Información Tecnológica
Vol. 25(4), 185-192 (2014)
doi: 10.4067/S0718-07642014000400020

Revisión de un Enfoque Modular de Programación Orientada a Aspectos

Cristian L. Vidal[(1)], Sabino E. Rivero[(2)], Rodolfo F. Schmal[(2)] y Jenny D. Morales[(3)]
(1) Departamento de Computación e Informática, Facultad de Ingeniería, Universidad de Playa Ancha, Avenida Leopoldo Carvallo 270, Playa Ancha, Valparaíso-Chile (e-mail: cristian.vidal@upla.cl)
(2) Escuela de Ingeniería Informática Empresarial, Facultad de Ciencias Empresariales, Universidad de Talca, Campus Lircay, Avenida Lircay S/N, Talca-Chile (e-mail: srivero@utalca.cl; rschmal@utalca.cl)
(3) Facultad de Ciencias Empresariales, Universidad Autónoma de Chile, Sede Talca, 5 Poniente 1670, Talca-Chile (e-mail: jmoralesb@autonoma.cl)

Recibido Ene. 14, 2014; Aceptado Feb. 12, 2014; Versión final recibida Feb. 27, 2014

Resumen

En este trabajo se muestra el enfoque modular de programación orientada a aspectos de interfaces de puntos de unión o JPI para el lenguaje de programación Java, Se analiza también las principales diferencias y ventajas de JPI respecto a la metodología clásica de programación orientada a aspectos para Java (AspectJ) con el fin de obtener una programación orientada a aspectos de tipo modular. Adicionalmente, se propone una extensión del diagrama de clases UML para el diseño estructural y conceptual de aplicaciones JPI. Como ejemplo de aplicación, se presenta y describe un caso de programación en AspectJ y JPI, junto con un modelo de diagramas de clases UML del programa ejemplo, usando la propuesta de modelación de este trabajo. Se concluye que la propuesta de diagramas de clases UML JPI permite la definición de clases ingenuas, un elemento esencial para lograr una completa adaptación y transformación de soluciones de programación orientada a aspectos en soluciones JPI.

Palabras clave: orientación a aspectos, AspectJ, JPI, diagrama de clases UML

A Review of a Modular Approach of Aspect-Oriented Programming

Abstract

In this paper the modular approach of aspect-oriented programming named join point interfaces or JPI for the Java programming language is discussed. Also, the main differences and advantages of JPI over the classical methodology of aspect-oriented programming for Java (AspectJ) are analyzed with the aim of achieving a modular aspect-oriented programming. In addition, an extension of UML class diagram for the structural and conceptual design of JPI applications is proposed. As a study case, a programming example in AspectJ and JPI, along with a UML class diagram model of the study case, using the modeling proposed in this work, is presented. It is concluded that the proposed class diagram UML JPI allows defining an ingenuous class, an essential element for achieving a complete adaptation and transformation of aspect-oriented programming into JPI solutions.

Keywords: aspect-oriented, AspectJ, JPI, UML class diagrams

INTRODUCCIÓN

La metodología de programación orientada a aspectos (POA) (Kiczales et al., 1997) permite encapsular o modularizar las denominadas "incumbencias cruzadas no modularizables" por metodologías de programación clásicas tales como programación orientada a objetos y programación estructurada. Las incumbencias cruzadas representan funcionalidades que se esparcen como parte de las funcionalidades de módulos, por ejemplo, en los métodos de las clases de un sistema, y su naturaleza funcional se mezcla con las funcionalidades propias del módulo. De esta forma, las incumbencias cruzadas representan elementos funcionales no modularizables por metodologías de desarrollo de software tradicionales, esto es, desarrollo de software estructurado y desarrollo de software orientado a objetos. Cabe señalar que, se habla de desarrollo de software orientado a aspectos dada la adaptación de otras fases del ciclo de desarrollo de software previas para soportar principios de base de la orientación a aspectos y de esta forma encapsular las denominadas incumbencias cruzadas.

Tal y como se aprecia en trabajos previos de uno de los autores, esto es, en Vidal et al. (2011), en "Aplicación de Modelación Orientada a Aspectos" de Vidal et al. (2012), y en "Extensión y Aplicación de AspectZ a la Administración de un Sistema de Fichas de Salud Electrónicas en Chile" de Vidal et al. (2012), la autentificación de usuario así como el registro de sus acciones son ejemplos clásicos de incumbencias cruzadas las cuales se pueden ser modularizar como aspectos. Es importante señalar que gracias a esta modularización de incumbencias cruzadas de un sistema de software orientado a objetos, entonces las clases y sus métodos logran respetar el principio de única responsabilidad (Wampler, 2007), esto es, las clases y sus métodos tienen una función y propósito claros y bien definidos, y existe una consecuencia y consistencia en el comportamiento de los métodos de clases, es decir, ninguno de los métodos de una clase presenta funcionalidades no propias con el propósito central de la clase. En resumen, el principio de única responsabilidad establece que una clase debería tener una única responsabilidad, y que esa responsabilidad debe ser completamente encapsulada por la clase. Esto representa un principio fundamental de modularización y de programación orientada a objetos (Martin, 2002).

La figura 1 muestra un diagrama de casos de uso UML que representa un sistema y dos casos de uso que son ejemplos de incumbencias cruzadas, usualmente presente en sistemas de software: caso de uso 'Registrar Acciones' y caso de uso 'Autentificarse'. Nótese que Jacobson (2003), Jacobson (2004), y Vidal et al. (2011) indican el uso de asociaciones de extensión entre casos de uso que representan incumbencias cruzadas y casos de uso base. En sistemas de software tradicionales, es común que antes de realizar una acción, el usuario deba autentificarse, y si dicha autentificación no es efectiva, la acción no procede. Además, es usual que los sistemas de información actuales mantengan un registro de las acciones realizadas en el sistema. En este contexto, la figura 2 presenta un diagrama de clases UML tradicional para el sistema ejemplo de la figura 1.

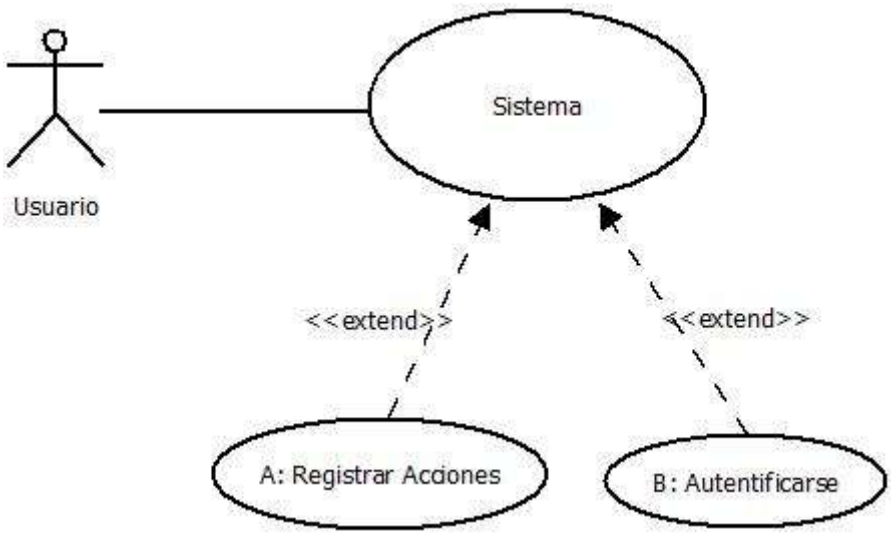

Fig. 1: Ejemplo de Incumbencia Cruzada de Registrar Acciones y Autenticarse en un Diagramas de Casos de Uso de un Sistema.

La figura 2 muestra dos clases, Clase1 y Clase2, con una asociación uno a muchos. Cada clase presenta dos atributos, y cuatro métodos. Como se aprecia, ambas clases presentan los métodos A(..) y B(..), esto es, Registrar Acciones y Autentificarse. Asumiendo que las funcionalidades y comportamiento asociado de los métodos A(..) y B(..) no son propios de Clase1 y Clase2, y como estas funcionalidades no pueden ser

representadas en clases independientes de manera que Clase1 y Clase2 respeten el principio de única responsabilidad, entonces los métodos A(..) y B(..) son ejemplos de incumbencias cruzadas.

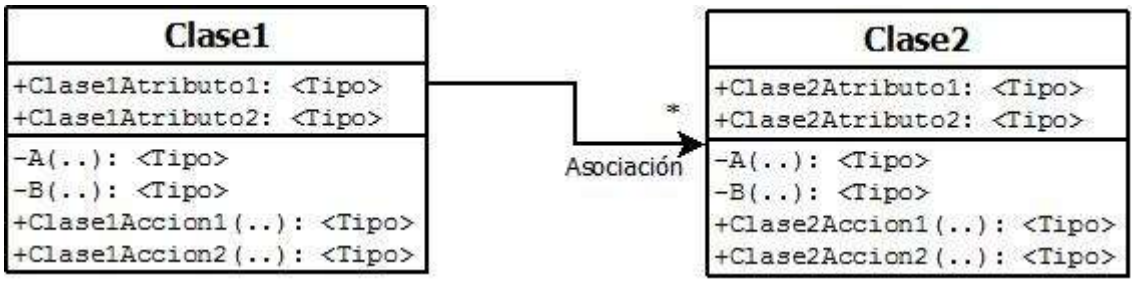

Fig. 2: Ejemplo de Diagrama de Clases UML con Incumbencias Cruzadas.

La figura 3 presenta el código Java asociado al diagrama de clases UML de la figura 2. Como se aprecia en este código de las clases Clase1 y Clase2, los métodos públicos de estas clases invocan explícitamente la ejecución de los métodos A(..) y B(..), lo que, se puede considerar, no corresponde a la esencia de la responsabilidad de los métodos públicos de Clase1 y Clase2. Es decir, ni las clases ni los métodos de este ejemplo respetan el principio de única responsabilidad.

```
public class Clase1 {
  public <Tipo> Clase1Atributo1;
  public <Tipo> Clase1Atributo2;

  private <Tipo> RegistrarAcciones(..){
          ...
  }
  private <Tipo> Autentificarse(..){
          ...
  }
  public <Tipo> Clase1Accion1([Arg1,., ArgN]) {
     Autentificarse(..);
     ...
     RegistrarAcciones(..);
   }
  public <Tipo> Clase1Accion2([Arg1,., ArgN]){
     Autentificarse(..);
     ...
     RegistrarAcciones(..);
  }
}
```

```
public class Clase2 {
  public <Tipo> Clase2Atributo1;
  public <Tipo> Clase2Atributo2;

  private <Tipo> RegistrarAcciones(..){
          ...
  }
  private <Tipo> Autentificarse(..){
          ...
  }
  public <Tipo> Clase2Accion1([Arg1,., ArgN]){
     Autentificarse(..);
     ...
     RegistrarAcciones(..);
  }
  public <Tipo> Clase2Accion2([Arg1,., ArgN]){
     Autentificarse(..);
     ...
     RegistrarAcciones(..);
  }
}
```

Fig. 3: Ejemplo de Programación Orientada a Aspectos – AspectJ.

Tal como lo señalan Kiczales et al. (1997), POA clásica o tradicional si permite la eliminación de incumbencias cruzadas en las clases de sistemas de software orientados a objetos, gracias a la modularización como aspectos de dichas incumbencias cruzadas. Sin embargo, así como lo señalan Bodden (2011), Instroza et al. (2011), y Bodden et al. (2013), en POA tradicional aún no es posible alcanzar una completa modularización debido a la existencia de dependencias implícitas entre aspectos y clases. Para la eliminación de estas dependencias implícitas entre clases y aspectos de la POA clásica, de manera de alcanzar un software con mayor modularidad, los trabajos de Bodden (2011), Inostroza et al. (2011), y Bodden et al. (2013) presentan interfaces de puntos de unión o JPI. En este contexto, el objetivo de este trabajo es aplicar JPI para modularizar el código ejemplo de la figura 3, y de esta forma describir las principales ventajas prácticas del uso de JPI para lograr software orientado a aspectos modular sin la presencia de dependencias implícitas entre clases y aspectos de la POA tradicional con AspectJ, además de proponer elementos para la modelación estructural de soluciones JPI.

Este trabajo se organiza de la siguiente forma: la próxima sección presenta el código AspectJ y JPI del ejemplo de la figura 3, donde se muestra que en JPI las clases no son necesariamente ingenuas de la presencia y posible inclusión de nuevo comportamiento de consejos de aspectos ya que las clases definen reglas de punto de corte para puntos de unión. Posteriormente, hay una sección de modelación orientada a aspectos que presenta diagramas de clases UML para modelar el ejemplo previamente presentado. En dicha sección se propone y aplica un diagrama de clases UML para software JPI sobre ele ejemplo previo.

Finalmente, como trabajo de investigación futura, se entregan ideas de modelamiento de comportamiento mediante diagramas de secuencias UML para soluciones JPI.

PROGRAMACIÓN ORIENTADA A ASPECTOS

Programación orientada a aspectos clásica

Tal como señala Kiczales et al. (1997), algunas de las características relevantes de la propuesta original de POA son punto de unión, punto de corte, consejo, aspecto, e introducción o declaración entre tipos. Además, se diferencia entre elementos base y aspectos, los que usualmente están asociados. La relación entre aspectos y clases se define mediante puntos de corte en los aspectos para así los aspectos puedan aconsejar métodos asociados a dicho punto de corte, o bien a través de la inclusión de nuevo comportamiento y atributos en las clases, esto es, introducciones o declaraciones entre tipos.

En POA clásica, los elementos base de un sistema son ingenuos respecto a su interacción con los aspectos del sistema, así como también a la posibilidad de ser aconsejados, esto es, de experimentar comportamiento adicional. De esta forma, una clase que no espera interrupciones y cambios en su funcionalidad, puede experimentar cambios en su comportamiento. Según Bodden et al. (2013), esta es una de las dependencias implícitas que existen en la POA clásica, y de la misma forma, se encuentra en cada una de las propuestas previas de modelación orientada a aspectos.

Dada la evolución del software, así como también el desarrollo de software en equipos de desarrollo independientes, por ejemplo un grupo desarrolladores a cargo de las clases del sistema y otro grupo encargado de los aspectos del sistema, entonces es posible que la firma de algunos elementos de las clases del sistema experimenten cambios, los que, si no son comunicados al equipo encargado de los aspectos, provocan aspectos no efectivos. Por ejemplo, cuando un método de clase, aconsejado por aspectos con puntos de corte que consideran la firma del método, recibe un nuevo parámetro, entonces este cambio produce aspectos no efectivos ya que los puntos de corte se definen en función de la firma de elementos de clases base. Como lo señala Bodden et al. (2013), esta es otra de las dependencias implícitas entre clases y aspectos de la POA clásica. Bodden et al. (2013) indica que una solución es un completo conocimiento de las clases del sistema por parte de los desarrolladores de aspectos, y además exista una comunicación constante entre estos equipos de desarrolladores ante cualquier cambio en los elementos base, así como también, para indicar los elementos de clases aconsejados. Esta completa comunicación parece viable en equipos pequeños de desarrollo de software, pero para grandes equipos de desarrollo, no siempre es posible el desarrollo de software entre equipos independientes, esto es, un equipo para desarrollar elementos base y un equipo para desarrollar aspectos del sistema.

Para capturar a un objeto que ejecuta uno de sus métodos, en AspectJ, se utiliza execution(..) en la definición de puntos de corte (Laddad, 2002; Jacobson, 2004). Por ejemplo, por medio del uso de execution(tipo C2.met(..)) && this(obj1) && target(obj2), obj1 y obj2 representan al mismo objeto de clase C2 que ejecuta el método met(..). De la misma forma, tal y como señalan Laddad (2002) y Jacobson (2004), además de capturar el objeto que ejecuta un método, también es posible capturar el objeto que invoca la ejecución de un método de un objeto determinado, que podría ser el mismo. Por ejemplo, en la definición de punto de corte call(tipo C2.met(..)) && this(obj1) && target(c2), obj1 representa al objeto que invoca la llamada del método met(..) de un objeto de clase C2, mientras que obj2 representa al objeto de la clase C2 que ejecuta el método met(..).

La figura 4 presenta la solución de código AspectJ para el código ejemplo de la figura 3. Como se aprecia en esta figura, el principio de única responsabilidad se respeta en las clases Clase1 y Clase2. Cabe señalar que, en este ejemplo, los métodos, métodos Clase1Accion1(..) y Clase1Accion2(..), así como Clase2Accion1(..) y Clase2Accion2(..), de las clases Clase1 y Clase2 respectivamente, son ingenuos respecto a la inclusión del comportamiento de aspectos. Esta ingenuidad de los métodos aconsejados resulta problemático para métodos que deben preservar intacto su comportamiento original lo que representa una de las dependencias implícitas entre clases y aspectos previamente señaladas. De la misma forma, si la firma de uno de los métodos públicos de las Clase1 o Clase2 cambiara, potencialmente Aspecto1 y Aspecto2 no serían efectivos, esto es, no existiría un punto de unión, y además, no habría errores de compilación asociados. Para eliminar estas dependencias entre aspectos y clases, la siguiente subsección describe el enfoque de JPI propuesto por Inostroza et al. (2011) y Bodden et al. (2013).

JPI

La principal diferencia de JPI respecto a la POA clásica es, como su nombre lo indica, el uso de interfaces de puntos de unión como puntos intermedios de la asociación entre clases y aspectos. De esta forma, JPI

permite la eliminación de clases ingenuas ya que para que una clase sea aconsejada por un aspecto, dicha clase explícitamente exhibe una interfaz de punto de unión y define una regla de punto de corte para la ocurrencia de dicha unión. De la misma forma, los aspectos implementan las interfaces de puntos de unión, y directamente no conocen las clases que son aconsejadas. Además, un aspecto debe implementar interfaces de puntos de unión para efectivamente aconsejar clases. Por lo tanto, JPI elimina las dependencias implícitas entre clases y aspectos, por lo cual, se permite alcanzar un mayor nivel de modularización respecto a POA clásica. Cabe señalar que JPI, como extensión de AspectJ, soporta código AspectJ tradicional, de manera de facilitar la adaptación de código AspectJ a JPI. JPI, al igual que POA tradicional, permite declaración entre tipos de aspectos sobre clases del sistema, para lo cual, no se requiere de una interfaz de punto de unión explícita, es decir, las clases continúan siendo ingenuas respecto a la introducción de atributos y comportamiento en ellas por parte de los aspectos.

```
public class Clase1 {
  public <Tipo> Clase1Atributo1;
  public <Tipo> Clase1Atributo2;
  public <Tipo> Clase1Accion1([Arg1,., ArgN]) {
     ...
   }
  public <Tipo> Clase1Accion2([Arg1,., ArgN]){
     ...
  }
}
```

```
public class Clase2 {
  public <Tipo> Clase2Atributo1;
  public <Tipo> Clase2Atributo2;
  public <Tipo> Clase2Accion1([Arg1,., ArgN]){
     ...
   }
  public <Tipo> Clase2Accion2([Arg1,., ArgN]){
     ...
   }
}
```

```
public aspect Aspecto1 {

pointcut pcAutentificarse(..):
    execution(<Tipo> Clase1.Clase1Accion1(
     [Arg1, .., ArgN])) ||
   execution(<Tipo> Clase1.Clase1Accion2(
     [Arg1, .., ArgN])) ||
   execution(<Tipo> Clase2.Clase2Accion1(
     [Arg1, .., ArgN])) ||
   execution(<Tipo> Clase2.Clase2Accion2(
     [Arg1, .., ArgN])) … ;

  before(..): pcAutentificarse(..){
        … //Autentificacion
     }
}
```

```
public aspect Aspecto2 {

pointcut pcRegistrarAcciones(..):
    execution(<Tipo> Clase1.Clase1Accion1(
     [Arg1, .., ArgN])) ||
   execution(<Tipo> Clase1.Clase1Accion2(
     [Arg1, .., ArgN])) ||
   execution(<Tipo> Clase2.Clase2Accion1(
     [Arg1, .., ArgN])) ||
   execution(<Tipo> Clase2.Clase2Accion2(
     [Arg1, .., ArgN])) … ;

  after(..): pcRegistrarAcciones(..){
        … //Registrar Acciones
     }
}
```

Fig. 4: Código POA AspectJ de Ejemplo.

La figura 5 muestra una solución JPI para las clases Clase1 y Clase2, y los aspectos del ejemplo de la figura 4. Es importante notar que, además de las clases y aspectos de la figura 4, la figura 5 presenta un nuevo cuadro de código para las interfacez de punto de unión JPIAutentificarse y JPIRegistrarAcciones. Como se aprecia en la figura 5, un aspecto, para ser efectivo aconsejando a una clase, requiere implementar a alguna interfaz de punto de unión exhibida por dicha clase. Así, Aspecto1 implementa la interfaz de punto de unión JPIAutentificarse, y Aspecto2 implementa la interfaz de punto de unión JPIRegistrarAcciones la cual es exhibida por las clases Clase1 y Clase2.

Como se mencionó anteriormente, no es necesaria una interfaz de punto de unión para la inclusión de nuevos métodos y atributos, esto es, para la declaración entre tipos, entre clases y aspectos. Es decir, para la declaración entre-tipos existe una asociación directa de aspecto a clase asociados, las que en este caso son ingenuas. Cabe señalar que además, JPI permite la definición de interfaces de punto de unión globales para así permitir un estilo AspectJ de POA ya que las interfaces de puntos de unión globales permiten la definición de clases ingenuas todavía. Ejemplos de puntos de unión globales se presentan en Bodden et al. (2013).

Como se aprecia en el código JPI de la figura 5, las clases Clase1 y Clase2 exhiben explícitamente las interfaces de punto de unión JPIAutentificarse y JPIRegistrarAcciones para los puntos de unión que son la ejecución de alguno de los métodos públicos de dichas clases. De esta forma, JPI permite eliminar las dependencias implícitas entre clases y aspectos de la POA tradicional. Primero, se pueden obtener clases no ingenuas ya que ellas indican sus métodos asociados a una interfaz de punto de unión, esto es, sus

métodos que pueden ser aconsejados. Segundo, si una clase que exhibe interfaces de punto de unión, considera la firma de los métodos asociados a dichas interfaces, entonces cuando uno de estos métodos experimenta algún cambio de firma el cual no se refleja en la exhibición de interfaces de punto de unión asociadas, entonces se obtendrá un error de compilación. Es decir, el equipo desarrollador de las clases debe indicar los cambios de firma de los métodos de clase en las exhibiciones de interfaces de puntos de unión asociadas. Entonces, en desarrollos POA JPI, claramente, con una definición clara de interfaces de punto de unión, podrían existir equipos de desarrollo de clases y aspectos.

```
public class Clase1 {
  exhibits JPIAutentificarse(..):
     execution(* Clase1Accion1([Arg1, ArgN]) ||
     execution(* Clase1Accion2([Arg1, ArgN])) &&
     args(..);
  exhibits JPIRegistrarAcciones(..):
     execution(* Clase1Accion1([Arg1, ArgN]) ||
     execution(* Clase1Accion2([Arg1, ArgN])) &&
     args(..);

  public <Tipo> Clase1Atributo1;

  public <Tipo> Clase1Atributo2;

  public <Tipo> Clase1Accion1([Arg1,., ArgN]) {
     ...
   }

  public <Tipo> Clase1Accion2([Arg1,., ArgN]){
     ...
  }
}
```

```
public class Clase2 {
  exhibits JPIAutentificarse(..):
     execution(* Clase2Accion1([Arg1, ArgN]) ||
     execution(* Clase2Accion2([Arg1, ArgN])) &&
     args(..);
  exhibits JPIRegistrarAcciones(..):
     execution(* Clase2Accion1([Arg1, ArgN]) ||
     execution(* Clase2Accion2([Arg1, ArgN])) &&
     args(..);
  public <Tipo> Clase2Atributo1;

  public <Tipo> Clase2Atributo2;

  public <Tipo> Clase2Accion1([Arg1,., ArgN]) {
     ...
   }

  public <Tipo> Clase2Accion2([Arg1,., ArgN]){
     ...
  }
}
```

```
public aspect Aspecto1 {

  before JPIAutentificarse(..){
       … //Autentificacion
     }
}
```

```
public aspect Aspecto2 {

  after JPIRegistrarAcciones(..){
       … //Registrar Acciones
     }
}
```

```
jpi JPIAutentificarse(..);

jpi JPIRegistrarAcciones(..);
```

Fig. 5: Código POA JPI de Ejemplo.

DIAGRAMAS DE CLASES UML ORIENTADOS A ASPECTOS JPI

Un diagrama de clases UML representa las clases de un programa o sistema computacional, con sus atributos y métodos, junto con las asociaciones entre dichas clases (Pender, 2003). Además, un diagrama de clases UML permite etiquetar clases y asociaciones entre clases mediante el uso de estereotipos. Por ejemplo, usualmente, una interfaz de clase se representa como una clase estereotipada como <<interface>>. De esta forma, un diagrama de clases UML tradicional es apto para la representación de una solución JPI.

A continuación, se definen reglas y nombre de elementos de un diagrama con este fin:
i) Cada clase, y asociaciones entre clases, se definen de manera usual como en un diagrama de clases UML.
ii) Una interfaz de punto de unión se declara con el estereotipo <<jpi>> o <<jpi global>> dependiendo de si las clases aconsejadas exhiben explícitamente o implícitamente dicha interfaz, respectivamente. En esta propuesta de diagramas de clases UML JPI, una interfaz de punto de unión no presenta atributos y métodos, es decir, una interfaz JPI representa un método sin firma.
iii) Un aspecto, que es una clase estereotipada con <<aspecto>>, permite definir una serie de variables y métodos de aspectos, así como también permite definir métodos de interfaces de punto de unión, y declaraciones entre tipos o introducciones.

iv) Las clases pueden exhibir interfaces de puntos de unión estereotipadas con <<jpi>>. De esta forma, cuando una clase exhibe una interfaz de punto de unión, hay una asociación de la clase a una interfaz de punto de unión. El rol de la clase de dicha asociación presenta una primera línea con el estereotipo <<exhibits>> junto con la firma de la interfaz, y una segunda línea con la regla de punto de corte.
v) Una interfaz global jpi incluye una asociación hacia la clase aconsejada con una línea que indica la firma del punto de unión y otra línea con la definición del punto de corte.
vi) Los aspectos, para efectivamente aconsejar esto es, para agregar comportamiento sobre el llamado o ejecución de métodos de clases a ser aconsejadas, deben implementar interfaces de puntos de unión. Por esta razón, cada aspecto, para ser efectivo, presenta una asociación hacia las interfaces de punto de unión asociadas. El rol del aspecto en estas asociaciones es 'implementa'.
vii) Los aspectos permiten declaración entre-tipos, esto es, agregar atributos y métodos a clases existentes. Para ello, se utiliza una asociación de aspecto a clase, donde el rol del aspecto es 'agregar'.

Debido a que la extensión de diagrama de clases UML JPI propuesta considera el uso de estereotipos y palabras clave especiales, para el diseño de diagrama UML JPI se puede utilizar cualquier herramienta de diseño UML. De esta forma, es posible modelar estructuralmente un programa y sistema JPI. La figura 6 presenta una aplicación de esta propuesta de diagramas de clase UML JPI sobre el ejemplo JPI de la figura 5. Nótese que JPIInterfazA se corresponde con JPIAutentificarse mientras JPIInterfazB se corresponde con JPIRegistrarAcciones. Similarmente, AspectA se corresponde con Aspecto1 y AspectoB con aspect2 del código JPI de la figura 5. Como se aprecia en esta figura, existe una clara analogía entre el número de componentes del diagrama de clases UML JPI, y la solución de código JPI correspondiente. Justamente, revisar esta propuesta de diagramas de clase UML JPI es parte de los trabajos futuros de los autores de este trabajo.

Cabe señalar que para diagrama de clases UML de AOP tradicional, ya existen propuestas tales como Kande et al. (2002) y Liu et al. (2008) que permiten usar herramientas de modelación UML existentes.

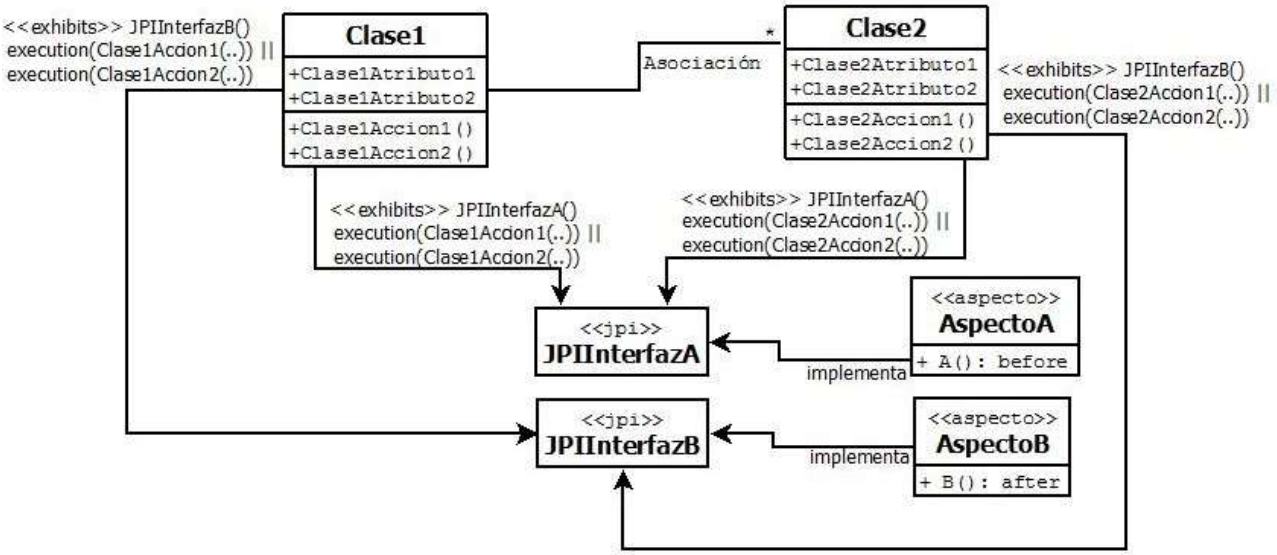

Fig. 6: Ejemplo de Aplicación Diagramas de Clases JPI.

CONCLUSIONES

JPI hace posible la generación de programas orientados a aspectos sin dependencias implícitas, lo que a su vez permite lograr un alto grado de modularización con respecto a la POA clásica. Como se menciona en este trabajo, JPI permite la definición de introducciones sin interfaces de puntos de unión, es decir, para clases ingenuas de la introducción de nuevos atributos y comportamiento, sin embargo estas clases ya no son ingenuas sobre cambios en el comportamiento de sus métodos mediante consejos de aspectos.

Para el modelamiento estructural de aplicaciones JPI, este trabajo extiende conceptos de JPI para lograr dicha modelación, es decir, este trabajo presenta una propuesta para la modelación de diagramas de clases UML para soluciones JPI. Como se presenta en el ejemplo de modelación, la propuesta de diagramas de clases UML para JPI captura elementos básicos de JPI tales como interfaces de puntos de unión, globales y no globales. Otros elementos de JPI, tales como puntos de unión cerrados y puntos de unión genéricos (Bodden, 2011; Inostroza et al., 2011; Bodden et al., 2013), son parte de extensiones futuras a esta propuesta de modelación. Además, esta propuesta de diagramas de clases UML JPI, gracias a la definición

de unidades estereotipadas con global JPI, permiten la definición de clases ingenuas, lo que es un elemento esencial para lograr una completa adaptación y transformación de soluciones POA en soluciones JPI.

Como trabajo futuro, los autores de este trabajo trabajan en una propuesta completa de modelamiento estructural de aplicaciones JPI, así como en ideas para modelar comportamiento de sistemas JPI por medio de diagramas de secuencias UML. Para esta última idea de trabajo futuro, se identifican los actores de cada escenario, donde los aspectos son claros participantes y, para los cuales, los objetos participantes se comunican en la existencia de puntos de unión, es decir, respetando las reglas de punto de corte asociadas. La idea es, modelar por medio de diagramas UML, completamente la estructura y comportamiento de soluciones JPI, de manera de trabajar en la generación de código JPI a partir de los modelos.

REFERENCIAS

Bodden, E., Closure Joinpoints: Block Joinpoints without Surprises, Actas de the 10th International Conference on Aspect-oriented Software Development, ACM, 117–128, Pernambuco, Brazil, Marzo (2011).

Bodden, E., Tanter, É. y Inostroza, M., Joint Point Interfaces for Safe and Flexible Decoupling of Aspects, por aparecer en ACM Transactions on Software Engineering and Methodology (TOSEM), 2013 (en línea). http://www.bodden.de/pubs/bti13jpi.pdf. Acceso: 24 de Noviembre (2013).

Inostroza, M., Tanter, É. y Bodden, E., Join Point Interfaces for Modular Reasoning in Aspect-Oriented Programs, Actas de ESEC/FSE ’11, European Software Engineerinsg Conference y ACM SIGSOFT Symposium on the Foundations of Software Engineering, 508-511, Szeged, Hungría, Septiembre (2011).

Jacobson, I., Use Cases and Aspects—Working Seamlessly Together, Journal of Object Technology, 2 (1), 7–28 (2003)

Jacobson, I., Aspect-Oriented Software Development with Use Cases, Addison Wesley Professional, Upper Saddle River, NJ, USA, (2004).

Kande, M., Kienzle, J. y Strohmeier, A., From AOP to UML – A Bottom– Up Approach, En las actas del Workshop on Aspect-Oriented Modelling with UML (AOSD’02), Enschede, Holanda (2002).

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. y Irwin, J., Aspect-Oriented Programming, In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Jyväskylä, Finlandia, 220-242, Junio (1997).

Laddad, R., AspecJ in Action, Practical Aspect-Oriented Programming, Manning Publications Co., Londres, Inglaterra (2003).

Liu, C., Chuang-Wen, C., A State-Based Testing Approach for Aspect-Oriented Programming, Journal of Information Science and Engineering, vol. 24, 11-31 (2008).

Martin, R. C., Agile Software Development, Principles, Patterns, and Practices, Prentice Hall, Octubre (2002).

Pender, T., UML Bible, John Wiley & Sons, Nueva York, Estados Unidos, (2003).

Vidal, C., Hernández, D., Gutiérrez, C., Meza, R. y López, L., Modelación Formal Orientada a Aspectos Usando AspectZ, Encuentro Chileno de Computación 2011, Curicó, Chile (2011).

Vidal, C., Hernández, D., Pereira, C. y Del Río, M., Aplicación de Modelación Orientada a Aspectos, Información Tecnológica, 23(1), 3-12 (2012).

Vidal, C., Del Río, C. y Saens, R., Extensión y Aplicación de AspectZ a la Administración de un Sistema de Fichas de Salud Electrónicas en Chile, Información Tecnológica, 23(5), 23-32 (2012).

Wampler, D., Aspect-Oriented Design Principles: Lessons from Object-Oriented Design, In Proceedings of Sixth International Conference on Aspect-Oriented Software Development (en línea), Vancouver, Canadá, Marzo (2007). http://aosd.net/2007/program/industry/I6-AspectDesignPrinciples.pdf. Acceso: 24 de Noviembre (2013).