Sin embargo, existen ciertos concerns que son ortogonales a las unidades que componen un sistema y que no pueden ser correctamente modularizados mediante los enfoques tradicionales de la ingeniería de software o paradigmas de programación como el orientado a objetos. Esta clase especial de concerns son llamados crosscutting concerns (CCCs) [6]. Ejemplos típicos de CCCs son el manejo de excepciones, logging y control de concurrencia.

Una alternativa para tratar la separación de concerns es la Programación Orientada a Aspectos (POA) [6]. La POA aumenta la modularidad del software reduciendo el impacto ante eventuales cambios en el código mediante el encapsulamiento de los crosscutting concerns en un nuevo componente llamado aspecto.

De esta forma se complementa a la Programación Orientada a Objetos (POO) permitiendo una mejor evolución del sistema.

Gracias a la aparición de la POA, han comenzado a migrarse sistemas legados a este paradigma. Para alcanzar el objetivo de migrar un sistema orientado a objetos (OO) a uno orientado a aspectos (OA) es necesario descubrir en su código fuente, aquellos crosscutting concerns que potencialmente puedan convertirse en aspectos, denominados aspectos candidatos, en una actividad conocida como aspect mining [5]. Una vez identificados los aspectos candidatos es necesario transformar el código orientado a objetos original en código orientado a aspectos en lo que se conoce como aspect refactoring [5].

Con el objetivo de facilitar la tarea de migración de un sistema orientado a objetos a un sistema orientado a aspectos, y específicamente, con el fin de transformar los crosscutting concerns identificados por el proceso de aspect mining en aspectos, hemos propuesto en trabajos anteriores un proceso iterativo de aspect refactoring [14,13]. Este proceso ayuda al desarrollador en el análisis de los aspectos candidatos y en su encapsulamiento en aspectos mediante el uso de diferentes tipos de aspect refactorings.

En este trabajo se presentan 2 mejoras a nuestro proceso de refactorización con el objetivo de que este posea una mayor automaticidad. Específicamente, se propone la automatización de la tarea de determinar el orden en el cual el código indicado como aspectizable por la actividad de aspect mining debe ser refactorizado. Además, se propone la identificación automática de reestructuraciones adicionales que pueden surgir cuando la aplicación de un aspect refactoring no es suficiente para encapsular un aspecto candidato. La automatización de las 2 tareas se realiza mediante el uso de modelos de Markov [11].

El resto de este trabajo se estructura de la siguiente manera. En la Sección 2, se describe el proceso de aspect refactoring. Luego, en la Sección 3, se presenta la automatización de las actividades mediante el uso de modelos de Markov. En la Sección 4, se describen los resultados de la refactorización de un caso de factibilidad. Finalmente, en la Sección 5, se presentan las conclusiones y trabajos futuros identificados.

2. Proceso de Aspect Refactoring

Con el objetivo de asistir al desarrollador durante el proceso de encapsular los aspectos candidatos identificados por aspect mining en aspectos, hemos propuesto un proceso iterativo el cual tiene como meta la refactorización de un sistema orientado a objetos en uno orientado a aspectos [14,12]. Este proceso de refactoring tiene como entrada el sistema orientado a objetos en Java a ser migrado y los aspectos candidatos identificados, y tiene como salida el sistema migrado a una orientación a aspectos en AspectJ (si bien se utiliza AspectJ como leguaje orientado a aspectos, el proceso es fácilmente aplicable a otros lenguajes). El proceso se centra en la actividad de aspect refactoring aplicando reestructuraciones en el código durante cada ciclo de las iteraciones. El proceso completo consta de un conjunto de pasos que tiene por resultado un sistema

orientado a aspectos cuyos crosscutting concerns no solo están encapsulados en aspectos, sino que además su estructura interna está construida con el objetivo de ser legible, modificable y extensible. Esto se logra mediante la aplicación de diferentes tipos de aspect refactorings (Aspect-Aware OO, construcciones POA y CCC [4]) durante el proceso de migración. Además, una herramienta que soporta el proceso llamada AspectRT (Aspect Refactoring Tool) ha sido construida. Esta herramienta es un plug-in para Eclipse[1] y se integra con AspectJ.

Este trabajo se enfoca en los pasos del proceso de refactorización relacionados con el encapsulamiento de código aspectizable en aspectos con el objetivo de analizar y mejorar la manera en la que los crosscutting concerns son encapsulados en aspectos. Como muestra la Figura 1, los pasos son los siguientes:

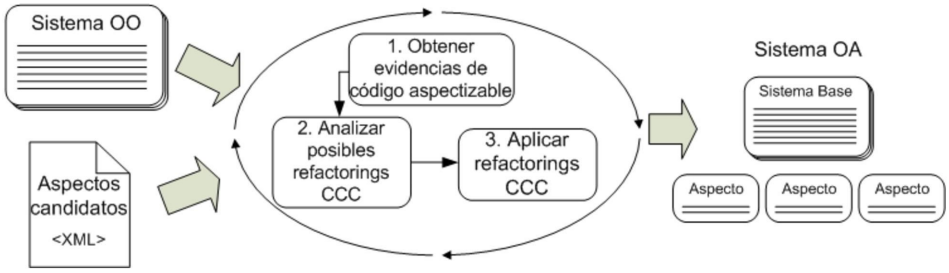

Figura 1. Proceso de refactorización acotado.

1. **Obtener evidencias de código aspectizable**. Este paso recupera el código que ha sido identificado como aspectizable por la fase de aspect mining. Los resultados de la actividad de aspect mining son transmitidos a través del uso de un archivo XML. Este archivo contiene la lista de crosscutting concerns con información relevante al código aspectizable. Cada CCC contiene uno o más aspectos candidatos que pueden estar diseminados en diferentes elementos Java tales como, clases, interfaces, métodos, implements declarations o fields. La estructura completa del archivo de aspectos candidatos se muestra en la Figura 2. Durante este paso un elemento Java perteneciente a un aspecto candidato es seleccionado con el objetivo de ser encapsulado en un aspecto.
2. **Analizar posibles refactorings CCC**. Este paso del proceso selecciona un tipo especial de aspect refactoring, llamado refactoring CCC [4], el cual tiene por objetivo el encapsulamiento de un fragmento de código aspectizable dentro de un aspecto. Para determinar un aspect refactoring apropiado de este tipo (o un conjunto de estos), debe realizarse un análisis del código a ser encapsulado.

[1] http://www.eclipse.org/

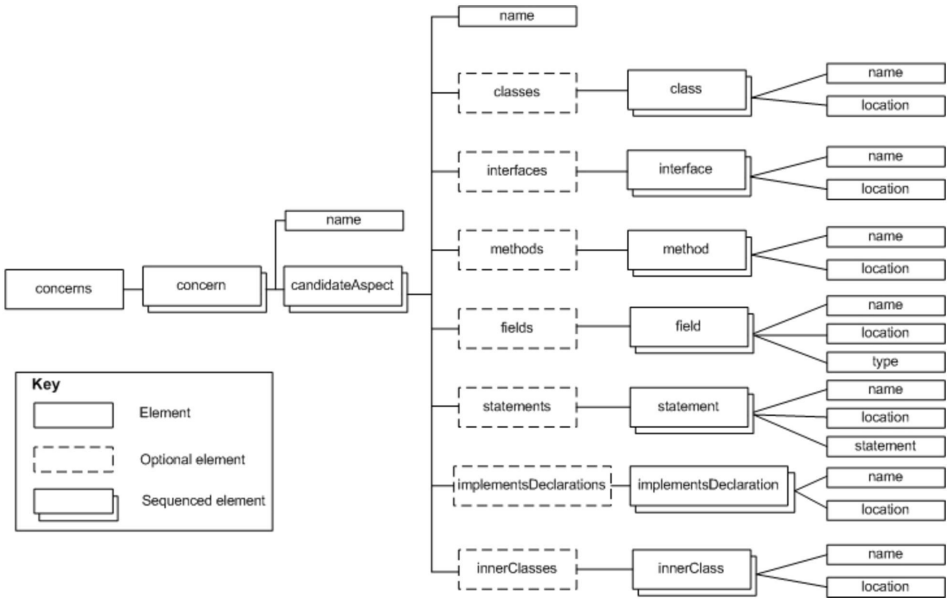

Figura 2. Estructura del archivo de aspectos candidatos.

3. **Aplicar refactorings CCC.** En este paso, el o los refactorings seleccionados anteriormente son aplicados. De esta forma los crosscutting concerns son extraídos del código orientado a objetos y son insertados en aspectos.

Existen algunas limitaciones en referencia a estos pasos en nuestros trabajos anteriores, específicamente aquellos en los cuales la intervención del desarrollador es muy requerida:

1. La selección de un elemento Java perteneciente a un aspecto candidato (Paso 1) es realizado de forma manual a través de la elección que realiza el desarrollador utilizando la interface provista por AspectRT.
2. El análisis y la selección de un aspect refactoring de CCC apropiado (Paso 2) es realizado por el desarrollador eligiendo de un menú de AspectRT los posibles refactorings a aplicar. Para esto se utiliza un enfoque basado en reglas, presentado en un trabajo anterior [13], que restringe los posibles refactorings a ser aplicados.
3. Aunque los aspect refactorings son aplicados automáticamente por el proceso (Paso 3), es necesaria la intervención del desarrollador para realizar algunas actividades en algunos casos. Si bien hay actividades que no pueden evitar ser delegadas al desarrollador (como por ejemplo, la elección de un aspecto en el cual se encapsulara un fragmento de código o el nombre de un nuevo pointcut), en algunas ocasiones, es necesario realizar reestructuraciones adicionales a las especificadas por un refactoring.

Como se dijo anteriormente, el problema común de las tres limitaciones es que el desarrollador debe intervenir demasiadas veces durante el proceso de refactorización. Esto representa demoras en la tarea de refactorización y el consumo de tiempo valioso. Si bien se han propuesto alternativas para solucionar la segunda limitación, no ha ocurrido lo mismo con las dos restantes. Por estas razones, este trabajo se centra en el análisis de técnicas que permitan resolver las limitaciones 1 y 3. De esta forma, se pretende proveer un proceso de refactorización con mayor automaticidad.

3. Asistencia Automática Utilizando Algoritmos de Markov

En algunas ocasiones la aplicación de un aspect refactoring, en el paso 3 del proceso (Figura 1), no es suficiente para encapsular un aspecto candidato por lo cual deben realizarse reestructuraciones adicionales de forma manual. Generalmente, estos cambios deben realizarse con el objetivo de completar el encapsulamiento de un aspecto candidato o para solucionar problemas en el código que permitan una correcta compilación.

En algunos casos, cuando un aspect refactoring no es suficiente para encapsular un elemento Java de un aspecto candidato, existen aspect refactorings específicos que podrían ser aplicados. Sin embargo, este tipo de refactorings (por ejemplo, los presentados por Laddad [7]) sólo son aplicables en situaciones muy específicas a diferencia de otros aspect refactorings (por ejemplo, los presentados en el catálogo de Monteiro [9]) que son aplicados sobre estructuras simples y pueden ser utilizados en muchas circunstancias.

Por las razones mencionadas anteriormente, se propone la identificación y la aplicación automática de estas actividades utilizando técnicas de inteligencia artificial. Se utilizan modelos de Markov [11] para capturar el conocimiento del desarrollador en determinados contextos y reconocer sus intenciones cuando se presentan casos similares.

Adicionalmente, se utilizan modelos de Markov para determinar el orden en el cual el código aspectizable debe ser refactorizado (Paso 1 del proceso de refactorización).

El uso de esta técnica en el proceso de refactoring se muestra en la Figura 3 y es explicada en detalle en las siguientes secciones.

3.1. Modelos de Markov

Un Modelo Oculto de Markov (MOM) es un proceso doblemente estocástico con un proceso subyacente que no es observable (oculto) pero que puede ser observado a través de otro conjunto de procesos estocásticos que generan la secuencia de observaciones [11]. Un modelo de Markov describe un proceso que se desarrolla a través de una secuencia de estados discretos. Se dice que el modelo es oculto debido a que el estado del modelo en el tiempo t no es observable directamente. Un modelo oculto de Markov cumple con la conjetura de Markov

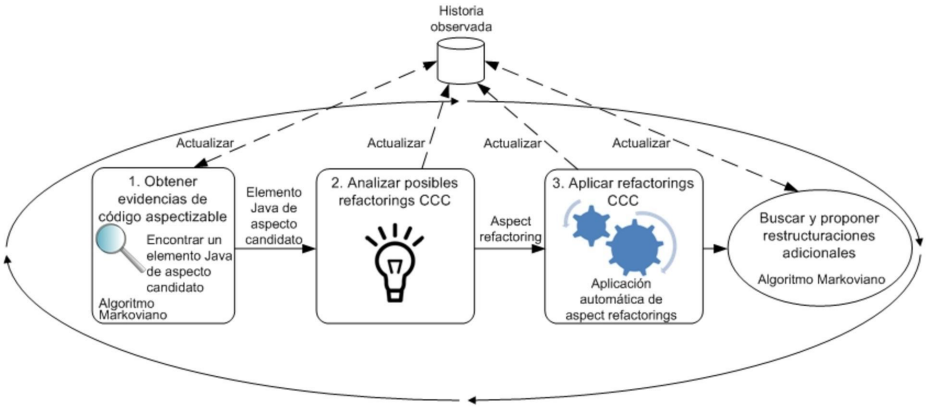

Figura 3. Proceso automático de refactorización.

de que, dado el estado actual, los estados futuros son independientes de los últimos estados.

Los principales elementos de un MOM son los siguientes [11]:

- N estados que son denominados individualmente como $\{S_1, S_2, \ldots, S_N\}$.
- El estado actual en el tiempo t es indicado como q_t.
- La distribución de las probabilidades de transición de un estado está dada por la matriz $A = \{a_{ij}\}$ donde a_{ij} es la probabilidad de que el modelo transicione del estado i al estado j.
- M diferentes acciones observables por estado que son denominadas individualmente como $V=\{v_1, v_2, \ldots, v_M\}$.
- La distribución de las probabilidades de las acciones está dada por la matriz $B = \{b_j(k)\}$ donde $b_j(k)$ es la probabilidad de observar la acción v_k cuando el modelo esta en el estado q_j.
- La acción observada en el tiempo t se indica como O_t.

3.2. Utilización de un Algoritmo Markoviano para Asistir al Desarrollador

Con el objetivo de identificar a través de modelos de Markov las acciones futuras del desarrollador cuando este reestructura un sistema, el algoritmo ONISI (On-line Implicit State Identification) [2,3] es utilizado. En el contexto de este trabajo, este algoritmo observa la interacción del desarrollador con AspectRT y dado un estado, predice la siguiente acción que realizará el desarrollador. ONISI asigna probabilidades a todas las posibles acciones en el estado que se observa. Estas probabilidades son calculadas estimando qué porcentaje de la historia observada soporta una acción en el contexto que se observa. Esta estimación se realiza utilizando un algoritmo de k vecinos más cercanos que realiza un ranking de las acciones que pueden suceder en un estado teniendo en cuenta la longitud

de las secuencias encontradas en la historia. De esta forma, ONISI busca cumplir con la conjetura de Markov para el contexto que se observa.

Como se mencionó anteriormente, cuando el contexto actual es observado por ONISI, las probabilidades de ocurrencia de todas las posibles acciones que pueden suceder desde el estado actual son calculadas y jerarquizadas mediante un ranking. Este ranking es calculado por ONISI como:

$$R_t(q_i, v_j) = \alpha \frac{l_t(q_i, v_j)}{\sum_n l_t(q_i, v_n)} + (1 - \alpha) \frac{f(q_i, v_j)}{\sum_n f(q_i, v_n)}$$

donde $l_t(q_i, v_j)$ es el promedio de las longitudes de las k secuencias de pares <estado, acción> de mayor longitud que terminan con la acción v_j en el estado q_i y que coinciden con la secuencia inmediatamente anterior al tiempo t; $f(q_i, v_j)$ es el número de veces que la acción v_j ocurrió en el estado q_i; y α es un valor entre 0 y 1 para indicar el peso que se le da a la longitud de las historias y la frecuencia en la que ocurren.

Dentro del proceso de refactorización el algoritmo ONISI es utilizado para ayudar al desarrollador en la tarea de qué elemento Java de un aspecto candidato debe ser refactorizado (Paso 1 de la Figura 3). Además, ONISI es utilizado para identificar aquellas reestructuraciones que deben realizarse en algunas ocasiones luego de aplicar un aspect refactoring (al finalizar el paso 3 de la Figura 3). Para realizar estas tareas fue creado un modelo que representa la interacción del desarrollador con AspectRT. El mismo tiene 3 estados (Figura 4):

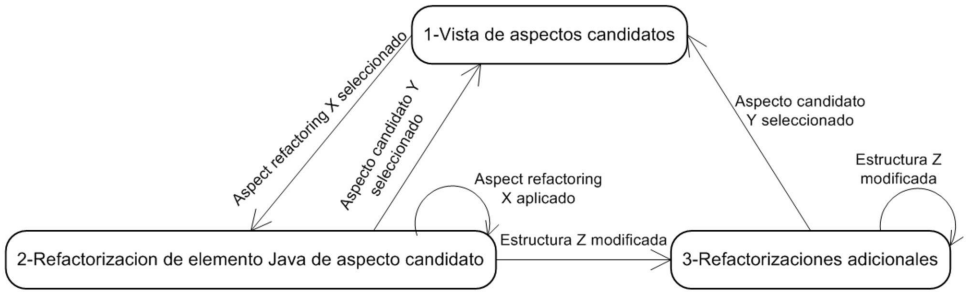

Figura 4. Modelo de AspectRT

1. **Vista de aspectos candidatos**. Este estado representa la situación en la cual un aspecto candidato ha sido seleccionado. Para transicionar hacia este estado un aspecto candidato debe ser seleccionado.
2. **Refactorización de elemento Java de aspecto candidato**. Este estado sucede cuando un aspect refactoring ha sido seleccionado y será o ha sido aplicado. Para transicionar hacia este estado un aspect refactoring debe ser seleccionado o aplicado.

3. **Refactorizaciones adicionales**. Este estado representa la situación en la cual se realizan reestructuraciones adicionales después de aplicar un aspect refactoring. Para transicionar a este estado debe modificarse una estructura de código tal como el modificador de acceso a una variable o clase (private, public, etc.) o un bloque try/catch. Por ejemplo, la transacción que informa que el aspect refactoring *Move Method from Class to Inter-type* [9] ha sido seleccionado y que debe transicionarse al estado 2 se representa como el par <estado, acción>:

<Refactorización de elemento Java de aspecto candidato, aspect refactoring Move Method from Class to Inter-type seleccionado>

Los estados y las acciones que ocurren en el modelo son actualizadas en cada paso del proceso de refactoring y son almacenados en una base de datos que contiene la historia observada (como se vio en la Figura 3).

Una vez definido el modelo de la aplicación y almacenada la historia observada debe ejecutarse el algoritmo que hace uso de la misma.

En el caso del paso 1 del proceso, cuando se refactoriza un aspecto candidato, generalmente, existe un conjunto de elementos Java que lo componen que deben ser encapsulados. El orden en el cual los elementos Java son seleccionados hacen posible o no el encapsulamiento de un aspecto candidato. Usualmente, este orden depende de la estructura del aspecto candidato. Mediante el uso del algoritmo ONISI se busca obtener un proceso iterativo que sea capaz de elegir automáticamente el orden en el cual los elementos Java de un aspecto candidato deben ser refactorizados. Con este objetivo, el algoritmo ONISI es ejecutado en el paso 1 del proceso de refactoring y el primer aspecto candidato resultante del ranking es propuesto al desarrollador (en caso de que no resten elementos Java del aspecto candidato de este tipo para ser encapsulados se propone el siguiente del ranking).

En cuanto a la situación que puede producirse al finalizar el paso 3 del proceso de refactoring, en la cual la aplicación de un refactoring no es suficiente para encapsular un elemento Java, se utiliza el algoritmo ONISI para identificar estas situaciones y automáticamente proponer los cambios necesarios. El algoritmo se utiliza para analizar situaciones pasadas en las cuales se agregó, eliminó o modificó una estructura, en una clase o un aspecto, luego de aplicar un aspect refactoring. A partir de este análisis se determina si se deben realizar o no reestructuraciones adicionales y en caso de ser así son propuestas al desarrollador.

4. Caso de Estudio

Con el objetivo de demostrar la aplicación de la propuesta y de medir la efectividad de la misma se presenta un pequeño caso de estudio. El caso de estudio consiste en la refactorización a aspectos del crosscutting concern E*xception Wrapping and Business Delegate* [1] presente en la aplicación *Java Pet Store*

Demo[2]. Este crosscutting concern se encuentra esparcido en más de 40 clases de la aplicación *Java Pet Store Demo* y se encuentra relacionado con el manejo de excepciones. Específicamente, cuando se encuentra una excepción se dispara una nueva de un tipo diferente. La solución propuesta es encapsular el código que maneja la excepción en un aspecto. En total el crosscutting concern se encuentra diseminado en más de 140 elementos Java que corresponden a 41 aspectos candidatos. Por ejemplo, la Figura 5 muestra la clase *AccountEJB* en la cual el método *ejbPostCreate* contiene un bloque try/catch. Este bloque luego de encontrar una excepción de tipo *NamingException* dispara una nueva de tipo *CreateException*. Además, con el fin de darle aún mayor complejidad se ha agregado una variable *errorData* que contiene información relevante a la excepción.

```
public abstract class AccountEJB implements javax.ejb.EntityBean {
    ...
    String errorData;
    public void ejbPostCreate(String status) throws CreateException {
            setStatus(status);
            try {
                    InitialContext ic = new InitialContext();
                    ...
            } catch (javax.naming.NamingException ne) {
                throw new CreateException("InfoEJB error: "+errorData);
            }
    }
    ...
}
```

Figura 5. Ejemplo crosscutting concern E*xception Wrapping and Business Delegate*.

Para llevar a cabo la refactorización de este CCC se realizó previamente un pequeño entrenamiento de la herramienta utilizando ejemplos del CCC como los presentados en [7]. Tanto durante el entrenamiento como en la refactorización de Java PetStore Demo se usó un valor de k=4 para utilizar el promedio de las longitudes de las 4 secuencias de pares <estado, acción> de mayor longitud. Además, se utiliza un valor de α=0.5 con el objetivo de darle igual importancia a la longitud de las historias y la frecuencia en la que ocurren.

Siguiendo el proceso de refactorización, para comenzar la migración del CCC debe cargarse el XML que contiene los aspectos candidatos. El mismo contiene en este caso una lista de los statements en los cuales se encuentran los bloques try/catch y las variables relacionadas con las excepciones (Figura 6). De esta forma, cada aspecto candidato posee una variable y uno o más statements. Una vez cargado el archivo XML, AspectRT comienza a iterar sobre el proceso selec-

[2] http://java.sun.com/developer/releases/petstore, versión 1.3.2

cionando el primer elemento Java de un aspecto candidato (Paso 1). En este caso la variable *errorData* es seleccionada y encapsulada (Paso 2) en un nuevo aspecto, llamado *AccountEJBException*, utilizando el aspect refactoring *Move Field from Class to Inter-type* [9]. Luego de realizar esta reestructuración el modelo de la aplicación se encuentra en el estado 2 y AspectRT propone la selección de un statement.

```
<candidateAspect>
  <concern>
    <name>ExceptionWrappingAndBusinessDelegate</name>
    <candidateAspect>
      <name>AccountEJBException</name>
      <fields>
        <field>
          <name>errorData</name>
          <location>/PetStore/src/.../AccountEJB.java</location>
          <type>String</type>
        </field>
      </fields>
      <statements>
        <statement>
          <name>ejbPostCreate(String)</name>
          <location>/PetStore/src/.../AccountEJB.java</location>
          <statement>0</statement>
        </statement>
      </statements>
    </candidateAspect>
    ...
  </concern>
</candidateAspect>
```

Figura 6. Aspectos candidatos para la clase *AccountEJB*.

La refactorización del CCC continúa con la selección del statement que lanza una nueva excepción dentro del bloque try/catch del método *ejbPostCreate* el cual debe encapsularse utilizando el aspect refactoring *Move Method from Class to Inter-type* [9]. El aspecto resultante luego de aplicar este refactoring se muestra en la Figura 7. Esta reestructuración, aunque correcta, no es suficiente ya que no se está tratando correctamente la excepción. Para completar el encapsulamiento de forma similar a *Extract Exception Handling* [7] (el cual es un refactoring específico para esta situación) debe tratarse la excepción mediante una estructura *declare soft* y luego removerse el bloque try/catch de la clase.

Dado que en este caso se realizó un entrenamiento previo, AspectRT sugiere, mediante la utilización del algoritmo ONISI, la adición de la estructura *declare soft* (como se vio en la Figura 7). Luego de realizar esta modificación, AspectRT propone la siguiente reestructuración adicional la cual es la eliminación del

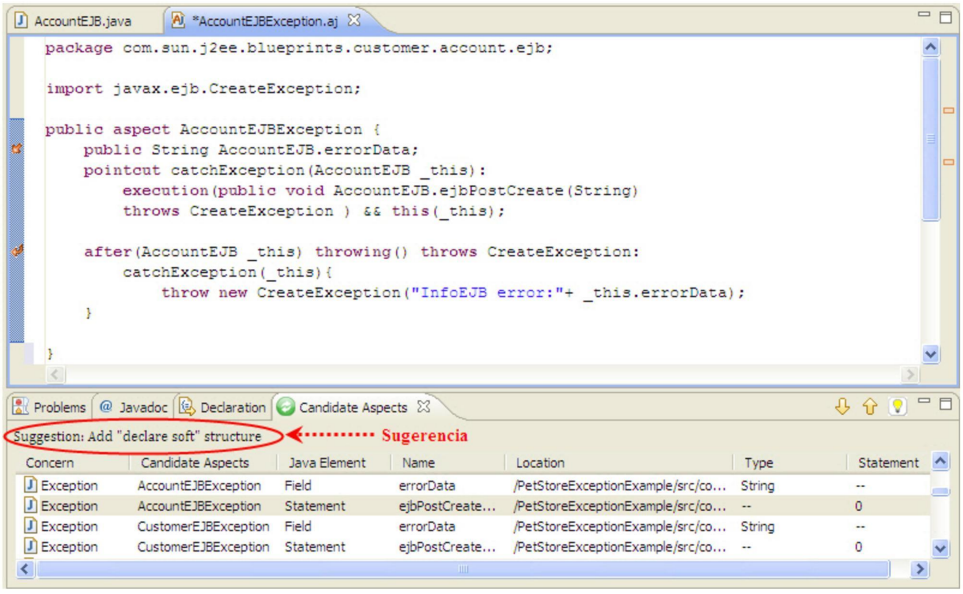

Figura 7. Sugerencia de paso adicional.

bloque try/catch del método *ejbPostCreate*. Luego de aplicar estas reestructuraciones se logra encapsular totalmente el aspecto candidato siendo el aspecto que lo contiene el presentado en la Figura 8.

El resto de los aspectos candidatos correspondientes al crosscutting concern E*xception Wrapping and Business Delegate* se encapsularon de forma similar. A continuación se describen los resultados generales de la refactorización.

El porcentaje total de elementos Java a ser refactorizados identificados correctamente por la herramienta (Paso 1 del proceso de refactoring) fue del 80 %. La mayoría de los problemas relacionados a este paso fueron identificados al comenzar la refactorización de un nuevo aspecto candidato. Se encontró que el algoritmo no identifica correctamente la finalización del encapsulamiento de un aspecto candidato. Por esta razón, falla al proponer correctamente un elemento Java para comenzar el encapsulamiento del próximo aspecto candidato.

El porcentaje de pasos adicionales de refactorización (luego del paso 3 del proceso de refactoring) propuestos correctamente por la herramienta fue del 84 %. En este caso, los mayores problemas se encontraron en situaciones en las que existen más de una sentencia catch para un try. Esto provoca que deban realizarse pasos adicionales diferentes al caso más general, en el que sólo existe una sentencia catch, por lo que sólo algunos pasos son identificados correctamente. Esto también se debe a que no hubo un entrenamiento específico en este tipo de casos. Como puede observarse en la Figura 9 estos casos se presentaron en pocos aspectos candidatos en los cuales fueron necesarios muchos pasos adicionales para completar el encapsulamiento.

```
public aspect AccountEJBException {
        private String AccountEJB.errorData;
        declare soft : javax.naming.NamingException :
            call(public void AccountEJB.ejbPostCreate(String)
            throws javax.naming.NamingException )
        pointcut catchException(AccountEJB _this):
            execution(public void AccountEJB.ejbPostCreate(String)
            throws CreateException)
        after (AccountEJB _this) throwing ()
            throws CreateException: catchException(_this){
                throw new CreateException("InfoEJB error: "+errorData);
            }
}
```

Figura 8. Refactorización del crosscutting concern E*xception Wrapping and Business Delegate*.

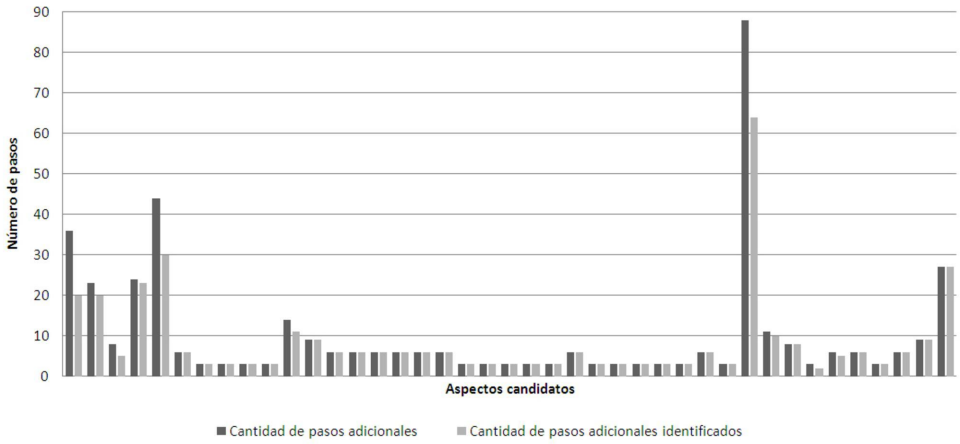

Figura 9. Identificación de pasos adicionales.

Los resultados totales de la refactorización del crosscutting concern indican que el 64 % de los elementos Java fueron refactorizados automáticamente (Figura 10), es decir, el elemento a encapsular fue identificado correctamente al igual que todos los pasos adicionales necesarios para realizar esta tarea. Del 36 % restante, el 19 % falló debido a que no se identifico correctamente el elemento Java a encapsular y el 17 % debido a que no se identificaron correctamente los pasos adicionales.

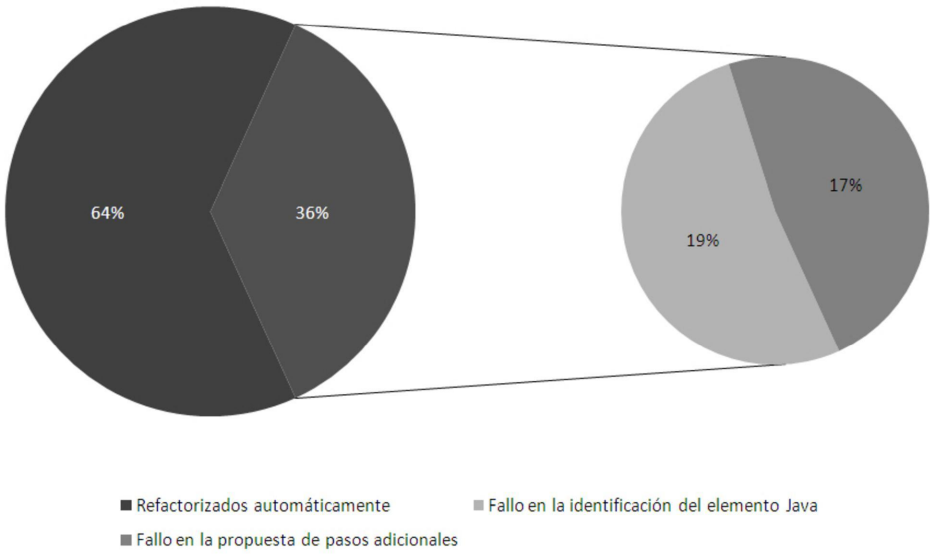

Figura 10. Resultados de la refactorización.

5. Conclusiones

En este trabajo se han presentado mejoras a un proceso de refactorización existente, el cual tiene por objetivo la migración de un sistema orientado a objetos en uno orientado a aspectos. Específicamente, las mejoras introducidas buscan aumentar la automaticidad del proceso utilizando modelos de Markov. Mediante esta técnica de inteligencia artificial se observa la interacción del usuario con el proceso, con el fin de construir un modelo estocástico que comprenda el proceso y las actividades que el usuario realiza al utilizarlo. De esta forma, por medio de la utilización del algoritmo ONISI se asiste al usuario durante el proceso de refactoring proponiéndole: (a) orden en el cual el código indicado como aspectizable por la actividad de aspect mining debe ser refactorizado y (b) actividades adicionales a llevarse a cabo con el fin de encapsular un elemento

Java correspondiente a un aspecto candidato cuando la aplicación de un aspect refactoring no es suficiente.

La principal ventaja de este trabajo es que mediante una mayor automatización del proceso se evitan demoras en la tarea de refactorización que consumirían tiempo valioso del desarrollador. Además, mediante la utilización de técnicas de inteligencia artificial para identificar actividades adicionales que deben llevarse a cabo se obtiene un proceso flexible que se adapta a diferentes tipos de reestructuraciones. Esto es beneficioso respecto de la posibilidad de agregar refactorings específicos, dado que estos últimos deberían ser programados e incluidos en la herramienta para cada una de las situaciones particulares que se presenten.

Aún existen algunos problemas pendientes por solucionar. Particularmente, en la experimentación llevada a cabo no se ha logrado identificar fehacientemente el primer elemento Java a ser encapsulado en un aspecto candidato. Por lo cual se pretende realizar mejoras en la forma en que se realiza esta identificación en trabajos futuros. Además, se prevé comparar el desempeño de la técnica cuando no se ha hecho ningún entrenamiento y cuando éste ha sido realizado. Finalmente, se pretende realizar experimentaciones de la técnica con otros sistemas reales.

References

1. Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
2. Peter Gorniak and David Poole. Predicting future user actions by observing unmodified applications. In *AAAI/IAAI*, pages 217–222. AAAI Press / The MIT Press, 2000.
3. P.J. Gorniak. *Keyhole state space construction with applications to user modeling*. PhD thesis, University of British Columbia, 2000.
4. Jan Hannemann. Aspect-oriented refactoring: Classification and challenges. In *LATE '06*, 2006.
5. A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development (TAOSD)*, IV (Special Issue on Software Evolution):143–162, 2007.
6. Gregor Kiczales, John Lamping, Anurag Mendheka, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Stein Gjessing and Kristen Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in Lecture Notes in Computer Science, Finland, June 1997. Springer.
7. Ramnivas Laddad. Aspect-oriented refactoring, 2003.
8. M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.
9. Miguel P.t Monteiro. Catalogue of refactorings for aspectj. Technical Report UM-DI-GECSD-200402, Universidade do Minho, December 2004.
10. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.
11. L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proc. of the IEEE*, 77(2):257 – 286, February 1989.

12. Santiago Vidal, Esteban S. Abait, Claudia Marcos, Sandra Casas, and J. Andrés Díaz Pace. Aspect mining meets rule-based refactoring. In *PLATE '09: Proceedings of the 1st workshop on Linking aspect technology and evolution*, pages 23–27, New York, NY, USA, 2009. ACM.
13. Santiago Vidal and Claudia Marcos. Identificación automática de refactorings. In *Tenth Argentine Symposium on Software Engineering (ASSE 2009), 38 JAIIO (Jornadas Argentinas de Informática)*, 2009.
14. Santiago Vidal and Claudia Marcos. Un proceso iterativo para la refactorización de aspectos. *Revista Avances en Sistemas e Informática*, 6(1):93–103, 2009.