A Modular Aspect-Oriented Programming Approach of Join Point Interfaces

Cristian Vidal[1*], Erika Madariaga[2*], Claudia Jiménez[3*], and Luis Carter[4*]

[1]Departamento de Administración, Facultad de Economía y Administración, Universidad Católica del Norte, Antofagasta, Chile,
[2]Ingeniería Informática, Facultad de Ingeniería, Ciencia y Tecnología, Universidad Bernardo O'Higgins, Santiago, Chile,
[3]Ingeniería Civil Informática, Escuela de Ingeniería, Universidad Viña del Mar, Viña del Mar, Chile,
[4]Ingeniería Civil Industrial, Facultad de Ingeniería, Universidad Autónoma de Chile, Chile,

***Abstract*—This paper describes and analyzes the main differences and advantages of the Join Point Interfaces (JPI) as an Aspect-Oriented Programming (AOP) approach for the modular software production concerning the standard aspect-oriented programming methodology for Java (AspectJ) to propose a structural modeling approach looking for modular software solutions. Using a Software Engineering point-of-view, we highlight the relevance of structural and conceptual design for JPI software applications. We model and implement a classic example of AOP using AspectJ and JPI as an application example to review their main difference and highlight the JPI consistency between products (models and code). Our proposal of UML JPI class diagrams allows the definition of oblivious classes which know about their JPI connections, an essential element to adapt and transform tradition like-AspectJ AOP solutions to their JPI version. Thus, for the modular software production and education, JPI seems an ideal software development approach.**

***Keywords*—Aspect-Oriented Programming; AspectJ; JPI; class diagrams; UML**

I. Introduction

The methodology of Aspect-Oriented Programming (AOP) [1] allows to encapsulate or modularize the so-called "non-modularizable cross-concerns" by classical programming methodologies such as object-oriented programming and structured programming. The cross-functionalities are functions spread as part of the modules such as in the methods of classes mixing their steps and nature. In this way, the cross-functionalities represent non-modularizable functional elements by traditional software development methodologies like structured and object-oriented software development. We talk of aspect-oriented software development given the adaptation of other phases in the software development process to encapsulate the so-called cross-concerns.

The users' authentication and registration or log of their actions are classic examples of cross-concerns [2] [3] [4]. By modularizing cross-concerns of an object-oriented software system, classes and their methods can respect the single-responsibility principle [5]), that is, classes and their methods own and know their function and well-defined purpose. Thus, consistency should exist in the behavior of classes' methods which do not presents actions out of their primary objective. In summary, the principle of single-responsibility states that a class should encapsulate only one responsibility. This represents a fundamental principle of modularization and object-oriented programming [6].

Fig. 1 shows a diagram of UML use cases that represents a system and two cases of use that are examples of cross-concerns, usually present in software systems: use cases 'Logging' and 'Authentication'. Note that Jacobson [7] [8], and Vidal et al. [2] indicate the use of 'extends' associations between use cases that represent cross-concerns and base-use cases. In traditional software systems, it is common that before acting, the user must authenticate, and if such authentication does not occur, the action does not proceed. Besides, it is usual for current information systems to keep a record of the actions performed by their users. In this context, Fig. 2 presents a traditional UML class diagram for the example system of Fig. 1.

Fig. 2 shows two classes, Class1 and Class2, with a one-to-many association. Each class has two attributes and four methods. We can appreciate that both classes present the methods A (..) and B (..), that is, Register Actions and Authenticate. Assuming that the functionalities and associated behavior of the methods A(..) and B(..) are not specific to Class1 and Class2, and how these functionalities cannot be represented in independent classes so that Class1 and Class2 respect the principle of only responsibility, then methods A(..) and B(..) are examples of cross-concerns.

Table I presents the Java code associated with the UML class diagram of Fig. 2. As can be seen in both classes, public methods explicitly invoke the execution of methods A(..) and B(..) which does not correspond to the responsibility nature of public methods of Class1 and Class2, respectively. That is, neither the classes nor the methods of this example respect the principle of sole responsibility.

Such as Kiczales et al. [1] pointed out, classic or traditional AOP permits the elimination of cross-concerns in classes of object-oriented software systems modularizing them as aspects. However, as Bodden [9] points out, Instroza et al. [10], and Bodden et al. [11], traditional AOP Aspect-style does not permit achieving a complete modularization for the existence of implicit dependencies between aspects and classes. For getting software products with greater modularity, the works of [9] [10] [11] present Join Point Interfaces (JPI) to eliminate implicit dependencies between classes and aspects of classic AOP AspectJ-style.

Considering the mentioned JPI benefits for developing

*Corresponding author

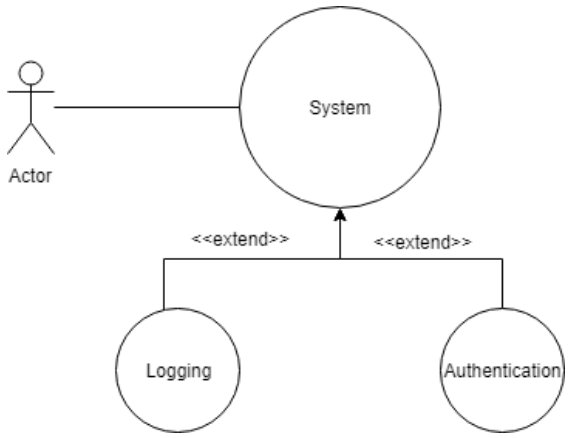

Fig. 1. Example of Cross-Concern for Logging the Actions and Authenticating in a Use Cases Diagram of a System.

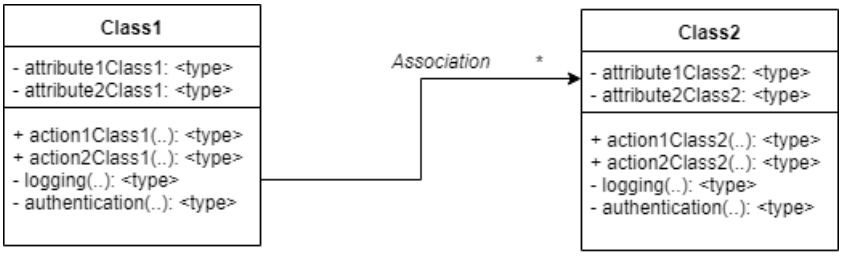

Fig. 2. Example of UML Class Diagram with Cross-Concerns.

TABLE I. JAVA EXAMPLE WITH CROSS-CONCERNS.

```
public class Class1 {
    public <type>attribute1Class1;
    public <type>attribute2Class1;

    public <type>action1Class1([Arg1,., ArgN]){
        authentication(..);
        ...
        logging(..);
    }

    public <type>action2Class1([Arg1,., ArgN]){
        authentication(..) ;
        ...
        logging(..);
    }
}
```

```
public class Class2 {
    public <type>attribute1Class2;
    public <type>attribute2Class2;

    public <type>action1Class2([Arg1,., ArgN]){
        authentication(..);
        ...
        logging(..);
    }

    public <type>action2Class2([Arg1,., ArgN]){
        authentication(..);
        ...
        logging(..);
    }
}
```

modular software, the main objective of this article is to apply JPI concepts on a base example to demonstrate and describe its practical advantages for getting modular aspect-oriented software solutions without implicit dependencies. Moreover, this article proposes an approach for the structural modeling of JPI solutions along with gives the bases for future research for behavioral modeling of JPI solutions.

This work organizes as follows: next section summarizes related works. Section 3 presents and exemplifies the AspectJ and JPI aspect-oriented programming approaches. Section 4 describes and exemplifies UML class diagrams along with to propose and exemplify a UML class diagram for JPI solutions. Finally, conclusions and future research work regarding behavioral modeling ideas using UML sequence diagrams for JPI solutions.

II. RELATED WORKS

Different works about AOP applications and extensions already exist such us [12] and [13] which define formal rules to specify AOP solutions and [14] that uses an AOP framework to monitor the run-time state and behavior of software applications. The works of [9] [10] and [11] are

bases and programming background of the JPI framework for building AOP solutions. The work of [15] represent one of the first article to describe and exemplify JPI modeling ideas. This article describe an exemplify a more precised and detailed structural modeling approach for JPI solutions which represent advances to look for a complete JPI software development process.

III. Aspect-Oriented Programming

This section summarizes AspectJ and JPI AOP styles.

A. AspectJ-Style AOP

Kiczales et al. [1] argue that the aspects, advice units, introductions, relations between data-types, join points, and pointcuts represent relevant characteristics of traditional AspectJ-like AOP. Also, there is a difference between base elements and aspects, which are usually associated. Pointcuts in aspects define the relations between aspects and classes, and aspects can advise methods related to their pointcuts, or through the inclusion of new behavior and attributes in advised classes, that is, by introductions or declarations among types in those classes.

In classical AOP, base elements of a system are oblivious about their interaction with aspects of the system, as well as the possibility of being advised, that is, base elements know nothing about any change or new behavior. In this way, a class that does not expect interruptions and changes in its functionality may experience unexpected changes. According to Bodden et al. [11], this is one of the implicit dependencies that exist in classic AOP as well as in any previous AOP like AspectJ modeling proposal.

Implicit dependencies between software modules complicate the software development process. Eliminating implicit dependencies between aspects and classes would facilitate the software development by independent groups, one in charge of system classes and another one in charge of aspects of the system. That approach seems adequate for software evolution. The work of [11] indicate that a solution is a complete knowledge of the classes of the system by the developers of aspects, and also, constant communication between development teams is necessary before any change in the base elements, as well as, to indicate all already advised class elements. This complete communication seems viable in small software development teams, but it is not always possible to develop software between large and multiple independent teams.

AspectJ uses execution(..) in the definition of pointcuts [16] [8] to capture defined objects which execute one of its methods. For example, by using execution(type C2.met (..)) && this (obj1) && target (obj2), obj1 and obj2 represent the same object of class C2 that executes the method met (..). In the same way, as pointed out by [16] and [8], in addition to capturing the object that executes a method, it is also possible to capture the object that invokes the execution of a method of a given object, which could be the same. For example, in the definition of the cutoff point call (type C2.met (..)) && this (obj1) && target (c2), obj1 represents the object that invokes the call of the method met (..) of an object of class C2, while obj2 represents the object of class C2 that executes the method met (..).

Table II presents the AOP AspectJ code solution for the example code of Fig. 3. As shown in this figure, the principle of sole responsibility is respected in Class1 and Class2 classes. In this example, the methods, methods action1Class1(..) and action2Class1(..), as well as action1Class2(..) and action2Class2(..), of the classes Class1 and Class2 respectively, are oblivious concerning the inclusion of aspects behavior. This ingenuity of the advised methods is problematic for methods that must preserve intact their original behavior. That represents one of the implicit dependencies between classes and previously mentioned aspects. In the same way, if the signature of one of the public methods of Class1 or Class2 changed, potentially Aspect1 and Aspect2 would not be effective, that is, there would be no join point, and besides, no compilation errors exist. The following subsection describes the JPI approach proposed by [10] and [11] to look for eliminating these dependencies between aspects and classes.

B. JPI

Such as [15] remark, the main difference of JPI concerning the classic POA is, as its name indicates, the use of join point interfaces as intermediate points of the association between classes and aspects. In this way, JPI allows the elimination of oblivious classes since advisable classes explicitly exhibit join point interfaces and define pointcut rules for the effectiveness of those unions. In the same way, aspects implement those interfaces and do not know directly about the advisable classes. Therefore, JPI eliminates the implicit dependencies between classes and aspects which permit reaching higher levels of modularization regarding classical AOP. JPI, as an extension of AspectJ, supports traditional AspectJ code to facilitate the adaptation of AspectJ code to JPI. JPI, like traditional AOP, allows declaration between types of classes in aspects which do not require explicit join point interfaces, that is, classes continue being oblivious regarding the introduction of attributes and behavior by aspects.

Table III shows a JPI solution for the classes and aspects of the AOP AspectJ-style example of Table II. Table III presents a new code box for the join point interfaces JPIAauthentication and JPILogging regarding the classes and aspects of Table II. As seen in Table III, each aspect, to be effective advising classes, requires implementing join point interfaces exhibited by those classes. Thus, Aspect1 implements the JPIAuthentication and Aspect2 implements the JPILogging join point interfaces respectively, both exhibited by Class1 and Class2.

As mentioned above, a join point interface can be used for the inclusion of new methods and attributes, that is, for the inter-type declaration in oblivious classes. JPI also allows defining global join point interfaces for an AspectJ style of AOP (Bodden et al. 2014).

We can appreciate in the JPI code of Table III, classes Class1 and Class2 explicitly exhibit the join point interfaces for the execution of any of their public methods. In this way, JPI allows the elimination of implicit dependencies between classes and aspects of traditional AOP. First, non-oblivious classes indicate their methods associated with a join point interface, that is, their advisable methods. Second, if a class that exhibits join point interfaces regarding some method's signature and that method undergoes some signature change

TABLE II. AOP ASPECTJ CODE FOR THE EXAMPLE.

```
public class Class1 {
    public <type>attribute1Class1;
    public <type>attribute2Class1;

    public <type>action1Class1([Arg1,., ArgN]){
        ...
    }

    public <type>action2Class1([Arg1,., ArgN]){
        ...
    }
}
```

```
public class Class2 {
    public <type>attribute1Class2;
    public <type>attribute2Class2;

    public <type>action1Class2([Arg1,., ArgN]){
        ...
    }

    public <type>action2Class2([Arg1,., ArgN]){
        ...
    }
}
```

```
public aspect Aspect1 {
    pointcut pcAuthentication(..):
    execution(<type>Class1.action1Class1(
        [Arg1, .., ArgN])) ||
    execution(<type>Class1.action2Class1(
        [Arg1, .., ArgN])) ||
    execution(<type>Class2.action1Class2(
        [Arg1, .., ArgN])) ||
    execution(<typo>Class2.action2Class2(
        [Arg1, .., ArgN])) ... ;

    before(..): pcAuthenticate(..){
        ... //Authentification
    }
}
```

```
public aspect Aspect2 {
    pointcut pcLogging(..):
    execution(<type>Class1.action1Class1(
        [Arg1, .., ArgN])) ||
    execution(<type>Class1.action2Class1(
        [Arg1, .., ArgN])) ||
    execution(<type>Class2.action1Class2(
        [Arg1, .., ArgN])) ||
    execution(<typo>Class2.action2Class2(
        [Arg1, .., ArgN])) ... ;

    after(..): pcLoggin(..){
        ... //Logging
    }
}
```

without updating the pointcut rule in the JPI exhibition, then a compilation error occurs. That is, the class developer team must indicate changes in the methods' signature in the associated joint point interface exhibition. Then, in JPI, with a clear definition of join point interfaces, classes and aspects development teams could exist.

IV. JPI UML CLASS DIAGRAM: PROPOSAL AND APPLICATION

A UML class diagram represents the classes of a software system along with their associations [17]. Such as Vidal et al. 2015 [15] and Torres et al. [18] argue, a UML class diagram allows classifying classes and their associations through the use of stereotypes besides. For example, usually, a class interface is represented by an <<interface>> stereotyped class. In this way, UML class diagrams seem suitable for the representation of JPI solutions.

Next, we define rules and names of elements for a JPI UML class diagram:

- Classes and their associations are defined in the usual way as in a UML class diagram.
- A join point interface is declared with the stereotype <<jpi>> or <<global jpi>> depending on whether the advisable classes explicitly or implicitly exhibit those interface, respectively. In this proposal of JPI UML class diagrams, a join point interface does not have attributes either methods, that is, a JPI interface represents a method without a signature.
- An aspect, which is a stereotyped class with <<aspect>>, allows to define a series of variables and methods of aspects, as well as to define methods of interfaces of point of union, and declarations between types or introductions.
- Classes can exhibit interfaces of stereotyped junctions with <<jpi>>. In this way, when a class exhibits a junction point interface, there is an association of the class to a join point interface. The role of the class of this association presents a first line with the stereotype <<exhibits>> together with the signature of the interface, and a second line with the pointcut rule.
- A global join point interface includes an association to the recommended class with a line that indicates the signature of the join point and another line with the definition of the pointcut rule.
- The aspects, to effectively advise this is, to add behavior on the call or execution of methods of advisable classes, they must implement join point interfaces. For this reason, each aspect, to be effective, presents an association towards the associated point of attachment interfaces. The role of the aspect in these associations is 'implements'.
- The aspects allow inter-type declaration, that is, to add attributes and methods to existing classes. Thus, an association between aspects and classes is used, where the role of the aspect is 'adding'.

TABLE III. AOP JPI CODE FOR THE EXAMPLE.

```
public class Class1 {
    exhibits JPIAuthentication(..):
        execution(* action1Class1([Arg1, ArgN]) ||
        execution(* action2Class1([Arg1, ArgN])) &&
        args(..);

    exhibits JPILogging(..):
        execution(* action1Class1([Arg1, ArgN]) ||
        execution(* action2Class1([Arg1, ArgN])) &&
        args(..);

    public <type>attribute1Class1;
    public <type>attribute2Class1;

    public <type>action1Class1([Arg1,., ArgN]){
        ...
    }

    public <type>action2Class1([Arg1,., ArgN]){
        ...
    }
}
```

```
public class Class2 {
    exhibits JPIAuthentication(..):
        execution(* action1Class2([Arg1, ArgN]) ||
        execution(* action2Class2([Arg1, ArgN])) &&
        args(..);

    exhibits JPILogging(..):
        execution(* action1Class2([Arg1, ArgN]) ||
        execution(* action2Class2([Arg1, ArgN])) &&
        args(..);

    public <type>attribute1Class2;
    public <type>attribute2Class2;

    public <type>action1Class2([Arg1,., ArgN]){
        ...
    }

    public <type>action2Class2([Arg1,., ArgN]){
        ...
    }
}
```

```
jpi JPIAuthentication(..);
jpi JPILogging(..);
```

```
public aspect Aspect1 {
    before JPIAuthentication(..){
        ... //Authentication
    }
}
```

```
public aspect Aspect2 {
    after JPILogging(..){
        ... //Logging Actions
    }
}
```

Because the proposed JPI UML class diagram extension considers the use of stereotypes and special keywords, any UML design tool can be used for the JPI UML diagram design. In this way, it is possible to model a JPI solution and system structurally. Fig. 3 presents an application of this proposal of UML JPI class diagrams on the JPI example of Table III. Note that JPIInterfaceA corresponds to JPIAauthentication while JPIInterfaceB corresponds to JPILogging. Similarly, AspectA corresponds to Aspect1 and AspectB with aspect2 of Table III. As Fig. 3 shows, there is a clear analogy between the number of components in the JPI UML class diagram and the JPI code solution. Precisely, to review the consistency and modular advantages of this proposal of UML JPI class diagrams is part of the future work for the authors of this work.

It should be noted that for traditional AOP UML class diagram, there are already proposals such as Kojarski et al. [19] and [20] which use existing UML modeling tools.

V. CONCLUSIONS

JPI makes possible the generation of aspect-oriented solutions without implicit dependencies, which in turn allows achieving a high degree of modularization concerning traditional AOP. As this paper mentioned, JPI enables the definition of introductions without join point interfaces, that is, for oblivious classes of the introduction of new attributes and behavior, however, those classes are no longer oblivious about changes in the behavior of their methods through aspects' advice units.

For the structural modeling of JPI applications, this work extends UML class diagrams using JPI concepts for the modeling JPI solutions. As presented in the modeling example, our UML class diagrams proposal for JPI captures basic elements of JPI such as global or non-global join point interfaces. Other elements of JPI, such as closure and generic join points [9], [10], [11], are part of future extensions to this modeling proposal. Also, this proposal of UML JPI class diagrams allows defining oblivious classes, which is an essential element to achieve a complete adaptation and transformation of AOP solutions into JPI solutions.

As future work, the authors of this article work on a complete proposal of structural modeling for JPI applications, as well as on ideas for modeling the behavior of JPI systems through UML sequence diagrams. For this last idea of future work, the actors of each scenario are identified, where the aspects are clear participants and, for which, the participating objects communicate in the existence of join points, that is,

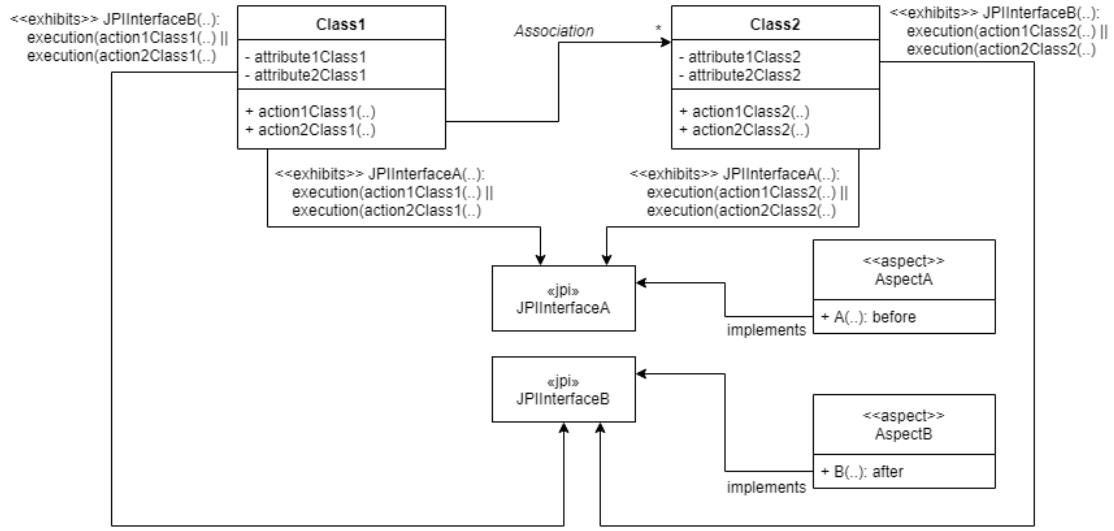

Fig. 3. Example of JPI UML Class Diagram.

respecting the pointcut rules. The idea is to model, through UML diagrams, the structure, and behavior of JPI solutions, to work on the generation of JPI code from the models. From a Software Engineering point-of-view, we want to use JPI as an approach for the modular and consistent software development approach.

REFERENCES

[1] G. Kiczales, "Aspect-oriented programming," *ACM Comput. Surv.*, vol. 28, no. 4es, Dec. 1996.

[2] C. V. Silva, R. Saens, C. D. Río, and R. Villarroel, "Aspect-oriented modeling: Applying aspect-oriented UML use cases and extending aspect-z," *Computing and Informatics*, vol. 32, no. 3, pp. 573–593, 2013.

[3] C. V. Silva, R. Villarroel, and C. P. Vasquez, "Jpiaspectz: A formal specification language for aspect-oriented JPI applications," in *33rd International Conference of the Chilean Computer Science Society, SCCC 2014, Talca, Maule, Chile, November 8-14, 2014*, 2014, pp. 128–131.

[4] C. V. Silva, R. Villarroel, R. S. Simón, R. Saens, T. Tigero, and C. D. Río, "Aspect-oriented formal modeling: (aspectz + object-z) = ooaspectz," *Computing and Informatics*, vol. 34, no. 5, pp. 996–1016, 2015.

[5] D. Wampler, *Aspect-oriented design principles: Lessons from object-oriented design*, 1st ed. Vancouver, Canada: Proceedings of the Sixth International Conference on Aspect-Oriented Software Development (AOSD'07), 2007.

[6] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[7] I. Jacobson, "Use cases and aspects-working seamlessly together," *Journal of Object Technology*, vol. 2, no. 4, pp. 7–28, 2003.

[8] I. Jacobson and P.-W. Ng, *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

[9] E. Bodden, "Closure joinpoints: Block joinpoints without surprises," in *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, ser. AOSD '11. New York, NY, USA: ACM, 2011, pp. 117–128.

[10] M. Inostroza, E. Tanter, and E. 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*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 508–511.

[11] E. Bodden, E. Tanter, and M. Inostroza, "Join point interfaces for safe and flexible decoupling of aspects," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 7:1–7:41, Feb. 2014.

[12] C. Vidal Silva, R. Saens, C. Del Río, and R. Villarroel, "Ooaspectz and aspect-oriented uml class diagrams for aspect-oriented software modelling (aosm)," *Ingeniería e Investigación*, vol. 33, no. 3, pp. 66–71, 2013.

[13] C. Vidal Silva, R. Villarroel, R. Schmal Simon, R. Saens, T. Tigero, and C. Del Rio, "Aspect-oriented formal modeling:(aspectz+ object-z)= ooaspectz," *Computing and Informatics*, vol. 34, no. 5, pp. 996–1016, 2016.

[14] A. O. AL-Zaghameem, "An aspect oriented programming framework to support transparent runtime monitoring of applications," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 6, 2019.

[15] C. V. Silva, L. López, R. Schmal, R. Villarroel, M. Bustamante, and V. R. Sanchez, "Jpi uml software modeling," *International Journal of Advanced Computer Science and Applications*, vol. 6, no. 12, 2015.

[16] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Greenwich, CT, USA: Manning Publications Co., 2003.

[17] T. Pender, *UML Bible*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[18] D. Torre, Y. Labiche, M. Genero, and M. Elaasar, "A systematic identification of consistency rules for uml diagrams," *Journal of Systems and Software*, vol. 144, pp. 121–142, 2018.

[19] S. Kojarski and D. H. Lorenz, "Modeling aspect mechanisms: A top-down approach," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 212–221.

[20] F. F. Silveira, A. M. da Cunha, and M. L. Lisbôa, "A state-based testing method for detecting aspect composition faults," in *Computational Science and Its Applications – ICCSA 2014*, B. Murgante, S. Misra, A. M. A. C. Rocha, C. Torre, J. G. Rocha, M. I. Falcão, D. Taniar, B. O. Apduhan, and O. Gervasi, Eds. Cham: Springer International Publishing, 2014, pp. 418–433.