Información Tecnológica
Vol. 25(5), 113-120 (2014)
doi: 10.4067/S0718-07642014000500016

Propuesta y Aplicación de Diagramas de Clases UML JPI

Cristian L. Vidal [(1)], Sabino E. Rivero [(2)], Leopoldo P. López [(3)], Cristian A. Pereira [(4)]
(1) Escuela de Ingeniería Informática, Facultad de Ingeniería y Administración, Universidad Bernardo O'Higgins, Avenida Viel 1497, Ruta 5 Sur, Santiago-Chile (e-mail: cristianvidal@docentes.ubo.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)
(3) Instituto de Investigación y Desarrollo Educacional, IIDE, Universidad de Talca, Campus Lircay, Avenida Lircay S/N, Talca-Chile (e-mail: llopez@utalca.cl)
(4) Simpartners Sofware, 1601 Sherman Ave, 3rd Floor Evanston, IL 60201, USA
(e-mail: cpereira@simpartners.com)

Recibido Feb. 25, 2014; Aceptado May. 12, 2014; Versión final recibida May. 28, 2014

Resumen

Este trabajo presenta una propuesta de modelación que asocia clases y aspectos con sus interfaces de punto de unión JPI correspondientes, y se define un rol para las clases y los aspectos. Esto es, una clase exhibe interfaces JPI para las cuales define puntos de corte para instancias de puntos de unión, mientras que los aspectos implementan las interfaces JPI. Las interfaces JPI, con un estilo de programación orientada a aspectos permiten modularizar incumbencias cruzadas. Como las clases explícitamente exhiben las interfaces JPI y los aspectos ya no están acoplados a las clases y métodos como los aconsejados en AspectJ. Este trabajo propone una extensión para el diagrama de clases UML denominada diagrama de clases UML JPI con el objetivo de adaptar herramientas de modelación para soportar desarrollo de software con programación orientada a aspectos con estilo JPI.

Palabras clave: POA, JPI, incumbencias cruzadas, UML, clases, componentes

Proposal and Application of UML JPI Class Diagrams

Abstract

This article presents a modeling proposal that associates classes and aspects along with their corresponding join point interfaces JPI, and defines a role for the classes and aspects. This is, a class exhibits JPI interfaces for which defines cut point units for join points instances, whereas aspects implements JPI interfaces. These JPI interfaces, by means of an aspect-oriented programming style permit modularizing crosscutting concerns. Because classes explicitly exhibit JPI interfaces and aspects are not coupled to classes and methods as those recommended for AspectJ. This article proposes an extension for the UML class diagram name UML JPI class diagram with the objective of adapting modeling tools for supporting software that uses aspect-oriented programming with JPI style.

Keywords: AOP, JPI, crosscutting concerns, UML, classes, components

INTRODUCCIÓN

Una aplicación software o programa computacional modular es un programa que se divide en módulos con el objetivo de dividir el problema en problemas más pequeños, y dar una solución a dichos problemas, de manera que la integración de estas soluciones cumpla a cabalidad con la solución del problema original. Claramente, este enfoque de solución representa una combinación de las estrategias de diseño y modularización incremental ascendente e incremental descendente de desarrollo de software (Pressman, 2005).

Las estrategias de desarrollo de software y programación antes mencionadas son parcialmente respetadas por la metodología de desarrollo de software orientada a objetos. Sin embargo, tal como lo indica Kickzales et al. (1997), en el desarrollo de software orientado a objetos, principalmente en la etapa de programación existen incumbencias cruzadas, esto es, propiedades y métodos ortogonales a las clases del sistema, no viables de ser modularizadas y que rompen la naturaleza de los métodos y de sus clases. Por esta razón, en POO no se respeta el principio de única responsabilidad de clases y métodos del sistema (Wampler, 2007).

Tal y como presentan Vidal et al. (2012), y Vidal et al. (2013), a nivel de requerimientos de usuario, un clásico ejemplo de incumbencia cruzada lo constituye la funcionalidad A de registro de acciones, así como también la funcionalidad B de autentificación como se muestra en la figura 1. Por esta razón, en POO, los métodos de un sistema requieren invocar a las funcionalidades A y B las que usualmente son incluidas en las mismas clases de los métodos invocadores, de manera que A y B representan incumbencias cruzadas. La figura 2 presenta un ejemplo de diagrama de clases para este sistema, donde Clase1 y Clase2 están asociadas y ambas clases presentan los métodos A y B como parte de su estructura. Entonces, si se asume que los métodos A y B no son parte de la naturaleza o esencia de Clase1 y Clase2, y si además, potencialmente, los métodos de Clase1 y Clase2 requieren invocar y reaccionar ante la ejecución de los métodos A y B, respectivamente, claramente, el principio de única responsabilidad (Wampler, 2007) no se respeta en este ejemplo de sistema. Los autores de este trabajo indican que este estilo de comportamiento es usual en sistemas de información.

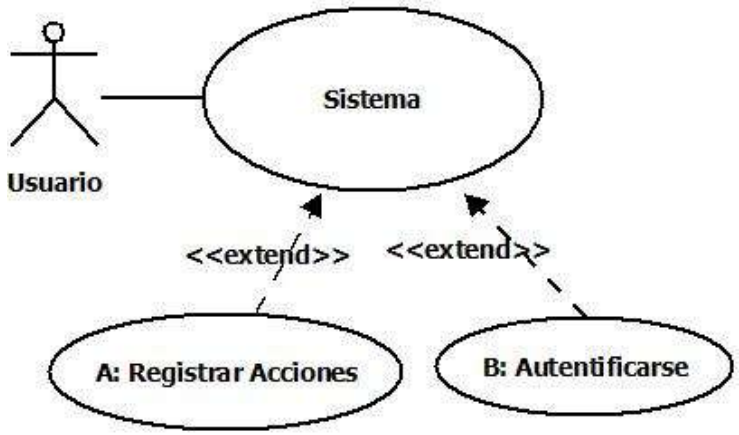

Fig. 1: Ejemplo de Incumbencia Cruzada de Registrar Acciones y Autenticarse en un Diagramas de Casos de Uso de un Sistema.

La metodología de programación orientada a aspectos permite modularizar las denominadas incumbencias cruzadas gracias a la separación de incumbencias. POA fue propuesta por Kickzales et al. (1997) como una extensión de un lenguaje de POO Java. Aun cuando POA permite obtener programas con mayor modularidad que dichos programas escritos con un lenguaje de POO, gracias a la modularización de clases y de incumbencias cruzadas como aspectos, tal y como lo indican Inostroza et al. (2011), Bodden et al. (2011), y Bodden et al. (2013), la POA clásica presenta dependencias implícitas entre clases y aspectos, principalmente porque las clases experimentan modificaciones a su comportamiento sin la existencia de una relación explícita, esto es, las clases dependen de los aspectos, y porque los aspectos dependen de la firma de los métodos de clases aconsejados para que sus consejos o funcionalidad que se agrega a los métodos de clase sean efectivos. Justamente, Inostroza et al. (2011), Bodden et al. (2011), y Bodden et al. (2013) presenta interfaces de puntos de unión entre clases y aspectos las cuales permiten eliminar estas dependencias implícitas.

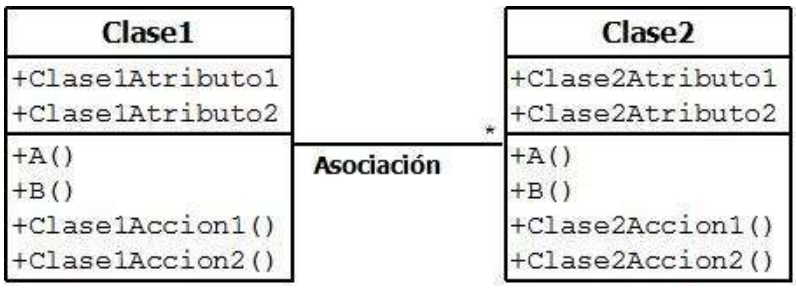

Fig. 2: Ejemplo de Pseudo-Código de Funcionalidad con Incumbencias Cruzadas.

JPI, como estilo de AOP, permite modularizar incumbencias cruzadas. Con el objetivo de adaptar herramientas de modelación para soportar un ciclo de desarrollo de software AOP con estilo JPI, este trabajo presenta una extensión para el diagrama de clases UML denominada diagrama de clases UML JPI. Para la definición de una solución JPI, como metodología de POA, se requiere definir la estructura y el comportamiento de la solución. Como los diagramas de clases UML permiten definir la estructura de soluciones POO, entonces, para definir la estructura de soluciones JPI es necesario definir diagramas de clases UML JPI. Este es el objetivo principal de este trabajo: proponer y aplicar diagramas de clases UML JPI sobre un ejemplo de estudio, y verificar la hegemonía del modelo con la solución JPI para así determinar su utilidad.

El trabajo se organiza de la siguiente forma: la siguiente sección introduce la metodología POA JP, Luego, la siguiente sección presenta el caso de estudio de sistema de compras, y su modelación con diagrama de clases UML tradicional. Posteriormente, la siguiente sección describe la propuesta de diagrama de clases UML JPI, y lo aplica a una versión extendida del caso de estudio de sistema de compras. Luego, una sección que presenta el código fuente JPI de la solución para revisar su hegemonía con su diagrama de clases UML JPI. Finalmente, una sección de conclusiones y trabajos futuros.

CÓDIGO JPI

La programación orientada a aspectos (POA) (Kiczales et al., 1997; Griswold et al., 2001; Ramnivas, 2003; Kiczales y Mezini, 2005) permite encapsular o modularizar incumbencias cruzadas, las cuales son no modularizables por metodologías de programación clásicas tales como POO y programación estructurada. Las incumbencias cruzadas representan funcionalidades ortogonales a las clases del sistema las cuales no permiten respetar el principio de única responsabilidad de clases y métodos del sistema (Wampler, 2007).

Tal como señalan Kiczales et al. (1997), Griswold et al. (2001), Ramnivas (2003), y Kiczales y Mezini (2005), punto de unión, punto de corte, consejo, aspecto, e introducciones o declaración entre tipos son algunas de las características relevantes de la propuesta original de POA (POA clásica). Además, en POA clásica, se diferencia entre elementos base y aspectos, y tal como señalan Inostroza et al. (2011), Bodden (2011), y Bodden et al. (2013), POA clásica define dependencias implícitas entre aspectos y clases de un sistema POA, con consecuencias para alcanzar una modularización y desarrollo POA óptimos.

Primero, en POA clásica, los elementos base son ingenuos respecto a su interacción con aspectos así como también de la posibilidad de ser aconsejados, esto es, de poseer nuevas propiedades y de experimentar nuevo comportamiento. Por la tanto, una clase que no espera interrupciones y cambios en su funcionalidad puede experimentar cambios en su comportamiento. Esta situación representa una dependencia implícita entre clases y aspectos en POA clásica, la que está presente en cada una de las propuestas de modelación de POA. Además, el desarrollo de software POA en equipos independientes de clases y aspectos, obliga a que los desarrolladores de aspectos tengan un completo conocimiento de las clases así como también de su privacidad para evitar cambios no deseados en dichas clases.

Segundo, en POA tradicional, dado que la relación clases y aspectos se define únicamente en los aspectos en los puntos de corte en función de la firma de elementos de clase, esto es, nombre, tipo y argumentos de métodos y atributos; entonces, nuevamente, para equipos de desarrollo de software POA independientes de clases y aspectos, para cambios en la firma de algunos elementos de las clases del sistema parte de la definición de puntos de corte en aspectos, cuando estos cambios no son comunicados al equipo de desarrollo de los aspectos del sistema, esto provoca aspectos no efectivos. Entonces, el desarrollo de software POA en equipos independientes de clases y aspectos, obliga a una completa comunicación entre los equipos de desarrollo, la cual parece viable para equipos pequeños y cercanos de desarrollo de software.

Para la eliminación de estas dependencias entre clases y aspectos de la POA clásica, así como para presentar nuevos elementos para producir software modular, los trabajos de Inostroza et al. (2011), Bodden (2011), y Bodden et al. (2013) presentan interfaces de puntos de unión o JPI. De esta forma, la principal diferencia de JPI respecto a la POA clásica es, como su nombre lo indica, el uso de interfaces de puntos de unión 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 exhibe una interfaz de punto de unión y define una regla de punto de corte para dicho punto de 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. De esta forma, JPI elimina las dependencias implícitas entre clases y aspectos, y así alcanzar un mayor nivel de modularización respecto a POA clásica. Cabe señalar que JPI, como extensión de AspectJ, soporta también código AspectJ tradicional, de manera de facilitar la adaptación de código AspectJ para soportar elementos de JPI.

CASO DE ESTUDIO: SISTEMA DE COMPRAS

Los diagramas de clases UML, para el modelamiento de software orientado a objetos, modelan los recursos necesarios para construir y operar un sistema. Principalmente, los diagramas de clases modelan cada recurso del sistema en términos de su estructura, comportamiento, y asociaciones con otros recursos. De esta forma, para sistemas de software, el modelamiento usando diagramas de clases UML representa una actividad esencial y primaria para la generación de código de sistemas (Pender, 2003).

Por ejemplo, considérese un software de ventas que preserva un registro de clientes, de los productos o ítems en stock, y de las transacciones de compra de los clientes. Cada sesión de compra incluye una o más transacciones de compra. Adicionalmente, este sistema requiere que, antes de cada transacción, la sesión de compra respectiva verifique el stock del ítem asociado. Además, cada sesión de compra aplica descuentos a las transacciones de los denominados clientes frecuentes. Entonces, cada cliente incluye un atributo privado para determinar si es o no un cliente frecuente, y un método público para obtener dicho valor. La figura 3 muestra el diagrama de clases asociado.

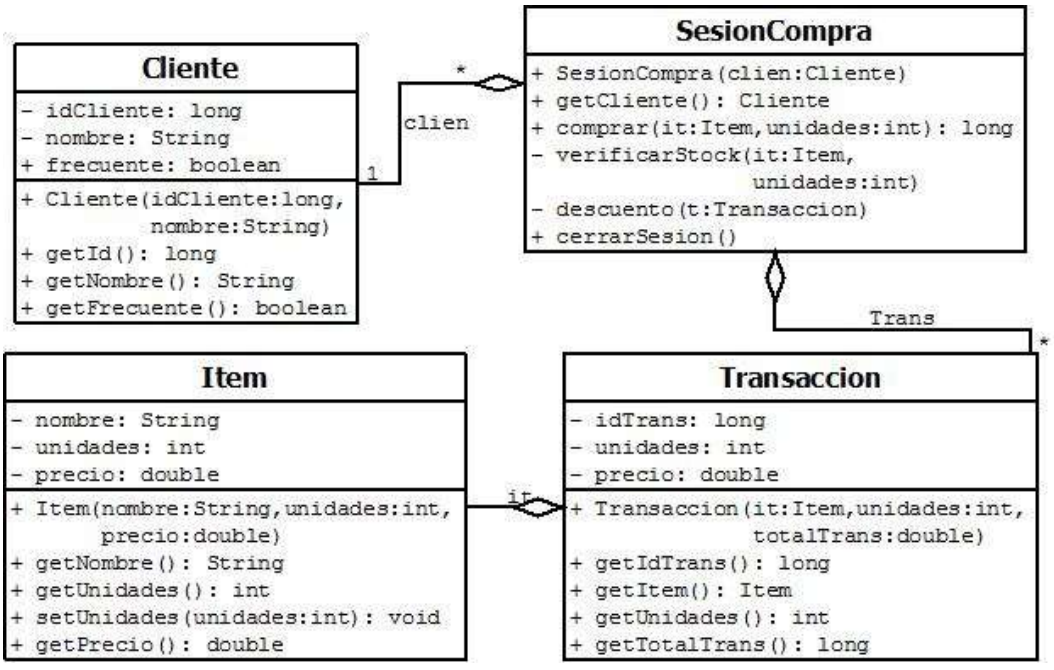

Fig. 3: Diagrama de Clases de Sistema de Compras.

Las operaciones verificarStock y descuento, se pueden considerar operaciones no propias en esencia de una instancia de SesionCompra, así como también, el atributo frecuente y método getFrecuente de la clase Cliente. Se debe considera que, cada instancia de SesionCompra, antes de proceder con la creación de una nueva transacción, debe invocar al método verificarStok. De la misma forma, al momento de crear una nueva transacción, cada instancia de SesionCompra debe potencialmente aplicar un descuento sobre la transacción para un cliente frecuente. Este operaciones en el sistema de compra representan incumbencias cruzadas. La próxima sección presenta diagramas de clases JPI, con lo que se logra separar y modularizar los elementos cruzados del sistema de compras.

DIAGRAMAS DE CLASES UML JPI

JPI define interfaces de punto de unión para la interacción de clases y aspectos. Por lo tanto, la propuesta de diagrama de clases UML JPI permite la definición de interfaces de punto de unión entre clases y aspectos, con asociaciones entre clases e interfaces de punto de unión (interfaces JPI), y entre aspectos e interfaces JPI, todas ellas en dirección hacia las mencionadas interfaces. Cada asociación entre una clase y una interfaz JPI presenta la definición de punto de corte, esto es, la definición para la ocurrencia efectiva de punto de unión, mientras que cada aspecto indica, en su rol en la asociación hacia una interfaz JPI, que el implementa dicha interfaz. Las interfaces JPI y los aspectos presentan el estereotipo <<jpi>> y <<aspecto>>, respectivamente. Como regla adicional, dado que cada aspecto, para ser efectivo, está asociado a una interfaz JPI, cada aspecto presenta al menos un método que implementa dicha interfaz JPI. Nótese que para cada método de un aspecto que implementa una interfaz JPI, recibe los atributos que indica la exhibición de dicha interfaz, cuando las clases definen puntos de corte. Además, cada método que implementa una interfaz JPI señala el tipo de consejo asociado al punto de unión, esto es, before (antes), after (después) o [Tipo] around (sobre o alrededor) donde tipo corresponde a la clase o tipo de dato que devuelve el consejo, asociado a dicho método. Un consejo de tipo around siempre devuelve el mismo tipo del método aconsejado.

El rol de una clase en su asociación con una interfaz JPI presenta dos líneas: la primera de ellas comienza con el estereotipo <<exhibit>> y luego se indica la interfaz JPI que se exhibe junto a los parámetros que esta recibe. La segunda línea corresponde a la definición del punto de corte, para el cual se utilizan elementos propios de los lenguajes de POA JPI y AspectJ tales como execution, call, y args. La figura 4 presenta un diagrama de clases UML JPI para la figura 2. Si bien es cierto, el diagrama de clases UML JPI de la figura 4 contiene mayor número de componentes que el diagrama de clases UML de la figura 2, se establecen claramente las asociaciones entre los nuevos componentes, y las clases ClaseA y ClaseB ya no presentan incumbencias cruzadas en su estructura.

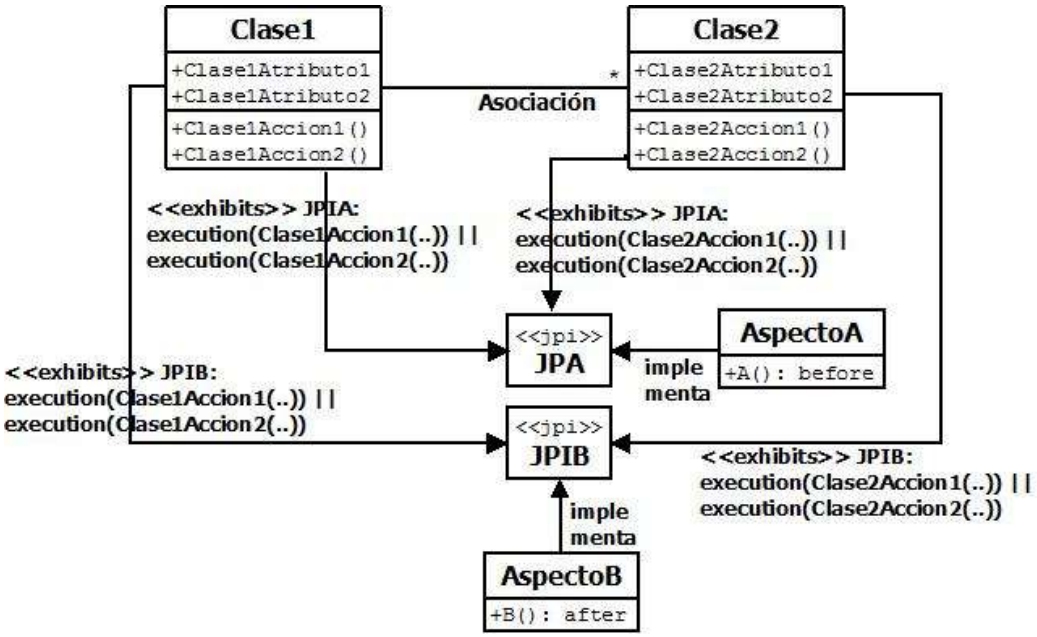

Fig. 4: Diagrama de Clases JPI de Ejemplo de Figura 1.

Para el ejemplo de sistema de compras, se considera una interfaz JPI denominada JPIVerificacionStock entre la clase SesionCompra y un aspecto PreCompra que implementa dicha interfaz JPI. En la asociación entre la clase SesionCompra y la interfaz JPI JPIVerificacionStock se define como punto de corte, la llamada al método comprar de SesionCompra, mientras que el aspecto PreCompra indica un tipo de consejo int around sobre el método JPIVerificacionStock. Nótese que el método comprar de la clase SesionCompra devuelve un int. Además, se define una interfaz JPI JPIDescuento entre la clase SesionCompra y el aspecto Descuento. En la asociación entre la clase SesionCompra y JPIDescuento se define como punto de corte una llamada al constructor de clase Transaccion, mientras que el aspecto indica sobre el método JPIDescuento un tipo de consejo Transaccion around, ya que el método aconsejado mediante la interfaz JPI corresponde al constructor de la clase Transaccion.

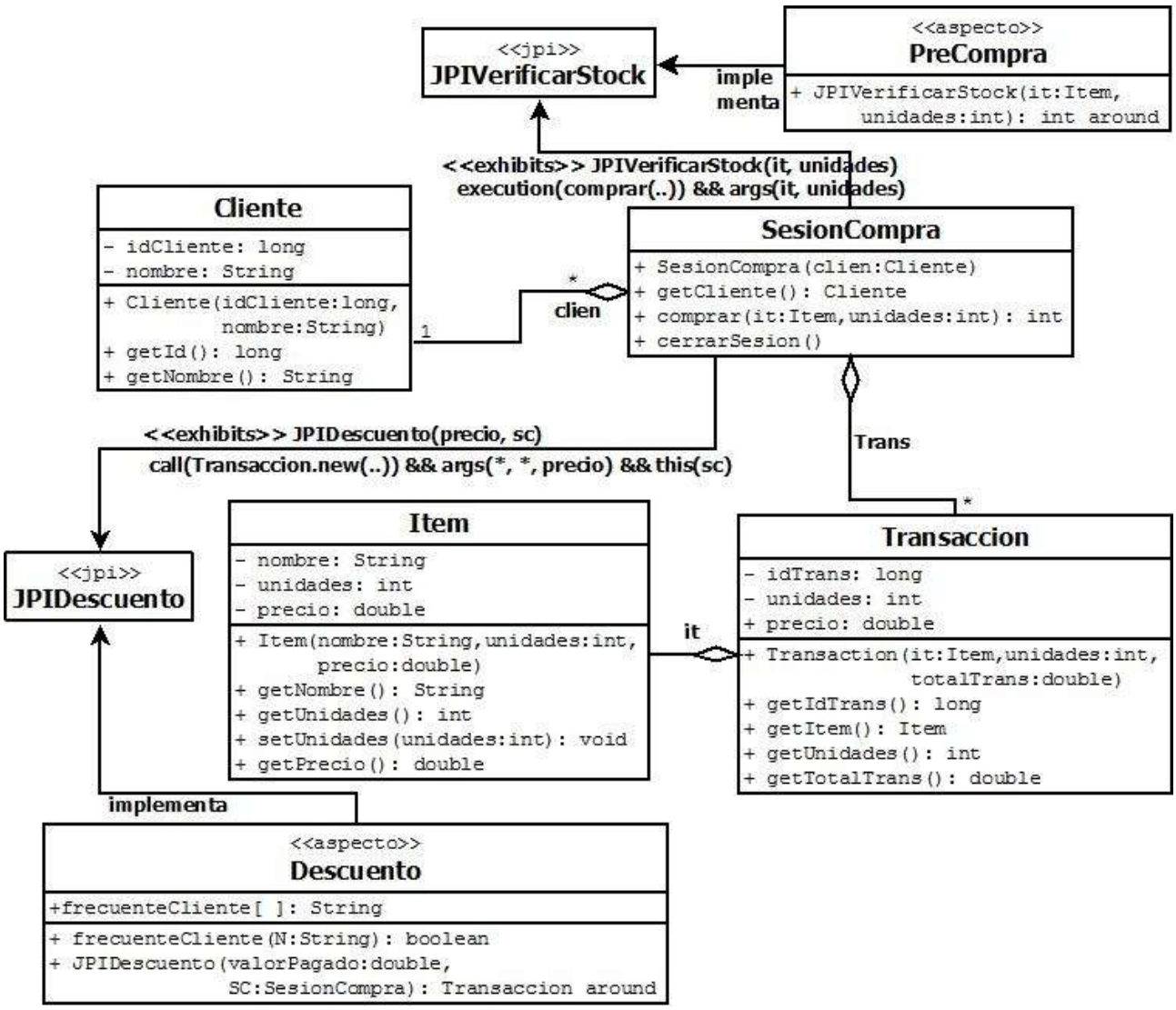

Fig. 5: Diagrama de Clases JPI de Sistema de Compras.

```
package aspects;
import classes.*;
import joinpointinterfaces.*;

public aspect PreCompra{
  Integer around JPIVerificarStock(Item it, int
unidades){
     if (it != null){
       if (unidades <= it.getUnidades())
           return proceed(it, unidades);
       }
     else
       it = new Item("item nulo", 0, 0);

     return proceed(it, 0);
     }
}
```

```
package aspects;
import classes.*;
import joinpointinterfaces.*;
public aspect Descuento {
  final String frecuenteCliente[] = {"Valeria", "Cristian"};
  boolean frecuenteCliente(String N){
    for(int i=0;i<frecuenteCliente.length; i++){
       if (frecuenteCliente[i].equals(N))
         return true;}
       return false;
  }
  Transaccion around JPIDescuento(double valorPagado,
SesionCompra sc){
    double factor = 1;
    if (frecuenteCliente(sc.getCliente().getNombre()))
       factor = 0.9;
    Transaccion T = proceed(valorPagado * factor, sc);
    return T;
  }
}
```

Fig. 6: Código de Aspectos de Sistema de Compras.

```
package classes;

public class Cliente {
  private long idCliente;
  private String nombre;
  public Cliente(long idCliente, String nombre) {…}
 public long getId(){…}
 public String getNombre(){…}
}
```

```
package classes;
public class Transaccion {
  private int idTrans;   private Item it;
  private int unidades;   private double precio;
  private static Integer actualIdTrans = 1;
  public Transaccion(Item it, int unidades, double
                                precio){...}
  public Integer getIdTrans() {...}
  public Item getItem() {...}
  public int getUnidades() {...}
  public double getTotalTrans(){...}
}
```

```
package classes;
import java.util.*; import joinpointinterfaces.*;
public class SesionCompra {
  private HashMap<Integer, Transaccion> trans;
  private Cliente clien;
  exhibits Integer JPIVerificarStock(Item it, int unidades):
    execution(Integer comprar(..)) && args(it, unidades);
  exhibits Transaccion JPIDescuento(double precio,
SesionCompra sc):
    call(Transaccion.new(..)) && args(*, *, precio) && this(sc);
  public SesionCompra(Cliente clien) {... }
  public Cliente getCliente() {...}
  public void cerrarSesion() {... }
  public Integer comprar(Item it, int unidades) {
    Transaccion T;
    Integer key;
    T = new Transaccion(it, unidades, it.getPrecio());
    key = T.getIdTrans();
    SesionCompraTrans.put(key, T);
    it.setUnidades(it.getUnidades()-unidades);
    return key;
   }
}
```

```
package classes;

public class Item {
  protected String nombre;
  protected int unidades;
  protected double precio;

  public Item(String nombre, int unidades, double
precio){...}

  public String getNombre() {...}
  public int getUnidades(){...}
  public void setUnidades(int unidades){...}
  public double getPrecio(){...}
}
```

```
package joinpointinterfaces;
import classes.*;
jpi Transaccion JPIDescuento(double precio,
SesionCompra SC);
jpi Integer JPIVerificarStock(Item it, int unidades);
```

Fig. 7: Código de Clases y Definición de Interfaces JPI de Sistema de Compras.

Para el ejemplo de sistema de compras, se considera una interfaz JPI denominada JPIVerificacionStock entre la clase SesionCompra y un aspecto PreCompra que implementa dicha interfaz JPI. En la asociación entre la clase SesionCompra y la interfaz JPI JPIVerificacionStock se define como punto de corte, la llamada al método comprar de SesionCompra, mientras que el aspecto PreCompra indica un tipo de consejo int around sobre el método JPIVerificacionStock. Nótese que el método comprar de la clase SesionCompra devuelve un int. Además, se define una interfaz JPI JPIDescuento entre la clase SesionCompra y el aspecto Descuento. En la asociación entre la clase SesionCompra y JPIDescuento se define como punto de corte una llamada al constructor de clase Transaccion, mientras que el aspecto indica sobre el método JPIDescuento un tipo de consejo Transaccion around, ya que el método aconsejado mediante la interfaz JPI corresponde al constructor de la clase Transaccion.

El rol de la clase SesionCompra en las asociaciones con las interfaces JPI JPIVerificarStock y JPIDescuento, en la definición de punto de corte, indica la existencia de puntos de unión cuando se ejecuta el método comprar de la clase SesionCompra con los argumentos it y unidades, y cuando se invoca al método constructor de la clase Transacciones donde se considera uno de sus argumentos, en este caso el precio, y se usa como argumento de la interfaz JPI la instancia de SesionCompra la que invoca al método constructor de clase Transaccion para poder obtener el cliente asociado.

La figura 5 presenta la aplicación de la propuesta de diagrama de clase UML JPI, mientras que las figuras 6 y 7 presentan el código completo de los aspectos, y el código de las clases junto con la definición de interfaces del sistema de compras, respectivamente. Nótese que el código de las clases del sistema de compras sólo presenta la firma de sus métodos. Como se aprecia en las figuras 5, 6 y 7, existe una clara analógica entre los componentes de la solución de código y los componentes del diagrama de clases UML JPI equivalente. La clase SesionCompra define (exhibe) las interfaces JPI JPIVerificacionStock y

JPIDescuento, mientras que los aspectos PreCompra y Descuento definen métodos que implementan dichas interfaces JPI, y definen además los tipos de consejo int around y Transaccion around, respectivamente. En la esquina inferior derecha de la figura 7, se muestra además la definición de interfaces JPI. Claramente, para el sistema de compra, hay una completa analogía entre el diagrama de clases UML JPI y los componentes de la solución de código JPI asociada, así como también de la asociación entre dichos componentes.

CONCLUSIONES

De lo presentado y discutido en este trabajo, se peden obtener las siguientes conclusiones principales:

i) JPI, como estilo de AOP, permite modularizar incumbencias cruzadas.

ii) JPI define interfaces de punto de unión entre clases y aspectos de manera que las clases ya no son ingenuas como en AspectJ.

iii) Se presenta una extensión para el diagrama de clases UML denominada diagrama de clases UML JPI.

iv) Existe una completa analogía entre esta propuesta de modelación estructural de soluciones JPI y su código asociado.
v) Mediante el uso de diagramas de clases UML JPI se logra una modelación de la estructura equivalente a la solución de código.

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, E. y Inostroza M., A Brief Tour of Join Point Interfaces, Actas de 12th Annual International Conference Companion on Aspect-Oriented Software Development, AOSD'13 Companion, Fukuoka, Japan, 19-22 (2013).

Griswold, B., Hilsdale, E., Hugunin, J., Isberg, W., Kiczales, G., Kersten, M.: Aspect-Oriented Programming with AspectJ™. AspectJ.org, Xerox PARC (2001).

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).

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. C., Irwin, J., Aspect-oriented programming, in proceedings of European Conference of Object-Oriented Programing ming, ECOOP97, SpringerVerlag, 220-242 (1997).

Kiczales, G., Mezini, M., Aspect-Oriented Programming and Modular Reasoning. Proceedings of the 27th International Conference on Software Engineering, ICSE '05, ACM, New York, NY, USA, 49-58 (2005).

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

Pressman, R., Ingeniería de Software: Un Enfoque Práctico, Sexta Edición, McGraw-Hill (2005).

Ramnivas, L., AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co. Greenwich, CT, USA (2003).

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

Vidal, C., Saens, R., Del Rio, C., Villarroel, R., Aspect-Oriented Modeling: Applying Aspect-Oriented UML Use Cases and Extending AspectZ., Computing and Informatics, Bratislava, Slovak, 573-593 (2013).

Wampler, D., Aspect-Oriented Design Principles: Lessons from Object-Oriented Design, Sexta Conferencia Internacional sobre Desarrollo de Software Orientado a Aspectos (AOSD'07), Vancouver, British Columbia, Marzo (2007).