# Mixing of Join Point Interfaces and Feature-Oriented Programming for Modular Software Product Line

Cristian Vidal
Universidad de Playa Ancha
Chile
cristian.vidal@upla.cl

David Benavides
University of Seville
Spain
benavides@us.es

Paul Leger
Universidad Católica del Norte
Chile
pleger@ucn.cl

José Angel Galindo
INRIA
France
jagalindo@inria.fr

Hiroaki Fukuda
Shibaura Institute of Technology
Japan
hiroaki@shibaura-it.ac.jp

## ABSTRACT

Feature-oriented programming (FOP) and aspect-oriented programming (AOP) focus on to modularize incremental classes behavior and crosscutting concerns, respectively, for software evolution. So, these software development approaches represent advanced paradigms for a modular software product lines production. Thereby, a FOP and AOP symbiosis would permit reaching pros and cons of both approaches.

FOP permits a modular refinement of classes collaboration for software product lines (SPL), an adequate approach to represent named heterogeneous crosscutting concerns. FOP works on changes of different functionality pieces for which to define join points is not a simple task. Similarly, AOP structurally modularizes in a refined manner homogeneous crosscutting concerns. Since traditional AOP like AspectJ presents implicit dependencies and strong coupling between classes and aspects, and the Join Point Interface JPI approach solves these classic AOP issues, this article presents JPI Feature Modules for the FOP + JPI SPL components modularization, i.e., collaboration of classes, aspects, and join point interfaces along with their evolution, for a SPL transparent implementation in a FOP + JPI context. In addition, this article shows JPI Feature Modules of a case study to highlight mutual benefits of FOP and JPI approaches for a modular SPL software conception.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*

## General Terms

Languages, Design

## Keywords

FOP, classic AOP, JPI, SPL, modular software, JPI-FM

## 1. INTRODUCTION

The separation of concerns principle [15] mentions that generating modular programs is one of the main goals of programming. In this context, for software modularity, Object-Oriented Programming OOP represents an evolution concerning structured programming because OOP encapsulates attributes and functions as members of classes, and defines information hiding rules to access class members. Nevertheless, OOP neither directly encapsulates features and product-line architectures [3] [19], nor crosscutting concerns as independent modules [17] for an organized software evolution.

Software development paradigms like Feature-Oriented Programming FOP [18] and Aspect-Oriented Programming AOP [17] look for to solve OOP issues:

- FOP modularizes collaboration of classes, named heterogeneous crosscutting concerns, i.e., by classes refining, FOP works on changes of different functionality pieces for which to define join points is not a simple task, as features, and permits the step-wise development of software product lines [3] [10]. In addition, as [3] [4] [6] indicate, FOP well modularizes static crosscutting concerns – new fields, methods, classes, and interfaces definition. Nevertheless, [17] [6] remark that FOP lacks adequate crosscutting modularity for software evolution since software has to change and adapt to fit non-predictable modifications. Particularly, for an homogeneous crosscutting concerns, i.e., for features which represent the same behavior that appear in different places in the features-tree hierarchy, "FOP does not modularize elegantly homogeneous crosscutting concerns" [3] [4] [6]. For dynamic crosscutting concerns, FOP supports only methods interception [7], i.e., in a class refinement, it is possible to call refined and non-refined methods.
- AOP, looking for a modular behavior of classes, defines oblivious advised classes and modularizes crosscutting concerns as aspects, i.e., orthogonal methods non-part

BICT 2015, December 03-05, New York City, United States

DOI 10.4108/eai.3-12-2015.2262534

of the advised classes nature [17]; so AOP solutions are able to respect the *single responsibility* OOP design principles [21]. Furthermore, AOP can modularize advanced dynamic crosscutting concerns not only methods interception. For example, AOP like AspectJ [9] permits the use of *cflow*, *if*, *execution* pointcuts. As [3] [4] [6] indicate, AOP modularizes homogeneous crosscutting concerns as aspects, but the modularization of classes collaboration by aspects results in software complex to evolve since aspects do not reflect the structure of the refined feature and its cohesion [7]. Moreover, AOP like AspectJ [9] solutions introduce implicit dependencies between aspects and classes. Hence, aspects do no respect the *information hiding* principle, so oblivious classes can experience non-expected behavior and properties changes, and by modifying the advised behavior of classes, aspects can be spurious and other non-effective can appear. So, in traditional AOP like AspectJ, aspects need clearly to know about advised classes before advising them, a great issue for independent development [13] [14] [16].

Since, main FOP and AOP work focus are different kinds of crosscutting concerns, thus, as [3] [5] [7] [18] argue, a symbiosis of FOP and AOP profit of each other to obtain modular software. Thus, looking for a FOP and traditional AOP like AspectJ symbiosis, regarding a collaboration-based design, [3] [8] propose Aspectual Feature Modules (AFM) to represent modular classes and aspects along with their association and evolution. Nonetheless, AFM preserves already mentioned classic AOP issues. So, given the FOP benefits to modularize SPL heterogeneous and static crosscutting concerns, as well as the JPI capability to respect OO principle and modularize dynamic and homogeneous crosscutting concerns [13] [14] [16], hence the main goals of this article are, 1st to present JPI Feature Modules JPI-FM to structurally model JPI + FOP solutions for classes and aspects collaboration and get associated benefits in the massive customized software production, and 2nd, apply JPI-FM on a simple graph to evaluate the obtainable modularity. This article gives more details and extends previous work of [20]. Specifically, this article presents a detailed study of an application example in order to discuss and find main points behind this proposal.

The rest of this paper is organized as follows. The next section presents main motivation for our research of proposing a mixing of JPI and FOP. Afterwards, next section presents FOP and AOP approaches as well as their traditional symbiosis, and an application example. After it, a section defines our JPI-FM approach to model and highlights its benefits and cons by means of an application example and shows part of its code solution. Finally, a section presents conclusions and future research work.

## 2. MOTIVATION

As was indicated before, traditional like AspectJ AOP and FOP look for to solve some OOP issues for the modular software production. This section details some pros and cons of these programming methodologies.

1st, traditional AOP in general modularizes crosscutting concerns of program modules, i.e., in an OOP context, orthogonal methods over classes. For OOP and AOP, since classes are oblivious of aspects, some OOP design principles for modular programs are respected such as the open-close and single responsibility [21]. Traditional AOP like AspectJ defines pointcut rules and advices to identify join points in program execution and, by implicit-invocation, aspects advise on these join points. Mainly, by the use of wildcards and pattern matching mechanisms, AOP permits refining multiple join points by only one declaration rule, i.e., AOP permits good modularization of homogeneous crosscutting concerns. In addition, since AOP provides mechanisms to advised programs based on their dynamic execution, AOP also permits good support of dynamic crosscutting concerns [3].

Since in traditional AOP like AspectJ, pointcut rules indicate textual references to advised code, changes on advised base code can generate spurious pointcut rules, i.e., non-effective aspects. Furthermore, for the mentioned dependency, aspects are usually non-reusable and unable for evolution. Conversely, base code can obliviously experience non-desirable changes by the aspect advices invasiveness, possibly breaking assumptions over advised and non-advised base code, i.e, advised code by aspects and related code [14]. Clearly, classic AOP like AspectJ languages violate the principle of information hiding [3] since an aspect may affect internals of other modules directly regardless privacy rules, possibly even breaking module interfaces.

Furthermore, as Bodden et al. [14] [13] and Inostroza et al. [16] indicate, for traditional AOP like AspectJ methodologies, the independent development is non-possible since aspects developer have to know all classes and their behavior to define pointcut rules and associated advices adequately, and changes on base code could demand applying changes on aspects as well. So, the independent development and evolution of base code and aspects is hugely compromised. Figure 1 [14] illustrates implicit dependencies of aspects and advised classes of a classic AOP solutions.

For a strong aspects and base code decoupling, Bodden et al. [14] [13] and Inostroza et al. [16] propose a new AOP methodology to define join point interfaces JPI between aspects and advised classes for their decoupling. In addition, JPI permits respecting the information hiding principle since aspects access received values from classes methods which exhibit those aspects. Figure 2 [14] shows, for JPI solutions, a join point interface between aspects and advised code.

2nd, FOP, by means of its hierarchical structure, permits an explicit stepwise refinement for software architecture. As was mentioned before, FOP modularizes features implemented by mixin layers. A mixin layer represents a set of collaborating mixins which implement class fragments [2] [6] [3], hence a mixin layer crosscut multiple classes. Thus, FOP presents a good support for modularizing heterogeneous crosscutting concerns.

According to Mezini et al. [18] and Apel et al. [2] [6] [7] [3], FOP does not present a good support to modularize homogeneous and dynamic crosscutting concerns; clear issues of FOP. Furthermore, since child mixins refine their parent

mixins, usually refined classes do not respect the single responsibility design principle. As a solution for these FOP issues, [18] and [2] [6] proposed Caesar and FeatureC++ respectively, languages that mix FOP and traditional AOP like AspectJ characteristics. Nevertheless, these solutions preserve mentioned AOP issues.

Thus, the main goals of this research are to analyze, propose, implement, and evaluate pros and cons of a symbiosis of JPI and a Java style FOP language to reach modular software.

## 3. COLLABORATION-BASE DESIGN

To represent features variation for SPL, FOP uses Feature Models FM [3] [11]. So, a FM displays all possible products of a SPL; but, traditionally, in a SPL context, as a conceptual model, a FM only presents composition, requires, or exclude features associations and does not show components structure. As Apel et al. [3] mention, collaboration-based diagrams are adequate to represent SPL hierarchical decomposition of classes structure and their collaboration associations.

Figure 3 [3] presents the feature model of a simple graph system, and Figure 4 [3] shows its classes, associations, and their evolution as a collaboration-based design.

Figure 5 [3] shows the code in Jak, a feature-oriented programming language. Feature **Colored** presents homogeneous crosscutting concerns, the highlighted repeated code; thus, in an AOP view, a more adequate solution should represent those crosscutting concerns as aspects.

As [3] [8] indicate, by means of an aspect-oriented instead of a FOP solution for features modularization, since aspects can merge different actions and behavioral roles, AOP solutions also can flatten inherent object-oriented structure of features. Thus, a mix of FOP and AOP seems more adequate because it would permit modularizing heterogeneous and homogeneous crosscutting concerns. Figure 6 [3] presents an AFM for the graph example in which heterogeneous crosscutting concerns continue being handled by a FOP solution meanwhile homogeneous crosscutting concerns, in this example, associated to a colored graph, are represented by a classic AOP solution. Figure 7 [3] presents AspectJ code for the aspect Colored.

Even though, FOP and classic AOP like AspectJ symbiosis solutions take into account modularization advantages of both paradigms, AFM retains issues of the last one. Thus, our JPI-FM should produce cleaner modular solutions.

## 4. JPI FEATURE MODULES: JPI-FM

Given the FOP and JPI benefits, this paper proposes JPI-FM looking for the symbiosis of both paradigms: 1st, FOP, for software evolution, for collaboration of classes and new elements of the system, heterogeneous and static crosscutting concerns; 2nd, JPI, to avoid code replication and adequately represent homogeneous and dynamic program behavior in terms of execution paths, dynamic homogeneous crosscutting concerns, and to respect information hiding and single responsibility OOP design principles.

Figure 8 illustrates an application of our JPI-FM proposal

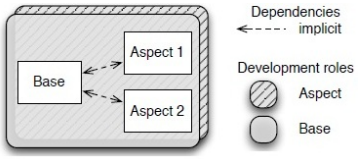


**Figure 1: Implicit Dependencies of Advised Base Code and Aspects of Classic AOP Solutions**

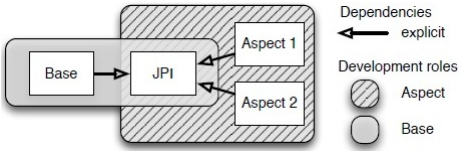


**Figure 2: Dependencies of Base Code and Aspects with JPI Solutions**

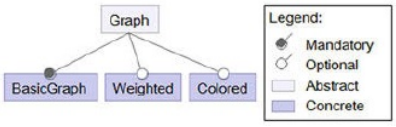


**Figure 3: Feature model of a graph**

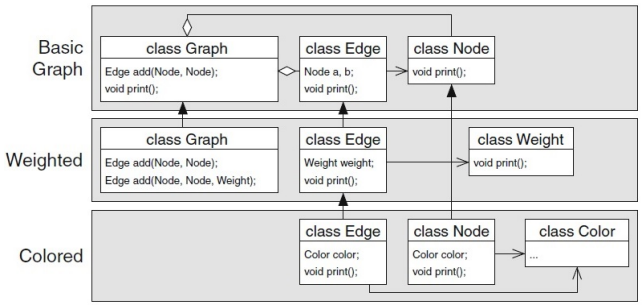


**Figure 4: Structural feature modules of a graph**

```
1  class Graph { /* ... */ }
2  class Node {
3    Color color = new Color();
4    Color getColor() { return color; }
5    int id = 0;
6    Node(int _id) { id = _id; }
7    void print() {
8      Color.setDisplayColor(getColor());
9      System.out.print(id);
10   }
11 }
12 class Edge {
13   Color color = new Color();
14   Color getColor() { return color; }
15   Node a, b;
16   Edge(Node _a, Node _b) { a = _a; b = _b; }
17   void print() {
18     Color.setDisplayColor(getColor());
19     System.out.print(" (");
20     a.print();
21     System.out.print(", ");
22     b.print();
23     System.out.print(") ");
24   }
25 }
26 class Color {
27   static void setDisplayColor(Color c) { /* ... */ }
28 }
```

**Figure 5: Code of FOP solution of feature Colored**

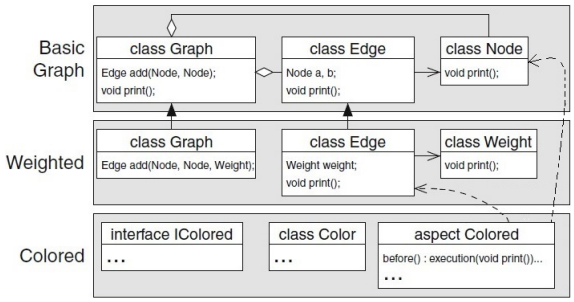


**Figure 6: AFM of a graph example**

```
1  aspect Colored {
2    interface IColored { Color getColor(); }
3    declare parents: (Node || Edge) implements IColored;
4    Color IColored.color;
5    Color IColored.getColor() { return color; }
6    before(IColored c) : execution(void print()) && this(c) {
7      Color.setDisplayColor(c.getColor());
8    }
9    static class Color { /* ... */ }
10 }
```

**Figure 7: AspectJ code of aspect Colored for the graph example**

for a system that presents a base layer and two layers of refinement:

- Base layer (Layer 1) presents a set of N classes, a set of M join point interfaces, and a set of M aspects, i.e., $Class_1$,..., $Class_N$; $\ll jpi \gg Interface1(< args >)$, ..., $\ll jpi \gg InterfaceM(< args >)$; $\ll aspect \gg Aspect_1$, ..., $\ll aspect \gg Aspect_M$ respectively. Note that JPI units are exhibited by classes and implemented by aspects, and aspects, like in traditional AOP like AspectJ present before, around, and after advices. Clearly, our modeling proposal stereotyped classes to identify JPI elements. In addition, our model tries to be consistent with JPI ideas [14]; hence, each class that exhibits a join point interface also defines a pointcut PC rule to identify join point occurrences, and aspects implement those $\ll jpi \gg$ interfaces.

- Layer 2 presents the refinement of previous layer elements, particularly for this example, $Class_1$, $Class_N$, $\ll jpi \gg Interface_M$, and $\ll aspect \gg Aspect_M$. Layer 2 presents the use of template to indicate a refinement of previous layer elements. Note that a refinement of join point interface requires a refinement of an associated advised class and aspect.

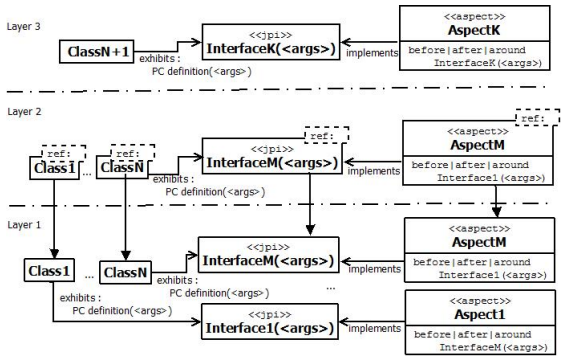


**Figure 8: General application of JPI-FM**

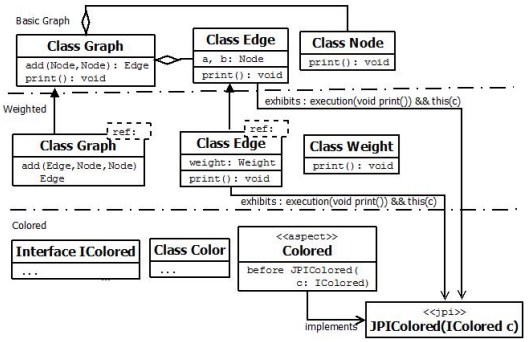


**Figure 9: JPI-FM of the graph example**

- Layer 3 preserves elements of Layer 2, and add a new class, $Class_{N+1}$, a new join point interface, $\ll jpi \gg InterfaceK(< args >)$, and a new aspect, $\ll aspect \gg Aspect_K$. Note that $Class_{N+1}$ exhibits $\ll jpi \gg InterfaceK(< args >)$, and $\ll aspect \gg Aspect_K$ implements that $\ll jpi \gg$ interface.

It is important to know that, for the product lines definition, classes of a layer or of previous layers can exhibit a join point interface JPI. Note that our proposal only considers join point interfaces definition between classes and aspects. As part of a current research project, additional characteristics of FOP and JPI are going to be added to our proposal. Next, Figure 9 illustrates an application of JPI-FM on the graph example, and Figure 10 shows part of the associated code for layers *Weighted* and *Colored*, specifically for class Edge, and aspect Colored along with JPIColored respectively. Note that class Edge of layer *Weighted* is a class refinement of itself from layer *Basic Graph* which exhibits the JPI JPIColored for the execution of its method print(), and aspect Colored implements JPIColored before. Thus, clearly our proposal mixes JPI and FOP nature for the SPL development.

# 5. CONCLUSIONS

JPI-FM continues using FOP main properties to produce massive customized software applications presenting modularization advantages for SPL respecting AFM since JPI exhibits notable modularization improvements over classic AOP. Thus, JPI-FM models a high-level massive modular software in a FOP-JPI symbiosis context. Exactly, to produce a complete FOP and JPI symbiosis is our future **main** goal. In this context JPI-FM has to support a symbiosis of FOP and closure join points [12].

In addition, to evaluate JPI-FM modularization pros and cons regarding its source FOP and JPI approaches, we want to define a case study to apply our whole SPL development proposal.

```
package aspects;

import classes.*;
import joinpointinterfaces.*;

layer Colored;

public aspect Colored{
   Interface IColored { Color getColor();}
   declare parenta: (Node || Edge) implements IColored;

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

   before JPIColored(IColored c){
          Color.setDisplayColor(c.getColor());
     }
  }
```

```
package joinpointinterfaces;
import classes.*;

layer Colored;

jpi void JPIColored(IColored c);
```

```
package classes;
import joinpointinterfaces.*;

layer Weighted;

refines public class Edge{
  exhibits JPIColored(Edge c): execution(void print()) && this(c);

  Node a,b;
  Edge(Node _a,  Node _b){ a = _a; b = -b}

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

**Figure 10: JPI-FM code for the graph example**

# 6. REFERENCES


[1] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.

[2] Sven Apel, Thomas Leich, Marko RosenmÃijller, and Gunter Saake. *FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C*. In Proceedings of the International Conference on Generative Programming and Component Engineering, Springer, 2005.

[3] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.

[4] Sven Apel, Don Batory, and Marko RosenmÃijller. On the structure of crosscutting concerns: Using aspects or collaborations. In *In Workshop on Aspect-Oriented Product Line Engineering*, 2006.

[5] Sven Apel and Christian Kästner. Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 2009.

[6] Sven Apel, Thomas Leich, Marko RosenmÃijller, and Gunter Saake. Combining feature-oriented and aspect-oriented programming to support software evolution. In Walter Cazzola, Shigeru Chiba, Gunter Saake, and Tom TourwÃl', editors, *RAM-SE*, pages 3–16. FakultÃd't fÃijr Informatik, UniversitÃd't Magdeburg, 2005.

[7] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual mixin layers: Aspects and features in concert. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 122–131, New York, NY, USA, 2006. ACM.

[8] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual Feature Modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008.

[9] Eclipse. The AspectJ Project. [Online]. *Available: https://eclipse.org/aspectj/*. [Accessed: 25- Sep- 2015].

[10] Don Batory. A tutorial on feature oriented programming and the ahead tool suite. In *Proceedings of the 2005 International Conference on Generative and Transformational Techniques in Software Engineering*, GTTSE'05, pages 3–35, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, September 2010.

[12] Eric Bodden. Closure joinpoints: Block joinpoints without surprises. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 117–128, New York, NY, USA, 2011. ACM.

[13] Eric Bodden, Eric Tanter, and Milton Inostroza. A brief tour of join point interfaces. In *Proceedings of the 12th Annual International Conference Companion on Aspect-oriented Software Development*, AOSD '13 Companion, pages 19–22, New York, NY, USA, 2013. ACM.

[14] Eric Bodden, Éric Tanter, and Milton Inostroza. Join point interfaces for safe and flexible decoupling of aspects. *ACM Transactions on Software Engineering and Methodology*, 23(1):7:1–7:41, February 2014.

[15] Edsger W. Dijkstra. The structure of the multiprogramming system. In *Communications of the ACM*, page 11(5):341âĂŞ346. Springer-Verlag, May 1968.

[16] Milton Inostroza, Éric Tanter, and Eric Bodden. Join point interfaces for modular reasoning in aspect-oriented programs. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 508–511, New York, NY, USA, 2011. ACM.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[18] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. *SIGSOFT Softw. Eng. Notes*, 29(6):127–136, October 2004.

[19] Christian P|rehofer. Feature-oriented programming: A fresh look at objects. pages 419–443. Springer, 1997.

[20] Cristian Vidal, David Benavides, José Galindo, and Paul Leger. Exploring the Synergies between Join Point Interfaces and Feature-Oriented Programming. JISBD 2015, Santander, Spain, 2015.

[21] Dean Wampler. Aspect-oriented design principles: Lessons from object-oriented design. In *Proceedings of the Sixth International Conference on Aspect-Oriented Software Development (AOSD'07)*, pages 615–636, Vancouver, British Columbia, March 2007.