Información Tecnológica
Vol. 29(1), 39-48 (2018)
http://dx.doi.org/10.4067/S0718-07642018000100039

Integración de Modelos de Análisis y Diseño de Interface de Punto de Unión JPI en la Búsqueda de un Desarrollo Modular de Software Orientado a Aspectos

Cristian L. Vidal-Silva(1)*, Trung T. Pham(2), Rodolfo H. Villarroel(3) y Andrew Philominraj(4)
(1) Escuela de Ingeniería en Informática, Facultad de Ingeniería, Ciencia y Tecnología, Universidad Bernardo O'Higgins, Avenida Viel 1497, Ruta 5 Sur, Santiago-Chile (e-mail: cristianvidal@docente.ubo.cl)
(2) Escuela de Ingeniería Informática Empresarial, Facultad de Economía y Negocios, Universidad de Talca, Av. Lircay S/N, Talca-Chile. (e-mail: tpham@utalca.cl)
(3) Escuela de Ingeniería Informática, Facultad de Ingeniería, Pontificia Universidad Católica de Valparaíso, Avenida Brasil 2241, Valparaíso-Chile. (e-mail: rodolfo.villarroel@pucv.cl)
(4) Departamento de Idiomas, Facultad de Ciencias de la Educación, Universidad Católica del Maule, Avenida San Miguel 3605, Talca-Chile. (e-mail: andrew@ucm.cl)

* Autor a quien debe ser dirigida la correspondencia

Recibido May. 24, 2017; Aceptado Jul. 27, 2017; Versión final Sep. 8, 2017, Publicado Feb. 2018

Resumen

En la búsqueda de un proceso de desarrollo de software JPI, para el modelado de soluciones JPI, este artículo presenta y aplica los lenguajes JPIAspectZ y diagramas de clases JPI UML para la especificación formal de requerimientos y el modelamiento de componentes estructurales de soluciones JPI. De esta forma se puede evaluar el nivel de hegemonía entre los productos de estos lenguajes respecto a la solución JPI equivalente en términos de clases, aspectos e instancias de JPI, para finalmente resaltar los resultados obtenidos. La principal característica de Join Point Interface (JPI) como una metodología de Programación Orientada a Aspectos (POA) es la definición de interfaces entre aspectos y clases aconsejables, característica que se ha considerado en este trabajo. El artículo presentado destaca la modularidad y consistencia entre los productos de soluciones JPI.

Palabras clave: POA, JPI, modelo JPI, JPIAspectZ, UML, diagrama de clase

Integration of Analysis and Designs Models of JPI Join Point Interface Looking for a Modular Aspect-Oriented Software Development

Abstract

Looking for a JPI software development process, for the modeling of JPI solutions, this article presents and applies the JPIAspectZ and JPI UML class diagrams languages for the formal specification of requirements and the modeling of structural components of JPI solutions. In this way, it is possible to evaluate the level of hegemony between the products of these languages regarding to the equivalent JPI solution in terms of classes, aspects and instances of JPI, to finally highlight the obtained results. The main feature of Join Point Interface (JPI) as an Aspect-Oriented Programming (AOP) methodology is the definition of interfaces between aspects and advisable classes, feature that has been considered in this work. This article also highlights the modularity and consistency among 32JPI solution products.

Keywords: AOP, JPI, JPI model, JPIAspectZ, UML, class diagram

INTRODUCCIÓN

La POA nació para modularizar mediante aspectos las denominadas incumbencias cruzadas en la fase de Programación Orientada a Objetos (POO) (Kiczales, 1996). Así, la POA direcciona una limitación central de la POO, principalmente un asunto de modularidad en módulos base cuya principal responsabilidad u objetivo se cruzan o combinan con otras incumbencias tales como seguridad, persistencia de datos, transacciones, registro de acceso, etc. (Wampler, 2016). Tal y como indican (Bodden et al., 2014; Inostroza et al., 2011), los componentes principales de POA, clases y aspectos, presentan una doble dependencia entre ellos, y como solución proponen y usan interfaces de puntos de unión o JPI. Las figuras 1 y 2 ilustran la estructura de soluciones POA y JPI, respectivamente (Bodden et al., 2014; Inostroza et al., 2011). Entonces, las soluciones POA tradicionales presentan una dependencia implícita entre sus módulos principales: 1. Módulos base ingenuos son aconsejados sin una petición explícita, además de usualmente exponer componentes privados y públicos (no se cumple el "principio de ocultamiento de la información") (Scott, 2009); 2. Puesto que aspectos usualmente se utilizan para aconsejar comportamiento, ellos dependen de la firma del comportamiento aconsejado (según la figura 1); mientras la figura 2 muestra que una solución JPI, por el uso de interfaces de puntos de unión entre módulos base y aspectos permite la eliminación de las dependencias implícitas descritas del POA tradicional.

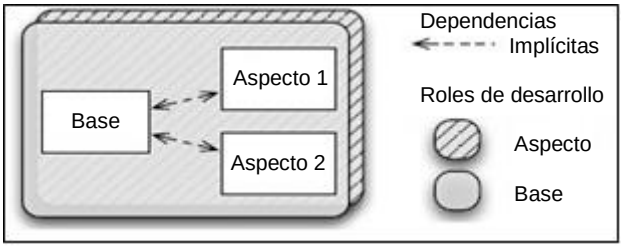

Fig. 1: Asociaciones entre módulo base y aspectos en POA tradicional.

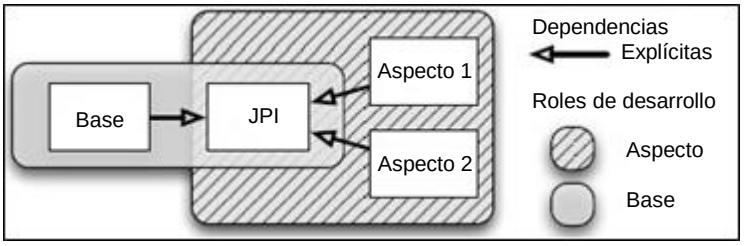

Fig. 2: Asociaciones entre modulo base y aspectos por medio de una interfaz de punto de unión en JPI

En consideración de la modularidad posible de obtener en el desarrollo de soluciones JPI en comparación a soluciones POA tradicionales, y en la búsqueda de la obtención de soluciones JPI consistentes con una transparencia de artefactos, esto es, una consistencia y transparencia entre conceptos y objetos en las fases de un ciclo de desarrollo de software JPI, entonces este trabajo presenta y ejemplifica el uso de JPIAspectZ, un lenguaje para la especificación formal de requerimientos de aplicaciones JPI, y de JPI diagrama de clases UML, una extensión de UML para modelar la estructura de soluciones JPI, junto con resaltar la consistencia entre dichos modelos.

PROGRAMACION ORIENTADA A ASPECTOS

A continuación, se describe la metodología de programación orientada a aspectos estilo AspectJ y la metodología de programación orientada a aspectos JPI, respectivamente.

POA estilo AspectJ

La Programación Orientada a Aspectos permite la modularización de incumbencias cruzadas en soluciones orientadas a objetos con el uso de aspectos (Kiczales, 1996). Los aspectos aconsejan clases según la ocurrencia de eventos (puntos de unión) de acuerdo a reglas de puntos de corte que los mismos aspectos definen. De esta forma, en POA, las clases son ingenuas de la presencia e interacción con aspectos. Tal y como remarcan (Kiczales, 1996; Bodden et al., 2014), las incumbencias cruzadas usualmente se corresponden con métodos ortogonales que se requieren en diferentes clases pero que no son parte de la naturaleza misma de dichas clases. Dentro de los beneficios de POA, el trabajo de (Wampler, 2007) ilustra como mediante soluciones POA es posible cumplir con buenos principios de diseño orientado a objetos

(Martin, 2003) tales como el “principio de única responsabilidad” y el “principio de abierto-cerrado”. Además, junto con modularizar incumbencias cruzadas estáticas tales como la inyección de nuevos atributos y métodos en clases aconsejadas, AO permite modularizar incumbencias cruzadas dinámicas avanzadas, esto es, aconsejar según un flujo de ejecución (uso de operandos *cflow*, *if*, *execution* entre otros para la definición de puntos de corte en AspectJ (Eclipse, 2017).

Según los trabajos de (Apel et al, 2006; Apel et al., 2013), las incumbencias cruzadas se pueden diferenciar entre incumbencias cruzadas homogéneas y heterogéneas, donde las primeras se refieren principalmente a la inyección de código y comportamiento no propio de las clases aconsejadas, mientras las segundas se refieren a colaboración entre clases. Justamente, el trabajo de (Apel et al., 2003; Apel et al., 2013) argumentan que una modularización de colaboración entre clases en POA usualmente no refleja la estructura de las características que se refinan ni tampoco la cohesión entre clases participantes. Además, tal y como antes se mencionó, POA estilo AspectJ introducen dependencias implícitas entre sus componentes. Claramente, en lenguajes de POA tradicional tal como AspectJ, los aspectos deben conocer acerca de las clases aconsejables antes de aconsejarlas, un gran asunto para lograr un desarrollo de software orientado a aspectos con equipos de trabajo independientes, esto es, un equipo de desarrollo para módulos base y otro equipo de desarrollo para módulos de aspectos (Bodden et al., 2014; Inostroza et al., 2011; Bodden, 2014). La figura 3 presenta un ejemplo de una clase HelloWorld con 2 métodos posibles de ser aconsejados por el aspecto BasicAspect. Como se mencionó antes, la clase aconsejable es ingenua de potenciales cambios en el comportamiento de sus métodos.

```
public class HelloWorld {

      public static void main(String[] args){
        say("Hello");
        sayToPerson("Hello", "Cristian");
    }

   public static void say(String message) {
        System.out.println(message);
    }

    public static void sayToPerson(String message,
                                   String name) {
        System.out.println(name + ", " + message);
    }
}
```

```
public aspect BasicAspect {
  pointcut callSayMessage():
     call(public static void HelloWorld.say*(..));

  before() : callSayMessage() {
     System.out.println("Good day!");
  }

  after() : callSayMessage() {
     System.out.println("Thank you!");
  }
}
```

Fig. 3: Ejemplo AspectJ de clase HelloWorld y Aspecto BasicAspect

Metodología de Programación *JPI*

Dadas las dependencias implícitas entre clases y aspectos en soluciones POA tradicionales estilo AspectJ, para lograr un aislamiento de las incumbencias cruzadas y la obtención de soluciones POA modulares sin las mencionadas dependencias implícitas, el trabajo de Inostroza et al. (2011) propone la metodología de programación JPI para la definición de una interfaz de punto de unión entre aspectos y clases aconsejables. Tal como en soluciones POA tradicionales (Kiczales, 1996), los aspectos en aplicaciones JPI representan incumbencias cruzadas, pero sin reglas de puntos de corte; los aspectos implementan las interfaces de puntos de unión junto con definir su instante de ejecución (before o antes, after o después y around o durante) en la ocurrencia de puntos de unión. Además, en aplicaciones JPI, las clases aconsejables ya no son ingenuas ya que ahora ellas explícitamente pueden exhibir interfaces de puntos de unión cuyas definiciones son análogas a las reglas de puntos de corte de POA estilo AspectJ, pero con la gran diferencia de que estas reglas ya no se definen en los aspectos si no en las clases aconsejables.

La figura 4 muestra los componentes de una solución JPI de la solución AspectJ de la figura 3, donde se aprecian la clase HelloWorld, el aspecto BasicAspect, y las interfaces de punto de unión entre dichos componentes. Los aspectos JPI estructuralmente son clases que se instancian automáticamente según su interacción con instancias aconsejables. Así mismo, los aspectos también son aconsejables, y también permiten preservar valores de sus variables de estado una vez instanciados.

LENGUAJES DE MODELAMIENTO JPI

Lograr una transparencia y consistencia de conceptos entre las etapas de análisis, diseño e implementación en el desarrollo de software JPI no parece simple, aun cuando ya existen lenguajes de modelamiento que soportan la POA tradicional; por ejemplo, diagramas de casos de uso orientados a aspectos (Jacobson y Ng,

2004; Vidal Silva et al., 2013), diagramas de clases UML orientadas a aspectos (Wimmer et al., 2011; Bustos y Eterovic, 2007; Vidal Silva et al., 2015a; y lenguajes formales de especificación de requerimientos orientados a aspectos (Vidal Silva et al., 2013; Nakajima y Tamai, 2004; Yu et al., 2005; Vidal Silva et al., 2015b). Como propuestas de solución, a continuación se presentan el lenguaje de modelamiento JPIAspectZ para la especificación formal de requerimientos de aplicaciones JPI, y el lenguaje de diseño de diagramas de clases JPI UML para el modelamiento estructural de soluciones JPI.

```
package classes;

import joinpointinterfaces.*;
import aspects.*;

public class HelloWorld {

    /////JPIs//////////
    exhibits void JPISayBefore(): call(* say*(..));
    exhibits void JPISayAfter(): call(* say*(..));
    ////////////////////////////

    public static void main(String[] args){
       say("Hello");
       sayToPerson("Hello", "Cristian");
    }

   public static void say(String message){
      System.out.println(message);
    }

   public static void sayToPerson(String message,
                                  String name){
      System.out.println(name + ", " + message);
    }
}
```

```
package aspects;

import classes.*;
import joinpointinterfaces.*;

public aspect BasicAspect {
  before JPISayBefore(){
    System.out.println("Good day!");
  }

  after JPISayAfter(){
    System.out.println("Thank you!");
  }
}
```

```
package joinpointinterfaces;
import classes.*;

jpi void JPISayBefore();
jpi void JPISayAfter();
```

Fig. 4: Solución JPI de Ejemplo AspectJ de clase HelloWorld, Aspecto BasicAspect e Interfaz de Punto de Unión.

JPIAspectZ

Los lenguajes Z (Woodcock y Davies, 1996) y Object-Z (Smith, 2000) se utilizan para la especificación formal de requerimientos de software, donde el segundo es una extensión del primero para soportar conceptos y expresiones del paradigma de POO. De manera similar, AspectZ (Vidal Silva et al., 2013; Yu et al., 2005; y OOAspectZ (Vidal Silva et al., 2015b) son extensiones para soportar elementos propios de POA y su integración con Z y Object-Z, respectivamente. Así, dados los componentes y sus relaciones de soluciones JPI, este trabajo presenta JPIAspectZ, una extensión de Z para modelar aplicaciones JPI y su integración con Object-Z. A continuación, se entregan detalles de los principales componentes de JPIAspectZ:

Módulo base: Para JPIAspectZ cada módulo base se corresponde con una clase Object-Z (Vidal Silva et al., 2015b) que, en el esquema de estado de la clase, junto con sus invariantes de estado, se permite incluir una regla *exhibits* para la definición de clase aconsejable según reglas de ocurrencia de puntos de unión. Para esto, JPIAspectZ considera elementos lógicos base de Z y Object-Z para la definición de condiciones: *call método*, *execute* método, conectores lógicos *&&*, ||, *!*, *args*(lista de argumentos) para hacer referencia a los argumentos de un método, *this*(objeto) para obtener referencia de objeto actual que solicita aconsejar a otro objeto (puede ser igual a si mismo), y *target*(objeto) para obtener referencia a objeto a ser aconsejado. Fig. 7 muestra una plantilla de esquema de clase JPIAspectZ. Nótese que OperationSchema* se refiere a la definición de múltiples esquemas de operación o métodos de clase.

Interfaces de punto de unión JPI: En JPIAspectZ, las interfaces de puntos de unión se definen en un esquema de operación de clase Object-Z cuyos nombres se inician con la palabra JPI junto a posible nombre de sistema y nombre de punto de unión, respectivamente. Además, los esquemas de puntos de unión solo presentan una sección de declaración para indicar los argumentos de la operación del punto de unión. Fig. 8 muestra una plantilla para definir esquemas de interfaz de punto de unión.

Aspectos: Los esquemas de aspectos (aspect-schemas) son similares a esquemas de clases, entonces permiten especificar esquemas de estado para definir atributos e invariantes del aspecto, esquemas de operación junto con su tipo de consejo (before, after o around) para indicar el orden sobre la ejecución o llamado de operaciones aconsejadas. El nombre de esquema de clase se inicia con la palabra 'Aspect'. Fig. 9 muestra una plantilla para la definición de esquemas de aspectos.

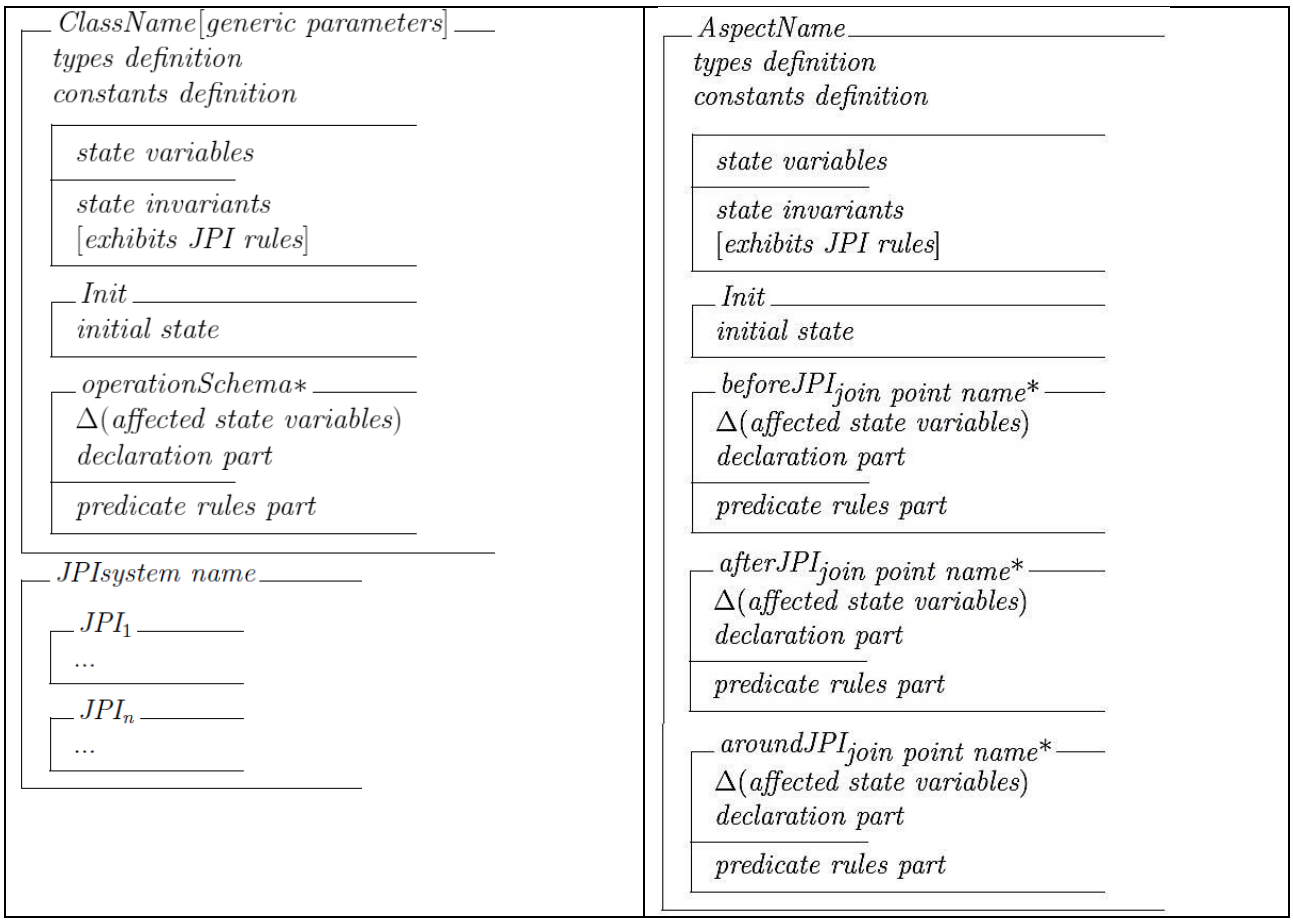

Fig. 5: Estructura y Componentes de Clase, Interfaz de Punto de Unión y Aspecto de Especificación JPIAspectZ.

Semánticamente, tal como en JPI (Bodden et al., 2014; Inostroza et al., 2011), a partir de las interfaces de puntos de unión, de las operaciones aconsejables que exhiben dichas interfaces de puntos de unión, y de los aspectos que implementan las mencionadas interfaces de puntos de unión, JPIAspectZ permite la obtención de esquemas tejidos como una integración de esquemas de los componentes ya mencionados.

Diagrama de Clases UML JPI

De acuerdo al trabajo de Pender (2003), 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. Además, un diagrama de clases UML permite etiquetar clases y las asociaciones entre clases mediante el uso de estereotipos. Por ejemplo, una interfaz de clase usualmente se representa como una clase estereotipada como <<interface>>. De esta forma, se pueden definir reglas de sintaxis y semántica sobre los diagramas de clases UML tradicionales para el diseño de modelos estructurales de soluciones JPI. A continuación, se presenta un primero conjunto de reglas para el diseño de diagramas de clases UML JPI. Cada clase y asociaciones entre clases se definen de manera usual como en un diagrama de clases UML.

i) 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.

ii) Un aspecto, que es una clase estereotipada con <<Aspect>>, permite definir una serie de atributos y métodos del aspecto, así como también permite definir métodos de interfaces de punto de unión, y declaraciones entre tipos o introducciones.

iii) Cuando una clase *A* exhibe una interfaz de punto de unión *B*, hay una asociación desde la clase *A* hacia la interfaz de punto de unión B. El rol de la clase en dicha asociación se presenta con la palabra exhibits junto con la firma de la interfaz y ":", para luego definir la regla de punto de corte.

iv) 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.

v) Los aspectos, para efectivamente aconsejar al llamado o ejecución de métodos de clases definidos, deben implementar interfaces de puntos de unión. Por esta razón, cada aspecto presenta una asociación hacia las interfaces de punto de unión asociadas para ser efectivo. El rol del aspecto en estas asociaciones es implements.

vi) 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 inter-type.

Dado lo anterior, cualquier herramienta de diseño UML es usable para el modelamiento estructural de aplicaciones JPI. Justamente, la parte superior de la figura 6 ilustra un ejemplo de diagrama de clases con incumbencias cruzadas, mientras que la parte inferior de la figura 6 muestra la versión de diagrama de clases UML JPI equivalente, donde se asume que las clases Class1 y Class2 requieren usar los métodos A y B durante (around) la ejecución del método Class1M1(..) y después (after) de la ejecución del método Class2M2(..), respectivamente.

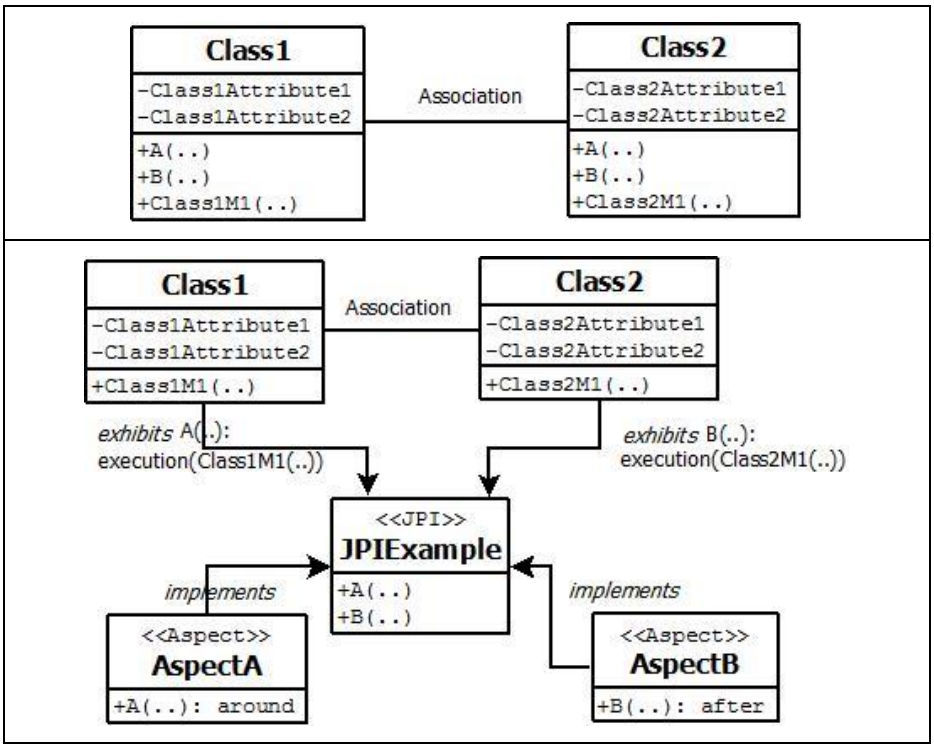

Fig. 6: Estructura de 2 Clases con Incumbencias Cruzadas y su Modelamiento con Diagrama de Clases UML JPI.

APLICACIÓN DE LA INTEGRACIÓN DE MODELOS JPIASPECTZ Y DIAGRAMA DE CLASES UML JPI

A continuación, se presenta un ejemplo de aplicación de carácter matemático, especialmente diseñada para dibujar puntos y líneas rectas en un plano cartesiano, representando el modelado en JPIAspectZ y posteriomente en diagramas de clases UML JPI. El caso de aplicación corresponde al sistema de pinturas Painting System (Vidal Silva et al., 2015b). Primero, se presenta la especificación JPIAspectZ de este ejemplo; para luego, mostrar y describir el diagrama de clase UML original de este caso de estudio y el equivalente diagramas de clase UML JPI y así resaltar la consistencia y homogeneidad entre los modelos de análisis y diseño JPI para su futura implementación.

El sistema Painting System permite dibujar puntos y líneas rectas en un plano cartesiano. En la especificación de este sistema, se indica que tanto un punto (Point) como una línea recta (Linea) son figuras (Shape), donde un punto tiene los atributos x e y, para indicar su posición en el plano cartesiano, mientras una línea recta se determina por su punto inicial y final. Las operaciones relevantes sobre un punto son establecer (setX – setY) y obtener (getX - getY) las coordenadas cartesianas de un punto a dibujar, mientras que para una línea son las operaciones propias de sus puntos. Toda figura se puede mover o cambiar la posición de un punto en el plano (moveBy), claro que mover una línea se corresponde con mover sus 2 puntos que la componen. En general, después de cualquier operación de cambio sobre una figura, esta debe ser dibujada. El diagrama JPIAspectZ de las figuras 7 y 8 son una especificación formal de este sistema. La figura 7 muestra la especificación de la interface Shape que define el método moveBy y a su vez exhibe este método para ser aconsejado. Luego se definen las clases Point y Line que implementan mediante herencia la interfaz Shape

(la clase Point por su firma entonces hereda también la regla de exhibición de la interface Shape), donde la clase Point exhibe los métodos setX y setY. De manera directa, la clase Line no es aconsejable, pero sí de manera indirecta por las operaciones sobre sus componentes.

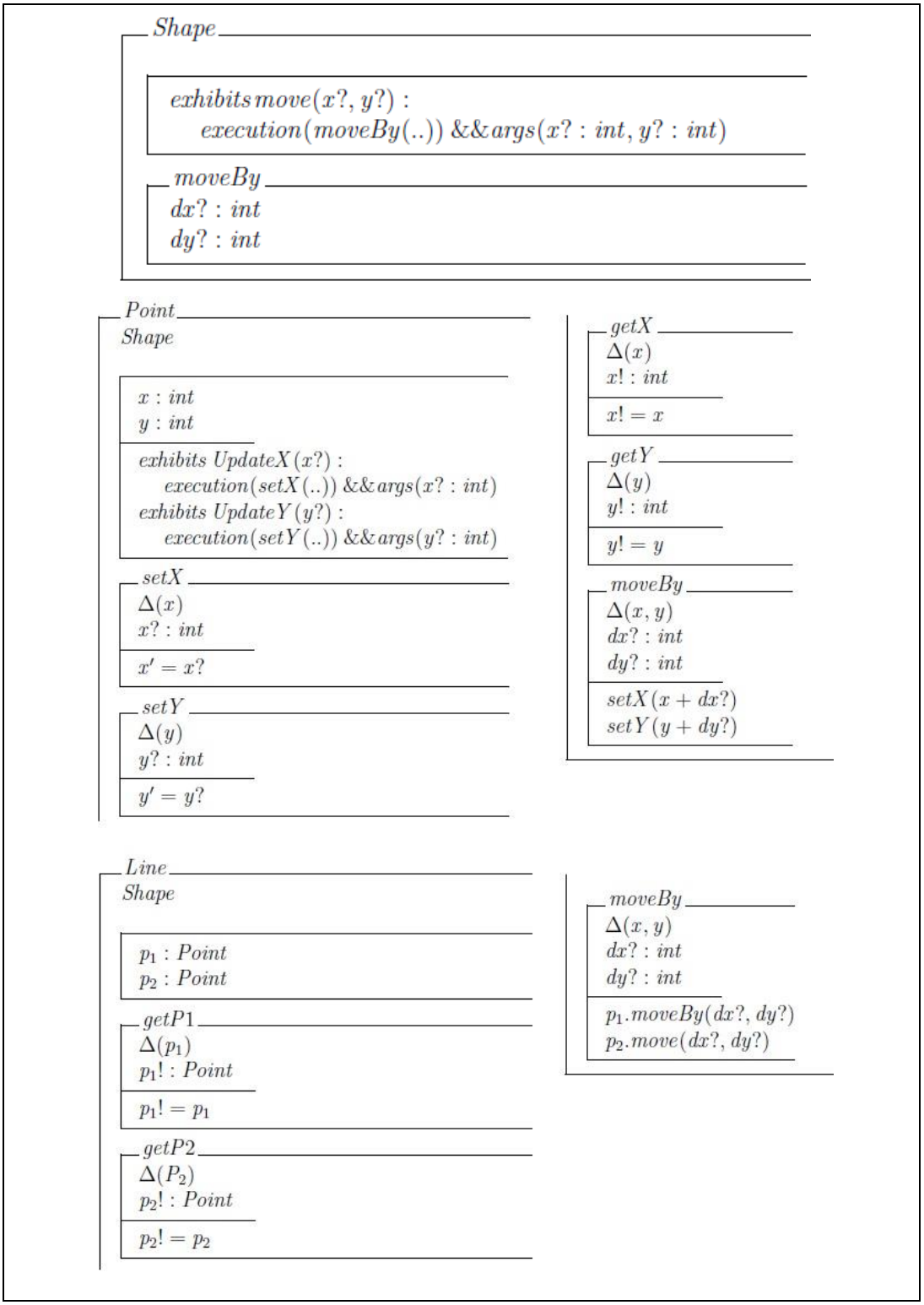

Fig. 7: Especificación JPIAspectZ de clases base de Painting System.

La figura 8 ilustra la interfaz de punto de corte JPIPainting y el método Aspect1Painting que implementa dicha interfaz después (after) de la ocurrencia de los puntos de unión UpdateX, UpdateY y move. La figura 9 muestra una versión previa de un diagrama de clases orientado a aspectos de Painting System para el modelamiento de una solución AspectJ. Aun cuando, se puede inferir que la figura 9 presenta una interfaz de punto de corte entre la clase Point y la interfaz Shape con el aspecto Change, entonces no existe una completa consistencia con su solución AspectJ ya que en AspectJ las clases son completamente ingenuas de la existencia y operaciones de los aspectos.

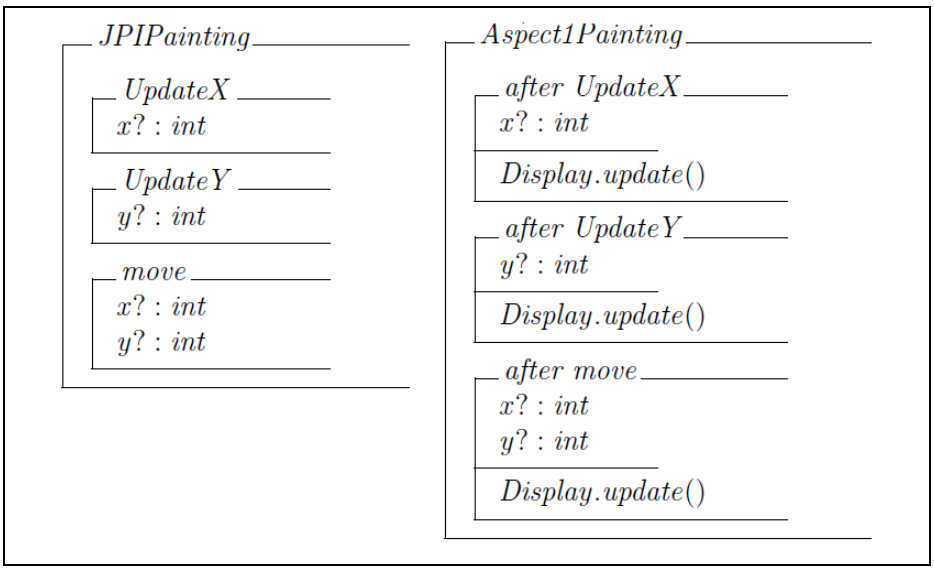

Fig. 8: Especificación JPIAspectZ de Interfaz de Punto de Unión y Aspecto de Painting System.

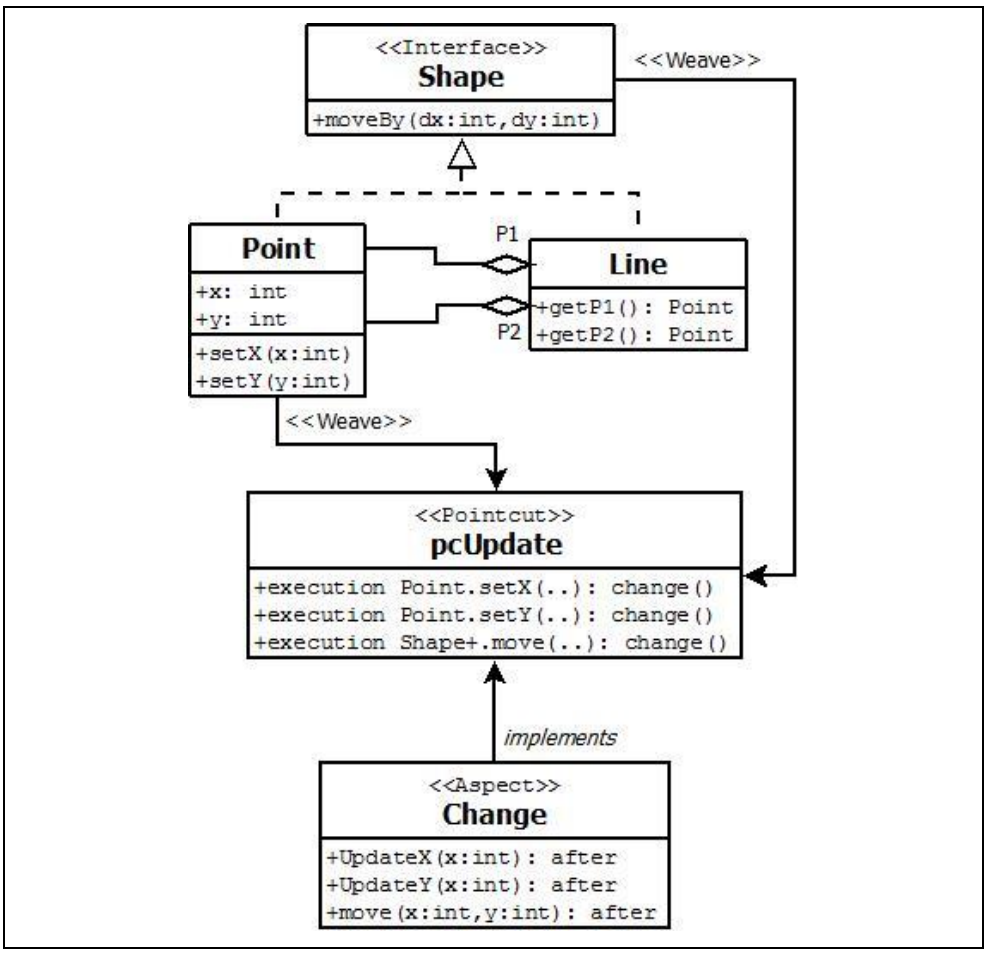

Fig. 9: Estructura de clases de aplicación Painting System.

La figura 10 muestra el diagrama de clases JPI de Painting System donde se aprecia que las clases aconsejables tal como Point y la interface Shape ya no son ingenuas ya que exhiben las interfaces de puntos de unión UpdateX(..), UpdateY(..) y move(..), mientras que el aspecto Change implementa dichas interfaces. Así, un diagrama de clases UML JPI es un claro reflejo de la estructura de una solución JPI. Existe consistencia entre los artefactos JPIAspectZ y diagramas de clase UML JPI, esto es clases, interfaces de punto de unión, aspectos y sus asociaciones, tal y como se presenta en el ejemplos de aplicación Painting System. Así, los módulos aconsejables de aplicaciones JPI ya no son ingenuos, y los aspectos no se refieren directamente a las clases y métodos a aconsejar, con lo cual se eliminan las dependencias previas entre ambos módulos, gracias a la definición de interfaces de punto de unión entre ellos.

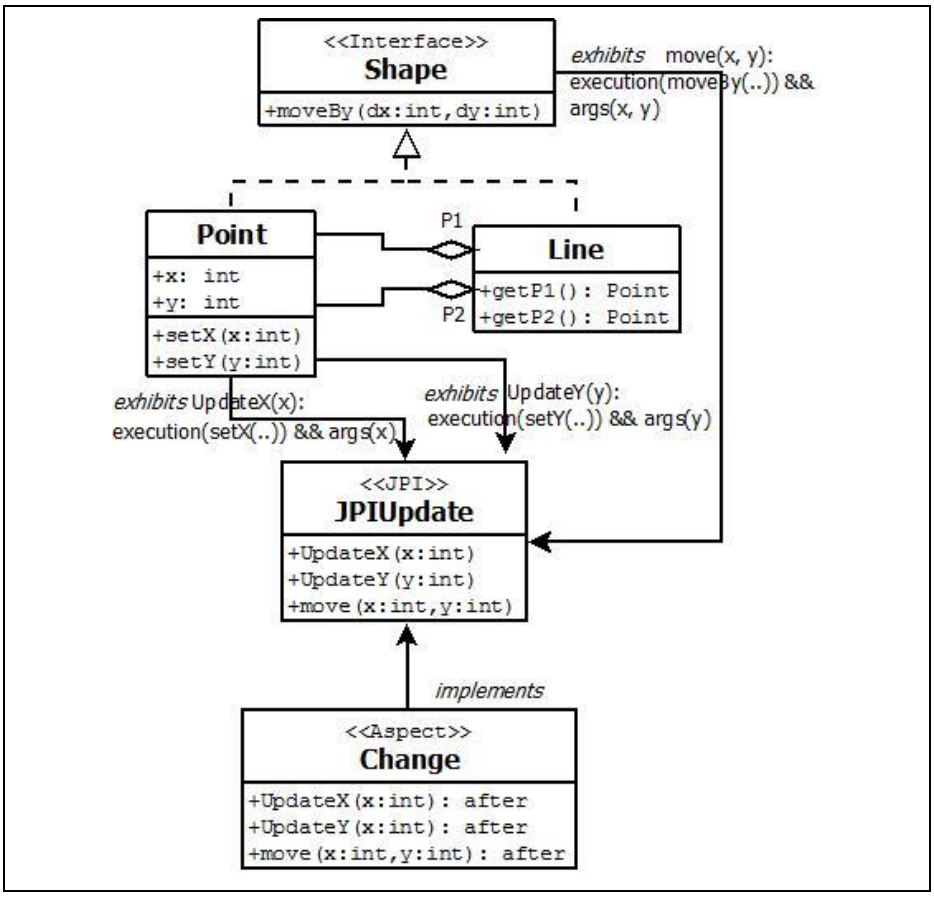

Fig. 10: Estructura de clases de JPI de aplicación Painting System.

Dada la consistencia entre los artefactos JPI, los autores buscan la extensión de nuevos diagramas UML para soportar la filosofía y componentes JPI como trabajo futuro. Así mismo, el equipo de investigación busca definir y aplicar métricas de producto y proceso para comparar cuantitativamente las ventajas y mejoras de JPI respecto a otros enfoques POA, además de trabajar en la implementación de una herramienta de validación JPIAspecctZ parte de CZT (CZT, 2016).

CONCLUSIONES

Este trabajo considera como un paso relevante, durante un proceso de desarrollo JPI, definir y preservar la consistencia de la filosofía y principios JPI entre artefactos, y así, dada la previa definición de interfaces de puntos de unión, potenciar un desarrollo entre equipos independientes, uno encargados de módulos base y otro de aspectos. Así, los lenguajes de modelamiento tales como JPIAspectZ y diagramas de clase UML JPI son adecuados en busca de un proceso de desarrollo de software orientado a aspectos modular.

Sin duda, JPIAspectZ y diagramas de clase UML JPI capturan los elementos principales de JPI para las fases de requerimientos y modelamiento estructural de aplicaciones JPI, y como este articulo resalta, es posible lograr una consistencia entre sus artefactos. Con ejemplos, se presentó que JPIAspectZ y diagramas de clase UML JPI, formalmente y conceptualmente preservan la filosofía y componentes de JPI, y que ambos lenguajes permiten la obtención de módulos base y aspectos sin dependencias implícitas entre ellos.

Gracias al uso de interfaces de punto de unión, JPI es un enfoque modular de POA ya que los aspectos implementan estas interfaces y los módulos aconsejables, tanto aspectos como clases, son quienes las exhiben; así, ya no hay dependencias implícitas entre clases y aspectos. Entonces, un proceso de Desarrollo de Software Orientado a Aspectos JPI (AOSD-JPI) parece altamente viable.

REFERENCIAS

Apel, S., D. Batory, C. Kstner y G. Saake. “Feature-Oriented Software Product Lines: Concepts and Implementation”. Springer Publishing Company, Incorporated (2013)

Apel, S., D. Batory y M. Rosenmüller. “On the Structure of Crosscutting Concerns: Using Aspects or Collaborations?” Actas de Workshop on Aspect-Oriented Product Line Engineering (2006)

Bodden, E. “Closure Joinpoints: Block Joinpoints Without Surprises,” AOSD'11, ACM, New York, NY, USA, pp. 117-128 (2011)

Bodden, E., E. Tanter y M. Inostroza. “Join Point Interfaces for Safe and Flexible Decoupling of Aspects”. ACM Transaction on Software Engineering and Methodology TOSEM, 23 (1), 1-41 (2014)

Bustos, A. y Y. Eterovic. “Modeling Aspects with UML's Class, Sequence and State Diagrams in an Industrial Setting”. SEA '07, ACTA Press, Anaheim, CA, pp. 403-410, USA (2007)

CZT. “CZT: Community of Z Tools”. En línea: http://czt.sourceforge.net/. Acceso: 17 de Diciembre (2016)

Eclipse. “The AspectJ Project”. En línea: https://eclipse.org/aspectj. Acceso: 21 de Febrero (2017)

Inostroza, M., E. Tanter y E. Bodden. “Join Point Interfaces for Modular Reasoning in Aspect-Oriented Programs”. ESEC/FSE'11, ACM, New York, NY, USA, pp. 508-511 (2011)

Jacobson, I. y P.W. Ng. “Aspect-Oriented Software Development with Use Cases”. Addison-Wesley Professional (2004)

Kiczales, G. “Aspect-Oriented Programming”. ACM Computing Survey, 28 (4es), Art. 154, Diciembre (1996)

Martin, R. “Agile Software Development: Principles, Patterns, and Practices,” Prentice Hall PTR, Upper Saddle River, NJ, USA (2003)

Nakajima, J. y T. Tamai. “Lightweight formal analysis of aspect-oriented models”. Proceedings of the 5th Aspect-Oriented Modeling Workshop in Conjunction with UML, pp. 120-127, Lisboa, Portugal (2004)

Pender, T. “UML Bible”. John Wiley & Sons, Inc., 1st Edition, New York, NY, USA (2003)

Scott, M. “Programming Languages Pragmatic”. Kauffmann Publishers Inc., 3rd Edition, pp. 132-143 (2009)

Smith, G. “The Object-Z Specification Language”. Kluwer Academic Publishers, Norwell, MA, USA (2000)

Vidal Silva, C., R. Villarroel, R. Schmal, R. Saens, C. Del Rio y T. Tigero. “Aspect-Oriented Formal Modeling: (AspectZ + Object-Z) = OOAspectZ”. COMPUTING AND INFORMATICS, 34 (5), Eslovaquia (2015a)

Vidal Silva, C., R. Villarroel, L. López, M. Bustamante, R. Schmal y V. Rea. “JPI UML Software Modeling: Aspect-Oriented Modeling for Modular Software”. International Journal of Advanced Computer Science and Application IJACSA 6 (12), Japón (2015b)

Vidal Silva, C., R. Saens, C. Del Rio y R. Villarroel. “Aspect-Oriented Modeling: Applying Aspect-Oriented UML Use Cases and Extending Aspect-Z”. COMPUTING AND INFORMATICS, 32 (3), Eslovaquia (2013)

Wampler, D. “Aspect-Oriented Design Principles: Lessons from Object-Oriented Design”. Proceedings of the Sixth International Conference on Aspect-Oriented Software Development AOSD'07, Vancouver, British Columbia, Canada, Marzo (2007)

Wampler, D. “The Role of Aspect-Oriented Programming in OMG’s Model-Driven Architecture”. Aspect Programming Inc., (2003). En línea: https://goo.gl/uLF6sY. Acceso: 17 de Diciembre (2016)

Wimmer, M., A. Schauerhuber, G. Kappel, W. Retschitzegger, W. Schwinger y E. Kapsammer, “A Survey on UML-based Aspect-oriented Design Modeling”. ACM Computing Survey, 43 (4) (2011).

Woodcock, J. y J. Davies. “Using Z: Specification, Requirement, and Proof”. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1996)

Yu, H., D. Liu, J. Jang y X. He. “Formal Aspect-Oriented Modeling and Analysis by Aspect-Z”. Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering SEKE'2005, Taipei, China, July (2005)