Refactorización Orientada a Aspectos de Aplicaciones Empresariales

Guillermo Zunino[1], Santiago Vidal[1, 2], Claudia Marcos[1, 3]

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

Resumen La importancia de las aplicaciones empresariales en las organizaciones de hoy en día, hace que sea necesario facilitar su mantenimiento y evolución a lo largo de su existencia. Este tipo de aplicaciones de gran complejidad cuentan con diversas funcionalidades y requerimientos que cortan transversalmente su estructura, para lo cual otros mecanismos de modularización más allá de la orientación a objetos resultarían convenientes. Dado que muchas de las aplicaciones empresariales se implementan utilizando el framework Spring puede utilizarse el módulo de AOP que este provee para obtener una mejor separación de los concerns. Con este objetivo, en este trabajo se presentan dos casos de estudios en los cuales se refactorizan los crosscutting concerns de dos aplicaciones empresariales con el propósito de encapsularlos en aspectos. Con el fin de simplificar la tarea de refactorizacion un conjunto de refactorings, inicialmente presentados para AspectJ fueron adaptados para ser utilizados con Spring/AOP.

1. Introducción

El término aplicación empresarial es el término utilizado para describir las aplicaciones o software que una organización podría usar para ayudar en la solución de problemas empresariales [7]. El software de nivel empresarial es software que ofrece funcionalidad de soporte de lógica de negocios para una empresa, generalmente en organizaciones comerciales, cuyo objetivo es mejorar la productividad y la eficiencia de la empresa. Los servicios prestados por este tipo de software suelen ser herramientas orientadas a los negocios tales como compras en línea y procesamiento de pagos en línea, catálogos de productos interactivos, sistemas automatizados de facturación, etc.

En el entorno corporativo de hoy, estas aplicaciones son complejas y de misión crítica. Pueden ser desplegadas en una variedad de plataformas a través de redes corporativas, intranets o Internet. Diseñar y desarrollar tales aplicaciones empresariales significa satisfacer cientos de requisitos diferentes. Las decisiones de desarrollo que se hacen para cumplir cada requisito afectan a muchos otros requisitos, a menudo en formas que son difíciles de comprender o predecir, y el

incumplimiento de cualquiera de estos requisitos puede significar el fracaso de todo un proyecto [6]. El mantenimiento y la evolución de estos sistemas una vez desarrollados es aún más complejo.

Para hacer frente a estas sofisticadas aplicaciones empresariales que a menudo requieren servicios tales como soporte de transacciones, seguridad, y computación distribuida se utiliza frecuentemente Spring [7]. El objetivo de Spring es facilitar el desarrollo de aplicaciones empresariales Java y promover código con bajo acoplamiento. Es un producto de software de alta calidad que se puede utilizar para construir software de clase empresarial en forma simple y sencilla.

La tendencia de los sistemas de software, y sobre todo los empresariales, es de un crecimiento rápido en la complejidad. La solución a la complejidad es la modularización [8]. Cuando se encuentran requisitos de software complejos, es muy probable que se tienda a dividirlos en varias partes, tales como lógica de negocio, acceso a datos y lógica de presentación. Dichas funcionalidades representan distintos concerns del sistema [8]. El desarrollo de software orientado a aspectos (DSOA) tiene por objeto mejorar la modularidad de los sistemas de software, mediante la captura de aquellas funcionalidades inherentemente transversales al código de la aplicación [11], denominadas crosscutting concerns, haciendo que la evolución de dichos sistemas sea más fácil y manejable. Spring/AOP es un módulo importante del framework Spring que presta servicios esenciales a nivel del sistema, promueve el bajo acoplamiento y permite que los crosscutting concerns sean separados de una manera más elegante.

Considerando la complejidad de las aplicaciones empresariales, la necesidad de su constante evolución, y la dificultad del mantenimiento de los sistemas existentes, en parte debido a las limitaciones de los mecanismos de modularización de la orientación a objetos, resulta evidente la necesidad de introducir las ventajas de la programación orientada a aspectos en estos sistemas para facilitar su evolución.

Si bien existen múltiples soluciones para refactorizar a aspectos, la mayoría se basan en el lenguaje AspectJ[1]. El catálogo de refactorings más conocido y utilizado es el de Monteiro [12] debido a su granularidad propia para una implementación automática. Para poder aprovechar los beneficios mencionados de Spring/AOP, como su rapidez para adoptar AOP en un proyecto existente, es necesario implementar refactorings propios para este framework. Para ello, en este trabajo se adaptaron los refactorings propuestos por Monteiro a las características propias de Spring/AOP, analizando su aplicabilidad, ventajas y limitaciones. Para migrar un sistema existente orientado a objetos hacia la orientación a aspectos, es necesario modularizar en aspectos todos los crosscutting concerns presentes, aplicando múltiples refactorizaciones sobre el código. Este proceso resulta muy complejo por sí mismo. Con el fin de simplificar la adopción de AOP en un sistema existente, se incorporaron los refactorings implementados dentro de la herramienta AspectRT [14], que sigue un proceso iterativo y ordenado de migración donde la entrada es código orientado a objetos con los aspectos candidatos identificados y su salida es una refactorización de este código a aspectos.

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

Para poder refactorizar un sistema empresarial a la orientación a aspectos se siguió el proceso iterativo de migración, realizando primero la minería de aspectos para identificar los crosscutting concerns presentes en el código, y luego, por medio de la aplicación de los refactorings adaptados para Spring/AOP, modularizar estos concerns en aspectos. Como casos de estudio se tomaron dos sistemas empresariales los cuales estaban implementados en Spring y fueron migrados a Spring/AOP analizando los resultados obtenidos.

El trabajo esta organizado de la siguiente manera, la Sección 2 presenta la adapatación de un conjunto de aspect refactorings, inicialmente presentados para AspectJ, para ser utilizados en Spring/AOP. En la Sección 3 se detallan las refactorizaciones realizadas en dos aplicaciones empresariales con el objetivo de encapsular sus crosscutting concerns en aspectos. Luego, en la Sección 4 se presentan los trabajos relacionados. Finalmente, en la Sección 5 se exponen las conclusiones del trabajo.

2. Refactorings para Spring/AOP

En esta sección se describen brevemente cómo los refactorings propuestos por Monteiro han sido adaptados para generar código Spring/AOP. Debido a que Spring/AOP carece del poder expresivo de AspectJ existen algunas limitaciones en la adaptación de los refactorings de Monteiro. Durante el proceso de adaptación de los refactorings de AspectJ a Spring/AOP es posible identificar tres grupos: aquellos refactorings que debieron ser adaptados con mayor o menor facilidad, aquellos que han podido ser utilizados directamente sin necesidad de adaptación y un tercer grupo de refactorings que no han podido ser adaptados debido a la carencia de poder expresivo de Spring/AOP. A continuación se describen brevemente estos tres grupos.

2.1. Refactorings Adaptados

Los refactorings más utilizados en un proceso de migración se han podido adaptar con algunos cambios mayores debido a las limitaciones de Spring/AOP. A continuación se describe la adaptación del refactoring Extract Fragment into Advice de manera tal de demostrar los cambios realizados para poder extraer un fragmento de código Spring en un advice de Spring/AOP. De manera similar se pudieron adaptar los refactoring Move Method form Class to Inter-type, Encapsulate Implements with Declare Parents, Move Field form Class to Inter-type, Inline Class with Aspect, y Extract Feature into Aspect.

Extract Fragment into Advice

Extract Fragment into Advice se aplica en la situación en que parte de un método está relacionado a un concern cuyo código quiere moverse a un aspecto. Para llevar a cabo este refactoring, se crea un pointcut que capture el joinpoint apropiado con su contexto, y se mueve el fragmento de código a un advice. A

diferencia de AspectJ que soporta una gran variedad de joinpoints Spring/AOP sólo permite joinpoints en la ejecución de métodos, en beans manejados por el framework [8]. Los casos en que puede llevarse a cabo el refactoring sin inconvenientes, son cuando las líneas de código pertenecientes al fragmento a extraer se encuentran al principio o al final del método, con lo cual puede aplicarse un before advice, after advice, o around advice. Para facilitar la automatización de este refactoring, es conveniente convertir la parte que quiere extraerse en un método propio, usando el refactoring de objetos Extract Method [2] y luego Move Method from Class to Inter-type. Luego, se crea un advice que invoque al nuevo método. Teniendo en cuenta estas consideraciones, los pasos que restan son los siguientes:

- Crear un método con lo anotación @Pointcut que capture la ejecución del método que contiene el fragmento de código a extraer.
- Asegurar que el pointcut también capture el contexto requerido por el fragmento de código, como parámetros del método, o referencias a this.
- Crear un método con la anotación correspondiente al tipo de advice y valor igual al nombre del pointcut creado anteriormente, y mover el código a extraer del método original en el cuerpo del advice.
- Reemplazar en el fragmento las referencias a la variable this por la variable correspondiente capturada en el pointcut.

Ejemplo Extracción simple de una invocación a un método. Se pretende extraer la llamada a display() de la clase TangledStack.

```
public class TangledStack{
  private int _top = -1;
  private Object[] _elements;
  public void display(){ //...}
  public void push(Object element){
    _elements[++_top] = element;
    display(); } //...}
```

Primero, la llamada al método display() es borrada del método push(Object). Luego, se define el pointcut que captura las llamadas al método push(Object) usando el annotation @Pointcut en un nuevo aspecto, denominado WindowView. Utilizando dicho pointcut el contexto del jointpoint es capturado por medio de la variable stack. Finalmente, debido a que la llamada al método display() se realiza al final del método push(Object) un advice es creado con el annotation @AfterReturning.

```
@Aspect
public class WindowView {
  @Pointcut("(execution(public void TangledStack.push(Object))) && this(
      stack)")
  private void stateChanged(TangledStack stack) {}
  @AfterReturning("stateChanged(stack)")
  public void displayState(TangledStack stack) {
    stack.display(); } }
```

2.2. Refactorings utilizados Directamente

Algunos refactorings presentados por Monteiro han sido utilizados para Spring/AOP sin necesidad de adaptación, debido a que no involucran demasiados conceptos de la orientación a aspectos. Los refactorings dentro de este grupo son Extract Inner Class to Standalone, Inline Interface within Aspect, y Change Abstract Class to Interface.

2.3. Refactorings que no han podido ser Adaptados

El único refactoring que no ha podido ser adaptado es Split Abstract Class between Aspect and Interface. Este refactoring es utilizado cuando una o más clases heredan de la misma clase abstracta impidiendo que hereden de otra clase. Adicionalmente, la clase abstracta no puede convertirse en una interface porque define miembros concretos. La solución para AspectJ consiste en mover todos los miembros concretos de la clase abstracta a un aspecto y luego convertir la clase abstracta a una interface.

La implementación en Spring/AOP de este refactoring no es posible, principalmente por la diferencia en el mecanismo de implementación de inter-type declarations entre uno y otro. Si bien podría pensarse en utilizar el mecanismo de introducción de Spring/AOP para poveer aquellos métodos de la clase abstracta a sus subclases, Spring/AOP requiere que se provea la implementación de la interface con todos sus métodos, y en este caso por tratarse de una clase abstracta hay métodos que no tienen implementación definida.

Una alternativa consistiría en definir una jerarquía de aspectos, con un aspecto definiendo los métodos de la clase abstracta, y un subaspecto por cada clase concreta conteniendo las implementaciones de métodos concretos y la "introducción" de la interface en las subclases originales. Pero esto tendría sentido únicamente si la clase abstracta representara un concern secundario, y el resultado sería la migración de todo el concern en una jerarquía de aspectos. Si este fuera el caso, podría lograrse mediante la implementación de Encapsulate Implements with Declare Parents para cada subclase concreta.

3. Casos de Estudio

Se han desarrollado dos casos de estudio para evaluar la propuesta presentada en este trabajo. Se analizaron los sistemas Finance Manager[2] y GridGain[3]. Debido a la complejidad de los sistemas es necesario aplicar una amplia variedad y cantidad de refactorings, con lo cual es necesario contar con herramientas que asistan durante el proceso. Por esta razón, se extendió la herramienta Aspect-RT [14] con los refactorings para Spring/AOP debido a que la misma generaba solamente código AspectJ.

[2] http://code.google.com/p/spring-finance-manager/
[3] http://www.gridgain.com/

Para realizar la evaluación de la migración se tuvieron en cuenta los efectos que las transformaciones aplicadas tuvieron sobre la calidad del sistema en general, así como también la facilidad provista para aplicar los sucesivos refactorings. Para ello se han utilizado un conjunto de métricas para medir por ejemplo el tamaño de alguna unidad en particular, como paquetes o clases (lineas estimadas de código ELOC, lineas estimadas de código por módulo ELOC/Modulo). Las métricas de complejidad se pueden definir como una medida que es directamente proporcional al esfuerzo necesario para entender el código y modificarlo correctamente. La complejidad se relaciona con la capacidad de prueba y mantenimiento del mismo. Dentro de este grupo están la complejidad ciclomática (CC) y complejidad ciclomática promedio. Adicionalmente, se utilizó la familia de métricas definida por Chindamber y Kemerer [1] destinadas a la orientación a objetos pero que pueden ser fácilmente adaptadas a la orientación a aspectos unificando clases y aspectos, así como métodos y advices. En está familia se encuentran operaciones ponderadas por módulo (WOM), acoplamiento entre componentes (CBC), respuesta para un módulo (RFM), falta de cohesión en operaciones (LCOO), etc.

3.1. Sistema Finance Manager

Spring Finance Manager es una herramienta simple de gestión de finanzas creada para proporcionar una idea de las mejores prácticas para el framework Spring. Antes de migrar la aplicación se identificaron los crosscutting concerns presentes en el sistema, realizando un análisis del fan-in [10] del código fuente con la herramienta FINT. El análisis dio como resultado la presencia de tres crosscutting concerns: chequeo de valores nulos, validación de restricciones en operaciones CRUD (crear, obtener, actualizar y borrar) y conversión de strings a tipos simples y viceversa. A continuación se realiza una breve descripción de cómo dichos concerns fueron migrados.

```
public class AccountController {
  // ...
  @RequestMapping(value = "/account", method = RequestMethod.POST)
  public String create(@ModelAttribute("account") Account account,
      BindingResult result) {
    Assert.notNull(account, "Account must be provided.");
    for (ConstraintViolation<Account> constraint : validator.validate(
        account)) {//...} //...}}
```

Figura 1. Concern de chequeo de valores nulos.

Concern Chequeo de Valores Nulos El método de mayor fan-in encontrado fue el método estático notNull() de la clase Assert, el cual tenia un total de 21 referencias distribuidas en 20 métodos de diferentes clases. El mismo comprueba que una variable dada no sea nula, y en caso de serlo arroja una excepción en

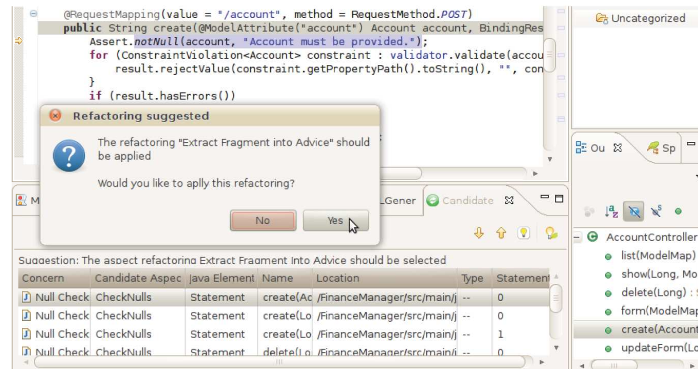

Figura 2. Refactorización mediante AspectRT.

tiempo de ejecución con el mensaje indicado. Todas las invocaciones se realizan como primera sentencia. Un ejemplo es la clase AccountController (Figura 1).

Para poder mover todas las verificaciones de valores nulos a un aspecto para evitar la repetición de código la herramienta AspectRT sugirió el uso del refactoring Extract Fragment into Advice (Figura 2). Luego, la herramienta asiste al desarrollador durante la aplicación del refactoring generando el código Spring/AOP correspondiente. De esta manera, se obtiene un aspecto con las comprobaciones de parámetros nulos, que antes estaban dispersos en el código fuente (Figura 3).

```
@Aspect
class CheckNulls {
  @Pointcut("(execution(* net.stsmedia.financemanager.web.mvc.*
      Controller.*(..,Long,..))) && args(id,..)")
  public void onControllerOpId(Long id) {}
  @Before("onControllerOpId(id)")
  public void checkNull(Long id) {
    Assert.notNull(id, "identifier must be provided."); }
  @Pointcut("(execution(*net.stsmedia.financemanager.web.mvc.*Controller
      .*(..,@Entity*,..))) && @args(ann,..) && args(value,..)")
  public void onControllerOp(Entity ann, Object value) {}
  @Before("onControllerOp(arg, param)")
  public void checkNull(Entity arg, Object param) {
    Assert.notNull(param, arg.name()+"must be provided."); }}
```

Figura 3. Aspecto para el concern chequeo de valores nulos.

Para poder migrar el concern chequeo de valores nulos se aplicaron 21 refactoring Extract Fragment into Advice y luego se necesitó realizar modificaciones en el aspecto resultante. El concern estaba disperso en 20 métodos de 3 clases y se pudo migrar completamente.

Concern Validación de Restricciones en Operaciones CRUD Se detectaron varios métodos con la misma cantidad de invocaciones y todas desde alguno de los tres controladores web presentes en el sistema. Los valores altos de fan-in se encontraban agrupados en métodos create o update y realizaban un conjunto de validación de restricciones de integridad sobre la entidad a crear o actualizar.

Se detectaron 10 métodos del tipo create o update, distribuidos en 3 clases implementando dicho concern. Para poder refactorizar el código es necesario modularizar el concern completo en un aspecto sacando de las clases involucradas los métodos create y update que realicen la validación de restricciones junto con una variable de instancia que pertenece al concern. Para esto se utilizó el refactoring Move Field from Class to Inter-Type (una vez por cada variable de las 3 clases), el cual creó un aspecto con la variable de instancia repetida en las 3 clases. Luego, se aplicó 10 veces el refactoring Extract Fragment into Advice creando un advice en el aspecto para representar los métodos involucrados en el concern. En este caso también se necesitó una reestructuración del aspecto resultante.

Concern Conversión a String Otro concern encontrado fue en las clases que heredaban de PropertyEditorSupport las cuales verifican que el valor a obtener o fijar no sea nulo, y luego convierten este valor a un objeto. Dos clases implementan dos métodos cada uno relacionado a dicho concern. Para extraer el concern se aplicó 4 veces el refactoring Extract Method into Advice, para poder extraer el código de cada uno de los métodos involucrados.

Resultados de la Migración Se calcularon un conjunto de métricas sobre el sistema original y sobre el sistema resultante luego de migrar los 3 concerns identificados (Tabla 1). Para el cálculo de las métricas sólo se tuvieron en cuenta aquellas clases afectadas por la migración.

Métricas de Finance Manager	Previo a migrar	Posterior a migrar
Número de Paquetes (Afectados[a]/Total[b])	2/9	2/9
Número de Módulos (Afectados/Total)	5/31	8/34
Líneas Estimadas de Código (ELOC) (Afectadas/Total)	370/1081	318/1029
Líneas Estimadas de Código por Módulo (Afectadas/Total)	74/34	39/30
Promedio de Complejidad Ciclomática (CC)	1.87	1.41
Promedio de Operaciones Ponderadas por Módulo (WOM)	14.20	9
Promedio de Respuesta para un Módulos (RFM)	8.60	7
Promedio de Falta de Cohesión en Operaciones (LCOO)	2.62	1.27

[a]Afectados: métrica calculada sobre módulos modificados por refactorings.
[b]Total: métrica calculada sobre todo el sistema.

Tabla 1. Comparación de métricas antes y después de migrar los concerns.

Las lineas de código fuente disminuyen considerablemente, al igual que la complejidad general (Figura 4) (Tabla 1). También se observa una disminución

importante en el acoplamiento (métrica RFM) y una mejora en la cohesión del sistema de acuerdo a la disminución de la última métrica (LCOO). El número de métodos y la complejidad de los métodos involucrados, representado por la métrica WOM, es un indicador del tiempo y el esfuerzo que se requeriría para desarrollar y mantener una clase y como puede verse en la comparación se reduce considerablemente, indicando también una mejora en la complejidad.

Debido a que la aplicación es open source se cuenta con el código fuente y una suite de casos de test de unidad. De manera tal de corroborar que la aplicación resultante era equivalente a la original se ejecutaron dichos tests. Esta validación otorga un mayor grado de validez sobre los refactorings efectuados y su preservación de comportamiento.

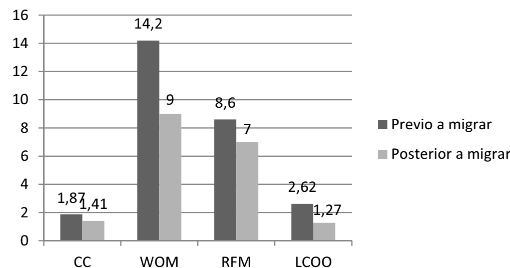

Figura 4. Gráfica de comparación de métricas antes y después de migrar los CCC.

3.2. Sistema GridGain

GridGain es una plataforma Java para el desarrollo de aplicaciones de "Cloud Computing" que proporciona a los desarrolladores de tecnologías para desarrollar y ejecutar aplicaciones de servicios por internet. El análisis de fan-in sobre su código fuente dio como resultado un conjunto de concerns que deberían ser modularizados en aspectos: Logging, Bean Management, Marshalling, Argument Checks, Before/After Calls y Bean Registration. Para la migración de dichos concerns se utilizó también la extensión de AspectRT. El proceso de refactorización aplicado fue similar al del caso de estudio descripto anteriormente.

A pesar que todos los concerns identificados requerian ser modularizados con aspectos, no fue posible en todos los casos, Marshalling y Argument Checks no pudieron ser migrados a la orientación a aspectos. Marshalling es el proceso por el cual se convierten los objetos en un formato apto para ser enviado a través de la red cuando se requiere de comunicación entre los nodos de la misma. Se identificaron alrededor de 63 invocaciones a métodos relacionados con este concern distribuidos en diferentes clases. A pesar de la gran cantidad de seeds involucrados no fue posible de realizar la migración del mismo. Esto se debió a que las invocaciones se encuentran mezcladas en diversas clases y métodos

intercaladas con otros concerns, pero sin un orden o comportamiento consistente. Es decir, no fue posible encontrar un patrón de aplicación del concern, tampoco se encontró una manera de refactorizar el código orientado a objetos original como para que permita encontrar dicho patrón.

El concern Argument Check involucra métodos para verificar que los parámetros pasados a un método no son nulos, están dentro de un rango, o no cumplen alguna condición. A pesar que se identificaron varios métodos e invocaciones a métodos para implementar dicho concern, tampoco fue posible identificar una consistencia en cuanto a los lugares o momentos en que se utilizan.

Los cuatro concerns restantes pudieron ser refactorizados con éxito. Para la implementación de estos 4 concerns había en el código original 131 clases y 266 métodos involucrados. Se aplicaron en total 208 veces el refactoring Extract Fragment into Advice, 4 veces el refactoring Move Method form Class to Inter-type, 2 veces Move Field from Class to Inter-Type y una vez Encapsulate Implements with Declare Parents. En la mayoría de los casos fue necesario realizar una modificación del código del aspecto resultante.

Con respecto a la comparación entre el código orientado a objetos original y el orientado a aspectos resultante los resultados pueden observarse en la Tabla 2. Un análisis de estos valores muestra una reducción de 370 líneas de código sobre

Métricas de GridGain	Previo a migrar	Posterior a migrar
Número de Paquetes (Afectados[a]/Total[b])	52/119	53/120
Número de Módulos (Afectados/Total)	55/479	58/482
Líneas Estimadas de Código (ELOC) (Afectadas/Total)	18061/46961	17691/46591
Líneas Estimadas de Código por Módulo (Afectadas/Total)	328/98	305/96
Promedio de Complejidad Ciclomática (CC)	2.98	2.89
Promedio de Operaciones Ponderadas por Módulo (WOM)	32.18	30.74
Promedio de Respuesta para un Módulos (RFM)	15.22	13.82
Promedio de Falta de Cohesión en Operaciones (LCOO)	78.82	75.92

Tabla 2. Comparación de métricas antes y después de migrar los concerns.

el sistema original y por ende una reducción en la cantidad de líneas por clase. A la vez que se ven reducidos los valores de complejidad ciclomática promedio del sistema y WOM, lo que implica la disminución de necesidad de testing y un aumento de la comprensibilidad en general. También se reduce el acoplamiento como indica el valor de RFM, lo que resulta en una mejor reutilización y una menor sensibilidad a los cambios en otras partes del diseño, mejorando el mantenimiento.

Por el tipo de modularizaciones realizadas, donde se extrajeron concerns secundarios de las clases originales, es de esperar una mejora en la cohesión general. Esto se ve claramente reflejado en la disminución del valor de LCOO que indica una reducción en la falta de cohesión del sistema, es decir una mejora en la cohesión que implica simplicidad y mayor reusabilidad.

4. Trabajos Relacionados

La mayoría de los trabajos de migración de sistemas orientados a objetos a sistemas orientados a aspectos se basan en el lenguaje AspectJ. En esta línea, [5] propone la refactorizacion de los crosscutting concerns hallados en la implementación de diversos patrones de diseño [3]. [13] presenta un enfoque para la refactorización de interfaces aspectizables. De forma similar a nuestro trabajo, este enfoque utiliza un conjunto de aspect refactorings para realizar las reestructuraciones. [9] propone una estrategia de refactorizacion basada en crosscutting concerns sorts. En este caso primero se describen los concerns como concerns sorts para luego refactorizarlos interactuando con el desarrollador. En este grupo de investigaciones basadas en el lenguaje AspectJ también se han presentado casos de estudios de la migración de aplicaciones empresariales [10,11].

Finalmente, algunos trabajos describen refactorizaciones orientadas a aspectos utilizando Spring/AOP. [8] describe el uso de la orientación a aspectos mediante Spring/AOP en algunos ejemplos concretos. Sin embargo, no se presentan estrategias de refactoring que puedan ser aplicadas para migrar aplicaciones orientadas a objetos a aplicaciones orientadas a aspectos. De forma similar, [4] presenta la implementación en Spring/AOP de crosscutting concerns como logging, seguridad y transacciones.

5. Conclusiones

La adopción exitosa de cualquier tecnología requiere que se tome un enfoque progresivo, con el fin de evitar un riesgo excesivo, y que tenga un costo inicial mínimo y ofrezca beneficios en cada etapa de su adopción. En aplicaciones empresariales basadas en el Framework Spring, incorporar la programación orientada a aspectos utilizando Spring/AOP es una manera libre de riesgos para introducir los conceptos de AOP. Pero incorporar estos nuevos conceptos en un proyecto existente, modularizando manualmente en aspectos aquellos crosscutting concerns presentes, sigue siendo un proceso complejo y propenso a errores. Mediante el uso de Spring y AOP se puede simplificar este proceso.

Para la implementación de refactorings semi-automáticos en una herramienta, se utilizó como base el catálogo de refactorings propuesto por Monteiro. Dicho catálogo genera código AspectJ por esta razón se adaptaron para generar código Spring/AOP. Luego, se incorporaron en un proceso iterativo de migración como parte de una herramienta para el ambiente de desarrollo Eclipse, con lo cual se logró la capacidad para migrar, con mayor facilidad, los crosscutting concerns presentes en sistemas existentes basados en Spring. Esto quedó demostrado con el desarrollo de los casos de estudio, dónde tanto para un sistema pequeño de ejemplo como Spring Finance Manager o para un sistema de gran complejidad como GridGain se lograron migrar mediante las herramientas desarrolladas la mayoría de aquellos crosscutting concerns encontrados en el código fuente. Con esta migración se logró mejorar la modularización de ambos sistema y sus atributos de calidad, lo que quedó demostrado con el cálculo de métricas de calidad y su posterior comparación antes y después de realizar la migración.

Las transformaciones a aspectos tienen muchas restricciones con respecto a la estructura y el patrón del código crosscutting, y sólo se puede aplicar directamente en determinadas circunstancias. Este requisito de uniformidad rara vez se puede esperar en un sistema grande, en especial en proyectos que han evolucionado durante un largo período de tiempo con múltiples programadores involucrados en su mantención. Sin bien en los sistemas utilizados como casos de estudio no se presentó en forma notable este problema, es de esperar que en sistemas que cumplen con las características descriptas anteriormente, el esfuerzo de migrar aquellos crosscutting concerns presentes en el código sea mucho mayor, por no poder detectar claros patrones de código, con lo cual se necesitaría hacer gran cantidades de refactorings de objetos previos a los refactorings de aspectos. Mayormente esto se debe a que la solución existente en objetos no es la ideal, con lo cual resulta difícil aplicar refactorizaciones a aspectos si la solución en objetos no es correcta.

References

1. Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
2. Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
3. Erich Gamma, Richard Helm, and Ralph E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1st ed., reprint. edition, 1995.
4. Ganesh Ghag. Implement crosscutting concerns using spring 2.0 aop, 2007.
5. Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37(11):161–173, 2002.
6. R. Johnson. *Expert One-on-one J2EE Design and Development.* Wiley & Sons, 2003.
7. Dhrubojyoti Kayal. *Best Practices and Design Strategies Implementing Java EE Patterns with the Spring Framework.* Apress, 2008.
8. Ramnivas Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications.* Manning Publications Co., Greenwich, CT, USA, 2nd edition, 2009.
9. Marius Marin, Arie Deursen, Leon Moonen, and Robin Rijst. An integrated crosscutting concern migration strategy and its semi-automated application to jhotdraw. *Automated Software Engg.*, 16(2):323–356, 2009.
10. Marius Marin, Arie Van Deursen, and Leon Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Trans. Softw. Eng. Methodol.*, 17(1):1–37, 2007.
11. Ali Mesbah and Arie van Deursen. Crosscutting concerns in j2ee applications. In *WSE '05: Proceedings of the Seventh IEEE International Symposium on Web Site Evolution*, pages 14–21, Washington, DC, USA, 2005. IEEE Computer Society.
12. Miguel P.t Monteiro. Catalogue of refactorings for aspectj. Technical Report UM-DI-GECSD-200402, Universidade do Minho, December 2004.
13. Paolo Tonella and Mariano Ceccato. Refactoring the aspectizable interfaces: An empirical assessment. *IEEE Transactions on Software Engineering*, 31(10):819–832, 2005.
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.