Mapping Connection Templates to Spring Aspects to Integrate Business Rules

Sandra Casas
Universidad Nacional de la Patagonia Austral
Piloto Rivera S/N
Río Gallegos, Argentina
54 2966 442313
scasas@unpa.edu.ar

Juan G. Enriquez
Universidad Nacional de la Patagonia Austral
Piloto Rivera S/N
Río Gallegos, Argentina
54 2966 442313
jenriquez@.unpa.edu.ar

ABSTRACT
AOP is a convenient approach to encapsulate business rule connection code since AOP reduces dependences and coupling; thus, best reusing is achieved and maintenance efforts reduced. In our previous works we proposed strategies to assist the developer in the operation face of software, to manage aspectual connections. These strategies are: a template (ACT) to document the aspectual connections and taxonomy of them. In this work, we propose a new step in this course, the implementation of ACT in XML, and a mapping process to generate automatically Spring aspects.

Categories and Subject Descriptors
D.2.3 [**Software Engineering**]: Coding Tools and Techniques - *Object-oriented programming, Pretty printers*, *Program editors*, *Reentrant code*, *Standards, Structured programming*

General Terms
Design. Languages. Documentation.

Keywords
Business Rule, Spring, Volatile concerns, AspectJ, AOP.

1. INTRODUCTION
Business rule connection code it is refer to the code not only in charge of triggering the application of the rules at certain events, but also gathering the necessary information for their application and incorporating their results in the rest of the core application functionality. It has been observed that, even when the decoupling of the business rules is successfully achieved, the connection code is still tangled with and scattered in the core application functionality [1]. This situation occurs independently of the concrete approach [2][3][4] used for representing the rules. Therefore, when the current business rules change and they need to be integrated in the existing application; or new business rules are added which need to be connected at unanticipated events of the core application, the source code of the core application must be adapted manually at different places. Consequently, it becomes difficult to localize, add, change or remove rule connections. Another issue is that the business rules are volatile concerns, that is to say, they change all the time. Then, the rule connection code must change too, therefore it is suitable that the rule be isolated and thus easy to localize.

AOP [5] is a convenient approach to encapsulate business rule connection code. AOP reduces dependences and coupling; thus, best reusing is achieved and maintenance efforts reduced. Some works [6][7][8][9][10] have studied the implementation of rule connection with AOP. Others works have considered the handling of volatile concerns in early stage of software development with aspects [11][12][13]. However, the mapping of rule connections to aspects has been less explored.

In [14], we present taxonomy of aspectual connections that can serve to identify the different elements of the possible aspectual connections and the situations where they can occur, in a commendable schema. Furthermore we have proposed a template to document the aspectual connections (ACT). However we have observed, mainly in complex applications such as business-to-business (B2B) and business-to-consumer (B2C) systems, where rules play an important role and aspectual connections link business rules with the core functionality, it is necessary to manage these connections to really assist the developers. The automatic mechanisms are needful too, in order to ease the business rules deployment, software maintenance and evolution.

In this work we provide a method to map the ACT to Spring AOP framework, automatically. The taxonomy [14] guides the mapping process. ACT and mapping process could be used in different stages of software development; design and implementation of volatile concerns, after core domain construction; or in order to maintain volatile concerns. Also, ACT could be used to generate source-code in other aspect-oriented languages.

The remainder of this work continues as follows: Section 2 presents the Spring AOP approach in brief. In Section 3 the aspectual connections taxonomy is explained, because of being a guide for next sections. In Section 4, we describe the ACT in XML format. In Section 5, we present the mapping process to transform ACT in Spring aspects and a simple example. Finally, in Section 6, we finish with conclusions, related and future works.

2. SPRING AOP IN A NUTSHELL
The J2EE specification and especially the EJB specification has been criticized for being too heavy-weight. There are popping up

EA'11, 21-MAR-2011, Porto de Galinhas, Brazil

open-source component frameworks that are trying to make it easier and more lightweight to create enterprise applications. The Spring Framework [15] is one such framework. The Spring Framework is based on the Inversion of Control pattern, or Dependency Injection (DIP) as it is called now. DIP is also called "The Hollywood Principle" – don't call us, we call you. Spring provides a simple AOP framework. This is mainly a means of inserting hooks into the composition. However, Spring is designed to be used with other AOP frameworks such as AspectJ [17]. It has well defined points and flows for pointcuts. Even though the AOP framework is simple, it still enables Spring to separate some complex crosscutting concerns. This includes remoting and declarative transactions. Spring´s support for AOP comes in four flavors: Classic Spring proxy-based AOP; @AspectJ annotation-driven aspects; Pure-POJO aspects (Schema-based AOP support); and Injected AspectJ aspects. The first three are all variations on Spring´s proxy-based AOP. Consequently, Spring´s AOP support is limited to method interceptions.

We have experienced the business rule connections with pure-POJO aspects approach [10]. We found that the instantiation aspect model (Singleton) is an important restriction, particularly when it is necessary to implement complex connections. For this reason, in this work, we adopt the @AspectJ annotation-driven aspects approach.

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. The @AspectJ style was introduced by the AspectJ project as part of the AspectJ 5 release. Spring 2.0 interprets the same annotations as AspectJ 5, using a library supplied by AspectJ for pointcut parsing and matching. The AOP runtime is still pure Spring AOP though, and there is no dependency on the AspectJ compiler or weaver.

Aspects (classes annotated with `@Aspect`) may have methods and fields just like any other class. They may also contain pointcut, advice, and introduction (inter-type) declarations (they should be annotated with @Pointcut, @Before, @After, etc.) A pointcut declaration has two parts: a signature comprising a name and any parameters, and a pointcut expression that determines exactly which method executions we are interested in. In the @AspectJ annotation-style of AOP, a pointcut signature is provided by a regular method definition, and the pointcut expression is indicated using the `@Pointcut` annotation (the method serving as the pointcut signature must have a `void` return type). For example, in Figure 1, the Logging class is an aspect, the LogPoint pointcut defines the interception of debit method of Account class. An after advice is defined to the LogPoint pointcut.

Aspect must be declared in the application context, like any other Spring bean. In this declaration also it is also possible to configure the scope and order properties. Scope or instantiation can be singleton, prototype, etc. Order property is used to solve the situation when two or more aspects have the same pointcut and advice, it is also know as interaction or conflict.

```
@Aspect
public class Logging {

@Pointcut("execution Account.debit(..)")
public void LogPoint ()
 {}

@After("LogPoint()")
public void register()
{ // register information of debit
 // operation    }
}
```

Figure 1. An aspect in @AspectJ annotation-driven aspects approach.

3. TAXONOMY OF ASPECTUAL CONNECTION AND SPRING SUPPORT.

An aspectual connection links a business rule with core functionality; it is any implementation mechanism that encapsulates: the object rule invocation; the transmission of the information required by the business rule; the interaction resolution among business rules and the information returned by the rule.

The aspectual connection must satisfy some requirements for business rule could be triggered. According to the imposed domain constrains, we clearly can identify three categories of aspectual connections: basic, query and change. Next we explain these categories and their implementation in @AspectJ annotation-style approach. The text in <> symbols is generic, they should be replaced by the code of particular cases. For all cases, it is assumed that the implementation of business rules classes follows the Object Rule pattern [2] whose interface consists of apply(..), condition(..) and action (..) methods, this interface is denominated Rule.

Basic Aspectual Connection: the aspectual connection triggers the business rule in a specific point of the core functionality (event), the information required by the business rule is either available in the event context or it is global system information.

In Spring this aspectual connection could be implemented as a class with a field for the rule (Figure 2). The mainEvent method is the pointcut. The triggerRule method is an advice that triggers the business rule.

```
@Aspect
public class <id_aspect_connection> {
  private Rule rule;

  @Pointcut("execution(<jointpoint> [and
                                    <context>]")
  public void mainEvent([<context>]) {}

  @<advice>("mainEvent([<context>])")
  public void triggerRule([<context>]) {
    this.rule.apply([<context>]);
  }
  public Rule getRule() {
   return rule;
  }
  public void setRule(Rule rule) {
   this.rule = rule;
  }
}
```

Figure 2. Implementation of basic aspectual connection.

Although it is necessary to configure the Application Context (XML file). First, it is necessary to inject the business rule to aspect bean. After, the aspect (a simple bean) should be instanced and scope and order properties should be defined.

Query Aspectual Connection: the aspectual connection triggers the business rule in a specific point of the core functionality but the information required by the business rule is not available in the event context. Then connection must first retrieve the information in order for it to be available when the business rule be applied. In this case, the aspectual connection should manage two event (pointcuts) and two advices, each one with different purpose.

In Spring this aspectual connection is similar to basic connection, (Figure 3), because of containing the same methods, but also a new pair of pointcut-advice is added to retrieve the information of another event. When this event is intercepted, the information is held in a field (<retrieved_field>) of the aspect. This field is used in the moment of triggering the rule.

```
@Aspect
public class <id_aspect_connection>  {
  private Rule rule;
  private <retrieved field>

  @Pointcut("execution(<jointpoint> [and
                                  <context>]")
  public void mainEvent([<context>]) {}
  <@advice>(“mainEvent [and <context>]”)
  public void triggerRule([Object obj]) {
    this.rule.apply([<context>,]
                    <retrieve_field);
  }
  @Pointcut("execution(<jointpoint> and
                                 <context>")
  public void queryEvent(<context>) {}
  <@advice>("queryEvent and <context>")
  public void retrieveContext(<context>) {
    // assign retrieve context to
    // <retrieved field>
  }
  // getRule and setRule methods
}
```

Figure 3. Implementation of query aspectual connection.

Change Aspectual Connection: the aspectual connection should add new properties (fields/methods) to the core functionality components, so that a business rule could be triggered. It means, the new business rule requires adapting the domain vocabulary. Then the connection must support the domain adaptation such as the addition of new fields and methods in existing classes.

In AOP this operation is well-know as introduction or inter-type declarations.

```
@Aspect
public class <id_aspect_connection>  {
  private Rule rule;
  @DeclareParents(value=<"change_class">,
     defaultImpl=<”class_new_properties”>)

  @Pointcut("execution(<jointpoint> [and
                                 <context>]")
  public void mainEvent([<context>]) {}
  <@advice>(“mainEvent [and <context>]”)
  public void triggerRule(<context>) {
    this.rule.apply(<context>);
  }
   // getRule and setRule methods
}
```

Figure 4. Implementation of change aspectual connection.

In Spring the way to modify a class, is to create a new class with the new methods and fields to add. In the aspect, the annotation sentence @DeclareParents has two arguments: the class to be adapted for value attribute and the class with new properties (Figure 4). The modification of domain class occurs when the system of Spring executes the configuration. After, a business rule can be applied to these new properties, as a basic connection.

4. ASPECTUAL CONNECTION TEMPLATES.

Initially we propose the ACT [14] as artifact to identify and document the required elements in order to define a connection between a business rule and base functionality. Also, ACT allows analyzing and classifying the aspectual connection before implementation. In this first version, the ACT was specified in a natural language format. Now, we propose implementing ACT in XML format. Due to the fact that we pretend to use it for later mapping process, we have to include more implementations details. Figure 5 presents a DTD.

```
<!ELEMENT act (aspect, businessRule, mainEvent,
queryEvent?, changeEvent?, relations?)>
<!ELEMENT businessRule (businessRuleClass,
require*, return?)>
<!ELEMENT mainEvent (eventJP, this?, arg*)>
<!ELEMENT queryEvent (eventJP, this?, arg*,
retrieve*)>
<!ELEMENT changeEvent (classBase, changeClassEvent,
addField*, addMethod*)>
<!ELEMENT addMethod (arg*)>

<!ATTLIST act category (basic|query|change)
"basic">
<!ATTLIST aspect instantiation (singleton |
prototype ) "singleton" >
<!ATTLIST aspect order CDATA #REQUIRED>
<!ATTLIST require type CDATA #REQUIRED>
<!ATTLIST this type CDATA #REQUIRED>
<!ATTLIST arg type CDATA #REQUIRED>
<!ATTLIST retrieve type CDATA #REQUIRED>
<!ATTLIST mainEvent when ( After | Before | Around)
"After">
<!ATTLIST queryEvent when ( After | Before |
Around) "After">
<!ATTLIST addField type CDATA #REQUIRED>
<!ATTLIST addMethod name CDATA #REQUIRED return
CDATA #REQUIRED>

<!ELEMENT aspect (#PCDATA)>
<!ELEMENT businessRuleClass (#PCDATA)>
<!ELEMENT require (#PCDATA)>
<!ELEMENT return (#PCDATA)>
<!ELEMENT eventJP (#PCDATA)>
<!ELEMENT this (#PCDATA)>
<!ELEMENT arg (#PCDATA)>
<!ELEMENT retrieve (#PCDATA)>
<!ELEMENT classBase (#PCDATA)>
<!ELEMENT changeClassEvent (#PCDATA)>
<!ELEMENT addField (#PCDATA)>
```

Figure 5. Document Type Definition (DTD) of ACT.

According to this definition, the root element “act” has a category attribute that indicates the category of taxonomy (basic, query or change). Inside, in the same level, there are six sections or tags:

aspect, businessRule, mainEvent, queryEvent, changeEvent and relations.

Aspect tag has two attributes; instantiation and order; value of this element corresponds to the aspect name. BusinessRule tag consists of several elements to describe the business rule as the class, inputs and outputs rule. MainEvent tag is a set of attributes and elements that describe the event and moment that business rule is triggered, also the context of this event it should be passed to aspect. QueryEvent tag consists of necessary elements to retrieve information of another context. ChangeEvent tag, indicates elements (fields and methods) that are necessary in order to adapt a domain to applied a business rule. When (after/before/around) and eventJP (jointpoint) are attributes of some tag, as mainEvent and queryEvent.

ACT category is basic if aspect, businessRule and mainEvent sections are completed; it is query if also queryEvent section is completed; and it is change if also changeEvent section is completed. Then aspect, businessRule and mainEvent sections are mandatory.

5. MAPPING ACT TO SPRING ASPECTS.

In this section, we explain the mapping process (rules), to transform the ACT to Spring aspects. We use the taxonomy (presented in Section 3) to define the necessary steps for each case. In Table 1, the actions are described; the text enclosed in <> symbols referred to elements mentioned in Section 3.

Table 1. Mapping Actions.

Basic ACT
1. Create empty class annotated with @Aspect Replace <id_aspect_connection> with value of aspect tag of ACT.
2. Create mainEvent method with @Pointcut annotation. Replace <joinpoint> with eventJP element of mainEvent tag of ACT.
3. Create triggerRule method with @<advice> annotation, replace <advice> with when attribute of mainEvent tag of ACT.
4. If this/arg tag of mainEvent element is not empty, then use to replace <context>
5. Configuration: Declare a bean to aspect class. Inject the business rule bean. Set bean aspect according instantiation and order attributes of aspect tag of ACT.
Query ACT
Apply step 1, 2, 3 and 4 for basic ACT.
5. Create a declaration for <retrieved_field> in aspect.
6. Create queryEvent method with @Pointcut annotation. Replace <joinpoint> with eventJP element of queryEvent tag of ACT.
7. Create retrieveContext method with @<advice> Annotation. Replace <advice> with when attribute of queryEvent tag of ACT.
8. Replace <context> with this/arg tag not Empty. Assign <context> to <retrieved_field>
9. Apply step 5 for basic ACT.

Change ACT
Apply step 1, 2, 3 and 4 for basic ACT.
5. Create @DeclareParents annotation. Replace <"change_class"> with classBase of ACT. Replace <"class_new_properties"> with changeClassEvent of ACT.
6. Apply step 5 for basic ACT.

The algorithms analyze the type of required data by business rules and the read data of the context events (main event or query event) and perform the necessary operations.

Next, we enlighten the mapping process, with very simple example of a store. The logic of the business dictates firstly, the customer orders are registered (Order), these operations include the customer data (Customer) and requested items (Item). Later on (the same day or another), when the customer wants to place the order and pay it, the invoice (Invoice) is performed, then the system calculates subtotal, discount and total. Each invoice maintains a copy of customer details (CustomerDetails) for print. After the invoice is created, customer purchase quantity is incremented (inv_count field in Customer class). The Figure 6 shows a simplify diagram class of the store.

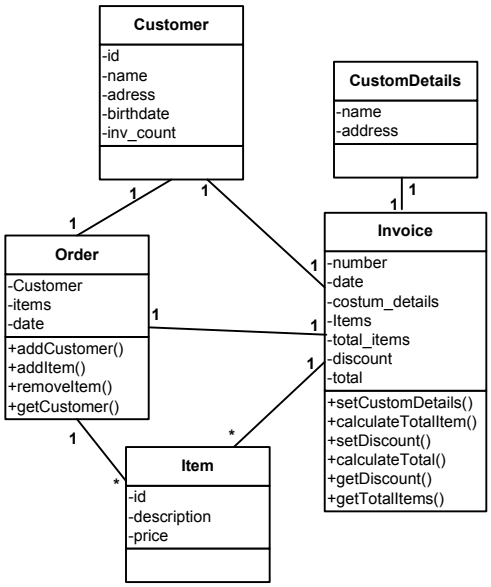

Figure 6. Summarized diagram of the Store.

The business rules that should be added in the application are classes that implement condition(), action() and apply() methods. These rules are:

BR#1: if today is store anniversary *then* apply a discount of 0,5% when perform the invoice.

BR#2: if invoice date is equivalent with customer birth date *then* apply a discount of 1% when perform the invoice.

According to the schema showed in Figure 6, for BR#1 is necessary a basic connection, and for BR#2 is necessary a query connection. ACT and the mapping process are showed in Figures 7 and 8, respectively. Each step of the mapping process is indicated with numbered arrows.

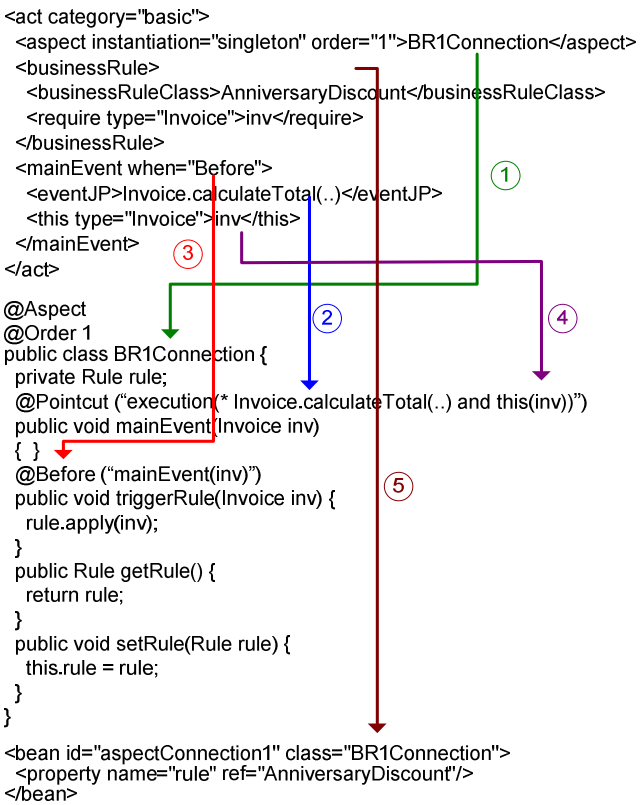

Figure7. Mapping process for connecting BR#1.

6. CONCLUSIONS

Some works have dealt with aspectual connections to compose business rules, but they have covered implementation approaches with different AOP tool, such as AspectJ[8], JasCo[9] and Spring AOP Framework [10]. These works, mainly analyze the reuse and flexibility of implemented code. [6] presents a template to implement the business rule with AspectJ. [7] presents an experience of refactoring business rule with AspectJ, in an important J2EE application. These contributions do not propose automatic method to generate aspect code.

Others contributions consider the handling of volatile concerns in early stage of software development. For example, an interesting contribution is [11], the authors present a method for handling volatile concerns during early lifecycle software modeling. The method consists of several steps: concern classification, requirements refactoring, model instantiation and model composition. These techniques improve the business rules ant their aspectual connection in modeling activities but not their implementation directly. Along the same line, a framework [12] [13] is proposed to identify volatile and crosscutting concerns at the requirements level. The identification of such concerns is based on a crosscutting pattern and simple matrix operations. Neither these works propose automatic methods to generate aspect code. Again, these contributions do not propose automatic method to generate aspect code.

Perhaps, the contributions more related with our work are [16] [18]. This approach consists of defining business rules and their connections to the existing applications in dedicated, high-level languages and expressing them in terms of the domain. In order to make these rules executable and integrate them with the existing application according to the connections, they follow a Model-Driven Engineering (MDE) approach: the rules and connections are automatically translated to object-oriented and aspect-oriented programs, respectively. The transformations use a mapping from the domain entities, used in the high-level rules and connections, to implementation elements in the existing application. The differences with our work are, i) the aspect-programming language destination of mapping is JasCo [19]; ii) they transform not only aspectual connections to aspects; they also transform the business rules. We assumed the business rules exist as reusable units.

Figure 8. Mapping process for connecting BR#2.

Another discussion, less evident, is the different approaches to implement aspectual connections. With the goal of achieving the reusability of aspects; Cibrian [8] proposes the next guidelines: i) code an aspect to express the event that triggers the business rule (pointcut); ii) code an aspect (optional) that captures and/or exposes the context not available or introduce information not anticipated; iii) code an aspect that unifies aspects created, and triggers the business rule. This implementation approach outlines some questions, when it is necessary to update, remove or add business rule very frequently: a) how many aspects must to be considered to perform operations correctly? b) Where and how the aspect relationships that form one connection are maintained,

so that the connection entity does not lose; c) how these different parts interact with other parts of others aspectual connections; d) how to manage the dependencies among business rules and their aspectual connections. In others words, this approach generates a proliferation of aspects, therefore it is difficult to maintain operations, and for this reason our approach proposes to implement each connection in one aspect.

As we have mentioned, in our previous work we have proposed taxonomy of aspectual connections, and a template to document them. These strategies allow to identify the elements of each category of connections, as events, pointcuts and advices required, additional declarations, etc. Both strategies are independent of any AOP tool. In this work, we use them to generate aspect code automatically in Spring. We chose Spring AOP framework for two reasons. First, it is a very popular tool in Java community developers; second, it is not very used in AOP community research, which is very influenced by AspectJ. Then in this sense, this work covers the expectative of a captive audience. Also a guideline to implements aspectual connections is provided in Section 3. Nevertheless, ACT implemented in XML, could be used to map the aspectual connections to others AOP languages, or to map following a different implementation approach, such as [8]. In this last case, the ACT serves too, to document and maintain the unit of connection that is implemented in several aspects. The ACT and the mapping process should be used in different stages of software development, as advanced design and implementation stage, or after, during maintenance stage.

At the present time we are developing a tool in order to integrate some techniques to manage aspectual connections: the mapping process presented here; and some methods to handle interactions and relations among aspectual connections in automatic way. Lastly we plan to design and implement a domain specific aspect language (DSAL), for the aspectual connections, in order to overcome semantic and syntactic restrictions of GPAL, such as AspectJ, reported in [8].

7. ACKNOWLEDGMENTS

This work was partially supported by the Universidad Nacional de la Patagonia Austral, Santa Cruz, Argentina.

8. REFERENCES

[1] M. A. Cibrán, D. Suvée, M. D'Hondt, W. Vanderperren, and V. Jonckers. 2004. Integrating Rules with Object-Oriented Software Applications using Aspect-Oriented Programming. In *Proceedings of the Argentine Conference on Computer Science and Operational Research (ASSE'04)*, Córdoba, Argentina.

[2] A. Arsanjani. 2001. Rule object 2001: A Pattern Language for Adaptive and Scalable Business Rule Construction.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns, Elementsof Reusable Object-Oriented Software*. Addison-Wesley.

[4] Business Rule Group. *Hybrid systems "Defining Business Rules: What Are They Really?"* http://www.businessrulesgroup.org/.

[5] Kiczales G., Lamping L., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J. 1997. Aspect-Oriented Programming. *In Proceedings ECOOP'97 – Object-Oriented Programming, 11th European Conference*. Finland, Springer-Verlag.

[6] Laddad R. (2003). "AspectJ in Action" Manning Publications Co.

[7] Kellens A., De Schutter K., D´Hondt T., Jonckers V. and Doggen H. 2008 "Experiences in modularizing business rules into aspects" *ICSM 24 th. IEEE International Conference on Software Maintenance*. Page(s):448 – 451. China.

[8] Cibrán M. and D'Hondt M. 2003. Composable and reusable business rules using AspectJ. *In Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT) at the International Conference on AOSD.* Boston, USA.

[9] Cibrán, M., D'Hondt, M., Suvee, D., Vanderperren, W. and Jonckers, V.(2005) "Linking Business Rules to Object-Oriented software using JAsCo#." Journal of Computational Methods in Sciences and Engineering, pp 13-27, IOS Press, Volume 5(1).

[10] Vidal G., Enriquez J. and Casas S. 2010. Integración de Reglas de Negocio con Conectores Aspectuales Spring. *11th Argentine Symposium on Software Engineering* - Argentina – 2010

[11] Moreira A., Araújo J., and Whittle J. 2006. Modeling Volatile Concerns as Aspects. E. Dubois and K. Pohl (Eds.): *CAiSE 2006,* LNCS 4001, pp. 544 – 558, 2006.Springer-Verlag Berlin Heidelberg.

[12] M. Conejero, J. H. Hernandez, A. Moreira and J. Araújo 2007. Discovering Volatile and Aspectual Requirements Using a Crosscutting Pattern. *15th IEEE International Requirements Engineering Conference*.

[13] K. van der Berg, J. M. Conejero, J. M., and J.Hernández, J. Analysis of Crosscutting in Early Software Development Phases based on Traceability. Transactions on AOSD, Special Issue on Early Aspects – 2007

[14] Casas S. 2010. Clasificación y Documentación de Conexiones Aspectuales para Reglas de Negocio. *I Encuentro Internacional de Computación e Informática del Norte de Chile*. Chile.

[15] Spring Framework Guide http://www.springsource.org/

[16] M. A. Cibrán and M. D'Hondt. A Slice of MDE with AOP: Transforming High-Level Business Rules to Aspects. In *International Conference on Model Driven Engineering Languages and Systems (MODELS'06)*, Genoa, Italy, October 2006. LNCS Springer.

[17] The AspectJ Prog. Guide, http://eclipse.org/aspectj

[18] M. A. Cibrán and M. D'Hondt. High-level specification of business rules and their crosscutting connections. In *8th International Workshop on Aspect-Oriented Modeling at the 5th International Conference on Aspect-Oriented Programming(AOSD'06)*, Bonn, Germany.

[19] Vanderperren, W., Suvee, D., Cibrán, M., Verheecke, B. and Jonckers, V. 2005. Adaptive Programming in JAsCo. *In Proceedings of AOSD*, ACM Press, Chicago, USA