JPIAspectZ: A Formal Requirement Specification Language for Joint Point Interface AOP Applications

Cristian Vidal-Silva  
Faculty of Economics and Administration,  
Catholic University of the North,  
Antofagasta, Chile  
cristian.vidal@ucn.cl

Erika Madariaga  
Faculty of Engineering,  
Bernardo O’Higgins University,  
Santiago, Chile  
erika.madariaga@ubo.cl

Trung Pham  
Faculty of Economics and Business,  
University of Talca,  
Talca, Chile  
tpham@utalca.cl

Franklin Johnson  
Computing and Information Department,  
University of Playa Ancha,  
Valparaiso, Chile  
franklin.johnson@upla.cl

Luis Alberto Urzua  
Faculty of Health,  
Santo Tomas University,  
Chile Talca, Chile  
lurzua@santotomas.cl

Luis Carter  
Ingenieria Civil Industrial,  
Facultad de Ingenieria,  
Universidad Autonoma de Chile, Chile  
luis.carter@uautonoma.cl

Abstract—Aspect-oriented software development (AOSD) solves a few issues of the object-oriented software development (OOSD) approach and adds a few more concerning modules and their relationships. Join point interface (JPI) is an AOSD methodology that by the definition of the interface between advised artifacts and aspects solves associated AOSD issues to get software with a high modularity level. Looking for a JPI software development approach, this article proposes and exemplifies the use of JPIAspectZ, an extension of the formal aspect-oriented language AspectZ for the software JPI requirement specifications. Mainly, JPIAspectZ looks for a concept and model consistency in a JPI software development process. Since the main JPI characteristics are the joining point interfaces definitions, i.e. explicit associations definition between aspects and advised modules, thus, by JPI, classes are no longer oblivious of possible interaction with aspects, and aspects, for their action effectiveness, do not depend anymore on signatures of advisable module components. JPIAspectZ fully supports these JPI principles. As JPI application examples, this article shows the formal requirements specification for classic aspect-oriented and JPI examples, along with describing the advantages and disadvantages of this language.

Keywords—JPIAspectZ; JPI; AspectZ; aspects; join point interface

I. INTRODUCTION


II. ASPECT ORIENTED PROGRAMMING AND JPI

Authors in [1] proposed aspect-oriented programming (AOP) to modularize crosscutting concerns as aspects in OOP. Aspects advise classes like events, i.e. aspects introduce behavior and structural elements such as methods and attributes into classes. Nevertheless, as authors in [10] indicate, AOP presents implicit dependencies between advised classes and...
aspects. First, aspects define pointcut rules (PCs) for advisable class behavior, and, as a result, instances of those classes are entirely oblivious of possible changes in their components, methods, and attributes. Second, aspects can be ineffective or spurious for signature changes on advised methods of target classes. As authors in [10, 11] mention, the last issue is known as the fragile pointcut problem. Authors in [10] indicate that traditional AOP like Aspect-J solutions compromise the independent development of base code and aspect modules since developers of base code, and aspects must obtain a global knowledge about all program components and their associations, i.e. they must know all the details about aspects, classes, and their relations. To isolate crosscutting concerns and get modular AOP programs without the mentioned implicit dependencies, authors in [10] describe the JPI programming methodology. JPI introduces the idea of join point interface on classic AOP. Like classic AOP [10, 11], for JPI applications, aspects represent crosscutting functionalities, but without PCs. Aspects in JPI only present their implementation of join point interfaces. Besides, in JPI, non-oblivious advised classes exhibit explicit join point interfaces, that is, classes know about potential changes on their methods. Figures 1 and 2 [10] illustrate dependencies between aspects and classes in classic AOP and JPI applications, respectively.

![Fig. 1. Base and aspects modules association in classic AOP.](Image)

![Fig. 2. Base and aspect modules association in JPI.](Image)

### III. JPIAspectZ Principles

Z [13] and Object-Z [14] are formal languages for software requirement specification. Specifically, Z is the classic formal specification language without the direct support of object-oriented abstractions like classes and inheritance, and Object-Z is an extension of Z to support OOSD principles. Likewise, AspectZ [5, 6] and OOAspectZ [7, 8] represent Z extensions for requirements specification of AOP applications and their integration with Z and Object-Z, respectively. Considering JPI ideas, this article describes JPIAspectZ, an OOAspectZ extension to model JPI applications and its integration with Object-Z. The main elements of a JPIAspectZ formal specification are:

**Base Modules:** Unlike AspectZ and OOAspectZ which present oblivious base modules, JPIAspectZ base modules are specified as Object-Z class modules which include an exhibit rule concerning advisable operations of advised class instances. Figure 5 shows the structure of a JPIAspectZ class schema, JPI schema, and Aspect schema. Since the declaration part of an Object-Z operation schema permits defining operation parameters, when looking for a transparency of concepts and design for JPI applications, an exhibit rule is definable in two sections: first, exhibits JPI for the join point interface instances which the class exhibits, and second, a set of conditions for the join point event. So far, JPIAspectZ considers basic AOP and JPI conditions for dynamic and static crosscuts, i.e. call operation, execution operation, logic connectors &&, ||, !; args (arguments list) to identify catchable method arguments, this(object) to determine the object on which the advisable method operates, and target(object) to identify the object owner of the advisable method.

Join Point Interface: In JPIAspectZ, operation schemas starting with the JPI initials represent join point interfaces (JPI schemas) for a system specification. For example, Figure 6 shows JPIUpdateX and JPIUpdateY. Furthermore, JPI schemas only present a declaration section to indicate their list of parameters.

Aspects: JPIAspectZ Aspects-schemas are like Object-Z class diagrams labeled with the phrase aspect. Aspect-schemas include state schemas to define attributes and invariants and operation schemas for the schema advice operations. As a distinction regarding class schemas, aspect-schemas can indicate the occurrence time for operations (before, after, and around) to specify the kind of advice. Semantically, aspect-schemas advise operation schemas, usually for inserting new methods in the advised classes, for adding behavior at the beginning, around, and end on advised operations schemas. From advised method schemas and associated aspect schemas, JPIAspectZ permits obtaining woven schemas. It is relevant to highlight the modular evolution from AspectZ, OOAspectZ, and JPIAspectZ schemas as Figure 3 [5], Figure 4 [7, 8], and Figure 5 respectively present. Note that for the first two, base schema, Z operation schema and Object-Z class schema, AspectZ aspects operate over oblivious advised elements. Nevertheless, for the JPI philosophy, in JPIAspectZ, the aspects and classes know about interfaces to implement and exhibit, respectively.

![Fig. 3. Classic AspectZ aspect schema](Image)

As Figure 5 describes, JPIAspectZ considers advisable classes which exhibit advisable operations in their state schema (Figure 5(a)), Join Point Interface (JPI) schemas as a link between advisable classes and aspects adviser, and aspects who...
will advise those classes in the occurrence of join points in objects of the class (Figure 5(b)).

![Fig. 4. OOAAspectZ aspect schema.](image)

**IV. APPLICATION EXAMPLES**

The Painting System [7] is a classic AO example that presents classes Point and Line which are Shapes, and each Line instance is composed of a few Point instances. The main idea is to illustrate the updating screen process as an external behavior. An illustration of the UML class diagram can be seen in [7], which presents an interface Shape to enclose Point and Line classes, and each Line instance is composed of two Point instances, P1 and P2. Note that for classic AO, both classes obliviously wait for advices from the aspect UpdateSignaling concerning the pointcut rules definition outside the classes. A JPI UML class diagram that includes the main JPI elements for the Painting system, that is, non-oblivious classes which exhibit JPI instances and aspect that implements those interfaces can be seen in [15]. Clearly, for exhibits and implements rules, classes are not more oblivious, and aspect does not directly refer to classes: classes exhibit JPI and aspects implement those interfaces. We recommend reviewing [15] for more details about JPI. As a JPI example, [10, 12] show a Shopping session ‘running example’ of an e-commerce system (ShoppingSession system). This example presents a join point interface checkingOut, a class ShoppingSession that exhibits checkingOut and an aspect Discount that implements checkingOut for around kind of advice.

![Fig. 5. Base and aspect modules association in JPIAspectZ: (a) Advised class schema, (b) JPI and aspect schemas](image)

**V. EXPERIMENTS AND RESULTS**

Figure 6 presents the JPIAspectZ specification for the Painting system: a Shape interface, Point and Line classes, JPI instances JPIUpdateX, JPIUpdateY, and JPIMove, and aspect Aspect1Painting. Point and Line classes exhibit JPI instances, class Point exhibit JPIUpdateX and JPIUpdateY, and class Line exhibit JPIMove, whereas Aspect1Painting implements these JPI instances. Figure 6 presents the model’s consistency.
VI. CONCLUSIONS

After reviewing JPIAspectZ examples for the Painting and ShoppingSession systems, we can argue a clear consistency exists between JPI models and JPI code. Furthermore, JPIAspectZ includes JPI practical details such as exhibits and association among JPI components, that i.e., aspects, JPI, and classes to represent complete JPI applications using called external advises is possible. Thus, JPIAspectZ permits formal models of JPI applications without closure join points [16].

This article presented JPIAspectZ that permits specifying formal requirements for JPI applications, i.e. nondependent classes and aspects according to the JPI central principle. Besides, this extended JPI abstraction demonstrates that consistency and transparency of models and concepts for a JPI software development process is attainable, specifically, a consistency between requirements and structural models, and requirements and code. To achieve real consistency between structural models and JPI solutions code modules seems direct. A formal proof of this consistency represents a future work scope for the authors. Furthermore, considering future work, authors want to continue proposing extensions on JPIAspectZ to model closure join points as well as more advanced dynamic crosscuts [16]. Besides, we propose the developing of a JPIAspectZ specification validation tool for automatic validation of JPIAspectZ specifications.

REFERENCES