Información Tecnológica
Vol. 30(3), 95-104 (2019)
http://dx.doi.org/10.4067/S0718-07642019000300095

En Búsqueda de un Procedimiento de Desarrollo de Software Modular. Simbiosis entre Programación Orientada a la Característica y Programación Orientada a Aspectos JPI

Cristian L. Vidal-Silva[1,5], Trung T. Pham[2]*, Sussan M. Sepúlveda[3] y Luis E. Carter[4]
(1) Ingeniería Civil Informática, Escuela de Ingeniería, Campus Rodelillo, Universidad Viña del Mar, Agua Santa 7055, Viña del Mar-Chile. (e-mail: cristian.vidal@uvm.cl)
(2) Facultad de Economía y Negocios, Escuela de Ingeniería Informática Empresarial, Universidad de Taca, Av. Lircay S/N, Talca-Chile. (e-mail: tpham@utalca.cl)
(3) Ingeniera de Servicios, Ingeniería, Asesoría y Servicios AyA SPA, Chile, 10 Oriente 1250, Talca – Chile.
(e-mail: ssepulvedad.ayascomputacion@gmail.com)
(4) Facultad de Ingeniería, Ingeniería Civil Industrial, Universidad Autónoma de Chile, Chile.
(e-mail: luis.carter@uautonoma.cl)
(5) 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)

* Autor a quien debe ser dirigida la correspondencia

Recibido Ago. 13, 2018; Aceptado Oct. 26, 2018; Versión final Dic. 5, 2018, Publicado Jun. 2019

Resumen

En la búsqueda de un método de desarrollo de software modular, este trabajo propone Interfaces de Punto de Unión JPI para el modelamiento colaborativo de soluciones modulares en una simbiosis JPI y Programación Orientada a la Característica, POC. Una simbiosis de POC y POA permite alcanzar las ventajas y alcances de ambos enfoques. Esto es, una alta modularidad para la colaboración heterogénea entre clases y alta repetición de comportamiento con POC. Se obtiene también, una alta modularidad para la colaboración homogénea entre clases y las dependencias implícitas entre componentes con POA tradicional. Se ejemplifica las ventajas de esta fusión en la representación modular y estructural de LPS, para destacar los beneficios de esta simbiosis. Los resultados obtenidos sólo avalan lo anterior, y constituyen la base para una metodología de desarrollo de software JPI- POC.

Palabras clave: software modular; incumbencia transversal; POC; POA*; JPI*

Looking for a Modular Software Development Methodology. Blending of Feature-Oriented Programming and Aspect-Oriented Programming JPI

Abstract

Looking for a methodology of modular software development, this work proposes JPI-FM for the collaborative modeling of modular solutions in a blending of Join Point Interfaces JPI and Feature-Oriented Programming FOP. A FOP and AOP symbiosis would allow to achieve the advantages and scope of both approaches. That is, a high modularity for the heterogeneous collaboration between classes and high repetition of behavior with FOP. Also, a high modularity for the homogeneous collaboration between classes and the implicit dependencies between components with traditional AOP, is also obtained. Examples are provided showing the advantages of this fusion in the modular and structural representation of SPL, to highlight the benefits of this symbiosis. The results only support the above statement and constitute the basis for a JPI-FOP software development methodology.

Keywords: modular software; crosscutting concern; FOP; AOP; JPI

INTRODUCCIÓN

El principio de separación de incumbencias (Dijkstra, 1968) busca la producción de programas modulares, una meta fundamental en la programación de computadores. La Programación Orientada a Objetos (POO) es una clara evolución de la programación estructurada: POO encapsula atributos y funciones como miembros de clases, y permite establecer reglas para el ocultamiento de información para el acceso a los miembros de clase, y también definir relaciones entre clases. Sin embargo, la POO no encapsula directamente las características y arquitectura de Líneas de Productos de Software (LPS) (Prehofer, 1997; Apel et al., 2013), ni tampoco modulariza de manera independiente las funcionalidades y atributos transversales de los módulos software (Kiczales, 1996). En búsqueda de soluciones modulares de software, los paradigmas de desarrollo de software Programación Orientada a la Característica POC (Feature-Oriented Programming, FOP) (Mezini y Ostermann, 2004) y Programación Orientada a Aspectos POA (Aspect-Oriented Programming, AOP) (Kiczales et al., 1997) buscan una solución a los asuntos de modularización de POO:

- POC modulariza la colaboración de módulos heterogéneos entre clases, para el desarrollo incremental de LPS (Batory, 2006; Apel et al., 2013); es decir, por el refinamiento de clases, POC define características como módulos que involucran cambios de diferentes piezas de funcionalidad donde no es simple la definición de puntos de unión. POC permite además modularizar módulos transversales estáticos; es decir, nuevas clases, nuevos atributos y métodos de clases, además de nuevas definiciones de interfaces (Apel et al., 2005b; Apel et al., 2006a; Apel et al., 2013). Sin embargo, POC no soporta una adecuada modularización de algunos módulos transversales para la evolución del software ya que requiere la evolución y cambio de diferentes módulos según eventos de cambio no predecibles (Kiczales, 1996; Apel et al., 2005b). Esto es así, particular con los denominados módulos transversales homogéneos; es decir, "POC no modulariza de manera elegante los módulos transversales homogéneos" (Apel et al., 2005b; Apel et al., 2006a; Apel et al., 2013) los que corresponden a características que representan el mismo comportamiento en diferentes lugares en la jerarquía de características. Para los módulos transversales dinámicos, POC sólo soporta métodos de interceptación (Apel et al., 2006b), es decir, en un refinamiento de clase, los métodos se pueden clasificar como refinados y no refinados.

- POA permite la definición de clases y aspectos para la modularización de funcionalidades transversales; esto es, los aspectos representan funciones y atributos ortogonales no parte de la naturaleza de las clases. Así, en busca de un comportamiento modular, según reglas de punto de corte y de punto de unión, los aspectos pueden agregar atributos y funciones no propias de la naturaleza de las clases de manera dinámica y estática (Kiczales, 1996). Así, las soluciones POA son capaces de respetar principios de buen Diseño Orientado a Objetos (DOO) tales como el principio de única responsabilidad (Wampler, 2007). Además, POA puede modularizar módulos transversales dinámicos avanzados, no sólo los métodos de interceptación. Por ejemplo, lenguajes de POA como AspectJ (Eclipse, 2016) permiten el uso de cflow, if, para la definición de puntos de corte. Sin embargo, POA permite una adecuada modularización de módulos transversales homogéneos, pero la modularización de colaboración entre clases como aspectos da lugar a una compleja evolución de software porque los aspectos no reflejan la estructura de cada característica refinada ni la cohesión de las mismas (Apel et al., 2006b). Lo anterior, sin considerar que las soluciones POA tradicional como AspectJ (Eclipse, 2016) introducen dependencias implícitas entre aspectos y clases, un gran problema para el desarrollo de software independiente (Inostroza et al., 2011; Bodden et al., 2013; Bodden et al., 2014).

Como POA y POC modularizan adecuadamente diferentes tipos de módulos transversales, homogéneos y heterogéneos respectivamente, una simbiosis de la POC y POA sería de gran beneficio para la producción de software modular (Mezini y Ostermann, 2004; Apel et al., 2006b; Apel y Kästner, 2009; Apel et al., 2013). Ya existen ya trabajos en busca de la simbiosis de POC y POA tales como (Apel et al., 2008; Apel et al., 2013) que proponen Módulos de Función Aspecto (MFA) para representar clases y aspectos modulares para su asociación y evolución. No obstante, MFA preserva los asuntos tradicionales de POA. Por los beneficios de POC para modular módulos transversales heterogéneos y estáticos de una LPS, y dada la capacidad de JPI para respetar principios de buen DOO con una adecuada modularización de módulos transversales dinámicos y homogéneos (Inostroza et al., 2011; Bodden et al., 2013; Bodden et al., 2014), los principales objetivos de este artículo son dar a conocer JPI-FM (Vidal et al., 2015) para modelar la estructura de soluciones JPI + POC en la producción masiva de software personalizado. Se aplica JPI-FM en un ejemplo simple para evaluar su modularidad.

SOFTWARE MODULAR ORIENTADO A LA CARACTERÍSTICA Y ORIENTADO A ASPECTOS

Las soluciones POA tradicionales estilo AspectJ y las soluciones POC buscan la producción de software modular (Vidal et al., 2018). A continuación, de mencionan ventajas y desventajas de estas metodologías de programación.

Ventajas y desventajas de POA.

Primero, POA tradicional en general modulariza módulos transversales de soluciones de programación en un contexto de POO, esto es, POA permite definir módulos para la contención de métodos y atributos ortogonales entre clases. Tal como señala (Wampler, 2007), en POA tradicional se respetan principios de buen diseño orientado a objetos tales como el principio abierto-cerrado y el principio de única responsabilidad, esto dado que las clases son ingenuas de la presencia de aspectos. Además, las soluciones de POA tradicional estilo AspectJ permiten la definición de reglas de punto de corte para identificar puntos de unión en la ejecución de programas, para la invocación implícita de advises o consejos, de manera que los aspectos pueden incluir propiedades y comportamiento sobre los módulos donde ocurren los puntos de unión (los módulos son aconsejados). Principalmente, mediante el uso de comodines y mecanismos de correspondencia de patrones, POA permite refinar múltiples puntos de unión por una sola regla de declaración, es decir, POA permite una buena modularización de asuntos transversales homogéneos. Además, dado que POA proporciona mecanismos definir consejos sobre programas basados en su ejecución dinámica, POA también permite un buen soporte de módulos transversales dinámicos (Apel et al., 2013).

Dado que en POA estilo AspectJ, las reglas de punto de corte indican referencias textuales a módulos a ser aconsejados, cambios en la interfaz de los módulos base a ser aconsejado pueden generar reglas de punto de corte posiblemente falsas o sin conexión, es decir, se producen aspectos no efectivos. Además, dada la mencionada dependencia, los aspectos suelen ser no reutilizables y de una evolución no simple. En el contexto del código base, claramente este puede experimentar inconscientemente cambios no deseables por la invasión de los consejos de los aspectos, posiblemente rompiendo los supuestos sobre el código base posible de ser aconsejado y no aconsejado (Bodden et al., 2014). Claramente, los lenguajes de POA clásicos tal como AspectJ violan el principio de la ocultación de la información (Apel et al., 2013), ya que un aspecto puede afectar a los miembros internos de otros módulos directamente, independientemente de las reglas de privacidad, e incluso con la posibilidad de romper las interfaces del módulo. Según (Inostroza et al., 2011; Bodden et al. 2013; Bodden et al., 2014), en POA tradicional estilo AspectJ, el desarrollo independiente no es posible ya que los desarrolladores de los aspectos deben conocer todas las clases y su comportamiento para definir reglas y consejos de manera adecuada, y los cambios en el código base también podrían exigir cambios en los aspectos. Por lo tanto, el desarrollo independiente junto con la evolución del código base y los aspectos está muy comprometida. La figura 1 (Bodden et al., 2014) ilustra las dependencias implícitas de los aspectos y las clases aconsejadas para soluciones en POA tradicional.

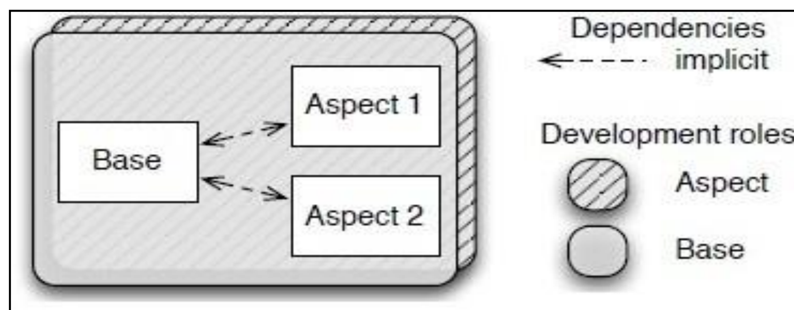

Fig. 1: Dependencias implícitas entre Código base aconsejable y aspectos en soluciones POA tradicionales.

Para un aspecto fuerte y desacoplamiento de código base, (Inostroza et al., 2011; Bodden et al. 2013; Bodden et al. 2014) proponen una nueva metodología POA para definir interfaces de puntos de unión JPI entre aspectos y clases aconsejadas para su desacoplamiento. Además, JPI permite respetar el principio de ocultación de la información ya que los aspectos de acceso recibieron valores de los métodos de clases que presentan esos aspectos. La figura 2 (Bodden et al. 2014) muestra, para las soluciones JPI, una interfaz de punto de unión entre los aspectos y el código aconsejado.

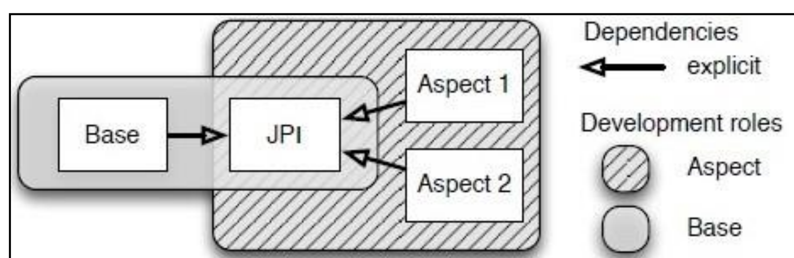

Fig. 2: Dependencias de Código base y aspectos en soluciones JPI.

Ventajas y desventajas de POC

En segundo lugar, la POC, por medio de su estructura jerárquica, permite un refinamiento paso a paso explícito para la arquitectura de soluciones software. Así, POC modulariza las características implementadas por una mezcla de diferentes capas jerárquicas. Una capa de una mezcla representa un conjunto de módulos colaboradores los que implementan fragmentos de clases (Apel et al., 2013; Apel et al., 2005a; Apel et al., 2005b). De esta forma, una mezcla de capas es transversal a varias clases. Por lo tanto, POC presenta un buen soporte para la modularización de módulos heterogéneos.

Según (Mezini et al., 2004; Apel et al., 2005a; Apel et al., 2005b; Apel et al., 2006b; Apel et al., 2013), los lenguajes de POC tales como Jak (Battory, 2004), no presenta un buen soporte para modularizar módulos transversales homogéneos y dinámicos; todo esto es un claro asunto de la POC. Además, como una capa representa el refinamiento de una capa superior, normalmente las clases refinadas no respetan el principio de única responsabilidad de un buen diseño orientado a objetos. Como una solución para estas cuestiones de POC, los trabajos de (Mezini y Ostermann, 2004; Apel et al., 2005a; Apel et al., 2005b) proponen Caesar y FeatureC++, respectivamente, que son lenguajes de programación que combinan POC y POA tradicional. Sin embargo, estas soluciones preservan los mencionados problemas de POA. Dado lo anterior, los objetivos principales de esta investigación son analizar, proponer, implementar y evaluar los pros y contras de una simbiosis de JPI y un lenguaje POC estilo Java para la producción de software modular.

PROPUESTA DE SIMBIOSIS FM-JPI

Para modelar la variabilidad de una LPS, POC utiliza modelo de características (Features Model, FM) (Benavides et al., 2010; Apel et al., 2013) para representar de manera conceptual mediante la composición de características y sus relaciones de cruzadas de requerimiento (requieres) o exclusión (excludes) todos los productos de una LPS, pero sin modelar la estructura de las características. Los diagramas de colaboración con estructura de diagramas de clases y con asociaciones de colaboración entre sus componentes sí son adecuados para representar la descomposición jerárquica de una LPS (Apel et al., 2013).

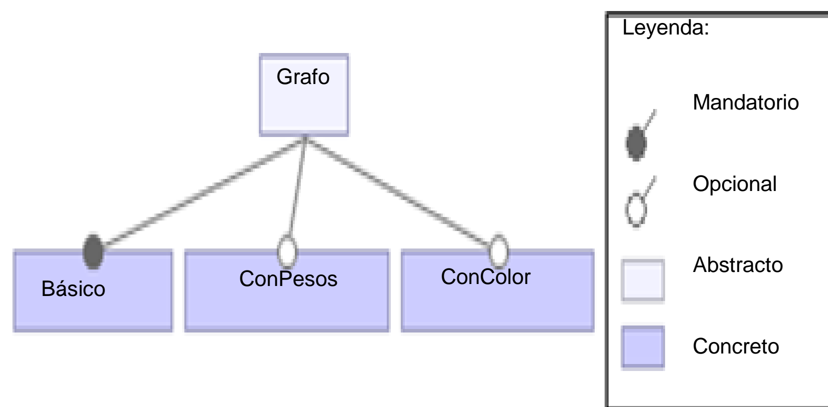

Fig. 3: Modelo de características de un grafo.

La figura 3 (Apel et al., 2013) muestra un ejemplo de FM de una LPS Grafo, mientras la figura 4 (Apel et al., 2013) muestra un diseño de colaboración entre clases, asociaciones y evolución de esta LPS. En la figura 4 se aprecia la evolución de capas desde la capa superior (capa padre) hasta la capa inferior. La capa superior es un grafo simple donde cada instancia de grafo está compuesta de nodos y aristas que se forman por nodos. La siguiente capa representa un grafo con pesos, en la que se define la clase Peso, y la extensión de las clases Grafo y Aristas para la consideración de pesos en las aristas.

Finalmente, la capa más inferior representa un grafo con colores para lo cual, con la base de la capa de un grafo con pesos, se incluye la clase Color y se redefinen las clases Arista y Nodo para la consideración de color. La figura 5 (Apel et al., 2013) muestra el código en Jak, una extensión de Java como lenguaje de programación orientado a la característica, para un grafo con color. Feature Colored presenta preocupaciones transversales homogéneas, el código resaltado repetido; Por lo tanto, en una perspectiva POA, una solución más adecuada debería representar esos aspectos transversales como aspectos.

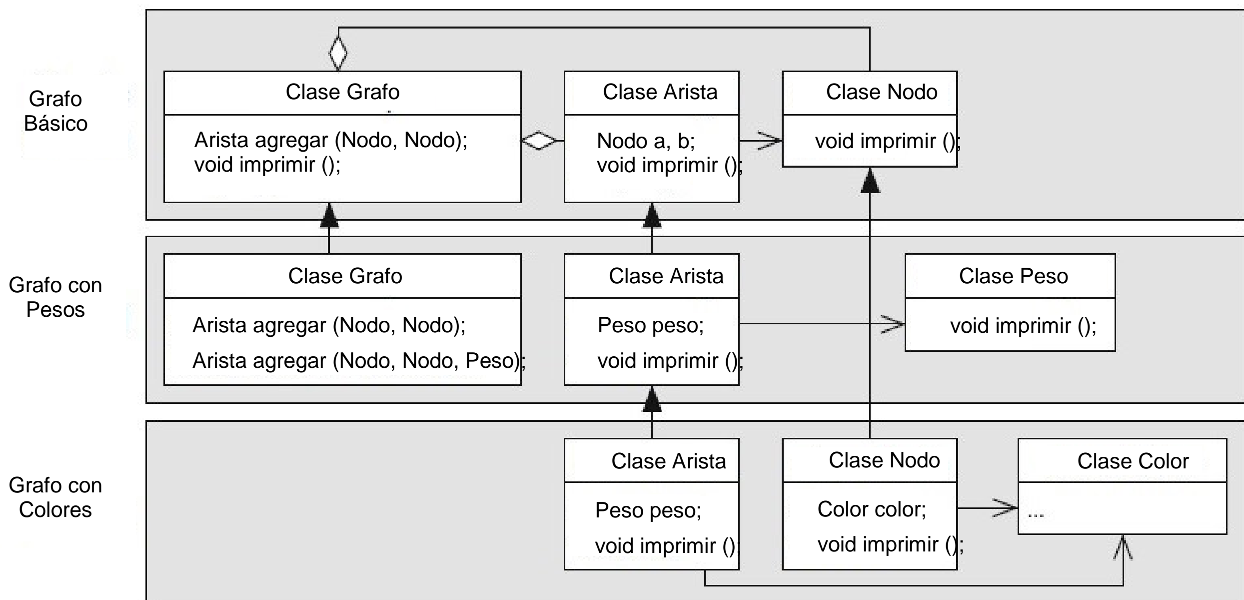

Fig. 4: Módulos estructurales de colaboración entre clases y características de un grafo.

```
public class Grafo { /*...*/}

public class Nodo {
  Color color = new Color();
  Color getColor(){ return color;}
  int id = 0;
  Nodo (int _id){ id = _id;}
  void imprimir(){
     Color.setDisplayColor(getColor());
     System.out.println(id);
  }
}
```

```
public class Arista {
  Color color = new Color();
  Color getColor(){ return color;}
  Nodo a, b;
  Arista(Nodo _a, Nodo _b){ a = _a;
                            b = _b;}
  void imprimir(){
     Color.setDisplayColor(getColor());
     System.out.println(" (");
     a.imprimir();
     System.out.println(", ");
     b.imprimir();
     System.out.println(") ");
  }
}
```

```
public class Color {
   public static void setDisplayColor(Color x){ /* ... */}

}
```

Fig. 5: Código Jak POC de solución de característica (capa) Grafo con Colores

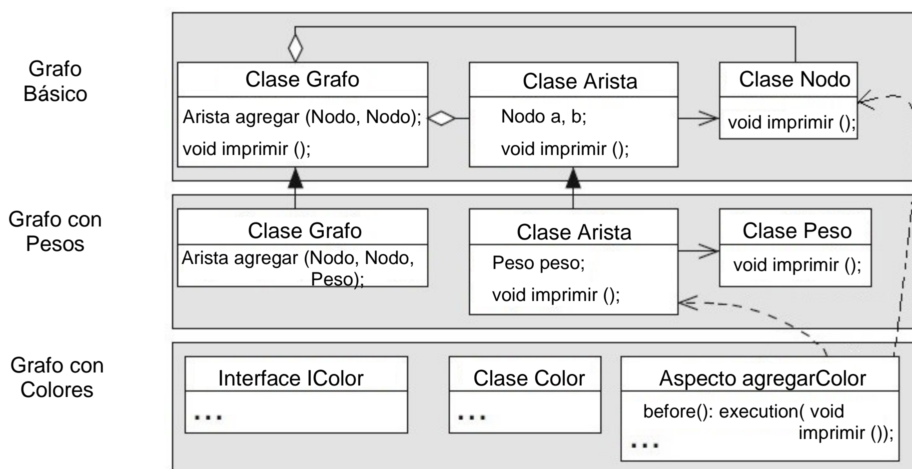

Fig. 6: Modelo estructural de colaboración entre clases para las etapas de un grafo.

Aun cuando los aspectos pueden combinar diferentes acciones y roles de comportamiento muy adecuados para modularizar componentes homogéneos principalmente. Las soluciones POA también pueden terminar con la estructura heterogénea inherente orientada a objetos de POC, por lo que una simbiosis de POC y POA parece más adecuada para la modularización heterogénea y homogénea de módulos transversales. La figura 6 (Apel et al., 2013) presenta un MFA para el ejemplo de LPS de Grafo donde los módulos transversales heterogéneos continúan siendo manejados por una solución POC, mientras que las los módulos transversales homogéneas son representadas por una solución clásica de POA. A pesar de que las soluciones de la simbiosis entre POC y POA clásico estilo AspectJ toman en cuenta las ventajas de modularización de ambos paradigmas, MFA retiene los problemas de POA tradicional. Por lo tanto, una propuesta de JPI-FM debe producir soluciones modulares más limpias, sin las dependencias propias de POA estilo AspectJ.

Programación POC-JPI

Dada los beneficios de POC y JPI, este trabajo propone JPI-FM para una simbiosis de ambos paradigmas: 1ero, POC, para la evolución del software, para la colaboración de clases y de nuevos elementos del sistema, módulos heterogéneos y estáticos transversales; 2do, JPI, para evitar la replicación de código y modularizar los atributos y comportamiento homogéneo y dinámico en términos de rutas de ejecución, junto con respetar los principios de ocultamiento de la información y de buen diseño orientado a objetos tales como única responsabilidad y abierto-cerrado. La figura 7 ilustra JPI-FM para un sistema que presenta una capa base y dos capas de refinamiento:

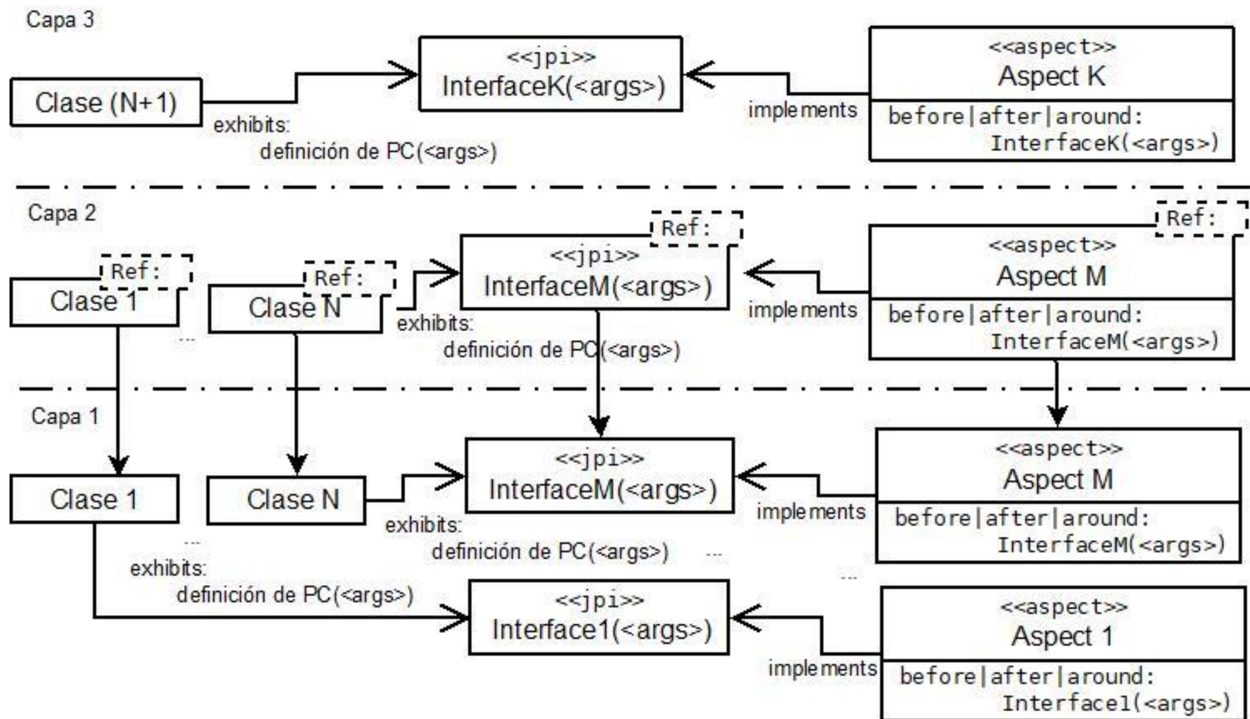

Fig. 7: Modelo general de aplicación de JPI-FM.

Capa base (Layer 1) representa un conjunto de N clases, un conjunto de M interfaces JPI, y un conjunto de M aspectos. Instancias JPI son exhibidas (exhibits) por clases e implementadas por aspectos, y los aspectos, tal como en POA estilo AspectJ, indican cuando ellos agregar comportamiento en el módulo aconsejado: before, after y around. Esta propuesta de modelamiento agrega estereotipos sobre clases para identificar elementos de la metodología POA JPI e intenta ser consistente con las ideas JPI (Bodden et al. 2014). Entonces, cada clase que exhibe una interfaz JPI también define una regla de punto de corte para identificar la ocurrencia de puntos de unión, y los aspectos implementan el comportamiento que dichas interfaces.

Capa 2 (Layer 2) presenta el refinamiento de elementos (clases, aspectos e interfaces) de la capa previa mediante modelos. Así, el refinamiento de una interfaz de punto de unión requiere el refinamiento de clases aconsejables (las que exhiben dicha interfaz), y de los aspectos que la implementan.

Capa 3 (Layer 3) preserva elementos de la capa 2, y la inclusión de nuevos elementos, una clase, un aspecto y una interfaz JPI.

Tal y como se aprecia en la figura 7, un elemento distintivo y diferenciador para el modelamiento conceptual de JPI-FMI es el refinamiento estructural en sentido vertical hacia arriba, es decir, en el sentido opuesto del refinamiento de los modelos estructurales de colaboración.

En la definición de una LPS, las clases en la jerarquía de capas pueden presentar interfaces JPI. Esta propuesta sólo considera la definición de JPI entre clases y aspectos. Como parte de un proyecto futuro está la inclusión de la definición de puntos de unión cerrados (closure join points) (Boddem, 2011). Como aplicación ejemplo de esta propuesta, la figura 8 ilustra el JPI-FM para la LPS Grafo y la implementación de algunos de sus componentes con refinamiento, y la figura 9 presenta su modelo JPI-FM, con lo cual se aprecia la modularidad de incumbencias homogéneas y heterogéneas de la aplicación de esta propuesta con detalles de la estructura de las características del modelo.

```
package aspects;

import classes;
import joinpointinterfaces.*;

layer GrafoconColor;

public aspect agregarColor {
    interface IColor{ Color getColor();}
    declare parenta:
        (Nodo || Arista) implements IColor;

    Color IColor.color;
    Color IColor.getColor(){ return color;}

    before JPIColor(IColor c):
      execution(void imprimir()) && this(c){
        Color.setDisplayColor(c.getColor());
    }
}
```

```
package joinpointinterfaces;
import classes.*;

layer GrafoconColor;

jpi void JPIColor(IColor c);
```

```
package classes;
import joinpointinterfaces.*;

layer GrafoconColor;

refines public class Arista{
  exhibits JPIColor(Arista c):
     execution(void imprimir()) &&
                         this(c);

   Nodo a, b;

   Arista(Nodo _a, Nodo _b){ a = _a;
                             b = _b;}

   void imprimir(){
      System.out.println(" (");
      a.imprimir();
      System.out.println(", ");
      b.imprimir();
      System.out.println(") ");
     }
}
```

Fig. 8: Código JPI-FM de aspecto agregarColor, de interfaz de punto de unión de Grafo con Colores y JPI-FM de clase Arista de Grafo con Colores.

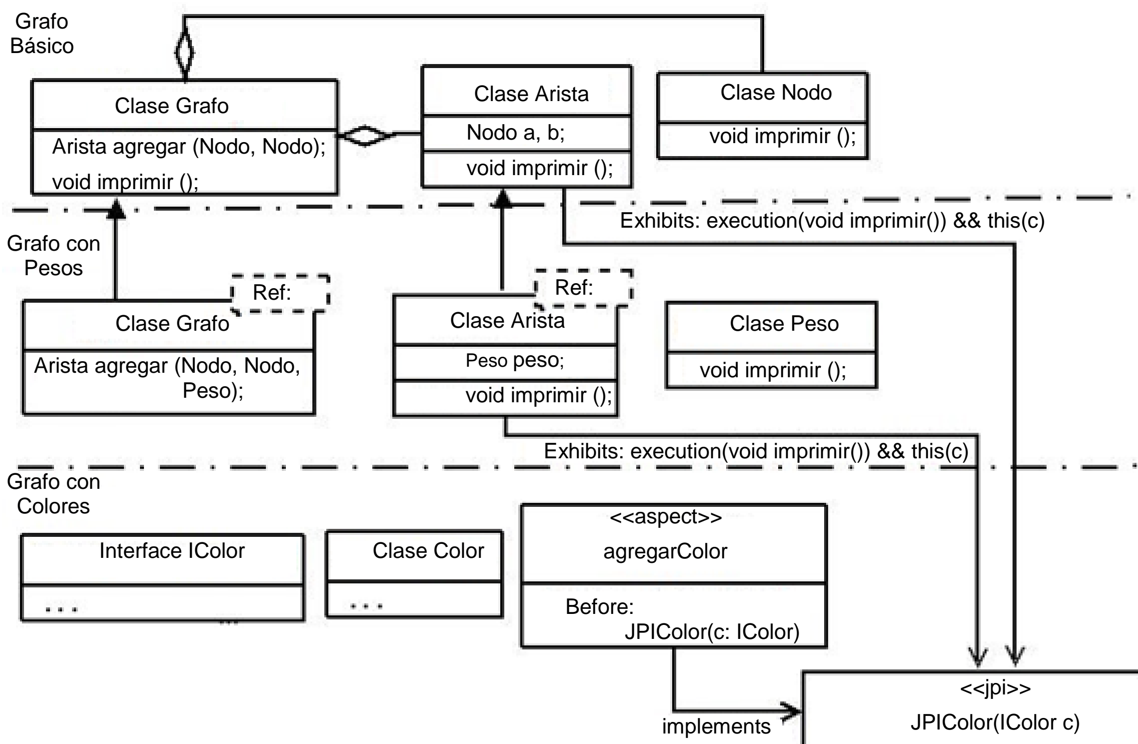

Fig. 9: Modelo JPI-FM de ejemplo de Grafo.

JPI-FM permite alcanzar altos niveles de modularidad conceptual en un enfoque de desarrollo JPI-POC, con lo cual sería posible una mejora en las métricas que relacionan características y la estructura modular de las soluciones tales como Dedicación, Disparidad y DIST (Abilio et al., 2016). Es trabajo actual de los autores la extensión de Eclipse para el soporte de programación para JPI- POC.

CONCLUSIONES

JPI-FM preserva las principales propiedades de POC para la producción de líneas de productos de software con una personalización masiva, lo cual presenta ventajas de modularización para LPS con respecto a MFA. Esto es así, ya que la metodología de POA JPI exhibe notables mejoras de modularización sobre POA tradicional estilo AspectJ. Considerando las ventajas de modularidad tato de JPI y POC, JPI-FM permite un modelamiento general y modular de alto nivel para la producción masiva de software modular.

Para evaluar los pros y contras de la modularización de JPI-FM con respecto a sus enfoques POC y JPI de origen, es necesaria la definición de casos de estudio para aplicar esta propuesta de desarrollo para LPS, y así evaluar sus ventajas prácticas, las que, desde un punto de vista teórico ya se aprecian, pero no hay garantía en ventajas de ejecución.

REFERENCIAS

Abilio, R., G. Vale, E. Figueiredo y H. Costa, Metrics for feature-oriented programming, doi: 10.1145/2897695.2897701, en Actas de 7th International Workshop on Emerging Trends in Software Metrics, 36 – 42, Austin, Texas, USA (2016)

Apel, S. y C. Kästner, Overview of¨ Feature-Oriented Software Development, Journal of Object Technology (JOT), 8(4), 1-36, (2009)

Apel, S., D. Batory y M. Rosenmüller, On the structure of crosscutting concerns: Using aspects or collaborations, en Actas de Workshop on Aspect-Oriented Product Line Engineering, Portland, Oregom, USA, Octubre (2006)

Apel, S., D. Batory, C. Kstner y G. Saake, Feature-Oriented Software Product Lines: Concepts and Implementation, 1-65, Springer Publishing Company, Incorporated (2013)

Apel, S., T. Leich y G. Saake, Aspectual Feature Modules, doi: 10.1109/TSE.2007.70770, IEEE Transactions on Software Engineering, 34(2), 162–180 (2008)

Apel, S., T. Leich y G. Saake, Aspectual mixin layers: Aspects and features in concert, doi: 10.1145/1134285.1134304, en Actas de 28th International Conference on Software Engineering, ICSE ’06, 122–131, New York, NY, USA (2006)

Apel, S., T. Leich, M. Rosenmüller y G. Saake, Combining feature-oriented and aspect-oriented programming to support software evolution, en Actas de Walter Cazzola, RAM-SE, 3–16 (2005)

Apel, S., T. Leich, M. Rosenmüller y G. Saake, FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C, doi; 10.1007/11561347_10, en Actas de 4th International Conference on Generative Programming and Component Engineering, GPCE 2005, Tallinn, Estonia (2005)

Batory, D., A tutorial on feature-oriented programming and the ahead tool suite, doi: 10.1007/11877028_1, en Actas de International Conference on Generative and Transformational Techniques in Software Engineering, GTTSE’05, 3–35, Berlin, Heidelberg, Springer-Verlag (2006)

Batory, D., Feature-Oriented Programming and the AHEAD Tool Suite, doi: 10.1109/ICSE.2004.1317399, en Actas de 26th International Conference on Software Engineering (ICSE’04), 702–703, Edinburgo, Inglaterra, Mayo (2004)

Benavides, D., S. Sergio y A. Ruiz-Cortés, Automated analysis of feature models 20 years later: A literature review, doi: 10.1016/j.is.2010.01.001, Journal Information Systems, 35(6), 615–636, Septiembre (2010)

Bodden, E., Closure joinpoints: Block joinpoints without surprises, doi: 10.1145/1960275.1960291, en Actas de Tenth International Conference on Aspect-oriented Software Development, AOSD’11, 117–128, New York, NY, USA (2011)

Bodden, E., E. Tanter y M. Inostroza, A brief tour of join point interfaces, doi: 10.1145/2457392.2457401, en Actas de 12th Annual International Conference Companion on Aspect-oriented Software Development, AOSD ’13 Companion, 19–22, New York, NY, USA (2013)

Bodden, E., E. Tanter y M. Inostroza, Join point interfaces for safe and flexible decoupling of aspects, doi: 10.1145/2559933, ACM Transactions on Software Engineering and Methodology, 23(1), 7:1–7:41, USA, Febrero (2014)

Dijkstra, E.W., The structure of the “THE”-multiprogramming system, Communications of the ACM, doi: 10.1145/363095.363143, 11(5), 341-346, USA, Mayo (1968)

Eclipse, The AspectJ Project 2013, https://eclipse.org/aspectj/ (2016)

Inostroza, M., E. Tanter y E. Bodden, Join point interfaces for modular reasoning in aspect-oriented programs, doi: 10.1145/2025113.2025205, en Actas de 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE ’11, 508–511, New York, NY, USA (2011)

Kiczales, G., Aspect-oriented programming, ACM Computing Surveys, doi: 10.1145/242224.242420, 28 (4es), USA (1996)

Mezini, M. y K. Ostermann, Variability management with feature-oriented programming and aspects, doi: 10.1145/1041685.1029915, SIGSOFT Software Engineering Notes, 29(6), 127–136, Octubre (2004)

Prehofer, C., Feature-oriented programming: A fresh look at objects, European Conference on Object-Oriented Programming ECOOP, Springer-Verlag, 419-443 (1997)

Vidal, C., D. Benavides, J. A. Galindo, P. Leger y H. Fukuda, Mixing of Join Point Interfaces and Feature-Oriented Programming for Modular Software Product Line, doi: 10.4108/eai.3-12-2015.2262534, BICT 2015, Nueva York, USA, 433-437 (2015)

Vidal, C., M. Bustamante, J. M. Rubio y L. Carter, Propuesta de Modelo de Características con Interfaz de Punto de Unión para el Modelamiento de Líneas de Productos de Software, http://dx.doi.org/10.4067/S0718-07642018000600213, Información Tecnológica, 29(6), 213-220 (2018)

Wampler, D., Aspect-oriented design principles: Lessons from object-oriented design, en Actas de Sixth International Conference on Aspect-Oriented Software Development (AOSD'07), 615–636, Vancouver, British Columbia, Marzo (2007)